

List of CILC 2008 papers

**Compliance Checking of Execution Traces to Business Rules:
an Approach Based on Logic Programming**

*Marco Montali, Paola Mello, Federico Chesani, Fabrizio Riguzzi, Sergio Storari,
and Maurizio Sebastianis*

Proof Methods for Conditional and Preferential Logics of Nonmonotonic Reasoning

Gian Luca Pozzato

ALC+T: Reasoning About Typicality in Description Logics

Laura Giordano, Valentina Gliozzi, Nicola Olivetti, and Gian Luca Pozzato

Hybrid Automata in System Biology: How far can we go?

Dario Campagna and Carla Piazza

External Point of View in Process Algebra for System Biology

Filippo Del Tedesco and Carla Piazza

Folding Transformation Rules for Constraint Logic Programs

Valerio Senni, Alberto Pettorossi, and Maurizio Proietti

Graded CTL Model Checking

Alessandro Ferrante, Margherita Napoli, and Mimmo Parente

Social Bugs Communities and Intelligent Agents: an Experimental Architecture

Stefania Costantini, Arianna Tocchio, and Mario Scotti Del Greco

Increasing Parallelism while Instantiating ASP Programs

Francesco Calimeri, Simona Perri, and Francesco Ricca

Experimental comparison of two tableau-based decision procedures for MLSS

Domenico Cantone, Pietro Ursino, and Rosario Terranova

The Decidability of the Bernays-Schoenfinkel-Ramsey Class for Set Theory

Eugenio Omodeo and Alberto Policriti

A Refined Calculus for Intuitionistic Propositional Logic

Mauro Ferrari, Camillo Fiorentini, and Guido Fiorino

Actions Over a Constructive Semantics for Description Logics

Loris Bozzato, Mauro Ferrari, and Paola Villa

Lifting Databases to Ontologies

Gisella Bennardo, Giovanni Grasso, Salvatore Maria Ielpa, Nicola Leone, and Francesco Ricca

A Graphical Representation of Relational Formulae with Complementation

Domenico Cantone, Andrea Formisano, Marianna Nicolosi Asmundo, and Eugenio Omodeo

Semantically Augmented DCG Analysis for Next-generation Search Engines

Stefania Costantini, and Alessio Paolucci

Experimenting with Stochastic Prolog as a Simulation Language

Enrico Oliva, Luca Gardelli, Mirko Viroli, and Andrea Omicini

A Nonmonotonic Soft Concurrent Constraint Language for SLA Negotiation

Stefano Bistarelli and Francesco Santini

Modeling and Selecting Countermeasures using CP-nets and Answer Set Programming

Stefano Bistarelli, Fabio Fioravanti, Pamela Peretti, and Irina Trubitsyna

A Java Wrapper for Answer Set Programming Inferential Engines

Giovanni Pirrotta and Alessandro Provetti

Towards Introducing Types in DLV*

Mario Ornaghi, Camillo Fiorentini, and Alberto Momigliano

Modeling Preferences on Resource Consumption and Production in ASP

Stefania Costantini and Andrea Formisano

Eliciting Multi-Dimensional Relational Patterns

Nicola Di Mauro, Teresa Basile, Grazia Bombini, Stefano Ferilli, and Floriana Esposito

Compiling Minimum and Maximum Aggregates into Standard ASP

Mario Alviano, Wolfgang Faber, and Nicola Leone

Normal Form Nested Programs

Annamaria Bria, Wolfgang Faber, and Nicola Leone

GASP: Answer Set Programming with Lazy Grounding

Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi

Compiling and Executing Declarative Modeling Languages in Gecode

Agostino Dovier, Raffaele Cipriano, and Jacopo Mauro

Generalizing Finite Domain Constraint Solving

Federico Bergenti, Alessandro Dal Palù, and Gianfranco Rossi

Compliance Checking of Execution Traces to Business Rules: an Approach based on Logic Programming ^{*}

Federico Chesani¹, Paola Mello¹, Marco Montali¹, Fabrizio Riguzzi², Maurizio Sebastianis³, and Sergio Storari²

¹ DEIS - University of Bologna, V.le Risorgimento 2, 40136 Bologna - Italy
{federico.chesani | paola.mello | marco.montali}@unibo.it,

² ENDIF - University of Ferrara, Via Saragat 1, 44100 Ferrara - Italy
{fabrizio.riguzzi | sergio.storari}@unife.it

³ Think3 Inc., Via Ronzani 7/29, 40033 Casalecchio di Reno (BO) - Italy
maurizio.sebastianis@think3.com

Abstract. Complex and flexible business processes are critical not only because they are difficult to handle, but also because they often tend to be less intelligible. Monitoring and verifying complex and flexible processes becomes therefore a fundamental requirement. We propose a framework for performing compliance checking of process execution traces w.r.t. expressive reactive business rules, tailored to the MXML meta-model. Rules are mapped to (extensions of) Logic Programming, to the aim of providing both monitoring and a-posteriori verification capabilities. We show how different rule templates, inspired by the ConDec language, can be easily specified and then customized in the context of a real industrial case study. We finally describe how the proposed language and its underlying a-posteriori reasoning technique have been concretely implemented as a ProM analysis plug-in.

1 Introduction

Recently, Workflow Management Systems (WfMS) have been increasingly applied by companies in order to efficiently implement their Business Processes. A plethora of tools, systems and notations have been proposed to cover all the phases of the Business Process Management life-cycle, from Process Design and Modeling to Execution and Monitoring/Analysis. To deal with needs and requirements of business users, two main dimensions have been recently tackled: *flexibility* and *complexity*. On one side, to be successfully employed WfMS should make a trade-off between controlling the way workers do their business and turning them loose to exploit their expertise during execution [1]; while constraining workers to follow a business process model, flexible WfMS support the

^{*} An short version of this paper, focused on the application, is currently submitted to the 4th Workshop on Business Process Intelligence (BPI2008), held in conjunction with BPM2008.

possibility of deviating from its prescriptions and even changing it at run-time. On the other side, business processes are exploited to model complex problems and domains under different perspectives (e.g. the control flow perspective and the organizational one); to have an idea of such a complexity, just take a look to the Workflow Patterns [2] initiative⁴.

Both dimensions are critical not only because they are difficult to handle, but also because they contribute to make the process less intelligible. Monitoring and verifying complex and flexible processes becomes therefore a fundamental requirement. On the one hand, as claimed in [3] “deviations from the ‘normal process’ may be desirable but may also point to inefficiencies or even fraud”, and therefore flexibility could lead the organization to miss its strategic goals or even to violate regulations and governance directives. On the other hand, as complexity increases it becomes important to provide support for a business manager in the task of analyzing past/ongoing process executions, in particular to verify whether they meet certain requirements or business rules. This analysis can help the business manager in the process of assessing business trends and consequently making strategic decisions.

In this paper, we focus on this specific task, proposing a framework for performing compliance checking of process execution traces w.r.t. reactive business rules. Such rules are specified by means of a powerful declarative language, inspired by the *SCIFF* one [4]. *SCIFF* is a framework based on Abductive Logic Programming, originally devised for modeling interaction within open Multi-Agent Systems and verifying whether interacting agents indeed comply with the prescribed model. Such a compliance verification can be seamlessly exploited at run-time, by dynamically acquiring and reasoning upon occurring events, or a-posteriori, by analyzing log traces of already completed executions.

In the last few years, *SCIFF* has been applied in the context of Business Process Management and Service Oriented Computing, by investigating its reasoning capabilities to verify a-priori interoperability between a service behavioral interfaces and a choreography [5] and to perform run-time monitoring of exchanged messages checking adherence to choreography rules of engagement [6]. Even more recently, it has been shown that *SCIFF* is able to formalize all core ConDec [7] constraints, supporting temporal-oriented extensions of such graphical languages and providing different underlying verification capabilities [8]. The work here presented is therefore part of a larger framework, called *CLIMB*⁵, which aims at applying (extensions of) Logic Programming for modeling and verifying business processes. Although *CLIMB* business rules stem from ConDec constraints, their expressiveness is extended not only as regards temporal aspects but also as regards event data, such as involved originators, event types and activity identifiers.

The need for a flexible and easy-comprehensible language to specify these kind of rules and the importance of a corresponding compliance verification

⁴ <http://www.workflowpatterns.com>

⁵ The interested reader is referred to <http://www.lia.deis.unibo.it/research/climb>.

framework are motivated by describing its concrete application on a real business process of Think3[®]⁶, a company working in the Computer Aided Design (CAD) and Product Life-cycle Management (PLM) market.

We sketch how the proposed language can be mapped to Logic Programming (SCIFF and Prolog in particular), enabling compliance verification both at runtime and a posteriori. The latter technique has been exploited to implement a ProM [9] plug-in, called SCIFFChecker, that classifies a set of MXML [10] execution traces as compliant/non-compliant w.r.t. a certain business rule, in the style of *LTL Checker* [3].

The paper is organized as follows. Section 2 grounds the compliance checking problem on the Think3 case study. Language and methodology for specifying and applying CLIMB business rules are presented in Section 3. Section 4 briefly sketches how CLIMB rules can be mapped to Logic Programming and illustrates the implementation of a-posteriori compliance checking inside ProM, reporting experiments made on the Think3 case study. Related works and conclusions follow.

2 An industrial case study

An important current challenge in the manufacturing industry is to handle, verify and distribute the technical information produced by the design, development and production processes of the company. The adoption of a system supporting the management of technical data and the coordination of the people involved is of key importance, in order to improve productivity and competitiveness. The main issue is to provide solutions for managing all the technical information and documentation (such as CAD projects, test results, photos, revisions), mainly focusing on the design phase, which produces most such data. Storing and tracking relevant information concerning an item is necessary in this context, because an important part of the design process is spent by testing, modifying and improving previously released versions.

Think3 is one of the leading global players in the field of CAD and PLM solutions: it provides an integrated software which bridges the gap between CAD modeling environments and other tools involved in the process of designing (and then manufacturing) products. All these tools are transparently combined with a non-intrusive information system which handles the underlying product workflow, recording all the relevant information and making it easily accessible to the workers involved, enabling its consultation, use and modification. Such an information system supplies a detailed, shared and constantly updated vision of the life-cycle of each product, as regards documentation of its features as well as traceability of the activities performed on it.

The underlying Think3 workflow centres around the design of a manufacturing product. Different activities can be executed to affect the progress status of an item, involving the modification and even the evolution of multiple co-existing

⁶ <http://www.think3.com>

versions of its corresponding project. Such a workflow can be adapted to each single Think3 client company in order to meet different specific requirements.

2.1 Compliance Checking and Decision Making Support: Think3 Requirements

To support a business manager in decision making, and in particular in the tasks of analyzing the life-cycle of different projects and pinpointing problems and bottlenecks, Think3 is investigating the development of a Business Intelligence dashboard. The feasibility of such a dashboard relies on the *traceability* provided by its solution: all the relevant technical and documental information as well as the history of involved executed activities are stored by the information system. Within the TOCAI.IT FIRB Project⁷, Think3 and the University of Bologna are collaborating for realizing one of the main dashboard components: a tool supporting compliance verification (both on and off-line) of design processes w.r.t. configurable business rules. This will facilitate the manager in the identification of behavioural trends and non-compliances to regulations or internal policies. In this particular case study, we elicited the following non-exhaustive lists of interesting properties:

- (Br1) Quantifying projects which have been subject to a certain activity within a given time interval (e.g., *How many projects have been modified between 01/2007 and 04/2007?*).
- (Br2) Evaluating the time relationship between the execution of two given activities (e.g. *Was a project committed by 18 days after its creation?*).
- (Br3) Identifying which projects passed too many times through a certain activity (e.g., *Which projects have been modified twice?*).
- (Br4) Analysing activities originators, i.e., workers involved in the process (e.g., *Was a project checked by a person different than the one who published it?*).

Note that such rules can be seamlessly exploited either to analyze process executions or in a prescriptive (deontic) manner; for example, rule (Br2) could be used to obtain an overview about projects throughput as well as to monitor workers in order to detect as soon as possible the violation of a certain deadline (and acting consequently).

3 CLIMB Business Rules

We propose a language, inspired by the SCIFF one [4], for specifying reactive business rules (called CLIMB business rules throughout the paper). Their structure resembles the one of ECA (Event-Condition-Action) rules [11, 12]; the main difference is that, since they are used for checking, they envisage expectations about executions rather than actions to be executed. Expectations represent

⁷ <http://www.dis.uniroma1.it/~tocai/>

events that should (not) happen. Therefore, CLIMB rules are used to constrain the process execution when a given situation holds. Both positive and negative constraints can be imposed on the execution, i.e., it is possible to specify what is mandatory as well as forbidden in the process.

Rules follow an **IF** *Body* **having** *BodyConditions* **THEN** *Head* structure, where *Body* is a conjunction of occurred events, with zero or more associated conditions *BodyConditions*, and *Head* is a disjunction of conjunctions of positive and negative expectations (or *false*). Each head element can be subject to conditions as well. An excerpt of the grammar is shown in Figure 1.

$$\begin{aligned}
\textit{Rule} & ::= [\textit{IF } \textit{Body} \textit{ THEN}] \textit{Head} \\
\textit{Body} & ::= \textit{Activity_Exec} \\
& \quad [\textit{AND } \textit{Activity_Exec}]^* [\textit{AND } \textit{Constraints}] \\
\textit{Activity_Exec} & ::= \textit{Simple_Activity} \mid \textit{Repeated_Activity} \\
\textit{Simple_Activity} & ::= \textit{activity } A_ID \textit{ is performed } [\textit{by } O_ID] [\textit{at time } O_T] \\
\textit{Repeated_Activity} & ::= \textit{activity } A_ID \textit{ is performed } N \textit{ times } [\textit{by } O_ID] \\
& \quad [\textit{between time } O_T \textit{ and time } O_T] \\
\textit{Head} & ::= \textit{Head_Disjunct} [\textit{OR } \textit{Head_Disjunct}]^* \\
\textit{Head_Disjunct} & ::= \textit{Activity_Exp} \\
& \quad [\textit{AND } \textit{Activity_Exp}]^* [\textit{AND } \textit{Constraints}] \\
\textit{Activity_Exp} & ::= \textit{Simple_Activity_Exp} \mid \textit{Repeated_Activity_Exp} \\
\textit{Simple_Activity_Exp} & ::= \textit{activity } A_ID \textit{ should } [\textit{not}] \textit{ be performed} \\
& \quad [\textit{by } O_ID] [\textit{at time } O_T] \\
\textit{Repeated_Activity_Exp} & ::= \textit{activity } A_ID \textit{ should be performed } N \textit{ times} \\
& \quad [\textit{by } O_ID] [\textit{between time } O_T \textit{ and time } O_T]
\end{aligned}$$

Fig. 1. An excerpt of the CLIMB rules grammar.

The underlying intuitive semantics is that whenever a set of occurred events makes *Body* (and the corresponding conditions *BodyConditions*) true, then also *Head* must eventually be satisfied⁸ (see Section 4). Furthermore, it is possible to specify rules without the **IF** part: such rules are used to impose what the business manager expects (not) to find inside the process instances in any case.

The concept of event is tailored to the one of audit trail entry in the MXML meta-model [10]. An event is atomic and is mainly characterized by:

- the identifier/name of the *activity* it is associated to;
- an *event type*, according to the MXML transactional model [10];
- an *originator*, identifying the worker who generated the event;
- an *execution time*;
- one or more involved *data* items (for simplicity, in the paper we will not take into account this aspect, but it can be seamlessly treated in our framework).

⁸ Therefore, rules having *false* in the head are used to express denials.

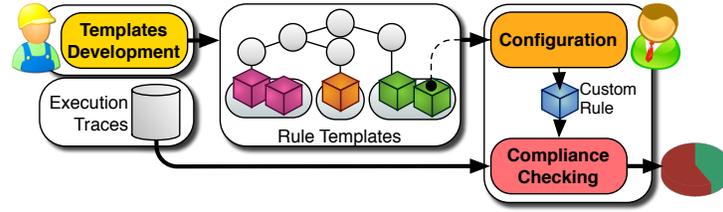


Fig. 2. A methodology for building, configuring and applying business rules.

The main distinctive feature of our rules is that all these parameters are treated, by default, as variables. To specify that a generic activity A has been subject to a whatsoever event, the rule body will simply contain a string like: *activity A is performed by O_A at time T_A* , where A stands for the activity's name, O_A and T_A represent the involved originator and execution time respectively, and *performed* is a keyword denoting any event type. To facilitate readability, the part concerning originator and execution time can be omitted if the corresponding variables are not involved in any condition.

Such a generic sentence will match with any kind of event, because all the involved variables (A , O_A and T_A) are completely free, and the event type is not specified. It can then be configured to constrain involved variables or ground them to specific values, and by fixing a specific event type. To deal with the first case, explicit conditions are attached to variables, whereas for the latter case it is enough to substitute the generic *performed* keyword with the specific one (e.g., *completed* to represent the completion of a certain activity).

Finally, positive and negative expectations are represented similarly to occurred events, by simply changing the *is* part with *should be* or *should not be* respectively.

3.1 A Methodology for Building Rules

To clarify the methodology we propose for developing rules, let us consider a completely configured business rule, namely the specification of the (Br4) rule:

IF activity A is performed by O_A having A equal to *Check*
 THEN activity B should NOT be performed by O_B (Think3-4)
 having B equal to *Publish* and O_B equal to O_A

By analyzing this rule, we can easily recognize two different aspects: on one hand, the rule contains generic elements, free variables and constraints, whereas on the other hand it specifically refers to concrete activities. The first aspect captures re-usable patterns: in this case, the fact that the same person cannot perform two different activities A and B , which is known as the *four-eyes principle*. The second aspect instantiates the rules in a specific domain, in this case grounding the four-eyes principle in the context of Think3's workflow.

To reflect such a separation, we foresee a three-step methodology to build, configure and apply business rules (see Figure 2): *(i)* a set of re-usable rules, called *rule templates*, are developed and organized into groups by a technical expert (i.e., someone having a deep knowledge of rules syntax and semantics); *(ii)* rule templates are further configured, constrained and customized by a business manager to deal with her specific requirements and needs; *(iii)* configured rules are exploited to perform compliance checking of company’s execution traces.

In the next sections we will introduce the different conditions supported by the language, and we will propose a core set of rule templates, inspired by ConDec constraints, showing how they can be easily customized to deal with the Think3 case study.

3.2 Specification of Conditions

As already pointed out, conditions are exploited to constrain variables associated to event occurrences and expectations inside business rules (namely activity names, originators and execution times). Two main families of conditions are currently envisaged: string and time conditions. String conditions are used to constrain an activity/originator by specifying that it is equal to or different than another activity/originator, either variable or constant. An example of a string condition constraining two originator variables is the “ O_B equal to O_A ” part in rule (Think3-4).

Time conditions are used instead to relate execution times, in particular for specifying ordering among events or imposing quantitative constraints, such as deadlines and delays. The semantics of constraints is determined by time operators, which intuitively capture basic time relationships (such as *before* or *at*). Absolute time conditions constrain a time variable w.r.t. a certain time/date, whereas relative time conditions define orderings and constraints between two variables. Relative conditions can optionally attach a displacement to the target time variable as well. A displacement is defined by a time duration and by an operator which determines whether the duration will translate the involved time variable forward or backward in time. For example, to specify that the time variable T_B must be within 2 days *after* T_A , we simply write T_B BEFORE $T_A+2days$.

3.3 Rule Templates

The first step in our methodology envisages the creation of rule templates, i.e. re-usable partially constrained rules. They typically fix the rule structure, e.g. deciding how many events are contained in the body, and use variable string conditions and/or relative time conditions. They do not involve absolute time and constant conditions, which are exploited to ground rules on a specific domain.

We have developed a hierarchy of rules which strictly resembles the one proposed for ConDec. Part of this hierarchy is depicted in Figure 3. Three basic groups, similar to the ConDec ones, are defined: *existence* rules, *IF... THEN* rules and *IF... THEN NOT* rules. Such hierarchy is not fixed: it can be adapted or even replaced by writing other rules and by organizing them differently.

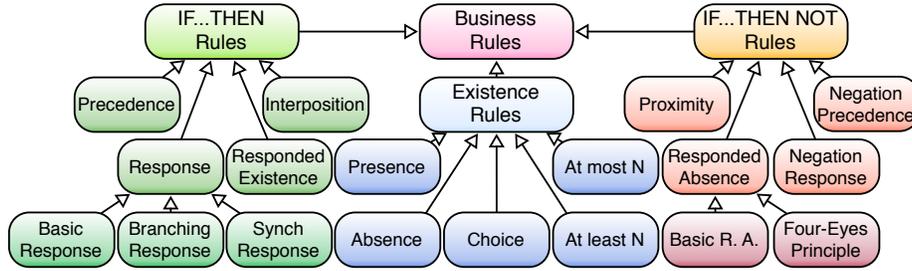


Fig. 3. A templates hierarchy inspired by Condec constraints.

Existence Rules impose the presence/absence of some events in the execution trace, independently from the occurrence of other events (except from the *At Most N* rule), they therefore have a *true* body. The *presence* (*absence*) template simply state that a certain event is expected to occur (not to occur), and is simply formalized as: *activity A should (NOT) be performed*. *Choice* extends *presence* by introducing disjunction of expectations. The *at least N* (*at most N*) rule extends the presence (absence) one by stating that the specified event should (not) be repeated *N* times. Such rules are useful for modeling the presence/absence of multiple instances of a certain event in the execution trace, as in the (Br3) Think3 example, but they are rather difficult to be represented, especially when *N* increases.

For this reason, we have extended the syntax of the language for supporting repetitions as first-class entities. To specify that activity *A* should be performed at least 3 times, we will then write: *activity A should be performed 3 times between T_{sA} and T_{cA}* . The two involved time variables extend the concept of execution time when dealing with multiple events, by identifying the two time points at which the repetition starts and completes⁹. Finally, to express that *A* must be performed at most 3 times, we can exploit repetitions as follows: *IF activity A is performed 4 times THEN false*.

IF... THEN Rules are positive relationships which specify that when certain events happen, then also other events should occur, satisfying the imposed time orderings. The simplest rule belonging to this group is the *responded existence* one, which simply states that when a certain event happens, then another event should happen too, either before or afterward. Starting from this rule, *response* and *precedence* templates extend it by adding respectively an *after* and *before* relative time condition among the involved execution times.

Response and precedence rules can then be specialized to express more complex event patterns, e.g. introducing conjunctions and disjunctions of events. For example, the following template represents a *synchronized response*, i.e. a

⁹ This kind of rules obviously involve multiple originators, but for space reasons we do not describe here how they can be constrained.

response triggered by the occurrence of two events:

```
IF activity A is performed at time  $T_A$ 
and activity B is performed at time  $T_B$ 
THEN activity C should be performed at time  $T_C$ 
having  $T_C$  after  $T_A$  and  $T_C$  after  $T_B$ .
```

By inverting the last temporal condition (i.e. by imposing T_C before T_B) the user can express an *interposition* template, which states that *C* should be performed between *A* and *B*. Finally, note that more complex ConDec constraints can be specified by combining two different templates: for example, the *alternate precedence* ConDec constraint is specified by means of CLIMB response and interposition.

IF...THEN NOT Rules are the negative version of positive relationships: they express events to be forbidden when other events happen. Roughly speaking, they replace positive expectations with negative ones; e.g., the negation of the interposition template is the *proximity* one, which states that between two given activities *A* and *B* another activity *C* cannot be performed¹⁰.

The *responded absence* pattern is a good example to illustrate templates which involve conditions about originators; in its basic form it states that **IF activity *A* is performed THEN activity *B* should not be performed**, but by adding an *equal to* constraint between the originators of *A* and *B*, it actually models the already cited four-eyes principle.

3.4 From Templates to Customized Business Rules

In a second phase, rule templates are configured by a business manager to deal with her specific requirements. This step exploits constant string conditions, absolute time conditions and relative time conditions with displacements, involving activities, originators and times specifically referring to the company's domain. For example, while Rule (Br3) simply grounds the *at least N* template with the *Modify* activity, rule (Br2) is formalized by adding a *before* relative time constraint with a displacement of 18 days, thus modeling the presence of a deadline:

```
IF activity A is performed at time  $T_A$ 
having A equal to Creation
THEN activity B should be performed at time  $T_B$  (Think3-2)
having B equal to Commit
and  $T_B$  after  $T_A$  and  $T_B$  before  $T_A + 18days$ .
```

¹⁰ If activity *C* is left unspecified also in the customized version, it will match with any performed activity, and therefore the rule will be used for checking that *A* and *B* are next to each other.

Rule (Br1) constrains the presence template with absolute time conditions:

activity A should be *performed* at time T_A
 having A equal to *Modify* and (Think3-1)
 T_A after 01/2007 and T_A before 04/2007.

4 Compliance Verification with Logic Programming

Having described syntax and features of our rules language, we now briefly sketch how they can be automatically mapped to two Logic Programming frameworks (namely *SCIFF* [4] and Prolog), enabling monitoring (run-time compliance verification) and a-posteriori compliance checking of process execution traces w.r.t. *CLIMB* rules.

Mapping to *SCIFF* is straightforward because of the deep similarities between the two languages. For space reasons, we report here a mapping example, referring to [8] for more details. Occurred events and positive/negative expectations are directly mapped to *SCIFF* happened events and expectations, represented respectively by the special predicates $\mathbf{H}(Ev, T)$ and $\mathbf{E}/\mathbf{EN}(Ev, T)$ where Ev is an event with the structure proposed in this paper and T is the corresponding execution time. For example, the customized four-eyes principle shown in rule (Think3-4) is simply specified in *SCIFF* as follows¹¹:

$$\begin{aligned}
 & \mathbf{H}(\text{event}(_, A, O_A), T_A) \wedge A = \text{'Check'} \\
 & \rightarrow \mathbf{EN}(\text{event}(_, B, O_B), T_B) \wedge A = \text{'Publish'} \wedge O_B \neq O_A.
 \end{aligned}$$

The constraints among the variables, and in particular those on the activity execution times, are treated in *SCIFF* by means of Constraint Logic Programming.

The declarative semantics of a *SCIFF* specification is given in terms of an Abductive Logic Program (ALP). In general, an ALP [13] is a triple $\langle P, A, IC \rangle$, where P is a logic program, A is a set of predicates named *abducibles*, and IC is a set of integrity constraints. Reasoning in abductive logic programming is usually goal-directed and it accounts to finding a set of abduced hypotheses Δ built from predicates in A such that $P \cup \Delta \models G$ and $P \cup \Delta \models IC$ where G is a goal. Abduction is exploited to dynamically *generate* the expectations and to perform the *compliance check*. Expectations are defined as abducibles, and are hypothesised by the abductive proof procedure, i.e. the proof procedure makes hypotheses about the happening of the events. A confirmation step, where these hypotheses must be confirmed by happened events, is then performed: if no set of hypotheses can be fulfilled, a violation is detected.

Given a set **HAP** representing a process execution trace, a knowledge base KB , the set \mathcal{E} of positive and negative expectations, and a set \mathcal{IC} (obtained

¹¹ The first parameter of event is an anonymous variable because it represents the event type, which is not specified when the keyword *performed* is used.

by translating CLIMB rules), we provide semantics to a SCIFF specification $\mathcal{S} \equiv \langle KB, \mathcal{E}, \mathcal{IC} \rangle$ by defining those sets $\mathbf{EXP} \subseteq \mathcal{E}$ ($\Delta \subseteq A$ in the abductive framework) of expectations which, together with the knowledge base and the happened events \mathbf{HAP} , imply an instance of the goal (Eq. 1) - if any - and *satisfy* the integrity constraints (Eq. 2).

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{G} \quad (1)$$

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{IC} \quad (2)$$

Moreover, we require the set \mathbf{EXP} (namely, an *abductive explanation*) to be also **E-consistent**: for any p , \mathbf{EXP} cannot include $\{\mathbf{E}(p), \mathbf{EN}(p)\}$ (an event cannot be both expected to happen and expected not to happen).

We define the *compliance* of a process execution trace \mathbf{HAP} with respect to a SCIFF specification (composed of a knowledge base KB and of a set \mathcal{IC} of rules) in terms of the fulfillment of a \mathbf{EXP} set of expectations: each positive expectation should have a matching happened event, and for each negative expectation there should not be any matching happened event.

Definition 1. (Fulfillment) *Given a process execution trace \mathbf{HAP} and a SCIFF specification, a set of expectations \mathbf{EXP} that is **E-consistent** is fulfilled if and only if for all (ground) terms p :*

$$\mathbf{HAP} \cup \mathbf{EXP} \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)\} \neq \text{false} \quad (3)$$

Definition 2. (Compliance) *Given an execution trace \mathbf{HAP} and a SCIFF specification \mathcal{S} , \mathbf{HAP} is compliant to \mathcal{S} if and only if there exists an **E-consistent** set \mathbf{EXP} such that Equations 1 and 2, and Definition 1 hold.*

SCIFF has an operational proof-theoretic counterpart whose main purpose is (i) to dynamically acquire occurred events during execution, (ii) to generate expectations when the corresponding rule's body is triggered by such happened events, and (iii) to finally verify if the execution actually complies with the expectations raised. SCIFF supports the compliance verification task both at run-time and a-posteriori. At run-time, the proof procedure is exploited by the SOCS-SI tool [4] to dynamically monitor and analyze the process behaviour. A posteriori, the same SCIFF proof can be exploited to analyze the execution trace of a process. While in the following we will concentrate on the verification a posteriori, the interested reader can refer to [4] for run-time verification.

4.1 SCIFFChecker: Compliance Checking in ProM

If compliance checking is performed a-posteriori (and therefore reactivity is not required anymore), then also pure Prolog can be exploited to reason upon execution traces, with an appreciable advantage in terms of performances with respect to the SCIFF proof procedure. In this setting, the execution trace under study is treated as a knowledge base storing each audit trail entry as a fact of the type

$happened(event(EventType, ActivityName, Originator), ExecutionTime).$

The rule used for checking is instead transformed into a Prolog query by computing the negation of the implication represented by the CLIMB rule. So, if the CLIMB rule is represented by the implication $B \rightarrow H$, then the query would be $B \wedge \neg H$. Such a query tries to find a set of occurred events in the execution trace that satisfy the rule body but violate the rule head. For example, rule (Br4) is translated to the following query:

$$\begin{aligned} ?-A = \text{'Check'}, & \text{happened}(event(-, A, O_A), T_A), \\ & \text{not}(B = \text{'Publish'}, O_B \neq O_A, \text{not}(\text{happened}(event(-, B, O_B), T_B))). \end{aligned}$$

Note that since the analysis is performed a-posteriori, positive expectations are flattened to occurred events, and negative expectations to the absence of events. If the query succeeds, then a counter-example which violates the rule has been found in the execution trace, which is then evaluated as non-compliant. Although Prolog does not provide any explanation about the reasoning outcome (as *SCIFF* does), it turns out to be very efficient, in particular if a huge number of execution traces have to be checked. However, the Prolog translation can be applied only to a subset of all the possible *SCIFF* rules: this is not a problem when translating CLIMB rules, since they belong to such a set.

Drawing inspiration from the *LTL Checker* [3], we have therefore developed a ProM [9] analysis plug-in, called *SCIFFChecker*, for the classification of MXML execution traces w.r.t. CLIMB business rules. In particular, we took inspiration from *LTL Checker* for what regards the functionalities offered, while not relying on temporal logic. *SCIFFChecker* relies on the three-steps methodology described in Section 3.1, providing a user-friendly graphical interface for the customization of rule templates.

At start-up, rule templates are loaded from a templates file: the available templates follows the classification proposed in Section 3.3. More templates can be added by simply extending the templates file. Once a template has been chosen, it can be easily customized by clicking the “configuration” button (Figure 4): the modifiable elements becomes highlighted, and the user can set specific parameters by clicking on them. To proceed with the compliance checking, the user has to choose also a time *granularity*, which ranges from milliseconds to months and defines the time unit for converting involved time quantities into coherent integer values.

For each execution trace contained in the considered MXML log, three steps are then automatically performed: (i) the execution trace is translated into a Prolog knowledge base, converting involved execution times; (ii) the CLIMB business rule is mapped to a Prolog query, repetitions are re-written as conjunction of simple sentences, and dates and time displacements are converted; (iii)

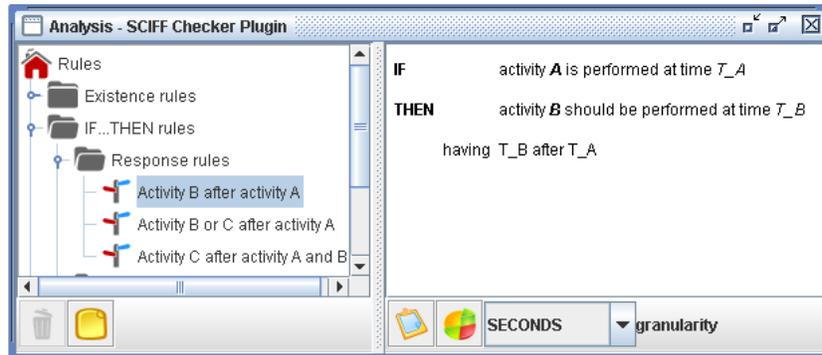


Fig. 4. A screenshot of the main *SCIFFChecker* window.

finally a Prolog engine based on SWI¹² is exploited to perform the compliance checking.

All verification outcomes are collected and a summarizing pie chart is shown¹³, together with the explicit list of compliant/non-compliant traces (Figure 5). At this point, the user can start a new classification by considering either the whole log or only the compliant/non-compliant execution traces. In this way, a conjunction of CLIMB rules can be verified by performing a sequence of tests, each dealing with only one rule, and by then selecting the compliant/non-compliant resulting set.

4.2 Applying *SCIFFChecker* to the Think3 Case Study

SCIFFChecker has been concretely applied to analyze execution traces of a Think3 client. We have first exploited the ProM Import tool¹⁴ in order to convert the relevant information from the client database into an MXML format, by considering the project name as the case identifier. In particular, we extracted a portion of 9000 execution traces, ranging from 4 to 15 events. We then used, together with a Think3 business manager, the plug-in to express and test the business rules of interest described in Section 3.4. The overall average time for performing compliance checking have been assessed to be around 10-12 seconds. The verification outcomes have been finally analyzed by the business manager, who found them useful in order to obtain a clear overview about the overall process behavior. For example, considering rules (Br3) and (Br4), we discovered that, fortunately, only the 2% of execution traces involved more than two project revisions, and that in only the 3.5% cases the same person was responsible for both publishing and checking the project.

¹² <http://www.swi-prolog.com/>

¹³ Thanks to the JFreeChart library, available from <http://www.jfree.org/jfreechart>.

¹⁴ <http://promimport.sourceforge.net>

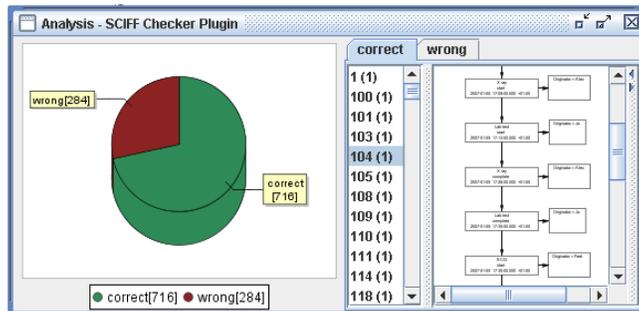


Fig. 5. Compliance chart produced by *SCIFFChecker* at the end of verification.

The verification of rules like (Br2) was found interesting especially by varying the deadline involved. Indeed, the business manager wanted to detect projects taking too much time as well as projects released too soon, to point out both possible bottlenecks and potential inaccuracies. This kind of rule would be even more useful in a monitoring perspective: it would allow to identify at run-time non-compliant projects and to take as soon as possible specific countermeasures or further analysis.

5 Related Works

The closest work to the one here presented is the ProM *LTL-Checker* [3], that shares with our approach the motivation and purposes. While *LTL-Checker* exploits linear temporal logic for the formalization of properties, our approach belongs to the Logic Programming setting; CLIMB business rules are more expressive for what concerns the possibility of expressing rules triggering when a conjunction of events occur, and regarding the support of relative time constraints. Furthermore, through the mapping to *SCIFF*, we can seamlessly check compliance at run-time, monitoring running process instances. A posteriori verification of execution traces has been deeply investigated also when the considered model is a procedural specification rather than a set of declarative business rules. For example, in [14] the authors tackle “conformance testing” between execution traces and a Petri Net-based process model, introducing metrics to measure how well a given execution fits into the model.

Our work, as part of the CLIMB framework, belongs to the recently investigated research stream aiming at exploiting declarative models and underlying formal methods for specifying and verifying IT-systems. The application of Logic Programming techniques to formalize and verify loosely-coupled processes and business rules has taken inspiration from ConDec [7].

An interesting research topic concerns the integration of *SCIFFChecker* with the process mining algorithm described in [15], which follows the opposite direction: it aims at discovering a set of declarative business rules, specified in the *SCIFF* language, starting from execution traces previously classified as correct

or wrong, s.t. the mined specification evaluates as compliant the correct subset and as non-compliant the wrong one. We could first mine a set of CLIMB business rules, use them to classify new traces, or exploit *SCIFFChecker* to split a given MXML log into the wrong and correct subsets required as input for the mining algorithm. The latter approach could be exploited to discover a declarative model giving an *explanation* of the *SCIFFChecker* classification.

6 Conclusions and Future Works

We have described a framework for checking the compliance of process execution traces to declarative reactive business rules, proposing a three-steps methodology for developing and applying rules. The approach has been tested on a real industrial case study, identifying what kind of rules the Think3 company would be able to check and showing how they can be easily expressed by customizing rule templates (re-usable patterns resembling ConDec constraints). In order to effectively use such rules for reasoning, we have sketched how they can be mapped to (extensions of) Logic Programming, namely *SCIFF* and Prolog. The two mappings allow the monitoring of process executions at run-time or the checking of the compliance of already completed execution traces, helping a business manager in the assessment of business trends and providing decision making support. The latter approach has been implemented as a ProM plug-in that classifies MXML execution traces w.r.t. CLIMB business rules.

Many ongoing and future works concern both the framework itself and its ProM implementation. Regarding the underlying formal framework, we plan to investigate the relationships between Logic Programming and other formal languages, such as temporal logics, and to evaluate to what extent CLIMB rules can be expressed as queries in temporal databases [11]. We are applying our plug-in to different domains, such as a regional screening protocol, and, in cooperation with ENEA, in the context of a chemo-physical process of wastewater treatment plants. The tool will be extended, by introducing the possibility of dealing with case and event data. Finally, we will investigate how *SCIFFChecker* and the mining algorithm presented in [15] can be jointly exploited to realize a declarative checking-mining cycle, discovering untrivial explanations for a given classification.

Acknowledgments This work has been partially supported by the FIRB project *TOCAI.IT: Tecnologie orientate alla conoscenza per aggregazioni di imprese in internet*.

References

1. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In Meersman, R., Tari, Z., eds.: Proceedings of the OTM 2007 Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS. Volume 4803 of LNCS., Springer (2007) 77–94

2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. *Distributed and Parallel Databases* **14**(1) (2003) 5–51
3. van der Aalst, W., de Beer, H., van Dongen, B.: Process Mining and Verification of Properties: An Approach based on Temporal Logic. In Meersman, R., Tari, Z., eds.: *Proceedings of the OTM 2005 Confederated International Conferences CoopIS, DOA, and ODBASE*. Volume 3760. (2005) 130–147
4. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic* **9**(4) (10 2008) To appear.
5. Alberti, M., Gavanelli, M., Lamma, E., Chesani, F., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In Bossi, A., Maher, M.J., eds.: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM (2006) 39–50
6. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Storari, S., Torroni, P.: Computational logic for run-time verification of web services choreographies: Exploiting the *socs-si* tool. In Bravetti, M., Núñez, M., Zavattaro, G., eds.: *Proceedings of the Third International Workshop on Web Services and Formal Methods (WS-FM 06)*. Volume 4184 of LNCS., Springer (2006) 58–72
7. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In Eder, J., Dustdar, S., eds.: *Proceedings of the BPM 2006 Workshops*. Volume 4103 of LNCS., Springer (2006) 169–180
8. Chesani, F., Mello, P., Montali, M., Storari, S.: Towards a decserflow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy (2007)
9. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W., Weijters, A.J.M.M.: ProM 4.0: Comprehensive Support for Real Process Analysis. In Kleijn, J., Yakovlev, A., eds.: *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*. Volume 4546. (2007) 484–494
10. van Dongen, B.F., van der Aalst, W.M.P.: A Meta Model for Process Mining Data. In Casto, J., Teniente, E., eds.: *Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop)*. Volume 2., FEUP, Porto, Portugal (2005) 309–320
11. Pissinou, N., Snodgrass, R.T., Elmasri, R., Mumick, I.S., Özsu, T., Pernici, B., Segev, A., Theodoulidis, B., Dayal, U.: Towards an infrastructure for temporal databases: report of an invitational arpa/nsf workshop. *SIGMOD Rec.* **23**(1) (1994) 35–51
12. Bailey, J., Bry, F., Eckert, M., Patranjan, P.L.: Flavours of xchange, a rule-based reactive language for the (semantic) web. In Adi, A., Stoutenburg, S., Tabet, S., eds.: *RuleML*. Volume 3791 of LNCS., Springer (2005) 187–192
13. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. *Journal of Logic and Computation* **2**(6) (1993) 719–770
14. Rozinat, A., van der Aalst, W.: Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In Bussler et al., C., ed.: *BPM 2005 Workshops (Workshop on Business Process Intelligence)*. Volume 3812. (2006) 163–176
15. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In Alonso, G., Dadam, P., Rosemann, M., eds.: *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*. Volume 4714 of LNCS., Springer (2007) 344–359

Proof Methods for Conditional and Preferential Logics of Nonmonotonic Reasoning

Gian Luca Pozzato

Dipartimento di Informatica - Università degli Studi di Torino
pozzato@di.unito.it

Abstract. Conditional and Preferential Logics have been applied in order to formalize nonmonotonic reasoning. In spite of their significance, very few proof methods have been proposed for these logics. In my PhD thesis [10], whose content is summarized in this paper, I have tried to partially overwhelm this gap, by introducing sequent and tableau calculi for Conditional and Preferential Logics.

1 Introduction

Many systems exhibiting a nonmonotonic behavior have been studied in the literature; negation as failure, Circumscription, Default logic are only some examples. Each of these systems deserves an independent interest, however it is not clear that any one of them really captures the whole properties of nonmonotonic reasoning. An alternative solution to the formalization of nonmonotonic reasoning consists in the application of *conditional and preferential logics*.

Conditional logics have a long history. They have been studied first by Lewis and Stalnaker [7] in order to formalize a kind of hypothetical reasoning (“if A were the case then B ”), that cannot be captured by classical logic with material implication. Moreover, they have been used to formalize *counterfactual sentences*, i.e. conditionals of the form “if A were the case then B would be the case”, where A is false.

In the last years, there has been a considerable amount of work on applications of conditional logics to various areas of artificial intelligence and knowledge representation. For instance, they have been used to formalize knowledge update and revision. Moreover, conditional logics have been used to model hypothetical queries in deductive databases and logic programming. In a related context, conditional logics have been used to model causal inference and reasoning about action execution in planning.

The application of conditional logics to nonmonotonic reasoning was firstly investigated by Delgrande [2] who proposed a conditional logic for prototypical reasoning; the understanding of a conditional $A \Rightarrow B$ in his logic is “the A 's have typically the property B ”. For instance, one could have: $\forall x(Penguin(x) \rightarrow Bird(x))$, $\forall x(Penguin(x) \rightarrow \neg Fly(x))$, $\forall x(Bird(x) \Rightarrow Fly(x))$. The last sentence states that birds typically fly. Observe that replacing \Rightarrow with the classical implication \rightarrow , the above knowledge base is consistent only if there are no penguins.

The study of the relations between conditional logics and nonmonotonic reasoning has gone much further since the seminal work by Kraus, Lehmann, and Magidor [5, 6] (KLM framework), who proposed a formalization of the properties of a nonmonotonic consequence relation. According to KLM framework, a defeasible knowledge base is represented by a (finite) set of nonmonotonic conditionals of the form $A \sim B$, whose reading is *normally (or typically) the A's are B's*. The operator “ \sim ” is nonmonotonic, in the sense that $A \sim B$ does not imply $A \wedge C \sim B$. For instance, a knowledge base K may contain $football_lover \sim bet$, $football_player \sim football_lover$, $football_player \sim \neg bet$, whose meaning is that people loving football typically bet on the result of a match, football players typically love football but they typically do not bet (especially on matches they are going to play...). If \sim were interpreted as classical implication, one would get $football_player \sim \perp$, i.e. typically there are not football players, thereby obtaining a trivial knowledge base. In KLM framework, the set of adopted inference rules defines some fundamental types of inference systems, namely, from the strongest to the weakest: Rational (**R**), Preferential (**P**), Loop-Cumulative (**CL**), and Cumulative (**C**) logic. In all these systems one can infer new assertions without incurring the trivializing conclusions of classical logic: in the above example, in none of the systems can one infer $football_player \sim bet$. In cumulative logics (both **C** and **CL**) one can infer $football_lover \wedge football_player \sim \neg bet$, giving preference to more specific information; in Preferential logic **P** one can also infer that $football_lover \sim \neg football_player$; in the rational case **R**, if one further knows that $\neg(football_lover \sim rich)$, that is to say it is not the case that football lovers are typically rich persons, one can also infer that $football_lover \wedge \neg rich \sim bet$. The logics of the KLM framework are also known as Preferential logics.

The connection between conditional and preferential logics has been largely investigated. It turns out that all forms of inference studied in KLM framework are particular cases of well-known conditional axioms [1]. In this respect the KLM language is just a fragment of conditional logics.

In spite of their significance, very few proof systems have been proposed for conditional and preferential logics. In my PhD thesis [10], whose content is summarized in this paper, I have tried to (partially) overwhelm this gap, by providing analytic calculi for some standard conditional logics and for all the four KLM systems.

2 Conditional Logics

Conditional logics are extensions of classical logic by the conditional operator \Rightarrow . We restrict our concern to propositional conditional logics.

A propositional conditional language \mathcal{L} is defined from a set of propositional variables ATM , the symbols of *false* \perp and *true* \top , and a set of connectives \neg , \wedge , \vee , \rightarrow , and \Rightarrow . Formulas of \mathcal{L} are defined as usual: \perp , \top and the propositional variables of ATM are *atomic formulas*; if A and B are formulas, then $\neg A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$ and $A \Rightarrow B$ are *complex formulas*.

Similarly to modal logics, the semantics of conditional logics can be defined in terms of possible world structures. In this respect, conditional logics can be seen as a generalization of modal logics where the conditional operator is a sort of modality indexed by a formula of the same language. We adopt the *selection function semantics*: intuitively, the selection function f selects, for a world w and a formula A , the set of worlds of \mathcal{W} which are *closer* to w given A . A conditional formula $A \Rightarrow B$ holds in a world w if the formula B holds in *all the worlds selected by f for w and A* .

Formally, a selection function model is a triple $\mathcal{M} = \langle \mathcal{W}, f, [\] \rangle$ where \mathcal{W} is a non empty set of *worlds*, $f: \mathcal{W} \times 2^{\mathcal{W}} \rightarrow 2^{\mathcal{W}}$ is the *selection function*, and $[\]$ is the *evaluation function*, which assigns to an atom $P \in \text{ATM}$ the set of worlds where P is true, and is extended to the other formulas as follows: $[\perp] = \emptyset$; $[\top] = \mathcal{W}$; $[\neg A] = \mathcal{W} - [A]$; $[A \wedge B] = [A] \cap [B]$; $[A \vee B] = [A] \cup [B]$; $[A \rightarrow B] = (\mathcal{W} - [A]) \cup [B]$; $[A \Rightarrow B] = \{w \in \mathcal{W} \mid f(w, [A]) \subseteq [B]\}$. Notice that we have defined f taking $[A]$ rather than A ; this is equivalent to define f on formulas, i.e. $f(w, A)$ but imposing that if $[A] = [A']$ in the model, then $f(w, A) = f(w, A')$. This condition is called *normality*.

The semantics above characterizes the *basic normal conditional system*, called CK. As in modal logic, extensions of the basic system CK are obtained by assuming further properties on the selection function. We consider the following standard extensions of the basic system CK:

System	Axiom	Model condition
ID	$A \Rightarrow A$	$f(w, [A]) \subseteq [A]$
MP	$(A \Rightarrow B) \rightarrow (A \rightarrow B)$	$w \in [A] \rightarrow w \in f(w, [A])$
CS	$(A \wedge B) \rightarrow (A \Rightarrow B)$	$w \in [A] \rightarrow f(w, [A]) \subseteq \{w\}$
CEM	$(A \Rightarrow B) \vee (A \Rightarrow \neg B)$	$ f(w, [A]) \leq 1$

2.1 Sequent Calculi SeqS

In Figure 1 sequent calculi SeqS for conditional logics are presented. S stands for $\{\text{CK, ID, MP, CEM, CS}\}$ and all their combinations (except for those containing both CEM and MP). The calculi make use of *labelled formulas*, where the labels are drawn from a denumerable set \mathcal{A} ; there are two kinds of formulas: *world formulas*, denoted by $x : A$, where $x \in \mathcal{A}$ and $A \in \mathcal{L}$; *transition formulas*, denoted by $x \xrightarrow{A} y$, where $x, y \in \mathcal{A}$ and $A \in \mathcal{L}$. A world formula $x : A$ is used to represent that A holds in the possible world represented by the label x ; a transition formula $x \xrightarrow{A} y$ represents that $y \in f(x, [A])$.

A *sequent* is a pair $\langle \Gamma, \Delta \rangle$, usually denoted with $\Gamma \vdash \Delta$, where Γ and Δ are multisets of labelled formulas. The intuitive meaning of $\Gamma \vdash \Delta$ is: every model that satisfies all labelled formulas of Γ in the respective worlds (specified by the labels) satisfies at least one of the labelled formulas of Δ (in those worlds).

In [10] it is shown that SeqS calculi are sound and complete with respect to the selection function semantics, i.e. a sequent $\Gamma \vdash \Delta$ is valid if and only if $\Gamma \vdash \Delta$ is derivable in SeqS. Moreover, SeqS calculi can be used to describe a

$(A\perp) \frac{\Gamma, x : \perp \vdash \Delta}{\Gamma, x : A \rightarrow B \vdash \Delta}$	$(AX) \Gamma, x : P \vdash \Delta, x : P \quad (P \in ATM)$	$(A\top) \Gamma \vdash \Delta, x : \top$
$(\rightarrow L) \frac{\Gamma \vdash \Delta, x : A \quad \Gamma, x : B \vdash \Delta}{\Gamma, x : A \rightarrow B \vdash \Delta}$		$(\rightarrow R) \frac{\Gamma, x : A \vdash \Delta, x : B}{\Gamma \vdash \Delta, x : A \rightarrow B}$
$(\Rightarrow L) \frac{\Gamma, x : A \Rightarrow B \vdash \Delta, x \xrightarrow{A} y \quad \Gamma, x : A \Rightarrow B, y : B \vdash \Delta}{\Gamma, x : A \Rightarrow B \vdash \Delta}$		$(\Rightarrow R) \frac{\Gamma, x \xrightarrow{A} y \vdash \Delta, y : B}{\Gamma \vdash \Delta, x : A \Rightarrow B}$
$(EQ) \frac{u : A \vdash u : B \quad u : B \vdash u : A}{\Gamma, x \xrightarrow{A} y \vdash \Delta, x \xrightarrow{B} y}$		
$(CEM) \frac{\Gamma, x \xrightarrow{A} y \vdash \Delta, x \xrightarrow{A} z \quad (\Gamma, x \xrightarrow{A} y \vdash \Delta)[y, z/u]}{\Gamma, x \xrightarrow{A} y \vdash \Delta}$		$(MP) \frac{\Gamma \vdash \Delta, x \xrightarrow{A} x, x : A}{\Gamma \vdash \Delta, x \xrightarrow{A} x}$
$(CS) \frac{\Gamma, x \xrightarrow{A} y \vdash \Delta, x : A \quad (\Gamma, x \xrightarrow{A} y \vdash \Delta)[x, y/u]}{\Gamma, x \xrightarrow{A} y \vdash \Delta}$		$(ID) \frac{\Gamma, x \xrightarrow{A} y, y : A \vdash \Delta}{\Gamma, x \xrightarrow{A} y \vdash \Delta}$

Fig. 1. Sequent calculi SeqS. Rules (ID), (MP), (CEM) and (CS) are only used in corresponding extensions of the basic system SeqCK. To save space, we omit the standard rules for the other boolean connectives.

decision procedure for conditional logics and to study their complexity, namely it is shown that provability in the studied logics is decidable in polynomial space.

3 Preferential Logics

We consider a propositional language \mathcal{L} defined from a set of propositional variables ATM , the boolean connectives and the conditional operator \sim . We use A, B, C, \dots to denote propositional formulas, whereas F, G, \dots are used to denote all formulas (including conditionals). The formulas of \mathcal{L} are defined as follows: if A is a propositional formula, $A \in \mathcal{L}$; if A and B are propositional formulas, $A \sim B \in \mathcal{L}$; if F is a boolean combination of formulas of \mathcal{L} , $F \in \mathcal{L}$.

In general, the semantics of KLM logics is defined by considering possible world (or possible states) structures with a *preference relation* $w < w'$ among worlds (or states), whose meaning is that w is preferred to w' . $A \sim B$ holds in a model \mathcal{M} if B holds in all *minimal worlds (states)* where A holds. This definition makes sense provided minimal worlds for A exist whenever there are A -worlds (A -states): this is ensured by the *smoothness condition* defined below. We recall the semantics of KLM logics [5, 6] from the strongest **R** to the weakest **C**. A *rational* model is a triple $\mathcal{M} = \langle \mathcal{W}, <, V \rangle$, where \mathcal{W} is a non-empty set of items called worlds, $<$ is an irreflexive, transitive and *modular*¹ relation on \mathcal{W} , and V is an evaluation function $V : \mathcal{W} \mapsto 2^{ATM}$, which assigns to every world w the set of atoms holding in that world. The truth conditions for a formula F are as follows: - if F is a boolean combination of formulas, $\mathcal{M}, w \models F$ is defined as for propositional logic; - let A be a propositional formula; we define $Min_{<}(A) =$

¹ A relation $<$ is modular if, for each u, v, w , if $u < v$, then either $w < v$ or $u < w$.

$\{w \in \mathcal{W} \mid \mathcal{M}, w \models A \text{ and } \forall w', w' < w \text{ implies } \mathcal{M}, w' \not\models A\}$; $\neg \mathcal{M}, w \models A \sim B$ if for all w' , if $w' \in \text{Min}_{<}(A)$ then $\mathcal{M}, w' \models B$. We also assume the *smoothness condition* on the preference relation: if $\mathcal{M}, w \models A$, then $w \in \text{Min}_{<}(A)$ or $\exists w' \in \text{Min}_{<}(A)$ s.t. $w' < w$. Validity and satisfiability of a formula are defined as usual. A *preferential* model is defined as the rational model, with the only difference that the preference relation $<$ is no longer assumed to be modular. Models for (loop-)cumulative logics also comprise states. A *(loop-)cumulative* model is a tuple $\mathcal{M} = \langle S, \mathcal{W}, l, <, V \rangle$, where S is a set of states and $l : S \mapsto 2^{\mathcal{W}}$ is a function that labels every state with a nonempty set of worlds; $<$ is defined on S , it satisfies the smoothness condition and it is irreflexive and transitive in **CL**, whereas it is only irreflexive in **C**. A propositional formula holds in a state s if it holds in *all* the worlds $w \in l(s)$; a conditional $A \sim B$ holds in a model if B holds in all minimal states where A holds.

3.1 Tableau calculi \mathcal{TS}^T

In Figure 2 we present the tableaux calculi \mathcal{TS}^T for KLM logics, where S stands for $\{\mathbf{R}, \mathbf{P}, \mathbf{CL}, \mathbf{C}\}$. The basic idea is simply to interpret the preference relation as an accessibility relation. The calculi for **R** and **P** implement a sort of runtime translation into (extensions of) Gödel-Löb modal logic of provability G. This is motivated by the fact that we assume the smoothness condition, which ensures that minimal A -worlds exist whenever there are A -worlds, by preventing infinitely descending chains of worlds. This condition therefore corresponds to the finite-chain condition on the accessibility relation (as in modal logic G). This approach is extended to the cases of **CL** and **C** by using a second modality L which takes care of states. The rules of the calculi manipulate sets of formulas Γ . We write Γ, F as a shorthand for $\Gamma \cup \{F\}$. Moreover, given Γ we define the following sets: $\Gamma^{\square} = \{\square \neg A \mid \square \neg A \in \Gamma\}$; $\Gamma^{\square^{\perp}} = \{\neg A \mid \square \neg A \in \Gamma\}$; $\Gamma^{\sim^{\pm}} = \{A \sim B \mid A \sim B \in \Gamma\} \cup \{\neg(A \sim B) \mid \neg(A \sim B) \in \Gamma\}$; $\Gamma^{L^{\perp}} = \{A \mid LA \in \Gamma\}$. As mentioned, the calculus for rational logic **R** makes use of *labelled* formulas, where the labels are drawn from a denumerable set \mathcal{A} ; there are two kinds of formulas: 1. *world formulas*, denoted by $x : F$, where $x \in \mathcal{A}$ and $F \in \mathcal{L}$; 2. *relation formulas*, denoted by $x < y$, where $x, y \in \mathcal{A}$, representing the preference relation. We define $\Gamma_{x \rightarrow y}^M = \{y : \neg A, y : \square \neg A \mid x : \square \neg A \in \Gamma\}$.

The calculi \mathcal{TS}^T are sound and complete wrt the semantics, i.e. given a set of formulas Γ of \mathcal{L} , it is unsatisfiable if and only if there is a closed tableau in \mathcal{TS}^T having Γ as a root [10]. Furthermore, it can be shown that the calculi \mathcal{TS}^T ensure termination only by preventing that the (\sim^+) rule is applied *more than once in the same world*. In [10] it is also shown that the problem of deciding validity for preferential logics is **coNP**-complete.

4 Conclusions

In this paper I have briefly summarized the contents of my PhD thesis [10]. Sequent calculi SeqS for some standard conditional logics and tableau calculi

<div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">$\mathcal{TR}^{\mathbf{T}}$</div> <div style="border: 1px solid black; padding: 2px;">$\mathcal{TP}^{\mathbf{T}}$</div> </div> <p style="text-align: center;">(AX) $\Gamma, x : P, x : \neg P$ with $P \in ATM$ (AX) $\Gamma, x < y, y < x$</p> $\frac{(\sim^+)}{\Gamma, u : A \vdash B, x : \neg A \quad \Gamma, u : A \vdash B, x : \neg \Box \neg A \quad \Gamma, u : A \vdash B, x : B} \frac{\Gamma, u : A \vdash B}{\Gamma, u : A \vdash B, x : \neg A}$ $\frac{(\sim^-)}{\Gamma, x : A, x : \Box \neg A, x : \neg B \text{ label} \quad \Gamma, x : \neg \Box \neg A \quad \Gamma, x : \neg \Box \neg A \quad \Gamma, y < x, y : A, y : \Box \neg A, \Gamma_{x \rightarrow y}^M \text{ label}} \frac{\Gamma, u : \neg(A \vdash B) \quad x \text{ new} \quad \Gamma, x : \neg \Box \neg A \quad y \text{ new}}{\Gamma, x : A, x : \Box \neg A, x : \neg B \text{ label} \quad \Gamma, y < x, y : A, y : \Box \neg A, \Gamma_{x \rightarrow y}^M \text{ label}}$ $\frac{(<)}{\Gamma, x < y, z < y, \Gamma_{y \rightarrow z}^M \quad \Gamma, x < y, x < z, \Gamma_{z \rightarrow x}^M \quad \{x < z, z < y\} \cap \Gamma = \emptyset} \frac{\Gamma, x < y}{\Gamma, x < y, z < y, \Gamma_{y \rightarrow z}^M \quad \Gamma, x < y, x < z, \Gamma_{z \rightarrow x}^M \quad \{x < z, z < y\} \cap \Gamma = \emptyset}$	<div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">$\mathcal{TC}^{\mathbf{T}}$</div> </div> <p style="text-align: center;">(AX) $\Gamma, P, \neg P$ with $P \in ATM$</p> $\frac{(\sim^+)}{\Gamma, A \vdash B \quad \Gamma, A \vdash B, \neg A \quad \Gamma, A \vdash B, \neg \Box \neg A \quad \Gamma, A \vdash B, B} \frac{\Gamma, A \vdash B}{\Gamma, A \vdash B, \neg A \quad \Gamma, A \vdash B, \neg \Box \neg A \quad \Gamma, A \vdash B, B}$ $\frac{(\sim^-)}{\Gamma, \neg(A \vdash B)} \frac{\Gamma, \neg(A \vdash B)}{\Gamma^{\Box}, A, \Box \neg A, \neg B}$ $\frac{(\Box^-)}{\Gamma, \neg \Box \neg A} \frac{\Gamma, \neg \Box \neg A}{\Gamma^{\Box}, \Gamma^{\Box}, \Gamma^{\Box}, A, \Box \neg A}$
<div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">$\mathcal{TC}^{\mathbf{L}}$</div> </div> $\frac{(\sim^+)}{\Gamma, A \vdash B, \neg LA \quad \Gamma, A \vdash B, \neg \Box \neg LA \quad \Gamma, A \vdash B, LB} \frac{\Gamma, A \vdash B}{\Gamma, A \vdash B, \neg LA \quad \Gamma, A \vdash B, \neg \Box \neg LA \quad \Gamma, A \vdash B, LB}$ $\frac{(\sim^-)}{\Gamma, \neg(A \vdash B)} \frac{\Gamma, \neg(A \vdash B)}{\Gamma^{\Box}, LA, \Box \neg LA, \neg LB} \quad \frac{(\Box^-)}{\Gamma, \neg \Box \neg LA} \frac{\Gamma, \neg \Box \neg LA}{\Gamma^{\Box}, \Gamma^{\Box}, \Gamma^{\Box}, LA, \Box \neg LA}$ $\frac{(L^-)}{\Gamma, \neg LA} \frac{\Gamma, \neg LA}{\Gamma^{L^1}, \neg A} \quad \frac{(L^-)}{\Gamma} \frac{\Gamma}{\Gamma^{L^1}} \text{ if } \Gamma \text{ does not contain negated } L \text{-formulas}$	<div style="display: flex; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">$\mathcal{TC}^{\mathbf{T}}$</div> </div> $\frac{(\sim^+)}{\Gamma, A \vdash B, \neg LA \quad \Gamma^{\Box}, \Gamma^{\Box}, A \vdash B, LA, \Box \neg LA \quad \Gamma, A \vdash B, LA, \Box \neg LA, LB} \frac{\Gamma, A \vdash B}{\Gamma, A \vdash B, \neg LA \quad \Gamma^{\Box}, \Gamma^{\Box}, A \vdash B, LA, \Box \neg LA \quad \Gamma, A \vdash B, LA, \Box \neg LA, LB}$ $\frac{(\sim^-)}{\Gamma, \neg(A \vdash B)} \frac{\Gamma, \neg(A \vdash B)}{\Gamma^{\Box}, LA, \Box \neg LA, \neg LB}$ $\frac{(L^-)}{\Gamma, \neg LA} \frac{\Gamma, \neg LA}{\Gamma^{L^1}, \neg A} \quad \frac{(L^-)}{\Gamma} \frac{\Gamma}{\Gamma^{L^1}} \text{ if } \Gamma \text{ does not contain negated } L \text{-formulas}$

Fig. 2. Tableau systems $\mathcal{TS}^{\mathbf{T}}$. To save space, we omit the standard rules for boolean connectives. For $\mathcal{TC}^{\mathbf{L}}$ and $\mathcal{TC}^{\mathbf{T}}$ the axiom (AX) is as in $\mathcal{TP}^{\mathbf{T}}$.

$\mathcal{TS}^{\mathbf{T}}$ for preferential logics have been presented. These calculi are sound, complete and terminating. They have been also implemented in SICStus Prolog [8, 4]. Recently, a free-variable version of the tableau calculus $\mathcal{TP}^{\mathbf{T}}$ for \mathbf{P} has been presented [3]. Moreover, the calculi SeqS have been used as the base of a goal-directed proof search mechanisms for conditional logics [9]. This extension might be the base of a language for reasoning hypothetically about change and actions in a logic programming framework.

Acknowledgements. This research has been partially supported by the projects “MIUR PRIN05: Specification and verification of agent interaction protocols” and “GALILEO 2006: Interazione e coordinazione nei sistemi multi-agenti”.

References

1. G. Crocco and P. Lamarre. On the connection between non-monotonic inference systems and conditional logics. In *Proceedings of KR'92 (Principles of Knowledge Representation and Reasoning)*, pages 565–571, 1992.
2. J. P. Delgrande. A first-order conditional logic for prototypical properties. *Artificial Intelligence*, 33(1):105–130, 1987.
3. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. An Implmentation of a Free-variable Tableaux for KLM Preferential Logic P of Nonmonotonic Reasoning: the Theorem Prover FreeP 1.0. In R. Basili and M.T. Paziienza, editors, *Proceedings of AI*IA 2007 (10th Congress of Italian Association for Artificial Intelligence)*, volume 4733 of *LNAI*, pages 84–96, Roma, Italy, September 2007. Springer-Verlag.
4. L. Giordano, V. Gliozzi, and G. L. Pozzato. KLMLean 2.0: A Theorem Prover for KLM Logics of Nonmonotonic Reasoning. In N. Olivetti, editor, *Proceedings of TABLEAUX 2007 (16th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 4548 of *LNAI*, pages 238–244, Aix En Provence, France, July 2007. Springer-Verlag.

5. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
6. Daniel Lehmann and Menachem Magidor. What does a conditional knowledge base entail? *Artificial Intelligence*, 55(1):1–60, 1992.
7. D. Lewis. *Counterfactuals*. Basil Blackwell Ltd, 1973.
8. N. Olivetti and G. L. Pozzato. CondLean 3.0: Improving Condlean for Stronger Conditional Logics. In Bernhard Beckert, editor, *Proceedings of TABLEAUX 2005 (Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 3702 of *LNAI*, pages 328–332, Koblenz, Germany, September 2005. Springer-Verlag.
9. Nicola Olivetti and Gian Luca Pozzato. Theorem proving for conditional logics: Condlean and goalduck. *Journal of Applied Non-Classical Logics (JANCL)*, to appear.
10. G. L. Pozzato. *Proof Methods for Conditional and Preferential Logics*. PhD thesis, Università degli Studi di Torino, February 2007.

$\mathcal{ALC} + \mathbf{T}$: Reasoning About Typicality in Description Logics

Laura Giordano¹, Valentina Gliozzi², Nicola Olivetti³, and Gian Luca Pozzato²

¹ Dipartimento di Informatica - Università del Piemonte Orientale “A. Avogadro” -
laura@mf.n.unipmn.it

² Dipartimento di Informatica - Università degli Studi di Torino -
{gliozzi,pozzato}@di.unito.it

³ LSIS-UMR CNRS 6168 Université “Paul Cézanne” - Aix-Marseille 3 -
nicola.olivetti@univ-cezanne.fr

Abstract. We extend the Description Logic \mathcal{ALC} with a “typicality” operator \mathbf{T} that allows us to reason about the prototypical properties and inheritance with exceptions. The resulting logic is called $\mathcal{ALC} + \mathbf{T}$. The typicality operator is intended to select the “most normal” or “most typical” instances of a concept. In our framework, knowledge bases may then contain, in addition to ordinary ABoxes and TBoxes, subsumption relations of the form “ $\mathbf{T}(C)$ is subsumed by P ”, expressing that typical C -members have the property P . The semantics of a typicality operator is defined by a set of postulates that are strongly related to Kraus-Lehmann-Magidor axioms of preferential logic \mathbf{P} . We first show that \mathbf{T} enjoys a simple semantics provided by ordinary structures equipped by a preference relation. This allows us to obtain a modal interpretation of the typicality operator. Using such a modal interpretation, we present a tableau calculus for deciding satisfiability of $\mathcal{ALC} + \mathbf{T}$ knowledge bases. Our calculus gives a nondeterministic-exponential time decision procedure for satisfiability of $\mathcal{ALC} + \mathbf{T}$. We then extend $\mathcal{ALC} + \mathbf{T}$ knowledge bases by a nonmonotonic completion that allows inferring defeasible properties of specific concept instances.

This paper has been also presented at the 14th Conference on Logic for Programming, Artificial Intelligence, and Reasoning “*LPAR 2007*” [10].

1 Introduction

The family of description logics (DLs) is one of the most important formalisms of knowledge representation. DLs are reminiscent of the early semantic networks and of frame-based systems. They offer two key advantages: a well-defined semantics based on first-order logic and a good trade-off between expressivity and complexity. DLs have been successfully implemented by a range of systems and they are at the base of languages for the semantic web such as OWL. A DL knowledge base (KB) comprises two components: (i) the TBox, containing the definition of concepts (and possibly roles), and a specification of inclusions relations among them, and (ii) the ABox containing instances of concepts and

roles, in other words, properties and relations of individuals. Since the very objective of the TBox is to build a taxonomy of concepts, the need of representing prototypical properties and of reasoning about defeasible inheritance of such properties easily arises. The traditional approach is to handle defeasible inheritance by integrating some kind of nonmonotonic reasoning mechanism. This has led to study nonmonotonic extensions of DLs [1–3, 5–7, 13]. However, finding a suitable nonmonotonic extension for inheritance reasoning with exceptions is far from obvious. Let us put forward some desiderata for such an extension:

1. The (nonmonotonic) extension must have a clear semantics and should be based on the same semantics as the underlying monotonic DL.
2. The extension should allow to specify prototypical properties in a natural and direct way.
3. The extension must be decidable, if so is the underlying monotonic DL and, possibly, computationally effective.

The nonmonotonic extensions proposed in the literature do not seem to fully satisfy all the above desiderata.

[1] proposes the extension of DL with Reiter’s default logic. However, the same authors have pointed out that this integration may lead to both semantical and computational difficulties. Indeed, the unsatisfactory treatment of open defaults via Skolemization may lead to an undecidable default consequence relation. For this reason, [1] proposes a restricted semantics for open default theories, in which default rules are only applied to individuals explicitly mentioned in the ABox. Furthermore, Reiter’s default logic does not provide a direct way of modeling inheritance with exceptions. This has motivated the study of extensions of DLs with prioritized defaults [13, 2].

A more general approach is undertaken in [6], where it is proposed an extension of DL with two epistemic operators. This extension allows one to encode Reiter’s default logic as well as to express epistemic concepts and procedural rules. However, this extension has a rather complicated modal semantics, so that the integration with the existing systems requires significant changes to the standard semantics of DLs.

In [3] the authors propose an extension of DL with circumscription. One of motivating applications of circumscription is indeed to express prototypical properties with exceptions, and this is done by introducing “abnormality” predicates, whose extension is minimized. The authors provide decidability and complexity results based on theoretical analysis. However, they do not provide a calculus for their logic. Moreover, the use of circumscription to model inheritance with exceptions is not that straightforward, as we remark below.

In this work, we propose a novel approach to defeasible inheritance reasoning based on the typicality operator \mathbf{T} . The intended meaning is that, for any concept C , $\mathbf{T}(C)$ singles out the instances of C that are considered as “typical” or “normal”. Thus assertions as “normally students do not pay taxes”, or “typically users do not have access to confidential files” [3] are represented by $\mathbf{T}(\text{Student}) \sqsubseteq \neg \text{TaxPayer}$ and $\mathbf{T}(\text{User}) \sqsubseteq \neg \exists \text{hasAccess. ConfidentialFile}$.

Before entering in the technical details, let us sketch how we intend to use the typicality operator and what kind of inferential services we expect to profit. We assume that a KB comprises, in addition to the standard TBox and ABox, a set of assertions of the type $\mathbf{T}(C) \sqsubseteq D$ where D is a concept not mentioning \mathbf{T} . The reasoning system should be able to infer or propagate prototypical properties of the concepts specified in the TBox, then to ascribe defeasible properties to individuals. For instance, let the KB contain:

$$\begin{aligned} \mathbf{T}(\textit{Student}) &\sqsubseteq \neg \textit{TaxPayer} \\ \mathbf{T}(\textit{Student} \sqcap \textit{Worker}) &\sqsubseteq \textit{TaxPayer} \\ \mathbf{T}(\textit{Student} \sqcap \textit{Worker} \sqcap \exists \textit{HasChild}.\top) &\sqsubseteq \neg \textit{TaxPayer} \\ \mathbf{T}(\textit{Unemployed}) &\sqsubseteq \neg \textit{TaxPayer} \end{aligned}$$

corresponding to the assertions: normally a student does not pay taxes, normally a working student pays taxes, but normally a working student having children does not pay taxes (because he is discharged by the government) etc...Observe that, if the same properties were expressed by ordinary inclusions, such as $\textit{Student} \sqsubseteq \neg \textit{TaxPayer}$ etc...we would simply get that there are not working students and so on, thus the KB would collapse. This collapse is avoided as we do not assume that \mathbf{T} is monotonic, that is to say $C \sqsubseteq D$ does not imply $\mathbf{T}(C) \sqsubseteq \mathbf{T}(D)$. To continue with the example, if the TBox contains $\textit{PersonWithNoIncome} \equiv \textit{Student} \sqcup \textit{Unemployed}$, then the system should be able to infer $\mathbf{T}(\textit{PersonWithNoIncome}) \sqsubseteq \neg \textit{TaxPayer}$. Suppose next that the ABox contains alternatively the following facts about *john*:

1. $\textit{student}(\textit{john})$
2. $\textit{student}(\textit{john}), \textit{worker}(\textit{john})$
3. $\textit{student}(\textit{john}), \textit{worker}(\textit{john}), \exists \textit{HasChild}.\top(\textit{john})$

Then the reasoning system should be able to infer the expected (defeasible) conclusion about *john* in each case: 1. $\neg \textit{TaxPayer}(\textit{john})$, 2. $\textit{TaxPayer}(\textit{john})$, 3. $\neg \textit{TaxPayer}(\textit{john})$. Observe that setting up a similar specification of the KB by using default logic or circumscription is not that simple: with default logic [1, 2, 5–7, 13], one has to specify a priority on default application (or one has to find a smart encoding of defaults giving priority to more specific information); with circumscription [3], one has to introduce abnormality predicates, and then establish which predicates are minimized, fixed, or variable, and finally for the minimized ones what is their priority wrt minimization (a total or a partial order). As a further step, the system should be able to infer (defeasible) properties also of individuals implicitly introduced by existential restrictions, for instance, if the ABox further contains $\exists \textit{HasChild}.\textit{Student}(\textit{jack})$, it should conclude (defeasibly) $\exists \textit{HasChild}.\neg \textit{TaxPayer}(\textit{jack})$.

Given the nonmonotonic character of the \mathbf{T} operator, there is a difficulty with handling irrelevant information, for instance, given the KB as above, one should be able to infer as well:

$$\begin{aligned} \mathbf{T}(\textit{Student} \sqcap \textit{SportLover}) &\sqsubseteq \neg \textit{TaxPayer} \\ \mathbf{T}(\textit{Student} \sqcap \textit{Worker} \sqcap \textit{SportLover}) &\sqsubseteq \textit{TaxPayer} \end{aligned}$$

as *SportLover* is irrelevant with respect to being a TaxPayer or not. For the same reason, the conclusion about *john* being a TaxPayer or not should not be influenced by the addition of *SportLover(john)* to the ABox. We refer to this problem as the problem of Irrelevance.

In this paper we lay down the base of an extension of DL with a typicality operator. Our starting point is a monotonic extension of the basic \mathcal{ALC} with the \mathbf{T} operator. The operator is supposed to satisfy a set of postulates that are essentially a reformulation of Kraus, Lehmann, and Magidor (KLM) axioms of preferential logic, namely the assertion $\mathbf{T}(C) \sqsubseteq P$ is equivalent to the conditional assertion $C \rightsquigarrow P$ of KLM preferential logic \mathbf{P} . It turns out that the semantics of the typicality operator can be equivalently specified by considering a preference relation (a strict partial order) on individuals: the typical members of a concept C are just the most preferred individuals, or “most normal”, of C according to the preference relation. The preference relation is the only additional ingredient that we need in our semantics. We assume that “most normal” members of a concept C always exist, whenever the concept C is non-empty. This assumption corresponds to the *Smoothness Condition* of KLM logics, or the well-known *Limit Assumption* in conditional logics. Taking advantage of this semantic setting, we can give a modal interpretation to the typicality operator: the modal operator \square has intuitively the same properties as in Gödel-Löb modal logic \mathbf{G} of arithmetic provability. We then define a tableau system for this extension based on this modal interpretation, thereby obtaining a decision procedure and an upper complexity bound (NEXPTIME). In future research we will further consider whether this bound is optimal or not.

These are the main results of the paper, however the monotonic extension is not enough to perform inheritance reasoning of the kind described above: (i) we need a way of inferring defeasible properties of individuals, (ii) we need a way of handling Irrelevance.

In the paper we deal with (i) by defining a completion of an ABox: the idea is that each individual is assumed to be a typical member of the most specific concept to which it belongs. Such a completion allows to perform inferences as the ones 1.,2.,3. above. The paper outlines how we intend to cope with typicality of all instances and with Irrelevance. In particular, dealing with Irrelevance (ii) requires a nonmonotonic mechanism. The idea is to complete a KB with a set of default rules. The default rules are not used to express defeasible properties of concepts (as in default extension of DLs), but to propagate defeasible properties of a concept to its subsumed concepts, e.g. to infer $\mathbf{T}(Student \sqcap SportLover) \sqsubseteq \neg TaxPayer$ from $\mathbf{T}(Student) \sqsubseteq \neg TaxPayer$. Thus in our approach the nonmonotonic mechanism is only needed to handle Irrelevance, whose treatment by means of default rules will be the object of future work.

2 The logic $\mathcal{ALC} + \mathbf{T}$: the typicality operator \mathbf{T}

We consider an alphabet of concept names \mathcal{C} , of role names \mathcal{R} , and of individuals \mathcal{O} . The language \mathcal{L} of the logic $\mathcal{ALC} + \mathbf{T}$ is defined by distinguishing *concepts*

and *extended concepts* as follows: (Concepts) $A \in \mathcal{C}$ and \top are *concepts* of \mathcal{L} ; if $C, D \in \mathcal{L}$ and $R \in \mathcal{R}$, then $C \sqcap D, C \sqcup D, \neg C, \forall R.C, \exists R.C$ are *concepts* of \mathcal{L} . (Extended concepts) if C is a concept, then C and $\mathbf{T}(C)$ are *extended concepts*, and all the boolean combinations of extended concepts are extended concepts of \mathcal{L} . A knowledge base is a pair (TBox, ABox). TBox contains subsumptions $C \sqsubseteq D$, where $C \in \mathcal{L}$ is either a concept or an extended concept $\mathbf{T}(C')$, and $D \in \mathcal{L}$ is a concept. ABox contains expressions of the form $C(a)$ and aRb where $C \in \mathcal{L}$ is an extended concept, $R \in \mathcal{R}$, and $a, b \in \mathcal{O}$.

In order to provide a semantics to the operator \mathbf{T} , we extend the definition of a model used in “standard” terminological logic \mathcal{ALC} :

Definition 1 (Semantics of \mathbf{T} with selection function). *A model is any structure $\langle \Delta, I, f_{\mathbf{T}} \rangle$, where: Δ is the domain; I is the extension function that maps each extended concept C to $C^I \subseteq \Delta$, and each role R to a $R^I \subseteq \Delta^I \times \Delta^I$. I is defined in the usual way (as for \mathcal{ALC}) and, in addition, $(\mathbf{T}(C))^I = f_{\mathbf{T}}(C^I)$. $f_{\mathbf{T}} : \text{Pow}(\Delta) \rightarrow \text{Pow}(\Delta)$ is a function satisfying the following properties:*

$$\begin{aligned}
 (f_{\mathbf{T}} - 1) \quad & f_{\mathbf{T}}(S) \subseteq S & (f_{\mathbf{T}} - 2) \quad & \text{if } S \neq \emptyset, \text{ then also } f_{\mathbf{T}}(S) \neq \emptyset \\
 (f_{\mathbf{T}} - 3) \quad & \text{if } f_{\mathbf{T}}(S) \subseteq R, \text{ then } f_{\mathbf{T}}(S) = f_{\mathbf{T}}(S \cap R) & (f_{\mathbf{T}} - 4) \quad & f_{\mathbf{T}}(\bigcup S_i) \subseteq \bigcup f_{\mathbf{T}}(S_i) \\
 (f_{\mathbf{T}} - 5) \quad & \bigcap f_{\mathbf{T}}(S_i) \subseteq f_{\mathbf{T}}(\bigcup S_i)
 \end{aligned}$$

Intuitively, given the extension of some concept C , $f_{\mathbf{T}}$ selects the *typical* instances of C . $(f_{\mathbf{T}} - 1)$ requests that typical elements of S belong to S . $(f_{\mathbf{T}} - 2)$ requests that if there are elements in S , then there are also *typical* such elements. The next properties constraint the behavior of $f_{\mathbf{T}}$ wrt \cap and \cup in such a way that they do not entail monotonicity. According to $(f_{\mathbf{T}} - 3)$, if the typical elements of S are in R , then they coincide with the typical elements of $S \cap R$, thus expressing a weak form of monotonicity (namely *cautious monotonicity*). $(f_{\mathbf{T}} - 4)$ corresponds to one direction of the equivalence $f_{\mathbf{T}}(\bigcup S_i) = \bigcup f_{\mathbf{T}}(S_i)$, so that it does not entail monotonicity. Similar considerations apply to the equation $f_{\mathbf{T}}(\bigcap S_i) = \bigcap f_{\mathbf{T}}(S_i)$, of which only the inclusion $\bigcap f_{\mathbf{T}}(S_i) \subseteq f_{\mathbf{T}}(\bigcap S_i)$ is derivable. $(f_{\mathbf{T}} - 5)$ is a further constraint on the behavior of $f_{\mathbf{T}}$ wrt arbitrary unions and intersections; it would be derivable if $f_{\mathbf{T}}$ were monotonic. We can prove the following proposition:

Proposition 1. $f_{\mathbf{T}}(S \cup R) \cap S \subseteq f_{\mathbf{T}}(S)$

We can give an alternative semantics for \mathbf{T} based on a preference relation. The idea is that there is a global preference relation among individuals and that the typical members of a concept C , i.e. selected by $f_{\mathbf{T}}(C^I)$, are the minimal elements of C wrt this preference relation. Observe that this notion is global, that is to say, it does not compare individuals wrt a specific concept (something like y is more typical than x wrt concept C). In this framework, an object $x \in \Delta$ is a *typical instance* of some concept C , if $x \in C^I$ and there is no C -element in Δ more typical than x . The typicality preference relation is partial since it is not always possible to establish which object is more typical than which other. The following definition is needed before we provide the Representation Theorem.

Definition 2. Given a relation $<$, which is a strict partial order (i.e. an ir-reflexive and transitive relation) over a domain Δ , for all $S \subseteq \Delta$, we define $Min_{<}(S) = \{x : x \in S \text{ and } \nexists y \in S \text{ s.t. } y < x\}$. We say that $<$ satisfies the Smoothness Condition iff for all $S \subseteq \Delta$, for all $x \in S$, either $x \in Min_{<}(S)$ or $\exists y \in Min_{<}(S)$ such that $y < x$.

We are now ready to prove the Representation Theorem below, showing that given a model with a selection function, we can define on the same domain a preference relation $<$ such that, for all $S \subseteq \Delta$, $f_{\mathbf{T}}(S) = Min_{<}(S)$. Notice that, as a difference wrt related results (Theorem 3 in [12]), the relation is defined on the same domain Δ of $f_{\mathbf{T}}$. On the other hand, if $<$ is a preference relation satisfying the Smoothness Condition, then the operator defined as $f_{\mathbf{T}}(S) = Min_{<}(S)$ satisfies the postulates of Definition 1.

Theorem 1 (Representation Theorem). Given any model $\langle \Delta, I, f_{\mathbf{T}} \rangle$, $f_{\mathbf{T}}$ satisfies postulates $(f_{\mathbf{T}} - 1)$ to $(f_{\mathbf{T}} - 5)$ above iff it is possible to define on Δ a strict partial order $<$, satisfying the Smoothness Condition, such that for all $S \subseteq \Delta$, $f_{\mathbf{T}}(S) = Min_{<}(S)$.

Proof. (“Only if” direction) Given $f_{\mathbf{T}}$ satisfying postulates $(f_{\mathbf{T}} - 1)$ to $(f_{\mathbf{T}} - 5)$, we define $<$ as follows: for all $x, y \in \Delta$, we let $x < y$ if $\forall S \subseteq \Delta$, if $y \in f_{\mathbf{T}}(S)$ then $x \notin S$, and $\exists R \subseteq \Delta$ such that $S \subset R$ and $x \in f_{\mathbf{T}}(R)$. We prove that:

1. $<$ is irreflexive. Easily follows by the definition of $<$.
2. $<$ is transitive. Let (a) $x < y$ and (b) $y < z$. Let $z \in f_{\mathbf{T}}(S)$ for some S , then by definition of $<$, $y \notin S$, and $\exists R$ s.t. $S \subset R$ and $y \in f_{\mathbf{T}}(R)$. Furthermore, $x \notin R$ and $\exists Q : R \subset Q$ and $x \in f_{\mathbf{T}}(Q)$. From this we can conclude that $x \notin S$ (otherwise $x \in R$), and $S \subset Q$, hence $x < z$.
3. $f_{\mathbf{T}}(S) \subseteq Min_{<}(S)$. Let $x \in f_{\mathbf{T}}(S)$. Suppose $x \notin Min_{<}(S)$, i.e. for some $y \in S$, $y < x$. By definition of $<$, $y \notin S$, contradiction, hence $x \in Min_{<}(S)$.
4. $Min_{<}(S) \subseteq f_{\mathbf{T}}(S)$. Let $x \in Min_{<}(S)$. Then $x \in S$, i.e. $S \neq \emptyset$. By $(f_{\mathbf{T}} - 2)$, $f_{\mathbf{T}}(S) \neq \emptyset$. Suppose $x \notin f_{\mathbf{T}}(S)$. Consider $\bigcup R_i$ for all $R_i \subseteq \Delta$ s.t. $x \in f_{\mathbf{T}}(R_i)$. By $(f_{\mathbf{T}} - 5)$, we have $x \in f_{\mathbf{T}}(\bigcup R_i)$. Consider now $f_{\mathbf{T}}(\bigcup R_i \cup S)$. We can easily show that $f_{\mathbf{T}}(\bigcup R_i \cup S) \not\subseteq \bigcup R_i$ (otherwise, by $(f_{\mathbf{T}} - 3)$ $f_{\mathbf{T}}(\bigcup R_i \cup S) = f_{\mathbf{T}}(\bigcup R_i)$, and by Proposition 1, $f_{\mathbf{T}}(\bigcup R_i) \cap S \subseteq f_{\mathbf{T}}(S)$, which contradicts the fact that $x \in f_{\mathbf{T}}(\bigcup R_i)$ but $x \notin f_{\mathbf{T}}(S)$). Consider hence $y \in f_{\mathbf{T}}(\bigcup R_i \cup S)$ s.t. $y \notin \bigcup R_i$. By definition of $<$, $y < x$. Furthermore, by $(f_{\mathbf{T}} - 1)$ $y \in S$ (since $y \in \bigcup R_i \cup S$ and $y \notin \bigcup R_i$). It follows that $x \notin Min_{<}(S)$, contradiction, hence $Min_{<}(S) \subseteq f_{\mathbf{T}}(S)$.
5. $<$ satisfies the Smoothness Condition. Let $S \neq \emptyset$ and $x \in S$. If $x \in f_{\mathbf{T}}(S)$ then by point 3 we have $x \in Min_{<}(S)$. If $x \notin f_{\mathbf{T}}(S)$ we can reason as for point 4 to conclude that there is $y \in f_{\mathbf{T}}(\bigcup R_i \cup S)$ s.t. $y \notin \bigcup R_i$ (hence $y \in S$), and $y < x$. By Proposition 1, we have $y \in f_{\mathbf{T}}(S)$, hence by point 3 we conclude $y \in Min_{<}(S)$.

The points above allow us to conclude.

(“If” direction) Given a strict partial order $<$ satisfying the Smoothness Condition, we can define $f_{\mathbf{T}} : Pow(\Delta) \rightarrow Pow(\Delta)$ by letting $f_{\mathbf{T}}(S) = Min_{<}(S)$. It

can be easily shown that $f_{\mathbf{T}}$ satisfies postulates $(f_{\mathbf{T}} - 1)$ to $(f_{\mathbf{T}} - 5)$. The proof is omitted due to space limitations. \blacksquare

Having the above Representation System, from now on, we will refer to the following semantics for $\mathcal{ALC} + \mathbf{T}$:

Definition 3 (Semantics of $\mathcal{ALC} + \mathbf{T}$). *A model \mathcal{M} is any structure $\langle \Delta, <, I \rangle$, where Δ and I are defined as in Definition 1, and $<$ is a strict partial order over Δ satisfying the Smoothness Condition (see Definition 2 above). As a difference wrt Definition 1, the semantics of the \mathbf{T} operator is: $(\mathbf{T}(C))^I = \text{Min}_{<}(C^I)$. For concepts (built from operators of \mathcal{ALC}), C^I is defined in the usual way.*

Definition 4 (Model satisfying a Knowledge Base). *Consider a model \mathcal{M} , as defined in Definition 3. We extend I so that it assigns to each individual a of \mathcal{O} an element a^I of the domain Δ . Given a KB $(TBox, ABox)$, we say that:*

- \mathcal{M} satisfies $TBox$ if for all inclusions $C \sqsubseteq D$ in $TBox$, and all elements $x \in \Delta$, if $x \in C^I$ then $x \in D^I$.
- \mathcal{M} satisfies $ABox$ if: (i) for all $C(a)$ in $ABox$, we have that $a^I \in C^I$, (ii) for all aRb in $ABox$, we have that $(a^I, b^I) \in R^I$.

\mathcal{M} satisfies a knowledge base if it satisfies both its $TBox$ and its $ABox$.

Notice that the meaning of \mathbf{T} can be split into two parts: for any object x of the domain Δ , $x \in (\mathbf{T}(C))^I$ just in case (i) $x \in C^I$, and (ii) there is no $y \in C^I$ such that $y < x$. In order to isolate the second part of the meaning of \mathbf{T} (for the purpose of the calculus that we will present in section 3) we introduce a new modality \square whose interpretation in \mathcal{M} is defined as follows.

Definition 5. $(\square C)^I = \{x \in \Delta \mid \text{for every } y \in \Delta, \text{ if } y < x \text{ then } y \in C^I\}$

The basic idea is simply to interpret the preference relation $<$ as an accessibility relation. By the Smoothness Condition, it turns out that the modality \square has the properties of Gödel-Löb modal logic of provability \mathbf{G} . The Smoothness Condition ensures that typical elements of C^I exist whenever $C^I \neq \emptyset$, by preventing infinitely descending chains of elements. This condition therefore corresponds to the finite-chain condition on the accessibility relation (as in \mathbf{G}). A similar correspondence has been presented in [8, 9] to interpret the preference relation in KLM logics. The following relation between \mathbf{T} and \square holds:

Proposition 2. *For all $x \in \Delta$, we have $x \in (\mathbf{T}(C))^I$ iff $x \in C^I$ and $x \in (\square \neg C)^I$*

Since we only use \square to capture the meaning of \mathbf{T} , in the following we will always use \square followed by a negated concept, as in $\square \neg C$.

We can establish the following equation between our typicality operator and the nonmonotonic (conditional) inference operator \sim (describing what can be *typically* derived from a given premise) by letting $C \sim D$ iff $\mathbf{T}(C) \sqsubseteq D$. It can be easily shown that there is a correspondence between the properties of \mathbf{T} and the properties of \sim in the system \mathbf{P} described in [12].

3 Reasoning

In this section we present a tableau calculus for deciding the satisfiability of a knowledge base. Given a KB (TBox, ABox), any concrete reasoning system should provide the usual reasoning services, namely *satisfiability of the KB*, *concept satisfiability*, *subsumption*, and *instance checking*. It is well known that the latter three services are reducible to the satisfiability of a KB.

We introduce a labelled tableau calculus for our logic $\mathcal{ALC} + \mathbf{T}$, which enriches the labelled tableau calculus for \mathcal{ALC} presented in [4]. The calculus is called $T^{\mathcal{ALC} + \mathbf{T}}$ and it is based on the notion of *constraint system*. We consider a set of *variables* drawn from a denumerable set \mathcal{V} . $T^{\mathcal{ALC} + \mathbf{T}}$ makes use of labels, which are denoted with x, y, z, \dots . Labels represent *objects*. An object is either a variable or an individual of the ABox, that is to say an element of $\mathcal{O} \cup \mathcal{V}$.

A *constraint* is a syntactic entity of the form $x \xrightarrow{R} y$ or $x : C$, where R is a role and C is either an extended concept or has the form $\Box \neg D$ or $\neg \Box \neg D$, where D is a concept. As we will define in Definition 6, the ABox of an $\mathcal{ALC} + \mathbf{T}$ -knowledge base can be translated into a set of constraints by replacing every membership assertion $C(a)$ with the constraint $a : C$ and every role aRb with the constraint $a \xrightarrow{R} b$. A tableau is a tree whose nodes are pairs $\langle S \mid U \rangle$, where:

- S contains constraints (or *labelled* formulas) of the form $x : C$ or $x \xrightarrow{R} y$;
- U contains formulas of the form $C \sqsubseteq D^L$, representing subsumption relations $C \sqsubseteq D$ of the TBox. L is a list of labels. As we will discuss later, this list is used in order to ensure the termination of the tableau calculus.

A node $\langle S \mid U \rangle$ is also called a *constraint system*. A branch is a sequence of nodes $\langle S_1 \mid U_1 \rangle, \langle S_2 \mid U_2 \rangle, \dots, \langle S_n \mid U_n \rangle \dots$, where each node $\langle S_i \mid U_i \rangle$ is obtained by its immediate predecessor $\langle S_{i-1} \mid U_{i-1} \rangle$ by applying a rule of $T^{\mathcal{ALC} + \mathbf{T}}$, having $\langle S_{i-1} \mid U_{i-1} \rangle$ as the premise and $\langle S_i \mid U_i \rangle$ as one of its conclusions. A branch is closed if one of its nodes is an instance of (Clash), otherwise it is open. We say that a tableau is closed if all its branches are closed.

Given a KB, we define its *corresponding constraint system* as follows:

Definition 6 (Corresponding constraint system). *Given an $\mathcal{ALC} + \mathbf{T}$ -knowledge base (TBox, ABox), we define its corresponding constraint system $\langle S \mid U \rangle$ as follows: $S = \{a : C \mid C(a) \in \text{ABox}\} \cup \{a \xrightarrow{R} b \mid aRb \in \text{ABox}\}$ and $U = \{C \sqsubseteq D^\emptyset \mid C \sqsubseteq D \in \text{TBox}\}$.*

Definition 7 (Model satisfying a constraint system). *Let \mathcal{M} be a model as defined in Definition 4. We define a function α which assigns to each variable of \mathcal{V} an element of Δ , and assigns every individual $a \in \mathcal{O}$ to $a^I \in \Delta$. \mathcal{M} satisfies $x : C$ under α if $\alpha(x) \in C^I$ and $x \xrightarrow{R} y$ under α if $(\alpha(x), \alpha(y)) \in R^I$. A constraint system $\langle S \mid U \rangle$ is satisfiable if there is a model \mathcal{M} and a function α such that \mathcal{M} satisfies under α every constraint in S and that, for all $C \sqsubseteq D \in U$ and for all x occurring in S , we have that if $\alpha(x) \in C^I$ then $\alpha(x) \in D^I$.*

$\langle S, x : C, x : \neg C \mid U \rangle \text{ (Clash)}$	$\frac{\langle S, x : \neg\neg C \mid U \rangle}{\langle S, x : C \mid U \rangle} (\neg)$
$\frac{\langle S, x : \mathbf{T}(C) \mid U \rangle}{\langle S, x : C, x : \Box\neg C \mid U \rangle} (\mathbf{T}^+)$	$\frac{\langle S, x : \neg\mathbf{T}(C) \mid U \rangle}{\langle S, x : \neg C \mid U \rangle \quad \langle S, x : \Box\neg C \mid U \rangle} (\mathbf{T}^-)$
$\frac{\langle S, x : \forall R.C, x \xrightarrow{R} y \mid U \rangle}{\langle S, x : \forall R.C, x \xrightarrow{R} y, y : C \mid U \rangle} (\forall^+)$ <p style="text-align: center; margin-left: 100px;">if $y : C \notin S$</p>	$\frac{\langle S, x : \exists R.C \mid U \rangle}{\langle S, x : \exists R.C, x \xrightarrow{R} y, y : C \mid U \rangle} (\exists^+)$ <p style="text-align: center; margin-left: 100px;">if $\exists z \prec x$ s.t. $z \equiv_{S, x \exists R.C} x$ and $\exists u$ s.t. $x \xrightarrow{R} u \in S$ and $u : C \in S$</p>
$\frac{\langle S, x : \neg\forall R.C \mid U \rangle}{\langle S, x : \neg\forall R.C, x \xrightarrow{R} y, y : \neg C \mid U \rangle} (\forall^-)$ <p style="text-align: center; margin-left: 100px;">if $\exists z \prec x$ s.t. $z \equiv_{S, x \neg\forall R.C} x$ and $\exists u$ s.t. $x \xrightarrow{R} u \in S$ and $u : \neg C \in S$</p>	$\frac{\langle S, x : \neg\exists R.C, x \xrightarrow{R} y \mid U \rangle}{\langle S, x : \neg\exists R.C, x \xrightarrow{R} y, y : \neg C \mid U \rangle} (\exists^-)$ <p style="text-align: center; margin-left: 100px;">if $y : \neg C \notin S$</p>
$\frac{\langle S, x : \Box\neg C \mid U \rangle}{\langle S, y : C, y : \Box\neg C, S_{x \rightarrow y}^M \mid U \rangle} (\Box^-)$ <p style="text-align: center; margin-left: 100px;">y new</p>	$\frac{\langle S \mid U, C \sqsubseteq D^L \rangle}{\langle S, x : \neg C \sqcup D \mid U, C \sqsubseteq D^{L,x} \rangle} (\text{Unfold})$ <p style="text-align: center; margin-left: 100px;">if x occurs in S and $x \notin L$</p>

Fig. 1. The calculus $T^{\mathcal{ALC}+\mathbf{T}}$. To save space, we omit the standard rules for \sqcup and \sqcap .

Proposition 3. *Given an $\mathcal{ALC} + \mathbf{T}$ -knowledge base, it is satisfiable if and only if its corresponding constraint system is satisfiable.*

Therefore, in order to check the satisfiability of (TBox, ABox), we build its corresponding constraint system $\langle S \mid U \rangle$, and then we use $T^{\mathcal{ALC}+\mathbf{T}}$ to check the satisfiability of $\langle S \mid U \rangle$. In order to check a constraint system $\langle S \mid U \rangle$ for satisfiability, our calculus $T^{\mathcal{ALC}+\mathbf{T}}$ adopts the usual technique of applying the rules until either a contradiction is generated (Clash) or a model satisfying $\langle S \mid U \rangle$ can be obtained from the resulting constraint system.

In order to take into account the TBox, we use a technique of *unfolding*, similar to the one described in [4]. Given a node $\langle S \mid U \rangle$, for each subsumption $C \sqsubseteq D^L \in U$ and for each label x that appears in the tableau, we add to S the constraint $x : \neg C \sqcup D$. As mentioned above, each formula $C \sqsubseteq D$ is equipped by the list L of labels in which it has been unfolded in the current branch. This is needed in order to avoid multiple unfolding of the same subsumption by using the same label, generating non-termination in a proof search.

Before introducing the rules of $T^{\mathcal{ALC}+\mathbf{T}}$ we need some more definitions. First, as in [4], we assume that labels are introduced in a tableau according to an ordering \prec , that is to say if y is introduced in the tableau, then $x \prec y$ for all labels x that are already in the tableau.

Given a tableau node $\langle S \mid U \rangle$ and an object x , we define $\sigma(\langle S \mid U \rangle, x) = \{C \mid x : C \in S\}$. Furthermore, we say that two labels x and y are *S-equivalent*, written $x \equiv_S y$, if they label the same set of concepts, i.e. $\sigma(\langle S \mid U \rangle, x) = \sigma(\langle S \mid U \rangle, y)$. Intuitively, *S-equivalent* labels can represent the same element in the model built by the rules of $T^{\mathcal{ALC}+\mathbf{T}}$. Last, we define $S_{x \rightarrow y}^M = \{y : C, y : \Box\neg C \mid x : \Box\neg C \in S\}$.

The rules of $T^{\mathcal{ALC}+\mathbf{T}}$ are presented in Figure 1. Rules (\exists^+) , (\forall^-) , and (\Box^-) are called *dynamic* since they introduce a new variable in their conclusions. The

other rules are called *static*. We do not need any extra rule for the positive occurrences of the \Box operator, since these are taken into account by the computation of $S_{x \rightarrow y}^M$. The side conditions on the rules (\exists^+) and (\forall^-) are introduced in order to ensure a terminating proof search, by implementing the standard *blocking* technique described below. The rules of $T^{\mathcal{ALC}+\mathbf{T}}$ are applied with the following *standard strategy*: 1. apply a rule to a variable $x \in \mathcal{V}$ only if no rule is applicable to a variable $y \in \mathcal{V}$ such that $y \prec x$; 2. apply dynamic rules ((\Box^-) first) only if no static rule is applicable. This strategy ensures that the variables are considered one at a time according to the ordering \prec . Consider an application of a dynamic rule to a variable x of a constraint system $\langle S \mid U \rangle$. For all $\langle S' \mid U' \rangle$ obtained from $\langle S \mid U \rangle$ by a sequence of rule applications, it can be easily shown that (i) no rule can be applied in $\langle S' \mid U' \rangle$ to a variable y s.t. $y \prec x$ and (ii) $\sigma(\langle S \mid U \rangle, x) = \sigma(\langle S' \mid U' \rangle, x)$. The calculus so obtained is sound and complete wrt to the semantics described in Definition 7 (we omit the proof to save space).

Theorem 2 (Soundness and Completeness of $T^{\mathcal{ALC}+\mathbf{T}}$). *Given a constraint system $\langle S \mid U \rangle$, it is unsatisfiable iff it has a closed tableau.*

Let us conclude this section by analyzing termination and complexity of $T^{\mathcal{ALC}+\mathbf{T}}$. In general, non-termination in labelled tableau calculi can be caused by two different reasons: 1. some rules copy their principal formula in the conclusion(s), and can thus be reapplied over the same formula without any control; 2. dynamic rules may generate infinitely-many labels, creating infinite branches.

Concerning the first source of non-termination (point 1), the only rules copying their principal formulas in their conclusions are (\forall^+) , (\exists^-) , (Unfold), (\forall^-) , and (\exists^+) . However, the side conditions on these rules avoid multiple applications on the same formula. Indeed, (Unfold) can be applied to a constraint system $\langle S \mid U, C \sqsubseteq D^L \rangle$ by using the label x only if it has not yet been applied to x in the current branch (i.e. x does not belong to L). Concerning (\forall^+) , the rule can be applied to $\langle S, x : \forall R.C, x \xrightarrow{R} y \mid U \rangle$ only if $y : C$ does not belong to S . When $y : C$ is introduced in the branch, the rule will not further apply to $x : \forall R.C$. The same for (\exists^-) , (\exists^+) , and (\forall^-) .

Concerning the second source of non-termination (point 2), we can prove that we only need to adopt the standard loop-checking machinery known as *blocking*, which ensures that the rules (\exists^+) and (\forall^-) do not introduce infinitely-many labels on a branch. Thanks to the properties of \Box , no other additional machinery is required to ensure termination. Indeed, we can show that the interplay between rules (\mathbf{T}^-) and (\Box^-) does not generate branches containing infinitely-many labels. Let us discuss the termination in more detail.

Without the side conditions on the rules (\exists^+) and (\forall^-) , the calculus $T^{\mathcal{ALC}+\mathbf{T}}$ does not ensure a terminating proof search. Indeed, given a constraint system $\langle S \mid U \rangle$, it could be the case that (\exists^+) is applied to a constraint $x : \exists R.C \in S$, introducing a new label y and the constraints $x \xrightarrow{R} y$ and $y : C$. If an inclusion $\mathbf{T}(\exists R.C) \sqsubseteq D$ belongs to U , then (Unfold) can be applied by using y , thus generating a branch containing $y : \neg \mathbf{T}(\exists R.C)$, to which (\mathbf{T}^-) can be applied introducing $y : \neg \Box \neg (\exists R.C)$. An application of (\Box^-) introduces a new variable

z and the constraint $z : \exists R.C$, to which (\exists^+) can be applied generating a new label u . (Unfold) can then be re-applied on $\mathbf{T}(\exists R.C) \sqsubseteq D$ by using u , incurring a loop. In order to prevent this source of non termination, we adopt the standard technique of *blocking*: the side condition of the (\exists^+) rule says that this rule can be applied to a node $\langle S, x : \exists R.C \mid U \rangle$ only if there is no z occurring in S such that $z \prec x$ and $z \equiv_{S, x : \exists R.C} x$. In other words, if there is an “older” label z which is equivalent to x wrt $S, x : \exists R.C$, then (\exists^+) is not applicable, since the condition and the strategy imply that the (\exists^+) rule has already been applied to z . In this case, we say that x is *blocked* by z . The same for (\forall^-) .

As mentioned, another possible source of infinite branches could be determined by the interplay between rules (\mathbf{T}^-) and (\Box^-) . This cannot occur, i.e. the interplay between these two rules does not generate branches containing infinitely-many labels. Intuitively, the application of (\Box^-) to $x : \neg\Box\neg C$ adds $y : \Box\neg C$ to the conclusion, so that (\mathbf{T}^-) can no longer consistently introduce $y : \neg\Box\neg C$. This is due to the properties of \Box (no infinite descending chains of $<$ are allowed). More in detail, if (Unfold) is applied to $\mathbf{T}(C) \sqsubseteq D$ by using x , an application of (\mathbf{T}^-) introduces a branch containing $x : \neg\Box\neg C$; when a new label y is generated by an application of (\Box^-) on $x : \neg\Box\neg C$, we have that $y : \Box\neg C$ is added to the current constraint system. If (Unfold) and (\mathbf{T}^-) are also applied to $\mathbf{T}(C) \sqsubseteq D$ on the new label y , then the conclusion where $y : \neg\Box\neg C$ is introduced is closed, by the presence of $y : \Box\neg C$. By this fact, we do not need to introduce any loop-checking machinery on the application of (\Box^-) .

Theorem 3 (Termination of $T^{\mathcal{ALC}+\mathbf{T}}$). *Let $\langle S \mid U \rangle$ be a constraint system, then any tableau generated by $T^{\mathcal{ALC}+\mathbf{T}}$ is finite.*

Since the calculus $T^{\mathcal{ALC}+\mathbf{T}}$ is sound and complete (Theorem 2), and since an $\mathcal{ALC} + \mathbf{T}$ -knowledge base is satisfiable iff its corresponding constraint system is satisfiable (Proposition 3), from Theorem 3 above it follows that checking whether a given $\mathcal{ALC} + \mathbf{T}$ -knowledge base is satisfiable is a decidable problem. Let us conclude this section with a complexity analysis of the calculus $T^{\mathcal{ALC}+\mathbf{T}}$:

Theorem 4 (Complexity). *Given an $\mathcal{ALC}+\mathbf{T}$ -knowledge base, checking whether it is satisfiable can be solved in nondeterministic exponential time.*

Proof. We first show that the number of labels generated on a branch is at most exponential in the size of KB. Let n be the size of a KB. Given a constraint system $\langle S \mid U \rangle$, the number of extended concepts appearing in $\langle S \mid U \rangle$, including also all the ones appearing as a subformula of other concepts, is $O(n)$. As there are at most $O(n)$ concepts, there are at most $O(2^n)$ variables labelling distinct sets of concepts. Hence, there are $O(2^n)$ non-blocked variables in S .

Let m be the maximum number of direct successors of each variable $x \in S$, obtained by applying dynamic rules. m is bound by the number of $\exists R.C$ concepts ($O(n)$) plus the number of $\neg\forall R.C$ concepts ($O(n)$) plus the number of $\neg\Box\neg C$ concepts ($O(n)$). Therefore, there are at most $O(2^n \times m)$ variables in S , where $m \leq 3n$. The number of *individuals* in the ABox is bound by n too, and each

individual has at most m direct successors. The number of *labels* in S is then bound by $O((2^n + n) \times m)$, and hence by $O(2^{2n})$.

For a given label x , the concepts labelled by x introduced in the branch (namely, all the possible subconcepts of the initial constraint system, as well as all boxed subconcepts) are $O(n)$. According to the standard strategy, after all static rules have been applied to a label x in phase 1, no other concepts labelled by x can be introduced later on a branch. Hence, the labelled concepts introduced on the branch is $O(n)$ for each label, and the number of all labelled concepts on the branch is $O(n \times 2^{2n})$. Therefore, a branch can contain at most an exponential number of applications of tableau rules.

The satisfiability of a KB can thus be solved by defining a procedure which nondeterministically generates an open branch of exponential size (in the size of KB). The problem is in NEXPTIME. \blacksquare

4 Reasoning about Typicality

Logic $\mathcal{ALC} + \mathbf{T}$ allows one to reason monotonically about typicality. In $\mathcal{ALC} + \mathbf{T}$ we can consistently express, for instance, the fact that three different concepts, as *student*, *working student* and *working student with children*, have a different status as taxpayers.

What about the typical properties of an individual *john* that we know being a working student, and having children? Of course, if we know that *john* is a typical instance of the concept $Student \sqcap Worker \sqcap \exists HasChild. \top$, i.e. if the ABox contains the assertion $(*) \mathbf{T}(Student \sqcap Working \sqcap \exists HasChild. \top)(john)$, then, in $\mathcal{ALC} + \mathbf{T}$, we can conclude that $\neg TaxPayer(john)$. However, in absence of $(*)$, we cannot derive $\neg TaxPayer(john)$.

In general, we would like to infer that individuals have the properties which are typical of the most specific concept to which they belong. To this purpose, we define a completion of the knowledge base which adds to the ABox, for each individual a occurring in the ABox, the assertion that a is a typical instance of the most specific concept C to which it belongs. Although in general ABoxes can contain typicality assertions about individuals, in practice we assume that typicality assertions are automatically generated by the system by means of the completion, and are not inserted by the user. From now on, we therefore assume that the initial ABox of a KB does not contain any typicality assertion.

Definition 8 (Completion of a Knowledge Base). *The KB $(TBox, ABox')$ is the completion of the KB $(TBox, ABox)$, if $ABox'$ is obtained from $ABox$ by adding to it, for all individual names a in the ABox, the assertion $\mathbf{T}(C_1 \sqcap \dots \sqcap C_j)(a)$, where C_1, \dots, C_j are all the concepts C_i such that: (1) C_i is a subconcept of any concept occurring in $(TBox, ABox)$; (2) C_i does not contain \mathbf{T} ; (3) a is an instance of C_i , i.e. $C_i(a)$ is derivable in \mathcal{ALC} from $(TBox, ABox)$.*

For instance, assuming that $Student(john)$, $Worker(john)$ and $\exists HasChild. \top(john)$ are the only assertions concerning *john* derivable from the KB, the completion

above would add $\mathbf{T}(Student \sqcap Worker \sqcap \exists HasChild.\top)(john)$ to the ABox, as $Student \sqcap Worker \sqcap \exists HasChild.\top$ is the most specific concept of which $john$ is an instance. From this, we can conclude in $\mathcal{ALC} + \mathbf{T}$ that $john$ does not pay taxes.

The completion adds $\mathbf{T}(C_1 \sqcap \dots \sqcap C_j)(a)$ by considering each $C_i(a)$ derivable in \mathcal{ALC} from the KB, rather than considering only $C_i(a)$ in the ABox. This is needed, for instance, to infer that $john$ does not pay taxes from the KB containing $Professor \sqsubseteq \forall HasChild.Student$, $Professor(paul)$, and $HasChild(paul, john)$.

As a matter of fact, if we had in the ABox the information that $john$ is a *TaxPayer*, this would not cause an inconsistent completion of the KB. Indeed, in such a case, $Student \sqcap Worker \sqcap \exists HasChild.\top \sqcap TaxPayer$ would be the most specific concept of which $john$ is an instance, so that the assertion $\mathbf{T}(Student \sqcap Worker \sqcap \exists HasChild.\top \sqcap TaxPayer)(john)$ would be added in the completion of the KB. This does not allow to infer that $\neg TaxPayer(john)$. Hence, no inconsistency arises. However, it could be the case that the KB obtained by the completion is inconsistent, even if the initial KB is consistent. For instance, the KB containing $\mathbf{T}(C) \sqsubseteq \forall R.E$, $\mathbf{T}(D) \sqsubseteq \forall R.\neg E$, $C(a)$, $D(b)$, $R(a, c)$, and $R(b, c)$ is consistent, whereas its completion, including also $\mathbf{T}(C)(a)$ and $\mathbf{T}(D)(b)$, is not. In this case, we keep the initial KB unaltered, instead of the one obtained by the completion.

Notice that the completion of the ABox only introduces $O(n)$ new formulas $a : \mathbf{T}(C_1 \sqcap \dots \sqcap C_j)$, one for each named individual a in the ABox. Furthermore, the size of each formula $\mathbf{T}(C_1 \sqcap \dots \sqcap C_j)$ is $O(n^2)$ as C_1, \dots, C_j are all distinct subformulas of the initial formula ($O(n)$), and each C_i has size $O(n)$. Hence, after the completion construction, the size of the KB is polynomial in n . Moreover, for each individual a ($O(n)$) and for each concept C ($O(n)$), we have to check whether $C(a)$ is derivable in \mathcal{ALC} from the KB, which is a problem in EXPTIME. Hence, the completion construction requires exponential time and produces a KB of size polynomial in the size of the original one:

Theorem 5. *The problem of deciding satisfiability of the knowledge base after completion is in NEXPTIME in the size of the original KB.*

As mentioned, given a consistent KB, its completion could be inconsistent. In this case, we choose to keep the original KB. As an alternative, we could consider all maximal consistent KBs (extensions) that can be generated by adding, for all individuals, the relative most-specific concept assumptions. We could then perform either a skeptical or a credulous reasoning with respect to such extensions.

Preferential logic allows to deal with some forms of inheritance among concepts, by the property of cautious monotonicity (which comes from the semantic property ($f_{\mathbf{T}} - 3$): if $\mathbf{T}(C) \sqsubseteq D$ and $\mathbf{T}(C) \sqsubseteq E$, then $\mathbf{T}(C \sqcap D) \sqsubseteq E$. Coming back to the example above, if we knew that all students typically have a teacher, i.e. $\mathbf{T}(Student) \sqsubseteq \exists HasTeacher.\top$, and that $john$ is a student and has a teacher ($Student(john)$ and $\exists HasTeacher.\top(john)$ are in the ABox) then, by the completion construction above, we would get $\mathbf{T}(Student \sqcap \exists HasTeacher.\top)(john)$, and, by cautious monotonicity, we would conclude that $john$ does not pay taxes.

5 Extensions

We consider this work as a first step. We plan to extend our approach in the following directions.

Inheritance with exceptions. Once the completion of a KB has been defined as above, the problem of inferring the typical properties of an individual is reduced to the problem of inferring the properties of the most specific concept to which it belongs. This can be done by reasoning on the typical properties of concepts in the TBox. Although Preferential Description Logic allows to capture - through cautious monotonicity - some form of inheritance of typical properties among concepts, there are cases in which cautious monotonicity is not strong enough to derive the intended conclusions. For instance, if we know that *jack* is a student who is a sport lover, we cannot conclude that *jack* is not a tax payer, as we do not have the property that typical students (or all students) are sport lovers, and hence cautious monotonicity is not applicable. Here we are faced with the problem of *Irrelevance*. Since the property of being a sport lover is irrelevant with respect to the property of paying taxes, we would like to infer that also $\mathbf{T}(Student \sqcap SportLover) \sqsubseteq \neg TaxPayer$, and therefore that *jack* is not a tax payer. In order to allow this form of inheritance among concepts, we can introduce a default rule of the following type:

$$\frac{\mathbf{T}(Student) \sqsubseteq \neg TaxPayer \quad : \mathbf{T}(Student \sqcap SportLover) \not\sqsubseteq TaxPayer}{\mathbf{T}(Student \sqcap SportLover) \sqsubseteq \neg TaxPayer} (IRR)$$

By the default rule above, if typical students are not tax payers, and *it is consistent* to assume that typical students who are sport lovers are not tax payers, then we could conclude that typical sport lover students are not tax payers. With this rule, the typical properties of a more general concept C are considered one by one, and are inherited by a more specific concept $(C \sqcap D)$ if it is consistent to do so. In order to deal with default rules like this one, we need to integrate our calculus with a standard mechanism to reason about defaults.

Reasoning on the typicality of all instances. The completion of a knowledge base, as defined above, only applies to individuals explicitly named in the ABox. However, we would like to reason on the typical properties of all individuals. Assume, for instance, that the ABox contains the assertions: $\exists HasChild.Worker(bill)$ and $\forall HasChild.Student(bill)$. Thus, *bill* has a child who is a student and is working. We want to be able to infer that *bill* has a child who is a tax payer. To this purpose, we need to assume that the *bill's* child is a typical working student.

To reason about the typicality of all individuals, we would need to assume that all individuals generated during the tableau construction are *typical* instances of the most specific concept to which they belong. To this purpose, we could think of applying the completion construction of Definition 8 to all generated individuals. The completion construction would be applied only when all relevant formulas $y : C_1, \dots, y : C_j$ with label y have already been introduced in the branch. According to the strategy described in section 3, the completion should be performed only after the application of the rules to all x s.t. $x \prec y$.

Extension to other DLs. We want to study the extension of our approach to more expressive description logics. For instance, we plan to extend $\mathcal{ALCN}\mathcal{R}$ considered in [4] with our typicality operator \mathbf{T} , and consider which is the complexity corresponding to this extension. Finally, we want to study the extension of the language of concepts by allowing arbitrary occurrences of the operator \mathbf{T} .

6 Conclusions

We have proposed an extension of \mathcal{ALC} for reasoning about typicality in Description Logic framework. The resulting logic is called $\mathcal{ALC} + \mathbf{T}$. We have proposed a calculus for deciding the satisfiability of a general knowledge base in $\mathcal{ALC} + \mathbf{T}$. The calculus, called $T^{\mathcal{ALC}+\mathbf{T}}$, is analytic, terminating, and allows us to decide the satisfiability of a knowledge base in $\mathcal{ALC} + \mathbf{T}$ in nondeterministic exponential time. The calculus is reminiscent of the tableaux calculi for KLM logics presented in [8, 9]. We have then shown how to complete the ABox by means of typicality assumptions, in order to infer prototypical properties of the individuals explicitly mentioned in the ABox. We have argued how to apply a similar completion also to individuals implicitly mentioned in the ABox, in order to infer their properties. Finally, we have sketched how to reason about the inheritance of typical properties from more general to more specific concepts handling with irrelevant information, by using appropriate default rules.

KLM logics are related to probabilistic reasoning. A probabilistic extension of DLs has been proposed in [11]. In particular, the notion of conditional constraint in [11] allows typicality assertions to be expressed (with a specified probability). We plan to compare in details this probabilistic approach to ours elsewhere.

Acknowledgements. This research has been partially supported by the projects “*MIUR PRIN05: Specification and verification of agent interaction protocols*” and “*GALILEO 2006: Interazione e coordinazione nei sistemi multi-agenti*”.

References

1. F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. *J. Autom. Reasoning*, 14(1):149–180, 1995.
2. F. Baader and B. Hollunder. Priorities on defaults with prerequisites, and their application in treating specificity in terminological default logic. *J. Autom. Reasoning*, 15(1):41–68, 1995.
3. P. A. Bonatti, C. Lutz, and F. Wolter. Description logics with circumscription. In *Proceedings of KR*, pages 400–410, 2006.
4. M. Buchheit, F. M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. Artif. Int. Research (JAIR)*, 1:109–138, 1993.
5. F. M. Donini, M. Lenzerini, D. Nardi, W. Nutt, and A. Schaerf. An epistemic operator for description logics. *Artif. Intell.*, 100(1-2):225–274, 1998.
6. F. M. Donini, D. Nardi, and R. Rosati. Description logics of minimal knowledge and negation as failure. *ACM Trans. Comput. Log.*, 3(2):177–225, 2002.

7. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proceedings of KR*, pages 141–151, 2004.
8. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Analytic Tableaux for KLM Preferential and Cumulative Logics. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proceedings of LPAR 2005 (12th Conference on Logic for Programming, Artificial Intelligence, and Reasoning)*, volume 3835 of *LNAI*, pages 666–681, Montego Bay, Jamaica, December 2005. Springer-Verlag.
9. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Analytic Tableaux Calculi for KLM Rational Logic R. In M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, editors, *Proceedings of JELIA 2006 (10th European Conference on Logics in Artificial Intelligence)*, volume 4160 of *LNAI*, pages 190–202, Liverpool, England, September 2006. Springer-Verlag.
10. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Preferential Description Logics. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of LPAR 2007 (14th Conference on Logic for Programming, Artificial Intelligence, and Reasoning)*, volume 4790 of *LNAI*, pages 257–272, Yerevan, Armenia, October 2007. Springer-Verlag.
11. R. Giugno and T. Lukasiewicz. P-*SHOQ(D)*: A Probabilistic Extension of *SHOQ(D)* for Probabilistic Ontologies in the Semantic Web. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Proceedings of JELIA 2002 (8th European Conference on Logics in Artificial Intelligence)*, volume 2424 of *LNAI*, pages 86–97, Cosenza, Italy, September 2002. Springer-Verlag.
12. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
13. U. Straccia. Default inheritance reasoning in hybrid kl-one-style logics. In *Proceedings of IJCAI*, pages 676–681, 1993.

Hybrid Automata in System Biology: How far can we go? *

Dario Campagna and Carla Piazza

Dept. of Mathematics and Computer Science
University of Udine
Via delle Scienze 206, 33100 Udine, Italy

Abstract. We consider the reachability problem on semi-algebraic hybrid automata. In particular, we deal with the effective cost that has to be afforded to solve reachability through first-order satisfiability. The analysis we perform with some existing tools shows that even simple examples cannot be efficiently solved. We need approximations to reduce the number of variables in our formulae: this is the main source of time computation growth. We study standard approximation methods based on Taylor polynomials and ad-hoc strategies to solve the problem and we show their effectiveness on the repressilator case study.

Introduction

Since their introduction (see, e.g., [1]), hybrid automata have initiated a new tradition, promising powerful tools for modeling and reasoning about complex engineered or natural systems (see, e.g., [2, 3]).

Intuitively, a hybrid automaton consists of a finite graph, whose nodes are called *locations*, together with a set of continuous variables which evolve according to continuous laws, called *dynamics*, characterising each discrete location. The continuous evolution of the hybrid automaton may change from location to location. Moreover, each location is characterised by an *invariant* condition which defines the allowed values for the continuous variables inside the location. Finally, each graph's *edge* is labelled by both an *activation* condition and a *reset* map. The edge can be crossed only if the continuous variables satisfy the activation condition and after crossing it the continuous variables are set accordingly to the reset map. The double nature, both discrete and continuous, of hybrid automata make them particularly suitable in the modeling of systems exhibiting a mixed behaviour which cannot be characterised in a proper way using either discrete or continuous formalisms.

In this context, one of the basic problems is the *reachability* one which requires to decide whether it is possible to move from a state (a pair consisting of a location together with a set of values for the continuous variables) to another.

* This work is partially supported by MIUR and will appear in the ENTCS proceedings of the workshop “From Biology To Concurrency and back” (FBTC’08). Corresponding author: campagnadario@alice.it, carla.piazza@dimi.uniud.it

Unfortunately, the flexibility and expressive power of hybrid automata soon lead to undecidability and complexity results [4] which cast doubts on their suitability as a general tool that can be algorithmized and efficiently implemented.

In order to control both undecidability and complexity one can either impose syntactic conditions and concentrate on classes of hybrid automata or define semantic approximation techniques.

In [5] the class of *semi-algebraic hybrid automata* has been introduced. The invariants, dynamics, activations, and resets of semi-algebraic automata have to be first-order formulæ over the theory of $(\mathbb{R}, 0, 1, +, *, <)$. On the one hand, such formulæ are decidable [6] and tools such as QEPCAD B [7] can be used to manage them. On the other hand, Taylor polynomials allow to use semi-algebraic formulæ to approximate with arbitrary precision any smooth function. As a consequence of the expressive power of semi-algebraic hybrid automata, the undecidability of the reachability problem for such class can be proved [8]. In particular, in this case, undecidability is a consequence of the fact that we cannot a-priori bound the number of edges we need to cross. Hence, we can see “the glass half full” saying that *bounded* (w.r.t. edge crossing) reachability is computable. Unfortunately, as noticed in [9] such computation results to be too time/space consuming due to the high computational complexity of semi-algebraic decomposition.

In this paper we start from the considerations presented in [9] concerning the effectiveness of bounded reachability computation on semi-algebraic hybrid automata and we show on some examples which kind of approximations are necessary to keep complexity under control. As done in [9] we may distinguish space and time discretizations in our work. As far as space discretizations are concerned, instead of implementing an ad-hoc algorithm, we try to exploit tools which allow approximate computations over the reals such as RSOLVER [10] and ECLⁱPS^e [11]. Unfortunately, this is not enough: space approximations which *separate* the continuous variables are necessary. We notice that time discretization and Taylor polynomials are essential ingredients in our approach.

The paper is organized as follows. In Section 1 we quickly overview the state of the art. Some basic notions about semi-algebraic hybrid automata and reachability find place in Section 2, while Section 3 is the core part of our work. In Section 4 we apply our analysis to the Repressilator case study. Some conclusions are drawn in Section 5.

1 Related Works

As mentioned in the introduction, we can control undecidability and complexity on hybrid automata in two ways: imposing syntactic constraints which limit the expressive power or introducing semantic approximation techniques.

In [12] Alur et al. introduced *multirate automata* as an extensions of *timed automata* [13]. Such hybrid automata are characterised by resets which are either identity or constant function zero. Moreover, their continuous variables evolve like clocks with rational rates. In the same work it has been proved that the reachability problem over multirate automata is not decidable in general. How-

ever, imposing a restriction on dynamics called *simplicity condition*, decidability for reachability problem and finite bisimulation are proved. Puri and Varaiya in [14] introduced *rectangular hybrid automata* whose dynamics can be characterised by a differential inclusion. They showed that, under a condition called *initialized condition*, reachability can be decided. Lafferriere, Pappas and Sastry introduced *o-minimal hybrid automata* in [15]. Such class of hybrid automata guarantee finite bisimulation quotient imposing both constant reset condition to all the edges and a unique o-minimal dynamic from each state. In [16] it has been proved that reachability is still decidable on semi-algebraic o-minimal automata when the conditions on the dynamics are relaxed allowing many possible continuous evolutions. Unfortunately, all the above mentioned classes have restrictions on both dynamics and resets and thus they are not suitable to verify properties of many interesting hybrid systems.

As far as approximation techniques are concerned, in [17] Halbwachs et al. suggested convex approximations as a way to verify linear hybrid systems, Dang and Maler proposed to verify hybrid automaton properties via face lifting in [18], Chutinan and Krogh showed in [19] how evolutions of polyhedral-invariant hybrid automata can be approximated using polyhedra, Asarin et al. gave in [20] a technique to approximate reachability analysis of piecewise-linear dynamical systems, Kurzhanski and Varaiya introduced ellipsoidal techniques in [21], Alur et al. proposed in [22] *predicate abstraction* as a technique to perform reachability analysis. Many tools, based on such techniques, have been developed in the last years. In particular, we can recall *HyTech* [23], *d/dt* [24], *Checkmate* [25], *UPPAAL* [26], and *KRONOS* [27]. Unfortunately, all these approximation methods and tools are again defined on restricted classes of hybrid automata. Such classes are clearly larger than the classes on which decidability has been proved. However, it is still necessary to check that the model satisfies all the required conditions before the method can be applied.

Semi-algebraic hybrid automata introduced in [5] intrinsically combine syntactic restrictions and semantics approximations. On the one hand Taylor polynomials can be used to approximate a large class of hybrid automata with semi-algebraic ones. In [28] Lanotte and Tini proposed an approximation technique for hybrid automata that exploits Taylor polynomials to obtain from an hybrid automaton H a polynomial hybrid automaton H' that over-approximate H . On the other hand, cylindrical algebraic decomposition (CAD) algorithms (see, e.g., [29–32]) can be used to reason on semi-algebraic hybrid automata. Such considerations are also at the basis of the abstractions and analysis techniques presented in [3].

The tool QEPCAD B [7] efficiently implements Collins' CAD-based algorithm [29] for quantifier elimination, transforming any given first-order semi-algebraic formula into an equivalent quantifier-free one and it can easily become the engine of a step-by-step reachability algorithm for semi-algebraic automata. Unfortunately, the computational cost is still too high. QEPCAD B is not the only tool which can be used to manage constraints over the reals. In particular, we recall: RSOLVER [10], a program for solving quantified inequality constraints

over the reals based on a *branch-and-prune* algorithm; ECLⁱPS^e [11], a software system for the development and deployment of constraint programming applications that contains a general interval propagation solver which can be used to solve problems over both integer and real variables; REDLOG [33], a package that extends the computer algebra system REDUCE to a system that provides algorithms for the symbolic manipulation of first-order formulæ with some syntactic restrictions on the quantified variables; CLP(RL) [34], a constraint solving system, implemented on top the computer logic system REDLOG, where the admissible constraints are arbitrary first-order formulæ.

2 Reachability in Semi-Algebraic Hybrid Automata

In this section we introduce the standard syntax and semantics of hybrid automata and describe the reachability problem on semi-algebraic hybrid automata.

We start with some notations and conventions we use on hybrid automata. Capital letters $Z_1, Z_2, \dots, Z_m, Z'_1, \dots, Z'_m, \dots$, denote variables ranging over \mathbb{R} . Analogously, Z denotes the vector of variables $\langle Z_1, \dots, Z_d \rangle$ and Z' denotes the vector $\langle Z'_1, \dots, Z'_d \rangle$. The temporal variables T, T', T'', \dots model time and range over $\mathbb{R}_{\geq 0}$. We use the small letters p, q, r, s, \dots to denote d -dimensional vectors of real numbers. Occasionally, we may use the notation $\varphi[X_1, \dots, X_m]$ to stress the fact that the set of free variables of the first-order formula φ is included in the set of variables $\{X_1, \dots, X_m\}$. By extension, if $\{Z_1, \dots, Z_n\}$ is a set of variable vectors, $\varphi[Z_1, \dots, Z_n]$ indicates that the free variables of φ are included in the set of components of Z_1, \dots, Z_n . Moreover, given a formula $\varphi[Z_1, \dots, Z_i, \dots, Z_n]$ and a vector p of the same dimension as the variable vector Z_i , the formula obtained by component-wise substitution of Z_i with p is denoted by $\varphi[Z_1, \dots, Z_{i-1}, p, Z_{i+1}, \dots, Z_n]$. When in φ the only free variables are the components of Z_i , after the substitution we can determine the truth value of $\varphi[p]$.

Hybrid automata have a mixed discrete and continuous behaviour. The discrete component is represented by a graph, while the continuous one is given as a set of continuous variables. For each node of the discrete graph we have an invariant condition and a dynamic law over the continuous variables. The dynamic law may depend on the initial conditions, i.e., on the values of the continuous variables at the beginning of the evolution in the state. The jumps from one discrete state to another are regulated by activation and reset conditions on the continuous variables.

Definition 1 (Hybrid Automata - Syntax). A hybrid automaton $H = (Z, Z', \mathcal{V}, \mathcal{E}, Inv, Dyn, Act, Res)$ of dimension d consists of the following components:

1. $Z = \langle Z_1, \dots, Z_d \rangle$ and $Z' = \langle Z'_1, \dots, Z'_d \rangle$ are two vectors of variables ranging over the reals \mathbb{R} ;
2. $\langle \mathcal{V}, \mathcal{E} \rangle$ is a graph. Each element of \mathcal{V} will be dubbed location.
3. Each vertex $v \in \mathcal{V}$ is labeled by the formulæ $Inv(v)[Z]$ and $Dyn(v)[Z, Z', T] \equiv Z' = f_v(Z, T)$, where $f_v : \mathbb{R}^d \times \mathbb{R}_{\geq 0} \longrightarrow \mathbb{R}^d$;

4. Each edge $e \in \mathcal{E}$ is labeled by the two formulæ $Act(e)[Z]$ and $Res(e)[Z, Z']$.

The semantics of hybrid automata regulates the time evolution of the continuous variables.

Definition 2 (Hybrid Automata - Semantics). A state ℓ of H is a pair $\langle v, r \rangle$, where $v \in \mathcal{V}$ is a location and $r = \langle r_1, \dots, r_d \rangle \in \mathbb{R}^{d(H)}$ is an assignment of values for the variables of Z . A state $\langle v, r \rangle$ is said to be admissible if $Inv(v)[r]$ is true.

The continuous reachability transition relation \xrightarrow{t}_C , with $t > 0$ is the transition elapsed time, between admissible states is defined as follows:

$\langle v, r \rangle \xrightarrow{t}_C \langle v, s \rangle$ iff it holds that $s = f_v(r, t)$, and for each $t' \in [0, t]$ the formula $Inv(v)[f_v(r, t')]$ is true.

The discrete reachability transition relation \xrightarrow{e}_D between admissible states is defined as follows:

$\langle v, r \rangle \xrightarrow{e}_D \langle u, s \rangle$ iff both $Act(e)[r]$ and $Res(e)[r, s]$ are true.

We use the notation $\ell \rightarrow \ell'$ to denote that either $\ell \xrightarrow{t}_C \ell'$ or $\ell \xrightarrow{e}_D \ell'$, for some $t \in \mathbb{R}_{\geq 0}$, $e \in \mathcal{E}$.

A trace is a sequence of continuous and discrete transitions. A point s is reachable from a point r if there is a trace starting from r and ending in s .

Definition 3 (Hybrid Automata - Reachability). A trace of H is a sequence of admissible states $[\ell_0, \ell_1, \dots, \ell_i, \dots, \ell_n]$ such that $\ell_{i-1} \rightarrow \ell_i$ holds for each $1 \leq i \leq n$.

The automaton H reaches a point $s \in \mathbb{R}^d$ (in time t) from a point $r \in \mathbb{R}^d$ if there exists a trace $tr = [\ell_0, \dots, \ell_n]$ of H such that $\ell_0 = \langle v, r \rangle$ and $\ell_n = \langle u, s \rangle$, for some $v, u \in \mathcal{V}$ (and t is the sum of the continuous transitions elapsed times). In such a case, we also say that s is reachable from r in H .

A path ph over a graph G is a sequence $[v_0, \dots, v_n]$ of nodes of G such that for each $1 \leq i \leq n$ there is an edge from v_{i-1} to v_i . Given a hybrid automaton H and trace, tr , of H , a corresponding path of tr is a path ph obtained by considering the discrete transitions occurring in tr .

We are interested in the reachability problem for hybrid automata, namely, given a hybrid automaton H , an initial set of points $I \subseteq \mathbb{R}^d$, and a final set of points $F \subseteq \mathbb{R}^d$ we wish to decide whether there exists a point in I from which a point in F is reachable.

An interesting class of hybrid automata is the class of *semi-algebraic hybrid automata* [5].

Definition 4 (Semi-Algebraic Automata). A hybrid automaton H of dimension d is semi-algebraic if $Dyn(v)$, $Inv(v)$, $Act(e)$, and $Res(e)$ are formulæ belonging to the first-order theory of $(\mathbb{R}, 0, 1, +, *, <)$ [6], also known as the theory of semi-algebraic sets.

Moreover, we say that H is continuous if $\forall v \in \mathcal{V} f_v(Z, T)$ is continuous on $\mathbb{R}^d \times \mathbb{R}_{\geq 0}$ and $f_v(r, 0) = r$, for each $r \in \mathbb{R}^d$.

In the rest of this paper we concentrate on continuous semi-algebraic hybrid automata, avoiding all the technical problems concerning the existence, uniqueness and continuity of dynamics (see [16] for more details).

The reachability problem for such class of automata is semi-decidable and it can be reduced to the satisfiability of a numerable disjunction of formulæ of the form $Reach(ph)[Z, Z']$ [16]. In particular, if H is a semi-algebraic automaton, then $q \in \mathbb{R}^d$ is reachable from $p \in \mathbb{R}^d$ in H through a trace whose corresponding path is ph if and only if the formula $Reach(ph)[p, q]$ holds. Unfortunately, as proved in [8], the reachability problem for semi-algebraic automata remains undecidable even if we consider computational models over the reals.

Now let us have a closer look at the first-order formulæ involved in the reachability computation. Inside a discrete location v the following formula expresses that Z reaches Z' :

$$Reach(v)[Z, Z'] \equiv Inv(v)[Z] \wedge \exists T \geq 0 (Z' = f_v(Z, T) \wedge \forall 0 \leq T' \leq T (Inv(v)[f_v(Z, T')]))$$

On the other hand, when we cross an edge $\langle v, u \rangle$ we have to consider the formula:

$$Reach(\langle v, u \rangle)[Z, Z'] \equiv Inv(v)[Z] \wedge Act(\langle v, u \rangle)[Z] \wedge Res(\langle v, u \rangle)[Z, Z'] \wedge Inv(u)[Z']$$

Combining the above formulæ, for each path ph we can easily construct the formula $Reach(ph)[Z, Z']$. For instance if we have the path $ph = [v, u]$, then:

$$Reach([v, u])[Z, Z'] \equiv \exists Z'', Z''' (Reach(v)[Z, Z''] \wedge Reach(\langle v, u \rangle)[Z'', Z'''] \wedge Reach(u)[Z''', Z'])$$

Example 1. Let $H_1 = (Z, Z', \mathcal{V}, \mathcal{E}, Inv, Dyn, Act, Res)$ where:

- Z, Z' are variables over \mathbb{R} ,
- $\mathcal{V} = \{v, u\}$ and $\mathcal{E} = \{e\}$, where e goes from v to u ,
- $Inv(v)[Z] \equiv 1 \leq Z \leq 10$ and $Inv(u)[Z] \equiv 10 \leq Z \leq 20$,
- $Dyn(v)[Z, Z', T] \equiv Z' = Z + (2Z^2 + Z)T$ and
 $Dyn(u)[Z, Z', T] \equiv Z' = Z + (3Z^2 + Z)T$,
- $Act(e)[Z] \equiv Z = 10$,
- $Res(e)[Z, Z'] \equiv Z' = Z$.

The formula for the path $ph = [v, u]$ is the following:

$$Reach([v, u])[Z, Z'] \equiv \exists Z'', Z''' \left(Inv(v)[Z] \wedge \exists T \geq 0 (Z'' = Z + (2Z^2 + Z)T \wedge \forall 0 \leq T' \leq T (Inv(v)[Z + (2Z^2 + Z)T']) \wedge Inv(v)[Z''] \wedge Act(e)[Z''] \wedge Res(e)[Z'', Z'''] \wedge Inv(u)[Z'''] \wedge \exists T'' \geq 0 (Z' = Z''' + (3Z'''^2 + Z''')T'' \wedge \forall 0 \leq T''' \leq T'' (Inv(u)[Z''' + (3Z'''^2 + Z''')T''']) \right)$$

3 Solving the Reachability Problem

In this section we describe some approximation methods for the reachability problem on semi-algebraic hybrid automata. All the computations have been performed on a Dual Core AMD Opteron™ Processor 275, 2205.042 MHz with 4 GB RAM, running CentOS.

The complexity of the reachability formulæ presented in Section 2 increases with the length of the discrete path. In particular, we can notice that the degree of the involved polynomials and the quantifier alternation remains bounded, while the number of variables linearly increases.

Since the first-order theory of $(\mathbb{R}, 0, 1, +, *, <)$ admits the quantifier elimination, we can try to bound the number of variables in each formulæ. When we apply the quantifier elimination procedure to $Reach(ph)[Z, Z']$ we obtain an equivalent first-order formula $\phi[Z, Z']$ involving only the variables Z and Z' . If we now add a step to the path $ph = [v_1, \dots, v_n]$, i.e., we consider the path $ph' = [v_1, \dots, v_n, v_{n+1}]$, we only have to apply quantifier elimination to the formula:

$$\exists Z'', Z''' (\phi[Z, Z''] \wedge Reach((v_n, v_{n+1}))[Z'', Z'''] \wedge Reach(v_{n+1})(Z''', Z'))$$

Proceeding in this way, it seems that we can keep under control the complexity of our method. Unfortunately, if we try to apply it, exploiting QEPCAD B to obtain quantifier free formulæ at each step, we cannot go far enough, as shown by the following example.

Example 2. Consider the following hybrid automaton.
 $H_2 = (Z, Z', \mathcal{V}, \mathcal{E}, Inv, Dyn, Act, Res)$ where:

- $Z = \langle Z_1, Z_2 \rangle$ and $Z' = \langle Z_1', Z_2' \rangle$, where Z_1, Z_2, Z_1', Z_2' variables over \mathbb{R} ,
- $\mathcal{V} = \{v, u\}$ and $\mathcal{E} = \{e\}$, where e goes from v to u ,
- $Inv(v)[Z] \equiv 1 \leq Z_1 \leq 10 \wedge 1 \leq Z_2 \leq 10$ and
 $Inv(u)[Z] \equiv 10 \leq Z_1 \leq 20 \wedge 10 \leq Z_2 \leq 20$,
- $Dyn(v)[Z, Z', T] \equiv Z_1' = Z_1 + (2Z_1^2 + Z_1)T \wedge Z_2' = Z_2 + (2Z_2^2 + Z_2)T$ and
 $Dyn(u)[Z, Z', T] \equiv Z_1' = Z_1 + (3Z_1^2 + Z_1)T \wedge Z_2' = Z_2 + (3Z_2^2 + Z_2)T$,
- $Act(e)[Z] \equiv Z_1 = 10 \wedge Z_2 = 10$,
- $Res(e)[Z, Z'] \equiv Z_1' = Z_1 \wedge Z_2' = Z_2$.

Suppose we want to apply the method described above with $ph = [v, u]$. First, we use QEPCAD B to compute a quantifier free formula $\phi[Z, Z']$ equivalent to the formula $Reach(v)[Z, Z']$. Then we construct the formula:

$$\exists Z'', Z''' (\phi[Z, Z''] \wedge Reach((v, u))[Z'', Z'''] \wedge Reach(u)(Z''', Z'))$$

When we try to compute an equivalent quantifier free formula with QEPCAD B we find out that we cannot obtain any result within 20 minutes of CPU time.

Using this method we are able to limit the number of variables in our formulæ, but we have an increasing number of polynomials and constraints in the computed quantifier free formulæ. This is one of the problems of this method,

since the complexity of the new constructed formulæ strongly depends on the number of polynomials and constraints occurring in computed quantifier free formulæ. Another problem of the method is that QEPCAD B could not give any result in reasonable time when used on formulæ of the form $Reach(v)[Z, Z']$, i.e., the reachability problem inside a location could be already too complex.

At this point the only possibility we have is that of introducing approximations. A first approximated approach to the reachability problem consists in the application of the above method exploiting RSOLVER instead of QEPCAD B. Acting in this way we hope to solve both the problems mentioned in Example 2. Unfortunately, this approach is less effective than the previous one.

Example 3. Consider the hybrid automaton H_2 of example 2. RSOLVER on the formula $Reach(v)[Z, Z']$ gives the following result:

```
True, volume ~[ 0., 0.]
False, volume ~[ 5905.08179397, 5905.08179397]
:
:
Unknown:
:
```

Since the **True** set is empty we do not know which values of Z and Z' satisfy the formula $Reach(v)[Z, Z']$ and we cannot proceed with the next step.

The results obtained with RSOLVER on formulæ of the form $Reach(v)[Z, Z']$ are too approximated for being used. However, we can use it to try to solve the problem related to the number of polynomials and constraints appearing in computed quantifier free formulæ. To do this we apply the previous method exploiting QEPCAD B with the add of an intermediate step that involves the use of RSOLVER.

More precisely, consider the path $ph' = [v_1, \dots, v_n, v_{n+1}]$ and suppose we have already computed a quantifier free formula $\phi[Z, Z']$ equivalent to a formula $Reach(ph)[Z, Z']$, where $ph = [v_1, \dots, v_n]$. Using RSOLVER we compute an approximation of the set of values for Z and Z' that satisfy $\phi[Z, Z']$, then we construct a first-order formula $\gamma[Z, Z']$ defining such approximation. Finally, we apply the quantifier elimination procedure to the formula:

$$\exists Z'', Z''' (\gamma[Z, Z''] \wedge Reach(\langle v_n, v_{n+1} \rangle)[Z'', Z'''] \wedge Reach(v_{n+1})(Z''', Z'))$$

It is still not enough, as shown by the following example.

Example 4. Consider again the hybrid automaton H_2 of Example 2. Let $\phi[Z, Z']$ be the quantifier free formula equivalent to $Reach(v)[Z, Z']$ computed using QEPCAD B. RSOLVER on the formula $\phi[Z, Z']$ gives the following result:

```
True, volume ~[ 0., 0.]
False, volume ~[ 5904.9114008, 5904.91140081]
:
:
Unknown:
:
```

As in Example 3 we obtain an empty `True` set and we cannot proceed with the next step.

Another approximated approach that we can consider consists in the application of this last described method using `ECLiPSe` instead of `RSOLVER` to compute the set of values that satisfy a quantifier free formula obtained with `QEPCAD B`.

Given a quantifier free formula we can define a constraint satisfaction problem with constraint on reals that can be solved by `ECLiPSe` through constraint propagation and search techniques. An answer to a problem on reals is called conditional solution. The number of conditional solutions returned vary according to the level of precision in the search procedure. For instance, given the problem defined from the formula $\phi[Z, Z']$ of Example 4 and using the predicate `locate/2` with final precision 1.0 `ECLiPSe` returns 51 answers, if we reduce the final precision to 0.1 we obtain more than 102 answers. Even if we find a way to use the values computed by `ECLiPSe` to construct the formula for the successive step, the method would not be effective, because we still have the problem that `QEPCAD B` could not give any result when used on formulæ of the form $Reach(v)[Z, Z']$.

All the above discussed methods share one problem: the high cost in terms of computation time that has to be afforded to compute a quantifier free formula from a formula of the form $Reach(v)[Z, Z']$ using `QEPCAD B`. We have to find an approximation strategy to solve this problem in order to obtain an effective method to compute approximated solutions for the reachability problem.

To achieve this goal we studied a method to over-approximated the set of values reachable inside a discrete location of an automaton with independent dynamics.

Definition 5 (Hybrid Automata with Independent Dynamics). *Let H be a continuous semi-algebraic hybrid automaton of dimension d , let $Z = \langle Z_1, \dots, Z_d \rangle$ and $Z' = \langle Z'_1, \dots, Z'_d \rangle$. H has independent dynamics if $\forall v \in \mathcal{V}$ the formula $Dyn(v)[Z, Z', T]$ is of the form:*

$$Z'_1 = f_{v,1}(Z_1, T) \wedge Z'_2 = f_{v,2}(Z_2, T) \wedge \dots \wedge Z'_d = f_{v,d}(Z_d, T)$$

Example 5. The automaton H_2 of Example 2 is a continuous semi-algebraic hybrid automaton with independent dynamics.

Given a discrete location v of an automaton with independent dynamics, we over-approximate the set of values Z' that can be reached inside v after time δ from values Z satisfying $Inv(v)[Z]$ applying the quantifier elimination procedure to the following formula

$$ReachApprox(v)[Z'] \equiv \exists Z (Inv(v)[Z] \wedge Z' = f_v(Z, \delta) \wedge Inv(v)[Z'])$$

This is an over-approximation of the sets of points reachable at time δ , since we did not check that at each time T' between 0 and δ the invariant is satisfied by $f_v(Z, T')$. Notice also that we can replace the condition $Inv(v)[Z]$ with a stronger

one if we are interested in a subset of starting points. Using this formula many times we can compute an over-approximation of all the values Z' that can be reached with δ -time steps from values Z satisfying $Inv(v)[Z]$. After each step of duration δ we can consider the following formula to check if the edge $\langle v, u \rangle$ can be crossed:

$$ReachApprox(\langle v, u \rangle)[Z'] \equiv \exists Z (\phi[Z] \wedge Act(\langle v, u \rangle)[Z] \wedge Res(\langle v, u \rangle)[Z, Z'] \wedge Inv(u)[Z'])$$

where $\phi[Z']$ is a quantifier free formula equivalent to $ReachApprox(v)[Z']$. We apply the quantifier elimination procedure to this formula. If it results to be false, we increase the value of δ to compute another quantifier free formula $\phi[Z']$. Otherwise, we obtain a quantifier free formula $\psi[Z']$ and we can move to the discrete location u where we can apply this procedure considering the formula:

$$ReachApprox(u)[Z'] \equiv \exists Z (\psi[Z] \wedge Z' = f_u(Z, \delta) \wedge Inv(u)[Z'])$$

Proceeding in this way we can keep under control the complexity of our formulæ, avoiding increases in the number of variables and polynomials. Exploiting QEPCAD B to apply this method on the automaton H_2 of Example 2 we can prove that the discrete location u can be reached from v . The result is obtained in 30 milliseconds. Using this method we are able to find approximated solutions for the reachability problem also on automata with independent dynamics with more continuous variables and more complex formulæ than the ones occurring in H_2 .

The method can be applied also to automata with non-independent dynamics, but it does not help us, as shown by the following example.

Example 6. Consider the following hybrid automaton with non-independent dynamics. $H_3 = (Z, Z', \mathcal{V}, \mathcal{E}, Inv, Dyn, Act, Res)$ where:

- $Z = \langle Z_1, Z_2 \rangle$ and $Z' = \langle Z_1', Z_2' \rangle$, where Z_1, Z_2, Z_1', Z_2' variables over \mathbb{R} ,
- $\mathcal{V} = \{v, u\}$ and $\mathcal{E} = \{e\}$, where e goes from v to u ,
- $Inv(v)[Z] \equiv 1 \leq Z_1 \leq 10 \wedge 1 \leq Z_2 \leq 8$ and
 $Inv(u)[Z] \equiv 8 \leq Z_1 \leq 50 \wedge 7 \leq Z_2 \leq 30$,
- $Dyn(v)[Z, Z', T] \equiv Z_1' = Z_1 + (2Z_1^2 + Z_1Z_2)T \wedge Z_2' = Z_2 + (7Z_2^2 + Z_2Z_1)T$
and
 $Dyn(u)[Z, Z', T] \equiv Z_1' = Z_1 + (3Z_1^2 + Z_1Z_2)T \wedge Z_2' = Z_2 + (4Z_2^2 + Z_2Z_1)T$,
- $Act(e)[Z] \equiv Z_1 \geq 8 \wedge Z_2 \geq 7$,
- $Res(e)[Z, Z'] \equiv Z_1' = Z_1 \wedge Z_2' = Z_2$.

Suppose we want to apply the method to find out if the discrete location u is reachable from v . First, we have to apply the quantifier elimination procedure to the formula $ReachApprox(v)[Z']$ with a fixed value δ . When we try to do this using QEPCAD B we are not able to obtain any result within 20 minutes of CPU time because of the presence of non-independent dynamics.

To find approximated solution for the reachability problem on automata with non-independent dynamics, we have to introduce further approximations in our last method. Let H be an automaton with non-independent dynamics. We have that $\forall v \in \mathcal{V}$ the formula $Dyn(v)[Z, Z', T]$ is of the form:

$$Z_1' = f_{v,1}(Z, T) \wedge Z_2' = f_{v,2}(Z, T) \wedge \dots \wedge Z_d' = f_{v,d}(Z, T)$$

Given the formula $ReachApprox(v)[Z']$ and $\delta > 0$, we compute $\forall = 1, \dots, d$ the minimum value ($min_i(v)$) and the maximum value ($max_i(v)$) that the function $f_{v,i}(Z, \delta)$ assume in the set defined by the formula $Inv(v)[Z]$. In order to determine an approximation of the values Z' that can be reached after time δ from values Z satisfying $Inv(v)[Z]$, we evaluate the following formula:

$$\bigwedge_{i=1, \dots, d} min_i(v) \leq Z_i' \leq max_i(v) \wedge Inv(v)[Z']$$

If, in the previous method, we use this procedure instead of the quantifier elimination procedure to obtain a formula $\phi[Z']$ from formula $ReachApprox(v)[Z']$, we have a new method that can find approximated solution to the reachability problem even in presence of non-independent dynamics.

Example 7. Consider the hybrid automaton of Example 6. Suppose we want to apply the approximated method described above exploiting QEPCAD B to find out if the discrete location u can be reached from v . First, we compute a formula $\phi[Z']$ using the procedure based on the calculus of minimum and maximum of each function in $Dyn(v)[Z, Z', T]$ described above. Then we apply the quantifier elimination procedure to the formula:

$$\exists Z (\phi[Z] \wedge Act(\langle v, u \rangle)[Z] \wedge Res(\langle v, u \rangle)[Z, Z'] \wedge Inv(u)[Z'])$$

We obtain a quantifier free formula representing the values for Z_1' and Z_2' in the discrete location u that can be reached starting from v . We succeed in proving the desired property (result obtained in 55 milliseconds).

We notice that solutions computed with this method are neither over nor under approximations.

Exploiting this last method together with QEPCAD B, we could not be able to obtain results on some automata. In particular, on automata whose dynamics are either very complex or not representable in QEPCAD B, e.g., functions where non integer or negative exponents appear. We can solve this problem introducing a further approximation to our method.

Consider a formula $Dyn(v)[Z, Z', T] \equiv \bigwedge_{i=1, \dots, d} Z_i' = f_{v,i}(Z, T)$. Instead of computing the maximum and the minimum of $f_{v,i}(Z, \delta)$ in the set defined by the formula $Inv(v)[Z]$, we can compute the maximum and the minimum of the linearization of $f_{v,i}$ that is the Taylor polynomial of degree one:

$$Z_i'(\delta) = f_{v,i}(Z(0), 0) + \frac{df_{v,i}}{dT}(Z(0), 0)\delta + R$$

where R is the reminder term. In order to compute the maximum and the minimum at time δ_j (where $\delta_0 = \delta$ and $\delta_j > \delta_{j-1}$) we consider the following expression:

$$Z'(\delta_j) = Z'(\delta_{j-1}) + \frac{df_{v,i}}{dT}(Z(0), \delta_{j-1})\delta_j + R$$

Notice that the derivative $df_{v,i}/dT$ has not to be computed for every time interval. Once computed we can obtain a function that can be used to calculate the values of the derivative for all the different δ_j .

4 The Repressilator Case Study

As a simple yet very interesting example, we consider the Repressilator system constructed by Elowitz and Leibler [35]. It consists of three proteins, namely *lacl*, *tetR*, and *cl*, and the corresponding genes. The protein *lacl* represses the gene which expresses *tetR*, *tetR* represses the gene which expresses *cl*, whereas *cl* represses the gene which expresses *lacl*, thus completing a feedback system. The dynamics of the network depend on the transcription rates, translation rates, and decay rates. Depending on the values of these rates the system might converge to a stable limit circle or become unstable.

We apply our method to compare the behaviour of two oscillating models proposed for the Repressilator.

First, we consider the hybrid automaton proposed in [36] to model the Repressilator. This hybrid automaton has 8 discrete locations, corresponding to all the possible combinations of genes being either *on* or *off*, and 9 variables. Three of them, A , B , C , represent the quantity of proteins in the system, the other six, $Y_{X,on}$, $Y_{X,off}$, where $X \in \{A, B, C\}$, control activation and deactivation of genes. For each discrete location v , $Inv(v) \equiv true$.

The differential equations governing proteins concentrations in each discrete location are decoupled: for instance, when gene A is *on* its dynamics is $\dot{A} = k_p - k_d A$, where k_p and k_d are constant parameters of the system.

The interactions between repressors and genes are confined to the activation conditions of the automaton transitions. Consider again gene A and suppose to be in a discrete location of the automaton where it is *on*. Then, the differential equation for $\dot{Y}_{A,off}$ is $\dot{Y}_{A,off} = k_b C$, the transition switching this gene *off* has an activation condition equal to $Y_{A,off} \geq 1 \wedge C \geq 1$ and a reset condition equal to $Y_{A,on} = 0 \wedge Y_{A,off} = 0$. The transition that turns gene A *on*, instead, has a constant rate k_u , hence its activation condition is $Y_{A,on} \geq 1$, $\dot{Y}_{A,on} = k_u$ is the differential equation for $\dot{Y}_{A,on}$ and the reset condition is equal to $Y_{A,on} = 0 \wedge Y_{A,off} = 0$, where k_b and k_u are constant parameters of the system.

In order to obtain a continuous semi-algebraic automaton from this hybrid automaton, we have only to define for each discrete location v a formula $Dyn(v)$ satisfying the conditions of Definition 4. To this aim we approximate the solutions of the differential equations in each discrete location with the corresponding Taylor polynomial of degree two. Consider, for instance, the differential equations for A , $Y_{A,off}$, and $Y_{A,on}$ in a discrete location where gene A is *on*, we

approximate their solution with the following polynomials:

$$\begin{aligned} A' &= A + (k_p - k_d A)T + (-k_d k_p + k_d^2 A)T^2/2 \\ Y'_{A,off} &= Y_{A,off} + (k_b C)T + (-k_b k_d C)T^2/2 && \text{if gene } C \text{ is } off \\ Y'_{A,off} &= Y_{A,off} + (k_b C)T + (k_b k_p - k_b k_d C)T^2/2 && \text{if gene } C \text{ is } on \\ Y'_{A,on} &= Y_{A,on} + k_u T \end{aligned}$$

The solution of the differential equation for A in a discrete location where the gene A is *off* is approximated with the following polynomial:

$$A' = A + (-k_d A)T + (k_d^2 A)T^2/2$$

The automaton we obtain has non-independent dynamics (see, e.g., the equation for $Y'_{A,off}$), hence we analyse it using the approximated method based on the computation of minimum and maximum values of the functions defining the dynamics. Starting from the discrete location where only gene A is active and with fixed values for proteins concentrations we succeed in simulating the automaton and observe an oscillatory behaviour (Fig. 1).

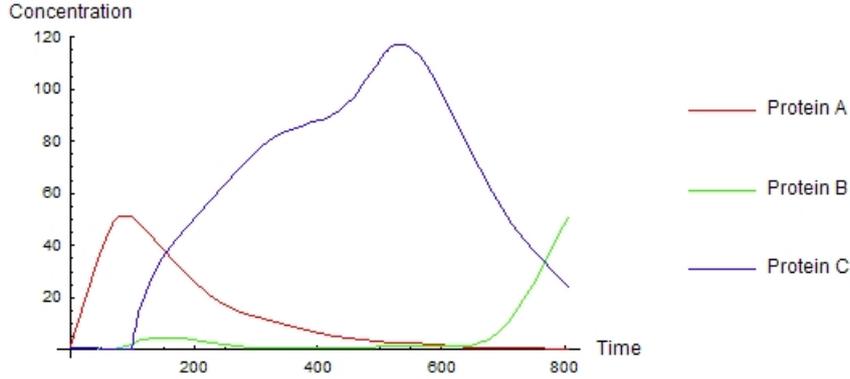


Fig. 1. Time trace of the hybrid automata with 8 discrete locations. Parameters are $k_p = 1$, $k_d = 0.01$, $k_b = 1$, $k_u = 0.01$.

The second hybrid automata we consider is the one that can be constructed from the following model for the Repressilator written in the S-System equations formalism [37] (see [38])

$$\begin{aligned} \dot{X}_1 &= \alpha_1 X_3^{-1} - \beta_1 X_1^{0.5}, && \alpha_1 = 0.2, \beta_1 = 1, \\ \dot{X}_2 &= \alpha_2 X_1^{-1} - \beta_2 X_2^{0.578151}, && \alpha_2 = 0.2, \beta_2 = 1, \\ \dot{X}_3 &= \alpha_3 X_2^{-1} - \beta_3 X_3^{0.5}, && \alpha_3 = 0.2, \beta_3 = 1. \end{aligned}$$

From this model we obtain an hybrid automaton with only one discrete location, no transitions and three variables, X_1 , X_2 , X_3 , representing proteins concentrations. For the unique discrete location v we have $Inv(v) \equiv true$, the dynamics in v are defined by the differential equations of the S-System model.

As in the previous case, to obtain a continuous semi-algebraic automaton we approximate the solutions of the differential equations in the discrete location with the corresponding Taylor polynomial of degree two. Consider for instance the differential equation for X_1 , we approximate its solution with the following polynomial:

$$X_1' = X_1 + (0.2X_3^{-1} - X_1^{0.5})T + (-0.04X_3^{-2}X_2^{-1} + 0.2X_3^{-1.5} - 0.1X_1^{-0.5}X_3^{-1} + 0.5)T^2/2$$

The automaton we obtain has non-independent dynamics with real exponents, hence we analyse it using the approximated method based on the computation of minimum and maximum values of the linearization of the functions defining the dynamics. We succeed in the simulation of the automaton, but we do not obtain any interesting result.

The analysis of the two models shows that the one obtained from the S-System does not permit to observe the oscillatory behaviour of the Repressilator, this because of the approximations introduced for simulation. The other model, instead, results to be less sensitive to approximation and simulating it we can observe the oscillations in proteins concentrations. This points out that in order to define robust models for biological systems it is important to distinguish from the beginning the discrete from the continuous parts of the systems. Hybrid automata allow to do this and hence to obtain simpler dynamics in each discrete location. Such dynamics are less sensible to the approximations which are necessary to carry out formal analysis.

5 Conclusions

In this paper we presented some experimental results on the reachability problem in semi-algebraic hybrid automata. Our results suggest that even if we try to exploit different techniques and powerful tools, we cannot go *far* enough, without introducing approximations.

However, it is easy to apply some standard, basic, approximation techniques. We showed on the Repressilator case study that the approximated results are coherent with the expected behaviour, even when we limit our approximations to the first and second degree, provided that intrinsic discrete nature of the system has been explicitly modeled. In particular, the approximations on the 8-states automaton show the oscillations, while this is not the case if we directly apply our method to the system of differential equations. Intuitively, the system of differential equations implicitly models the discrete nature of the system exploiting more complex dynamics whose simulation requires more sophisticated techniques. The hybrid automaton allows to keep the dynamics more simple and more *robust* to approximations.

References

1. Alur, R., Henzinger, T.A., Ho, P.H.: Automatic Symbolic Verification of Embedded Systems. In: Proceedings of IEEE Real-Time Systems Symposium, IEEE Computer Society Press (1993) 2–11
2. Alur, R., Belta, C., Ivancic, F., Kumar, V., Mintz, M., Pappas, G.J., Rubin, H., Schug, J.: Hybrid Modeling and Simulation of Biomolecular Networks. In: Proceedings of Hybrid Systems: Computation and Control (HSCC'01). Volume 2034 of LNCS., Springer-Verlag (2001) 19–32
3. Ghosh, R., Tiwari, A., Tomlin, C.: Automated Symbolic Reachability Analysis; with Application to Delta-Notch Signaling Automata. In Maler, O., Pnueli, A., eds.: Proceedings of Hybrid Systems: Computation and Control (HSCC'03). Volume 2623 of LNCS., Springer-Verlag (2003) 233–248
4. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing (STOC '95), New York, NY, USA, ACM Press (29 May–1 June 1995) 373–382
5. Piazza, C., Antoniotti, M., Mysore, V., Policriti, A., Winkler, F., Mishra, B.: Algorithmic Algebraic Model Checking I: Challenges from Systems Biology. In Etessami, K., Rajamani, S.K., eds.: Proceedings Computer Aided Verification (CAV'05). Number 3576 in LNCS, Springer-Verlag (2005) 5–19
6. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. Univ. California Press (1951)
7. Brown, C.W.: QEPCAD B: A program for computing with semi-algebraic sets using CADs. ACM SIGSAM Bulletin **37**(4) (2003) 97–108
8. Mysore, V., Piazza, C., Mishra, B.: Algorithmic Algebraic Model Checking I: Decidability of Semi-Algebraic Model Checking and its Applications to Systems Biology. In D. A. Peled, Y.T., ed.: Proceedings Third International Symposium on Automated Technology for Verification and Analysis (ATVA 2005). Number 3707 in LNCS, Springer-Verlag (October 2005) 217–233
9. Mysore, V., Mishra, B.: Algorithmic Algebraic Model Checking III: Approximate Methods. Electronic Notes in Theoretical Computer Science **149**(1) (2006) 61–77
10. Ratschan, S.: Efficient Solving of Quantified Inequality Constraints over the Real Numbers. ACM Transactions on Computational Logic **7**(4) (2006) 723–748
11. Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press (2006)
12. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science **138**(1) (1995) 3–34
13. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
14. Puri, A., Varaiya, P.: Decidability of hybrid systems with rectangular differential inclusions. In Dill, D.L., ed.: Proceedings of International Conference on Computer Aided Verification (CAV'94). Volume 818 of LNCS., Springer-Verlag (1994) 95–104
15. Lafferriere, G., Pappas, G.J., Sastry, S.: O-Minimal Hybrid Systems. Mathematics of Control, Signals, and Systems **13** (2000) 1–21
16. Casagrande, A., Piazza, C., Mishra, B.: Semi-Algebraic Constant Reset Hybrid Automata -SACoRe. In: Proc. of the 44rd Conference on Decision and Control (CDC'05), IEEE Computer Society Press (2005) 678–683

17. Halbwachs, N., Proy, Y.E., Raymond, P.: Verification of linear hybrid systems by means of convex approximations. In Le Charlier, B., ed.: *Static Analysis Symposium*. Springer-Verlag (1994) 223–237
18. Dang, T., Maler, O.: Reachability analysis via face lifting. In Henzinger, T.A., Sastry, S., eds.: *HSCC*. Volume 1386 of LNCS., Springer-Verlag (1998) 96–109
19. Chutinan, A., Krogh, B.: Verification of Polyhedral-Invariant Hybrid Automata Using Polygonal Flow Pipe Approximations. In Vaandrager, F.W., van Schuppen, J.H., eds.: *Proceedings of Hybrid Systems: Computation and Control (HSCC'99)*. Volume 1569 of LNCS., Springer-Verlag (1999) 76–90
20. Asarin, E., Dang, T., Maler, O., Bournez, O.: Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems. In Krogh, B., Lynch, N., eds.: *Proceedings of Hybrid Systems: Computation and Control (HSCC'00)*. Volume 1790 of LNCS., Springer-Verlag (2000) 20–31
21. Kurzhanski, A.B., Varaiya, P.: On verification of controlled hybrid dynamics through ellipsoidal techniques. In: *Proceedings of the 44rd Conference on Decision and Control and European Control Conference (CDC-ECC'05)*, Seville, Spain, IEEE Computer Society Press (December 2005) 4682–4687
22. Alur, R., Dang, T., Ivancic, F.: Progress on reachability analysis of hybrid systems using predicate abstraction. In Maler, O., Pnueli, A., eds.: *HSCC*. Volume 2623 of LNCS., Springer-Verlag (2003) 4–19
23. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HYTECH: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer* **1**(1–2) (1997) 110–122
24. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: *Proceedings of the Fourteenth International Conference on Computer Aided Verification (CAV'02)*. LNCS, London, UK, Springer-Verlag (2002) 365–370
25. Silva, B.I., Krogh, B.H.: Formal verification of hybrid system using checkmate: A case study. In: *Proceedings of the American Control Conference 2000 (ACC'00)*, Chicago, Illinois, IEEE Computer Society Press (June 2000) 678–683
26. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal - a tool suite for automatic verification of real-time systems. In Alur, R., Henzinger, T.A., Sontag, E.D., eds.: *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop*. Volume 1066 of LNCS., Springer-Verlag (1996) 232–243
27. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The tool KRONOS. In Alur, R., Henzinger, T.A., Sontag, E.D., eds.: *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop*. Volume 1066 of LNCS., Springer-Verlag (1996) 208–219
28. Lanotte, R., Tini, S.: Taylor approximation for hybrid systems. *Inf. Comput.* **205**(11) (2007) 1575–1607
29. Collins, G.E.: Quantifier Elimination for the Elementary Theory of Real Closed Fields by Cylindrical Algebraic Decomposition. In: *Proceedings of the Second GI Conference on Automata Theory and Formal Languages*. Volume 33 of LNCS., Springer-Verlag (1975) 134–183
30. Grigorév, D., Vorobjov, N.: Counting connected components of a semialgebraic set in subexponential time. *Computational Complexity* **2**(2) (1992) 133–186
31. Renegar, J.: On the computational complexity and geometry of the first-order theory of the reals. Part III: Quantifier elimination. *Journal of Symbolic Computation* **13**(3) (1992) 329–352

32. Basu, S.: An improved algorithm for quantifier elimination over real closed fields. In: Proceedings of the Thirty-Eighth Annual Symposium on Foundations of Computer Science (FOCS '97), Washington, DC, USA, IEEE Computer Society Press (1997) 56–65
33. Dolzmann, A., Sturm, T.: REDLOG: Computer algebra meets computer logic. SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation) **31**(2) (1997) 2–9
34. Sturm, T.: Quantifier Elimination-Based Constraint Logic Programming. Technical Report MIP-0202, Fakultät für Mathematik und Informatik, Universität Passau (2002)
35. Elowitz, M., Leibler, S.: A Synthetic Oscillatory Network of Transcriptional Regulators. *Nature* **403** (2000) 335–338
36. Bortolussi, L., Policriti, A.: Hybrid approximation of Stochastic Concurrent Constraint Programming. In: International Federation of Automatic Control World Congress, (IFAC'08). (2008) To appear.
37. Voit, E.O.: Computational Analysis of Biochemical Systems. A Practical Guide for Biochemists and Molecular Biologists. Cambridge University Press (2000)
38. Antonioti, M., Policriti, A., Ugel, N., Mishra, B.: Model Building and Model Checking for Biological Processes. *Cell Biochemistry and Biophysics* (2003)

External Point of View in Process Algebra for Systems Biology [★]

Filippo Del Tedesco¹ and Carla Piazza¹

DIMI, Università di Udine, Via delle Scienze, 206, 33100 Udine, Italy

Abstract. A critical aspect in the modeling of biological systems is the description view point. On the one hand, the Stochastic π -calculus formalism provides an intuitive and compact representation from an internal perspective. On the other hand, other proposed languages such as Hybrid Automata and Stochastic Concurrent Constraint Programming introduce in the system description an external control and provide more structured models.

This work aims at bridging the above discussed gap. In particular, we propose a different approach for the encoding of biological systems in Stochastic π -calculus in the direction of introducing the above mentioned external control and comparing different formalisms. We show the effectiveness of our method on some biochemical examples.

Introduction

Systems Biology exploits different languages and formalisms mainly inherited from mathematics and computer science with the aim of modeling and analyzing complex biological systems in terms of their components and their interactions. A formal modeling and understanding of the underlying laws of life are at the basis of all the scientific research in biology. We just mention here the impact that systems biology could produce on the development of new therapies and drugs.

The huge complexity of biological systems calls for the definition of a range of modeling languages operating at different levels of description (e.g., different space/time scales). However, it is also essential to compare and integrate such languages in a global framework in which information is shared and analyzed from many perspectives.

Among the formalisms proposed for systems biology, the *Stochastic π -calculus*, proposed by Priami in [1], has received growing attention in the last years [2–5]. Many advantages come from the use of Stochastic π -calculus: if we consider the problem of describing a system, it allows to model biological entities (e.g., molecules, genes, proteins) as *processes* and entity interactions as *communications* between processes. Such approach has some interesting peculiarities: for

[★] This work is partially supported by MIUR. Corresponding author: pippodt@libero.it, carla.piazza@dimi.uniud.it.

instance it is strongly compositional, and it explicitly models the biological meaning (functional activity) of each entity. In this sense we can say that it represents the system from an *internal point of view*, since each entity has only knowledge about its interactions. Moreover, formal techniques are available to automatically study properties of stochastic π -calculus processes. Unfortunately, it has also few critical aspects, such as the fact that describing interactions by communications forces to consider only binary exchanges, while communications among three or more principals are not immediately encodable inside the language.

On the other hand, *Ordinary Differential Equations* (see, e.g., [6]) describe the system from a different perspective. If x_i is an entity and X_i is the quantity of x_i in the system, the differential equation $\dot{X}_i = f(X_1, \dots, X_n)$ represents the time evolution of x_i , while the biological meaning of x_i has to be reconstructed by looking at all the right hand sides of the system equations. Roughly speaking we can say that the differential equation $\dot{X}_i = f(X_1, \dots, X_n)$ allows to syntactically represent an *external view* of the system: from outside we can measure the global quantities of each entity. Such external point of view in the Stochastic π -calculus models [2–5] is hidden at a semantic level, where the reaction rates depend on the total amount of reactants. Again there are both advantages and disadvantages. It is very difficult to find exact solutions for differential equations and, at the same time and the standard model-checking techniques are hard to apply.

Other formalisms known in literature such as *Stochastic Concurrent Constraint Programming* (sCCP) [7,8] and *Hybrid Automata* [9–11] push even forward the discrepancy between the internal and the external perspectives by allowing at the syntactic level to model changes in dynamic laws according on global constraints (e.g., activation and reset conditions in the case of hybrid automata).

In this paper we consider a different use of Stochastic π -calculus in the modeling of biological systems with the aim of joining its positive aspects with the possibility of introducing an external control. To obtain this, we model the entities as messages on a “memory” channel and the interactions as processes. We focus on chemical reactions and we use one process for each reaction type. The processes have an external view on the system, since they “count” the total amount of entities in memory. In a sense our approach try to bridge the gap between the Stochastic π -calculus models described in [2–5] and other formalisms such as sCCP and hybrid automata, moving the control on the global state of the system from the semantic to the syntactic level. Our main aim is that of comparing / integrating different modeling languages. This should suggest us how to extend the standard analysis techniques of process algebra to other formalisms. Interestingly, our approach also allows to easily model n -ary chemical reactions. Since, our work is still embryonal we do not prove any semantic equivalence with other works, but we limit our comparisons to some examples, showing that our simulation results are very close with those described in the stochastic simulator SPiM [4].

The paper is organized as follows. In Section 1 we recall the syntax and semantics of both Stochastic π -calculus and SPiM [4], while we present our

proposal in Section 2. Some examples are discussed in Section 3. Section 4 ends the paper with some remarks and future work proposals.

1 Stochastic Pi-Calculus, Biology, and Simulations

1.1 Stochastic Pi-Calculus

Stochastic π -calculus is an extension of the π -calculus process algebra. It has been introduced in [1] with the aim of modeling performances of dynamically reconfigurable or mobile networks. It inherits all the syntax of π -calculus and enriches this last with the possibility of associating to each action a probability distribution. As a consequence, we can associate to each prefix a time duration, represented by the value of a random variable, which follows the above mentioned probability distribution. In the case of memoryless processes exponential distributions can be used and actions take the form (a, r) , also denoted by a_r , where a is the action name and r (*activity rate*) is the parameter characterizing the exponential distribution, i.e., the average duration of (a, r) is $1/r$. Notice that the language allows also instantaneous actions, corresponding to $r = \infty$, in which the rate is omitted.

Let us briefly recall the syntax of Stochastic π -calculus.

Definition 1 (Stochastic π -calculus – Syntax). Let $\mathcal{N} = \{a, b, \dots, x, y, \dots\}$ be an infinite set of names. A Stochastic π -calculus process is an expression of the following grammar:

$$P ::= 0 \mid (\pi, r).P \mid (\nu x)P \mid [x = y]P \mid P|P \mid P + P \mid P(y_1, \dots, y_n)$$

where $r \in \mathbb{R}^+ \cup \{\infty\}$.

The intuitive meaning of the operators is essentially the same as in π -calculus. In particular:

- π is either $x(y)$ or $\bar{x}y$ or τ . $x(y)$ denotes that we are waiting for a message on the channel x and y acts as a placeholder, which will be replaced with the received message. $\bar{x}y$ represents the output of the message y on the channel x . τ is the silent action. All the standard considerations about free and bound names hold.
- $P + Q$ stochastically behaves as either P or Q . The choice depends on the time durations of the actions occurring in P and Q . The fastest will win the race. This is one of the main differences with π -calculus, where $+$ is a nondeterministic choice operator.
- In general the behavior of processes depends on *race conditions*: among all the executable activities we will activate the one which has the shortest duration.

A complete presentation of the operational semantics of the Stochastic π -calculus is outside the scope of this paper and can be found in [1]. We also implicitly adopt all the standard syntactic conventions.

Example 1. Consider the following toy example, representing the interaction between a man and a coffee/tea machine.

$$\begin{aligned} \text{Man} &\stackrel{\text{def}}{=} \overline{\text{coin}} \text{ am.} \overline{\text{choi}} \text{ cof.} \overline{\text{drink}}(\text{smthg}) \\ \text{Machine} &\stackrel{\text{def}}{=} \text{coin}(m).\text{choi}(c).(([\text{c} = \text{cof}] \overline{\text{drink}} \text{ cof}) + ([\text{c} = \text{tea}] \overline{\text{drink}} \text{ tea})) \\ \text{System} &\stackrel{\text{def}}{=} \text{Man} \mid \text{Machine} \end{aligned}$$

In this case all the actions are instantaneous and there are no stochastic effects.

The following example uses the stochastic features of the calculus on a mutual exclusion protocol.

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \overline{\text{down}}_{r_1} \text{ pars1}.0 \\ P_2 &\stackrel{\text{def}}{=} \overline{\text{down}}_{r_2} \text{ pars2}.0 \\ \text{mutex} &\stackrel{\text{def}}{=} (\text{down}_{r_1}(p1).0 + \text{down}_{r_2}(p2).0) \\ \text{race} &\stackrel{\text{def}}{=} P_1 \mid P_2 \mid \text{mutex} \end{aligned}$$

The fastest process win.

1.2 Stochastic Pi-Calculus in Systems Biology

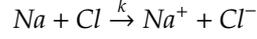
Many works have pointed out the usefulness of Stochastic π -calculus in the modeling of biological systems (see, e.g., [2–5]). We start focusing on [5]. Cardelli observes that the available description languages for biochemical system (e.g., state transition diagrams) are in a sense similar to process algebras, since in both cases labeled transition systems represent concurrent systems. This is not the case in differential equation models. Hence, the use of process algebras in systems biology is natural and provides double advantages: on the one hand, the representation is incremental and compositional; on the other hand, the models support formal verification techniques such as behavioral equivalences and model checking.

Then, the paper formally establishes connections between discrete and continuous descriptions of systems of chemical reactions. The discrete representations are given as processes of a fragment of the Stochastic π -calculus, while the continuous models are systems of ordinary differential equations.

Let us formally introduce the above mentioned representations:

- Chemical reactions are expression of the form $A \xrightarrow{k} B_1 + \dots + B_n$ or $A_1 + A_2 \xrightarrow{k} B_1 + \dots + B_n$ or $A + A \xrightarrow{k} B_1 + \dots + B_n$. They represent the problem from a *macroscopic* point of view.
- Ordinary differential equations can be automatically inferred from chemical reactions and describe the dynamic of a reactant concentration in terms of the other concentrations.
- The *chemical ground form* (CGF) is a subset of the Stochastic π -calculus. It does not allow the use of the restriction operator (νx) and of the test operator $[x = y]$. It is expressive enough to represent each chemical entity involved in the system and hence it provides a *microscopic* description.

Example 2. Let us consider the *Na-Cl* ionization example. The chemical reaction describing the system is the following:



The corresponding differential equations are:

$$\begin{aligned} d[Na^+]/dt &= d[Cl^-]/dt = k[Na][Cl] \\ d[Na]/dt &= d[Cl]/dt = -k[Na][Cl] \end{aligned}$$

In the CGF we have the parallel composition of the processes:

$$Cl = \bar{i}_{k/g}().0 \quad Na = i_{k/g}().(Na^+|Cl^-)$$

where g is a constant for the dimensional conversion (see [5]).

In [5] equivalences between the discrete and continuous semantics of chemical reactions, ordinary differential equations, and CGF are proved. This formally clarifies the connections between the microscopic and macroscopic points of view.

A graphical representation of the above described approach is presented in [3] on a variant of the Stochastic π -calculus. This is at the basis of the current implementation of the stochastic simulator SPiM [4]. In particular, in [3] the language is specialized for the biological context. A system has the form $E \vdash P$, where E is a *constant environment*, i.e., a library of entity definitions, and P is the *test tube* containing all the copies of the entities. Semantics equivalences between the language proposed in this paper and the original Stochastic π -calculus are proved.

Example 3. Let us consider a system consisting of a gene G which can synthesize a protein A in time τ_t . Moreover, protein B , produced by another gene, can either inhibit gene G for time τ_u or decay in time τ_d . The description of the system in Graphical Stochastic π -calculus is the following:

$$\begin{aligned} E : G &\stackrel{def}{=} \tau_t.(G|A) + b_r().\tau_u.G \quad P : G|B \\ B &\stackrel{def}{=} \bar{b}_r.B + \tau_d \end{aligned}$$

1.3 SPiM

The *Stochastic Pi-Machine* (SPiM) simulates the behavior of Stochastic π -calculus processes. The latest versions of SPiM have been optimized for the simulation of processes representing biological systems [4]. The input for the simulator is a process. SPiM determines its time evolution by establishing which action has the highest probability and by performing it. At the basis of SPiM computation there is a variant of Gillespie's algorithm [12], a Monte Carlo procedure which numerically simulates the time evolution of a given system of chemical reactions.

SPiM's input language is a high level translation of the Stochastic π -calculus. Since, we will use it in the remaining part of this work, we briefly introduce here part of its syntax. We address the reader to [4] for all the details.

Definition 2 (SPiM – Syntax).

$$\begin{aligned}
Dec &::= \text{new } x\{\text{@}r\} : \text{chan}\{(Type_1, \dots, Type_n)\} \mid \text{run } P \mid \text{let } D_1 \text{ and } \dots \text{ and } D_N \\
P &::= X(v_1, \dots, v_n) \mid (P_1 \mid \dots \mid P_M) \mid () \mid \text{do } \pi_1\{; P_1\} \text{ or } \dots \text{ or } \pi_M\{; P_M\} \\
&\quad \mid \text{if } b \text{ then } P \{\text{else } P\} \mid n \text{ of } P \mid \pi\{; P\} \\
D &::= X(m_1, \dots, m_N) = P \\
\pi &::= !x\{v_1, \dots, v_N\} \mid ?x\{m_1, \dots, m_N\} \mid \text{delay}@r
\end{aligned}$$

Let us give some intuitions about the meaning of the operators with an example.

Example 4. Let us consider again Example 2 where we described the chemical reaction $Na + Cl \xrightarrow{k} Na^+ + Cl^-$. The SPiM program representing it can be written as follows.

```

new i@r:chan (*declaration of channel i with rate r=k/g*)
let Na()=?i;(Nap()|Clm()) (*declaration of Na molecule*)
  and Cl()=!i;()
  and Nap()=()
  and Clm()=()

run (Na()|Cl())

```

Similarly Example 3 can be translated in SPiM as follows.

```

new a@0.4:chan
new b@0.4:chan
let G()=do delay@1.0;(G()|A())
  or ?b;delay@0.5;G()
  and A()=do !a;A()
  or delay@0.3;()
  and B()=do !b;B()
  or delay@0.3;()
run (G()|B())

```

1.4 Internal versus External

We already noticed that compositionality is one of the main advantages of the use of Stochastic π -calculus in the modeling of biological systems. As emerged from the examples, once we have defined the actors of the system and their rules (interactions, delays, ...), we only have to let them play. This represents an internal perspective on the system, in the sense that each actor only knows his rule, while the movie director is hidden in the semantics and he does not directly talk to the actors.

On the opposite side we can find formalisms based on external perspectives with their own advantages and disadvantages.

How can we compare and integrate internal and external perspectives?

In the next section we try to partially answer to this question, proposing a different use of Stochastic π -calculus which is closer to formalisms with external perspectives.

2 Memory and Pi-Molecules

2.1 Some Intuitions

We start with an informal description of our approach for modeling biochemical systems in (Graphical) Stochastic π -calculus. The idea is that of introducing an external control mechanism in the model. We aim to: (1) keep the representation simple and compositional; (2) remain inside the Graphical Stochastic π -calculus language.

The approaches described in the previous section model the reactants of biochemical systems as *active* entities. Each of them plays a fundamental role in the system evolution.

A first idea could be that of introducing a control process in the system and asking to the reactants to communicate only with the control process. The control process should acquire information from the reactants and give them back the orders. Unfortunately, this make life very complex, since too many communications and stochastic rates are involved. As a matter of fact in this way both the reactants and the reactions are *active* players.

So we propose to have *active* reactions and *passive* reactants.

Let us focus on the *passive* reactants. We model them as messages which are sent and received on a *memory* channel. In particular, for each reactant we have a message representing its quantity in the system.

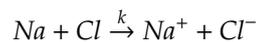
Now we only have to model the *active* reactions. We call them π -molecules, since they do not represent real molecules, but they control the molecules behaviors. Let us consider again Example 2, in which Na becomes Na^+ and Cl becomes Cl^- . The memory channel knows the current number of Na and Cl atoms. The π -molecule has to: (1) Read the memory; (2) Check whether there is at least one atom of Na and one of Cl ; (3) If the answer is positive, modify the memory; (3) Go back to item (1).

Of course this intuitive explanation lacks of the technical details necessary to evaluate the admissibility of our proposal. In particular, all the details concerning the reaction rates are missing. We will see in the next section that all the communications are instantaneous, and the rates are modeled through silent delays. However, already at this level of description the compositionality and scalability of the approach seems guaranteed: the number of π -molecules in the system is proportional to the number of chemical reactions.

2.2 Strategies for the Implementation of the Memory and Pi-Molecules

In this section we describe all the technical details of our method by producing the SPiM program for Example 2. The example allows us to keep the presentation simple. Its generalization is immediate.

We have to model the chemical reaction



The *memory* has to be a resource which can be accessed and modified instantaneously. It has not to contribute on the reaction time, since it is not a *real* part of the the physical system. Moreover, it has to be a permanent resource. All these conditions are satisfied in the SPiM language if we model the memory as an output on a polyadic channel memory having infinite rate. Hence, in our context the implementation of memory is the following: `!mem(na, cl, nap, clm)`, where the parameters `na`, `cl`, `nap` and `clm` represent the quantities of the chemical reactants Na , Cl , Na^+ , and Cl^- , respectively.

Let us now model the π -molecule. It plays the role of the chemical reaction. This includes both the transformation of the reactants and the simulation of the reaction time. Hence, we can distinguish two phases in the evolution of a π -molecule: an instantaneous computational phase and a delay phase.

At this point we have to recall that in general a large number of molecules and many reactions are involved. We say that a reaction is *enabled* if there are enough reactants in the system to fire it, and *disabled* otherwise. Each reaction has to check whether it is enabled or not. The enabled reactions are then delayed (the delays are proportional to both constant rates and quantities). When a reaction is ready to be fired, i.e., after its delay, it takes control and modifies the system. This means that all the π -molecules will read the memory to check their preconditions and, later, some of them will try to change the system state. It is clear that we need to ensure that there are not two π -molecules reading the memory at the same time. Since, we implement the memory as an output on a channel this condition is immediately satisfied. Once a π -molecule is fired it has to check again that the memory is consistent with its input requirements, because in the meanwhile of its delay some changes may have happened. If this is the case the π -molecule can change the memory state, by subtracting its inputs and adding its outputs. Translating the above reasoning in SPiM we get:

```
let Pim()=
  ?mem(na,cl,nap,clm);
  if(na>0.0 * cl>0.0)
  then
    (!mem(na,cl,nap,clm)|
    (delay@ k*na*cl;
    ?mem(_na,_cl,_nap,_clm);
    if (_na>0.0 * _cl>0.0)
    then
      !mem(_na-1.0,_cl-1.0,_nap+1.0,_clm+1.0)|
      Pim())
    else
      (!mem(_na,_cl,_nap,_clm)|Pim()))))
```

We still have to specify the behavior of a π -molecule in the case in which there are not enough reactants. In this case the π -molecule cannot proceed and it enters into a *waiting* state, until the next completed reaction reactivates it. This avoids that reactions which cannot proceed participate to the race for the execution. In SPiM this can be implemented as follows:

```

let Pim()=
?mem(na,c1,nap,clm);
if(na>0.0 * c1>0.0)
then
  (!mem(na,c1,nap,clm)|
  (delay@ k*na*c1;
  ?mem(_na,_c1,_nap,_clm);
  if (_na>0.0 * _c1>0.0)
  then
    ?wait(m);
    (Unlock(m)|
    !mem(_na-1.0,_c1-1.0,_nap+1.0,_clm+1.0)|
    Pim()))
  else
    (!mem(_na,_c1,_nap,_clm)|Pim()))))
else
  (!mem(na,c1,nap,clm)|
  ?wait(n);
  (!wait(n+1);())|
  ?prel;Pim()))

let Unlock(int m)=
if(m>0)
then
  (!prel|Unlock(m-1))
else
  !wait(0)

```

where `Unlock(m)` reactivates the m π -molecules which were in the waiting state.

For the sake of completeness we report also the channel declarations which are necessary in SPiM:

```

new mem:chan(float,float,float,float)
new wait:chan(int)
new prel:chan()

```

We can notice that we only need to declare three instantaneous channels. We conclude with the initial conditions necessary for the system simulation:

```

run( !wait(0)| !mem(<#Na>,<#Cl>,0,0) | Pim())

```

Notice that from the structural point of view our approach is compact and modular. We will see in the next section that a generic system involving m chemical reactants and n reactions is implemented with a memory having m parameters and n π -molecule definitions. Since in our approach we focus more on the chemical reactions than on the reactants, in principle, in our models the complexity should not depend on the reactant quantities. The kinetic behavior

is syntactically (explicitly) declared in the delay rates, which depend also on the reactant concentrations.

There are some critical aspects in our approach which need further investigations. First, we need a deeper understanding on the initial conditions of the system. We have one π -molecule for each reaction. However, one could notice that when the reactant quantities are high it is reasonable to have a “proportional” number of π -molecules. We experimentally noticed that there are differences in the simulations when we change the initial number of π -molecules. We still have to study the relationships between simulations with different initial conditions on the π -molecules. Another related issue is that of the regeneration of the π -molecules. In our approach each π -molecule regenerate itself. There are other possibilities, e.g., the π -molecule P generates the π -molecules P_1, \dots, P_b of the reactions whose inputs include some of the outputs of P . Finally, it is possible to avoid the second memory check for the fired π -molecule, by implementing a booking mechanism. We run some tests on a variant of our approach in which the second memory check is replaced by a booking mechanism. We did not notice substantial differences in the simulations.

We plan to investigate on these considerations on a formal basis, by analyzing the possible semantics equivalences between our approach and other formalisms.

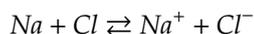
3 Some Biochemical Examples

In this section we present some case studies taken from the SPiM test suite, available at <http://research.microsoft.com/~aphillip/spim/Examples.pdf>.

For each example we show both the results of our simulation and those of the simulation of the model included in the SPiM test suite. These last are based on the approach described in [2–5]. The constant reaction rates used in our simulations are those provided in the corresponding SPiM examples. Notice that, since in our approach we count the number of messages on the memory channel after each reaction, while in the classical approach the number of processes is measured on a given time window, there are some discrepancies in the time scales of our graphs.

3.1 Na-Cl Ionization

We consider again the reaction:



Notice that it is now a reversible reaction.

The simulation results of our approach are presented in Figure 1 on the left, while the simulation results for the classical approach are on the right.

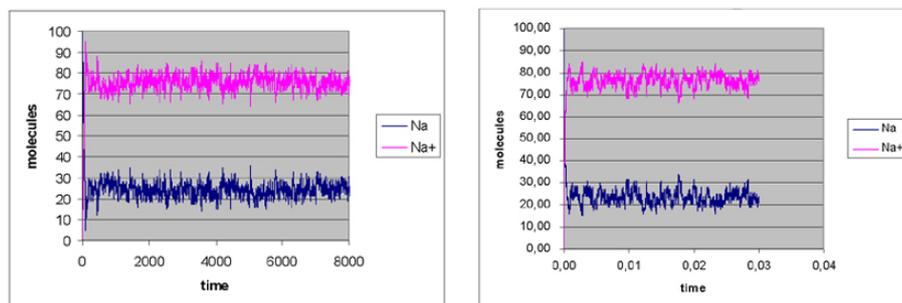
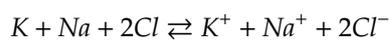


Fig. 1. *Na-Cl* Ionization.

3.2 *K-Na-Cl* Ionization

A slight more sophisticated example is:



The simulation results of our approach are presented in Figure 2 on the left, while the simulation results for the classical approach are on the right.

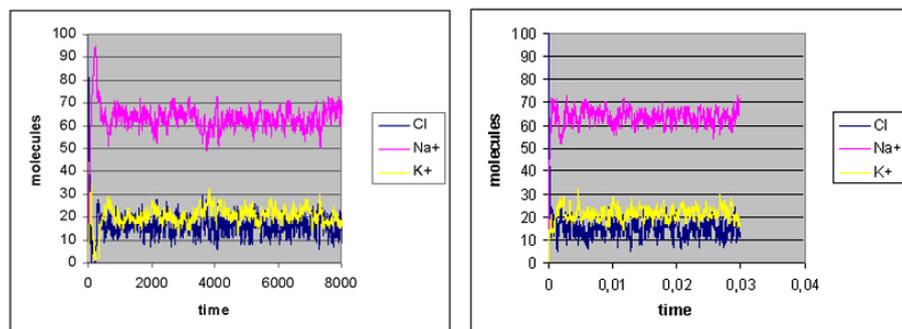
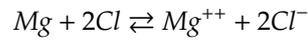


Fig. 2. *K-Na-Cl* Ionization.

3.3 Mg-Cl Ionization

Finally, we consider the reaction:



The simulation results of our approach are presented in Figure 3 on the left, while the simulation results for the classical approach are on the right.

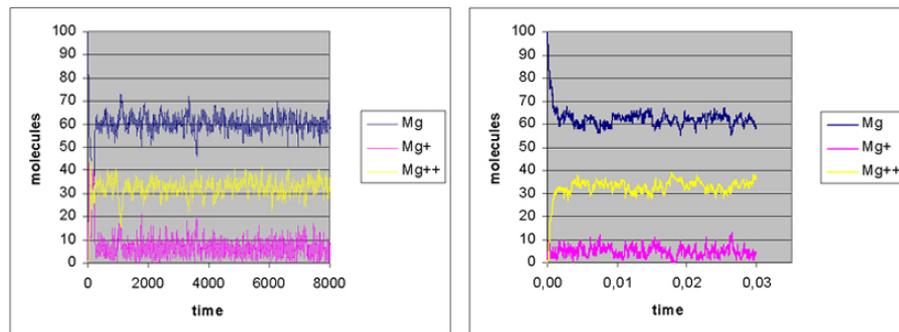


Fig. 3. Mg-Cl Ionization. Two steps simulations.

In this case our approach also allows to write a model for the complete reaction, i.e., without intermediate steps. We report in the next page the code. Notice that it contains also the routine Printmem() we used in all our examples to get the simulation results.

```
(* Mg + 2Cl <==> Mg++ + 2Cl- *)

new mem:chan(float,float,float,float)
new sinc:chan(float,float,float,float,int)
new wait:chan(int)
new prel:chan()
(*Memory contains: Mg, Mg++, Cl, Cl-*)

let
Unlock(m:int)=
  if (m>0)
  then
    (!prel|Unlock(m-1))
  else
    !wait(0)
```

```

let Pimion()=
?mem(mg,mgpp,cl,clm);
if(mg<1.0+cl<2.0)
then
(!mem(mg,mgpp,cl,clm)|
?wait(n);
(!wait(n+1)|?prel;Pimion()))
else
(!mem(mg,mgpp,cl,clm)|
(delay@ 500.0*mg*cl*cl; (*10*100*1/2*mg*cl^2*)
?mem(_mg,_mgpp,_cl,_clm);
if(_mg>0.0*_cl>1.0)
then
?wait(m);
(Unlock(m)|
!mem(_mg-1.0,_mgpp+1.0,_cl-2.0,_clm+2.0)|
!sinc(_mg-1.0,_mgpp+1.0,_cl-2.0,_clm+2.0,0))
else
(!mem(_mg,_mgpp,_cl,_clm)|Pimion()))))

let Pimnoion()=
?mem(mg,mgpp,cl,clm);
if(mgpp<1.0+clm<2.0)
then
(!mem(mg,mgpp,cl,clm)|
?wait(n);
(!wait(n+1)|?prel;Pimnoion()))
else
(!mem(mg,mgpp,cl,clm)|
(delay@ 25.0*5.0*mgpp*clm*(clm); (*5*50*1/2*mg++*cl-^2*)
?mem(_mg,_mgpp,_cl,_clm);
if(_mgpp>0.0*_clm>1.0)
then
?wait(m);
(Unlock(m)|
!mem(_mg+1.0,_mgpp-1.0,_cl+2.0,_clm-2.0)|
!sinc(_mg+1.0,_mgpp-1.0,_cl+2.0,_clm-2.0,1))
else
(!mem(_mg,_mgpp,_cl,_clm)|Pimnoion()))))

let Printmem()=
?sinc(b,d,f,g,e);
print(show b);println(show d);
if(e=0)

```

```

then
  (Pimion() | Printmem())
else
  (Pimnoion() | Printmem())

run (!mem(100.0,0.0,100.0,0.0);() | Printmem() | 1 of Pimion() |
    1 of Pimnoion() | !wait(0))

```

The simulation results of the one step reaction are depicted in Figure 4.

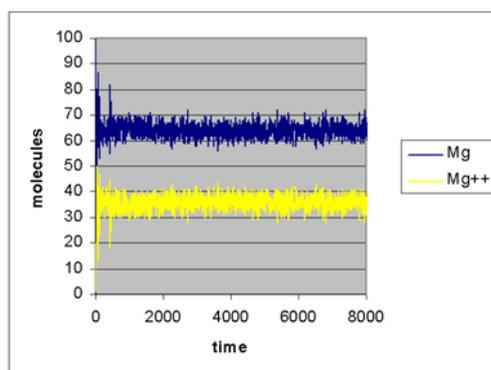


Fig. 4. *Mg-Cl* Ionization. One step simulation.

All the above considered examples show that our simulations are comparable with the ones provided by the classical approach.

4 Conclusions

We described an alternative *reaction centric* use of Stochastic π -calculus.

Our idea is based on the use of passive reactants and active reactions: this choice injects an external point of view inside a typical internal point of view framework, but retains the structure of the language and its available theoretical tools. We got this result by translating chemical reactions in delay transactions, following the chemical meaning of the formers. The pure chemical based models (such as ODEs) are difficult to use in the context of formal analysis (e.g., model checking techniques). Our approach should allows to get round this problem.

As a by product of our approach we notice that reactions involving more than two inputs can be directly simulated.

In the long period our main aim is that of providing a deeper understanding and integration between different formalisms. In particular, we are interested in translations from Hybrid Automata into Stochastic π -calculus processes. In this sense the external point of view seems necessary for the translation of the global constraints (invariant, activations, and resets). Translations from sCCP programs into Hybrid Automata have been considered in [13].

We intend to study possible semantic equivalences between our approach and the classical *reactant centric* use of Stochastic π -calculus.

References

1. Priami, C.: Stochastic pi-Calculus. The Computer Journal **38**(7) (1995) 578–589
2. Priami, C., Regev, A., Shapiro, E.Y., Silverman, W.: Application of a stochastic name-passing calculus to representation and simulation of molecular processes. Information Processing Letters **80**(1) (2001) 25–31
3. Phillips, A., Cardelli, L., Castagna, G.: A Graphical Representation for Biological Processes in the Stochastic pi-Calculus. (2006) 123–152
4. Phillips, A., Cardelli, L.: Efficient, Correct Simulation of Biological Processes in the Stochastic Pi-calculus. In Calder, M., Gilmore, S., eds.: Proc. of Int. Conf. on Computational Methods in Systems Biology (CMSB'07). Volume 4695 of LNCS., Springer (2007) 184–199
5. Cardelli, L.: On process rate semantics. Theoretical Computer Science **391**(3) (2008) 190–215
6. Voit, E.O.: Computational Analysis of Biochemical Systems. A Practical Guide for Biochemists and Molecular Biologists. Cambridge University Press (2000)
7. Bortolussi, L.: Stochastic Concurrent Constraint Programming. Electronic Notes in Theoretical Computer Science **164**(3) (2006) 65–80
8. Bortolussi, L., Policriti, A.: Stochastic Concurrent Constraint Programming and Differential Equations. Electronic Notes in Theoretical Computer Science **190**(3) (2007) 27–42
9. Alur, R., Belta, C., Ivancic, F., Kumar, V., Mintz, M., Pappas, G.J., Rubin, H., Schug, J.: Hybrid Modeling and Simulation of Biomolecular Networks. In: Proc. of Hybrid Systems: Computation and Control (HSCC'01). Volume 2034 of LNCS., Springer (2001) 19–32
10. Ghosh, R., Tiwari, A., Tomlin, C.: Automated Symbolic Reachability Analysis; with Application to Delta-Notch Signaling Automata. In Maler, O., Pnueli, A., eds.: Proc. of Hybrid Systems: Computation and Control (HSCC'03). Volume 2623 of LNCS., Springer (2003) 233–248
11. Casagrande, A., Piazza, C., Mishra, B.: Semi-Algebraic Constant Reset Hybrid Automata - SACoRe. In: Proc. of the 44rd Conference on Decision and Control (CDC'05), IEEE Computer Society Press (2005) 678–683
12. Gillespie, D.: Exact Stochastic Simulation of Coupled Chemical Reactions. Journal of Physical Chemistry **81** (1977) 2340–2361
13. Bortolussi, L., Policriti, A.: Hybrid approximation of Stochastic Concurrent Constraint Programming. In: International Federation of Automatic Control World Congress, (IFAC'08). (2008) To appear.

Folding Transformation Rules for Constraint Logic Programs

Valerio Senni¹, Alberto Pettorossi¹, and Maurizio Proietti²

- (1) DISP, University of Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy
{senni,pettorossi}@disp.uniroma2.it
(2) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
proietti@iasi.rm.cnr.it

Abstract. We consider the folding transformation rule for constraint logic programs. We propose an algorithm for applying the folding rule in the case where the constraints are linear equations and inequations over the rational or the real numbers. Basically, our algorithm consists in reducing a rule application to the solution of one or more systems of linear equations and inequations. We also introduce two variants of the folding transformation rule. The first variant combines the folding rule with the clause splitting rule, and the second variant eliminates the existential variables of a clause, that is, those variables which occur in the body of the clause and not in its head. Finally, we present the algorithms for applying these variants of the folding rule.

1 Introduction

Rule-based program transformation is a program development technique which has been first proposed by Burstall and Darlington in the context of functional programming [4], and then it has been extended to logic programming [18] and to other programming paradigms as well (see [13] for an overview).

In this paper we consider constraint logic programs [8] and the unfold/fold transformation rules presented in [3,5,7,10]. In particular, we focus our investigations on the folding rule, which can be defined (in a declarative style) as follows.

Let (i) H and K be atoms, (ii) c and d be constraints, and (iii) G and B be goals (that is, conjunctions of literals). Given a clause $\gamma: H \leftarrow c \wedge G$ and a clause $\delta: K \leftarrow d \wedge B$, if there exist a constraint e , a substitution ϑ , and a goal R such that $H \leftarrow c \wedge G$ is equivalent (in a given theory of constraints) to $H \leftarrow e \wedge (d \wedge B)\vartheta \wedge R$, then γ is folded into the clause $\eta: H \leftarrow e \wedge K\vartheta \wedge R$.

In the literature, the folding rule is presented with respect to a generic theory of constraints and no algorithm is provided to determine whether or not the suitable e , ϑ , and R needed for applying that rule exist. In this paper we assume that the constraints are linear equations and inequations over the rational numbers (but the techniques we will present are valid without significant changes also when the equations and inequations are over the real numbers), and we

propose an algorithm based on linear algebra and term rewriting techniques for computing e , ϑ , and R , if they exist. For instance, let us consider the clauses:

$$\gamma: p(X, Y) \leftarrow X > 1 \wedge X > T \wedge q([\], T) \wedge r(Y)$$

$$\delta: s(U, V, A) \leftarrow U > 1 \wedge V > 0 \wedge U > W \wedge q(A, W)$$

and suppose that we want to fold γ using δ . Our folding algorithm computes: (i) the constraint $e: X > 1 \wedge X \geq U$, (ii) the substitution $\vartheta: \{A/[\], W/T\}$, and (iii) the goal $R: r(Y)$, and the clause derived by folding γ using δ is:

$$\eta: p(X, Y) \leftarrow X > 1 \wedge X \geq U \wedge s(U, V, [\]) \wedge r(Y)$$

(The correctness of folding will be stated in Section 3. At this point the reader can check it by unfolding the atom $s(U, V, [\])$ in the clause η using δ and, indeed, that unfolding returns clause γ , modulo equivalence of constraints.)

In general, there may be zero or more triples $\langle e, \vartheta, R \rangle$ that satisfy the conditions for the applicability of the folding rule. For this reason, our folding algorithm is nondeterministic and thus, given the clauses γ and δ , it may return, in different computations, different folded clauses.

In this paper we also introduce two variants of the standard folding rule presented above, which are often very useful for transforming programs, and we propose two algorithms for their application.

The first variant of the folding rule corresponds to some applications of the *clause splitting* rule [7] followed by some applications of the standard folding rule. Clause splitting consists in replacing a clause $H \leftarrow c \wedge G$ by two clauses of the form: $H \leftarrow c \wedge d_1 \wedge G$ and $H \leftarrow c \wedge d_2 \wedge G$ such that in the given theory of constraints, the universal closure of $d_1 \vee d_2$ is equivalent to *true*. This variant of the folding rule combines in a single rule application: (i) several applications of the clause splitting rule, by which from clause γ we derive the n clauses $\gamma_1, \dots, \gamma_n$, and (ii) n subsequent applications of the standard folding rule, by which we fold the clauses $\gamma_1, \dots, \gamma_n$ using clause δ and we derive the n clauses η_1, \dots, η_n . In the paper we will also propose an algorithm for determining the suitable applications of the clause splitting rule which allow the subsequent applications of the standard folding rule.

The second variant of the standard folding rule eliminates the *existential variables* which occur in the clause to be folded [14,15]. Recall that the existential variables are those variables which occur in the body of a clause but not in its head.

The paper is structured as follows. In Section 2 we recall some basic definitions concerning constraint logic programs. In Sections 3, 4, and 5 we introduce the standard folding rule [3,5,7,10] and two variants of this rule. We present the three algorithms for applying the folding rule and its two variants. For these algorithms we also prove the soundness and completeness results with respect to the declarative specifications of the folding rules. Finally, in Section 6 we discuss the related work and we indicate some directions for future investigation.

2 Preliminary Definitions

In this section we recall some basic definitions concerning constraint logic programs, where the constraints are conjunctions of linear equations and inequations over the rational numbers. As already mentioned, the results we will present in the following sections, are valid also when the constraints are conjunctions of linear equations and inequations over the real numbers. For notions not defined here the reader may refer to [8,9].

Let us consider a first order language \mathcal{L} given by a set Var of variables, a set Fun of function symbols, and a set $Pred$ of predicate symbols. We assume that the function symbols $+$ and \cdot denoting addition and multiplication, respectively, belong to Fun , and every rational number in \mathbb{Q} is a constant symbol (that is, a 0-ary function symbol) belonging to Fun . We also assume that the predicate symbols \geq , $>$, and $=$ denoting inequality, strict inequality, and equality, respectively, belong to $Pred$.

In order to distinguish terms representing rational numbers from other terms, we assume that \mathcal{L} is a typed language [9]. We have two basic types: **rat**, that is, the type of rational numbers, and **tree**, that is, the type of finite trees. We also consider types constructed from basic types by using the type constructors \times and \rightarrow . A variable $X \in Var$ has either type **rat** or **tree**. We denote by $Var_{\mathbf{rat}}$ and $Var_{\mathbf{tree}}$ the set of variables of type **rat** and **tree**, respectively. A predicate symbol of arity n and a function symbol of arity n in \mathcal{L} have types of the form $\tau_1 \times \dots \times \tau_n$ and $\tau_1 \times \dots \times \tau_n \rightarrow \tau_{n+1}$, respectively, for some types $\tau_1, \dots, \tau_n, \tau_{n+1} \in \{\mathbf{rat}, \mathbf{tree}\}$. In particular, the predicate symbols \geq , $>$, and $=$ have type **rat** \times **rat**, the function symbols $+$ and \cdot have type **rat** \times **rat** \rightarrow **rat**, and the rational numbers have type **rat**. The function symbols in $\{+, \cdot\} \cup \mathbb{Q}$ are the only symbols whose type is $\tau_1 \times \dots \times \tau_n \rightarrow \mathbf{rat}$, for some types τ_1, \dots, τ_n .

A *term of type rat* is a *linear polynomial* and has the following syntax:

$$p ::= a \mid X \mid a \cdot X \mid p_1 + p_2$$

where a is a rational number and X is a variable in $Var_{\mathbf{rat}}$. We will also write the term $a \cdot X$ as aX . A *term of type tree* has the following syntax:

$$t ::= X \mid f(t_1, \dots, t_n)$$

where X is a variable in Var , f is a function symbol of type $\tau_1 \times \dots \times \tau_n \rightarrow \mathbf{tree}$, and t_1, \dots, t_n are terms of type τ_1, \dots, τ_n , respectively. A *term* is either a term of type **rat** or a term of type **tree**.

A *constraint* is a finite conjunction of *atomic constraints*, which are linear equations and inequations over the rational numbers. Constraints have the following syntax:

$$c ::= p_1 \geq p_2 \mid p_1 > p_2 \mid p_1 = p_2 \mid c_1 \wedge \dots \wedge c_n$$

When $n = 0$ we write $c_1 \wedge \dots \wedge c_n$ as *true*. We denote by $LIN_{\mathbb{Q}}$ the set of all constraints.

An *atom* is of the form $r(t_1, \dots, t_n)$, where r is a predicate symbol of type $\tau_1 \times \dots \times \tau_n$ not in $\{\geq, >, =\}$, and t_1, \dots, t_n are terms of type τ_1, \dots, τ_n , respectively. A *literal* is either an atom or a negated atom. An atom is also called a

positive literal and a negated atom is also called a *negative literal*. A *goal* is a conjunction $L_1 \wedge \dots \wedge L_n$ of literals, with $n \geq 0$. Similarly to the case of constraints, we will denote by *true* the conjunction of 0 literals. A *constrained goal* is a conjunction $c \wedge G$ where c is a constraint and G is a goal. A *clause* is of the form $H \leftarrow c \wedge G$, where H is an atom and $c \wedge G$ is a constrained goal. A *constraint logic program* is a set of clauses. A *formula* of the language \mathcal{L} is constructed as usual in first order logic from the symbols of \mathcal{L} by using the logical connectives $\wedge, \vee, \neg, \rightarrow, \leftarrow, \leftrightarrow$, and the quantifiers \exists, \forall .

If e is a term or a formula then by $\text{Vars}_{\text{rat}}(e)$ and $\text{Vars}_{\text{tree}}(e)$ we denote, respectively, the set of variables of type **rat** and of type **tree** occurring in e . By $\text{Vars}(e)$ we denote the set $\text{Vars}_{\text{rat}}(e) \cup \text{Vars}_{\text{tree}}(e)$. By $\text{Vars}(e_1, e_2)$ we denote the set $\text{Vars}(e_1) \cup \text{Vars}(e_2)$. By $F\text{Vars}(e)$ we denote the set of free variables of e . Given a constraint c occurring in the body of a clause γ , a variable which occurs in c and not elsewhere in γ , is said to be a *local variable of c in γ* . Given a clause $\gamma: H \leftarrow c \wedge G$, by $E\text{Vars}(\gamma)$ we denote the set $\text{Vars}(c \wedge G) - \text{Vars}(H)$ of the *existential variables* of γ . Given a set $X = \{X_1, \dots, X_n\}$ of variables and a formula φ , by $\forall X \varphi$ we denote the formula $\forall X_1 \dots \forall X_n \varphi$ and by $\exists X \varphi$ we denote the formula $\exists X_1 \dots \exists X_n \varphi$. By $\forall(\varphi)$ and $\exists(\varphi)$ we denote the *universal closure* and the *existential closure* of φ , respectively.

In the following we will use the notion of substitution as defined in [9] with the following extra condition: for any substitution $\{X_1/t_1, \dots, X_n/t_n\}$, for $i = 1, \dots, n$, the type of X_i is equal to the type of t_i .

Let \mathcal{L}_{rat} denote the sublanguage of \mathcal{L} given by the set Var_{rat} of variables, the set $\{+, \cdot\} \cup \mathbb{Q}$ of function symbols, and the set $\{\geq, >, =\}$ of predicate symbols. The formulas of \mathcal{L}_{rat} are the formulas of \mathcal{L} where all variables, function symbols, and predicate symbols belong to \mathcal{L}_{rat} . The interpretation \mathcal{Q} for \mathcal{L}_{rat} is defined as follows: (i) \mathcal{Q} assigns to the function symbols $+$ and \cdot the usual addition and multiplication operations in \mathbb{Q} , (ii) \mathcal{Q} assigns to every $a \in \mathbb{Q}$ the element a itself, and (iii) \mathcal{Q} assigns to the predicate symbols $\geq, >$, and $=$ the usual inequality, strict inequality, and equality relations on \mathbb{Q} , respectively. For a formula φ of \mathcal{L}_{rat} (in particular, for a constraint), the satisfaction relation $\mathcal{Q} \models \varphi$ is defined as usual in first order logic. Recall that the problem of checking, for any formula φ of \mathcal{L}_{rat} , whether or not $\mathcal{Q} \models \varphi$ holds, is decidable.

By D_{rat} we denote the set \mathbb{Q} of the rational numbers and by D_{tree} we denote the set \mathbb{H} of ground terms constructed from the function symbols in $\text{Fun} - \{+, \cdot\}$. A \mathcal{Q} -*interpretation* is an interpretation I for the typed language \mathcal{L} defined as follows. For $\tau_1, \dots, \tau_n, \tau_{n+1} \in \{\text{rat}, \text{tree}\}$, (i) I assigns to a function symbol of type $\tau_1 \times \dots \times \tau_n \rightarrow \tau_{n+1}$ a function from $D_{\tau_1} \times \dots \times D_{\tau_n}$ to $D_{\tau_{n+1}}$. In particular, (i.1) to any function symbol in $\{+, \cdot\} \cup \mathbb{Q}$, I assigns the same function as \mathcal{Q} , and (i.2) to any function symbol f of type $\tau_1 \times \dots \times \tau_n \rightarrow \text{tree}$, I assigns the function that maps $\langle d_1, \dots, d_n \rangle \in D_{\tau_1} \times \dots \times D_{\tau_n}$ to $f(d_1, \dots, d_n) \in \mathbb{H}$. (ii) I assigns to any predicate symbol of type $\tau_1 \times \dots \times \tau_n$ a relation on $D_{\tau_1} \times \dots \times D_{\tau_n}$. In particular, to the symbols $\geq, >$, and $=$, I assigns the same relation as \mathcal{Q} . For any formula φ of \mathcal{L} (and thus, for a constraint logic program), the satisfaction relation $I \models \varphi$ is defined as usual in typed first order logic. For any \mathcal{Q} -interpretation I and formula

φ of \mathcal{L}_{rat} , we have that $I \models \varphi$ iff $\mathcal{Q} \models \varphi$. We say that a \mathcal{Q} -interpretation I is a \mathcal{Q} -model of a program P if for every clause $\gamma \in P$ we have that $I \models \forall(\gamma)$. Similarly to the case of logic programs, we can define *stratified* constraint logic programs and we can show that every stratified program P has a *perfect* \mathcal{Q} -model [7,8,10], denoted by $M(P)$.

A *solution* of a set C of constraints is a ground substitution σ of the form $\{X_1/a_1, \dots, X_n/a_n\}$, where $\{X_1, \dots, X_n\} = \text{Vars}(C)$ and $a_1, \dots, a_n \in \mathbb{Q}$, such that $\mathcal{Q} \models c\sigma$ for every $c \in C$. We assume that we are given a function *solve* that takes a constraint c in $LIN_{\mathbb{Q}}$ and returns a solution σ of c , if c is satisfiable, and **fail** otherwise. The function *solve* can be implemented by using, for instance, the Fourier-Motzkin algorithm or the Khachiyan algorithm [17].

We assume that we are also given a function *project* such that for every constraint $c, d \in LIN_{\mathbb{Q}}$ and for every finite set of variables $X \subseteq \text{Var}_{\text{rat}}$, if $\text{project}(c, X) = d$ then $\mathcal{Q} \models \forall X ((\exists Y c) \leftrightarrow d)$, where $Y = \text{Vars}(c) - X$ and $\text{Vars}(d) \subseteq X$. Also the *project* function can be implemented by using the Fourier-Motzkin algorithm.

A clause $\gamma: H \leftarrow c \wedge G$ is said to be in *normal form* if (i) the terms of type **rat** occurring in G are distinct existential variables, and (ii) c has no local variables in γ . It is always possible to transform any clause γ into a clause γ_1 in normal form such that γ and γ_1 have the same \mathcal{Q} -models. (In particular, given a clause γ , all local variables of a constraint in γ can be eliminated by applying the *project* function.) The clause γ_1 is called a *normal form* of γ .

Since every clause can be transformed into an equivalent clause in normal form, when presenting the folding rule and the corresponding algorithm for its application we will assume, without loss of generality, that the clauses are in normal form.

Given two clauses γ_1 and γ_2 , we write $\gamma_1 \cong \gamma_2$ if there exist a normal form $H \leftarrow c_1 \wedge B_1$ of γ_1 , a normal form $H \leftarrow c_2 \wedge B_2$ of γ_2 , and a variable renaming ρ such that: (1) $H = H\rho$, (2) $B_1 =_{AC} B_2\rho$, and (3) $\mathcal{Q} \models \forall (c_1 \leftrightarrow c_2\rho)$, where $=_{AC}$ denotes equality modulo the equational theory of associativity and commutativity of conjunction. We refer to this theory as the AC_{\wedge} theory [1].

Proposition 1. (i) \cong is an equivalence relation.

(ii) If $\gamma_1 \cong \gamma_2$ then, for every \mathcal{Q} -interpretation I , $I \models \gamma_1$ iff $I \models \gamma_2$.

(iii) If γ_2 is a normal form of γ_1 then $\gamma_1 \cong \gamma_2$.

(iv) Let γ_1 be $H \leftarrow c \wedge B$ and γ_2 be $H \leftarrow d \wedge B$, then $\gamma_1 \cong \gamma_2$ iff $\mathcal{Q} \models \forall (\exists V_1 c \leftrightarrow \exists V_2 d)$, where V_1 is the set of local variables of c in γ_1 and V_2 is the set of local variables of d in γ_2 .

The \cong equivalence relation can be extended to sets of clauses as follows. Let $\{\gamma_1, \dots, \gamma_m\}$ and $\{\delta_1, \dots, \delta_n\}$ be two sets of clauses. We write $\{\gamma_1, \dots, \gamma_m\} \cong \{\delta_1, \dots, \delta_n\}$ if there exist $H, B_1, B_2, c_1, \dots, c_m, d_1, \dots, d_n$ such that, for $i = 1, \dots, m$, $H \leftarrow c_i \wedge B_1$ is a normal form of γ_i , for $j = 1, \dots, n$, $H \leftarrow d_j \wedge B_2$ is a normal form of δ_j , and there exists a variable renaming ρ such that: (1) $H = H\rho$, (2) $B_1 =_{AC} B_2\rho$, and (3) $\mathcal{Q} \models \forall ((c_1 \vee \dots \vee c_m) \leftrightarrow (d_1 \vee \dots \vee d_n)\rho)$.

3 The Standard Folding Rule

In the literature there are several, slightly different versions of the folding transformation rule for constraint logic programs [3,5,7,10]. These versions differ on the applicability conditions and on the number of clauses that can be folded by a single rule application. In this section we introduce the rule **Fold₁**, whose applicability conditions are the ones given in [7]. However, unlike the folding rule presented in [7], the rule **Fold₁** can be applied to one clause at a time, and in this respect it is similar to the folding rule considered in [3,5]. The general case, where $n(\geq 1)$ clauses can be folded at once, can be addressed as a straightforward generalization of the methods we will present here.

Definition 1 (Rule Fold₁). Let γ and δ be clauses of the form

$$\begin{aligned}\gamma &: H \leftarrow c \wedge G \\ \delta &: K \leftarrow d \wedge B\end{aligned}$$

such that γ and δ are in normal form and without variables in common. Suppose that there exist a constraint e , a substitution ϑ , and a goal R such that:

- (1) $\gamma \cong H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$;
- (2) for every variable X in $EVars(\delta)$, the following conditions hold: (2.i) $X\vartheta$ is a variable not occurring in $\{H, e, R\}$, and (2.ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X .

By *folding clause γ using clause δ* we derive the clause $\eta: H \leftarrow e \wedge K\vartheta \wedge R$.

In Theorem 1 below we establish the correctness of the folding rule **Fold₁** w.r.t. the perfect model semantics. That result follows immediately from [3,5,7].

A *transformation sequence* is a sequence P_0, \dots, P_n of programs such that, for $k = 0, \dots, n-1$, program P_{k+1} is derived from program P_k by an application of one of the following transformation rules: *definition*, *unfolding*, *folding* (Here we refer to the definition rule and the unfolding rule as they are presented in [7]). By $Defs_n$ we denote the set of clauses introduced by applying the definition rule during the construction of P_0, \dots, P_n .

Program P_{k+1} is derived from program P_k by an application of the folding rule **Fold₁** if $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$, where γ is a clause in P_k , δ is a clause in $Defs_k$, and η is the clause derived by folding γ using δ as described in Definition 1.

Theorem 1. [7] *Let P_0 be a stratified program and let P_0, \dots, P_n be a transformation sequence. Suppose that, for $k = 0, \dots, n-1$, if P_{k+1} is derived from P_k by folding clause γ using clause δ in $Defs_k$, then there exists j , with $0 < j < n$, such that $\delta \in P_j$ and program P_{j+1} is derived from P_j by unfolding δ w.r.t. a positive literal in its body. Then $P_0 \cup Defs_n$ and P_n are stratified and $M(P_0 \cup Defs_n) = M(P_n)$.*

Now we will present an algorithm for determining whether or not a clause $\gamma: H \leftarrow c \wedge G$ can be folded using a clause $\delta: K \leftarrow d \wedge B$, according to Definition 1. Our folding algorithm finds, if they exist, a substitution ϑ , a constraint e , and a goal R that satisfy the clause equivalence $H \leftarrow c \wedge G \cong H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$

of Point (1) of Definition 1 with the extra conditions of Point (2) of that same definition. If it is not possible to fold clause γ using clause δ , our algorithm returns **fail**.

Note that since Definition 1 does not determine in a unique way ϑ , e , and R , our folding algorithm is nondeterministic.

Our algorithm can be viewed as a solver of a matching problem [19] and it consists of two procedures which we will present below: (i) the *goal matching procedure*, called **GM**, which matches goals modulo the AC_\wedge theory [1], and (ii) the *constraint matching procedure*, called **CM₁**, which matches constraints modulo the theory of constraints.

Now let us present the goal matching procedure **GM**, which given two clauses $\gamma: H \leftarrow c \wedge G$ and $\delta: K \leftarrow d \wedge B$, either computes, if they exist, two new clauses $\gamma': H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta': K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$ such that G is equal to $B\vartheta_1 \wedge R$ modulo the AC_\wedge theory, for some substitution ϑ_1 and goal R , or returns **fail**, otherwise.

Goal Matching Procedure: GM

Input: two clauses $\gamma: H \leftarrow c \wedge G$ and $\delta: K \leftarrow d \wedge B$ in normal form without variables in common.

Output: (A) two clauses $\gamma': H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta': K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$ in normal form such that: (1) $\gamma \cong \gamma'$ and (2) for every variable X in $EVars(\delta)$, the following conditions hold: (2.i) $X\vartheta_1$ is a variable not occurring in $\{H, R\}$, and (2.ii) $X\vartheta_1$ does not occur in the term $Y\vartheta_1$, for every variable Y occurring in $d \wedge B$ and different from X , if such two clauses γ' and δ' exist, and (B) **fail**, otherwise.

Step 1. Compute an injection μ from the multiset \overline{B} of the literals occurring in B , to the multiset \overline{G} of the literals occurring in G such that: (i) μ is different from any previously computed injection from \overline{B} to \overline{G} , if any, and (ii) for all $L \in \overline{B}$, L and $\mu(L)$ are either both positive or both negative literals, and have the same predicate symbol with the same arity. If there exists no such an injection μ then return **fail**.

Step 2. Consider the set of equations $\overline{S} = \{L = \mu(L) \mid L \in \overline{B}\}$. Transform the set \overline{S} by applying as long as possible the following rewriting rules:

- (i) $\{\neg A_1 = \neg A_2\} \cup \overline{S} \implies \{A_1 = A_2\} \cup \overline{S}$;
- (ii) $\{a(s_1, \dots, s_n) = a(t_1, \dots, t_n)\} \cup \overline{S} \implies \{s_1 = t_1, \dots, s_n = t_n\} \cup \overline{S}$;
- (iii) $\{a(s_1, \dots, s_m) = b(t_1, \dots, t_n)\} \cup \overline{S} \implies \mathbf{fail}$, if a is different from b ;
- (iv) $\{a(s_1, \dots, s_n) = X\} \cup \overline{S} \implies \mathbf{fail}$, if X is a variable;
- (v) $\{X = t_1\} \cup \overline{S} \implies \mathbf{fail}$, if X is a variable and in the current set \overline{S} of equations there is an equation $X = t_2$ such that t_1 is different from t_2 .

Step 3. Let R be the conjunction of all literals in $\overline{G} - \{\mu(L) \mid L \in \overline{B}\}$ and S be the final set of equations (possibly, **fail**) which is derived at the end of Step 2.

IF S is not **fail** and for every equation $X = t$ in S such that $X \in EVars(\delta)$ we have that: (i) the term t is a variable not occurring in $\{H, R\}$ and (ii) there is no $Y \in Vars(d \wedge B)$ different from X such that $Y = r$ is an equation in

S and $t \in \text{Vars}(r)$ THEN return the clauses $\gamma' : H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta' : K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$, where ϑ_1 is the substitution $\{X/t \mid X = t \in S\} \cup \{V_1/s_1, \dots, V_\ell/s_\ell\}$, where $\{V_1, \dots, V_\ell\} = \text{Vars}_{\text{tree}}(K) - \text{Vars}(B)$ and s_1, \dots, s_ℓ are arbitrary ground terms of type **tree** ELSE go to Step 1.

Note that the goal matching procedure **GM** is nondeterministic because, in general, more than one injection μ may lead to the construction of a matching substitution ϑ_1 and different mappings μ may determine different matching substitutions.

Moreover, the goal matching procedure **GM** has a ‘generate-and-test’ structure in the sense that at Step 1 it looks for an injection μ and then at Step 3 it tests whether or not that injection μ allows the construction of the matching substitution ϑ_1 . It could be desirable to avoid the inefficiency due to that ‘generate-and-test’ structure, but one should recall that the problem of matching modulo the equational theory AC_\wedge has been shown to be NP-complete [1,2].

Note also that, since the input clauses γ and δ are in normal form, the substitution ϑ_1 computed by the goal matching procedure **GM** is a renaming substitution when restricted to the variables of type **rat**.

Finally, note that the arbitrary ground substitution $\{V_1/s_1, \dots, V_\ell/s_\ell\}$ for the variables of $\text{Vars}_{\text{tree}}(K) - \text{Vars}(B)$ can be chosen before executing the **GM** procedure, so that **GM** does *not* compute distinct substitutions ϑ_1 differing only on the choice of $\{V_1/s_1, \dots, V_\ell/s_\ell\}$.

The following theorem states that the goal matching procedure **GM** is sound and complete, that is, **GM** finds a suitable matching substitution ϑ_1 and a goal R if and only if they exist.

Theorem 2 (Termination, Soundness, and Completeness of the Goal Matching Procedure). *Let $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$ be two clauses in normal form and without variables in common, given in input to the goal matching procedure **GM**. Then the following properties hold:*

- (i) **GM** terminates;
- (ii) *If the output of **GM** is the pair of clauses $\gamma' : H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta' : K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$, then*
 1. γ' and δ' are in normal form,
 2. $\gamma \cong \gamma'$, and
 3. the substitution ϑ_1 is such that, for every variable X in $E\text{Vars}(\delta)$, the following conditions hold: (i) $X\vartheta_1$ is a variable not occurring in $\{H, R\}$, and (ii) $X\vartheta_1$ does not occur in the term $Y\vartheta_1$, for every variable Y occurring in $d \wedge B$ and different from X ;

(iii) *If there exist a substitution ϑ_1 and a goal R such that:*

1. $\gamma \cong H \leftarrow c \wedge B\vartheta_1 \wedge R$, and
2. for every variable X in $E\text{Vars}(\delta)$, the following conditions hold: (i) $X\vartheta_1$ is a variable not occurring in $\{H, R\}$, and (ii) $X\vartheta_1$ does not occur in the term $Y\vartheta_1$, for every variable Y occurring in $d \wedge B$ and different from X ,

*then the output of **GM** is not fail.*

Now we present the constraint matching procedure \mathbf{CM}_1 which takes in input the two clauses γ' and δ' which are produced in output by the goal matching procedure. When presenting the procedure \mathbf{CM}_1 we will assume that the clauses γ' and δ' are of the following form:

$$\begin{aligned}\gamma' &: H \leftarrow c \wedge B' \wedge R \\ \delta' &: K' \leftarrow d' \wedge B'.\end{aligned}$$

The constraint matching procedure \mathbf{CM}_1 returns either a clause $\gamma'': H \leftarrow e \wedge d' \wedge B' \wedge R$ such that $\gamma' \cong \gamma''$ and $EVars(\delta') \cap Vars(e) = \emptyset$ (see Condition 2.i of Definition 1), or **fail**.

Note that, by Point (iv) of Proposition 1, the equivalence $\gamma' \cong \gamma''$ holds iff $\mathcal{Q} \models \forall(\exists W(e \wedge d') \leftrightarrow c)$, where W is the set of the local variables of $e \wedge d'$ in γ'' . As a consequence of this fact and the fact that, without loss of generality, we may assume that the constraint e has no local variables in γ'' , one can show that $Vars(e) = Vars(c, d') - (Vars(c) \cap Vars(d'))$.

Now let us consider the formula \hat{e} defined as follows.

Definition 2. Let γ' and δ' be the output clauses of the goal matching procedure \mathbf{GM} . By \hat{e} we denote the formula $\forall Z(d' \rightarrow c)$, where: (i) c and d' are the constraints occurring in the clauses γ' and δ' , respectively, and (ii) Z is the set of variables $Vars(c) \cap Vars(d')$.

From Definition 2, we immediately get that $FVars(\hat{e}) = Vars(c, d') - (Vars(c) \cap Vars(d'))$. In the following Lemma 1 we show that, among all formulas e in $\mathcal{L}_{\mathbf{rat}}$ such that $\mathcal{Q} \models \forall(\exists W(e \wedge d') \leftrightarrow c)$, where W is the set of the free variables of $e \wedge d'$ not occurring in $\{H, B' \wedge R\}$, the formula \hat{e} is the most general one in the sense that $\mathcal{Q} \models \forall(e \rightarrow \hat{e})$.

Lemma 1. Let $\gamma': H \leftarrow c \wedge B' \wedge R$ and $\delta': K' \leftarrow d' \wedge B'$ be the two clauses in normal form which are the output clauses of the goal matching procedure \mathbf{GM} . Let Y be $FVars(\hat{e} \wedge d') - Vars(H, B' \wedge R)$. The following properties hold:

1. $\mathcal{Q} \models \forall((\exists Y(\hat{e} \wedge d')) \rightarrow c)$;
2. For all formulas e in $\mathcal{L}_{\mathbf{rat}}$ such that $FVars(e) = FVars(\hat{e})$,
if $\mathcal{Q} \models \forall((\exists Y(e \wedge d')) \rightarrow c)$ then $\mathcal{Q} \models \forall(e \rightarrow \hat{e})$ and
if $\mathcal{Q} \models \forall((\exists Y(e \wedge d')) \leftrightarrow c)$ then $\mathcal{Q} \models \forall((\exists Y(\hat{e} \wedge d')) \leftrightarrow c)$.

Note that, in general, the formula \hat{e} is not a constraint and, therefore, the constraint matching procedure cannot return $H \leftarrow \hat{e} \wedge d' \wedge B' \wedge R$. Thus, starting from the formula \hat{e} , we need to construct a constraint e such that $FVars(e) = FVars(\hat{e})$ and $\mathcal{Q} \models \forall(\exists Y(e \wedge d') \leftrightarrow \exists Y(\hat{e} \wedge d'))$ where $Y = FVars(\hat{e} \wedge d') - Vars(H, B' \wedge R)$. Note also that it is always possible to rewrite the formula \hat{e} into a finite disjunction $e_1 \vee \dots \vee e_m$ of constraints such that $\mathcal{Q} \models \forall(\hat{e} \leftrightarrow (e_1 \vee \dots \vee e_m))$ and $FVars(\hat{e}) = Vars(e_1 \vee \dots \vee e_m)$. Thus, we have the following property.

Lemma 2. Let \hat{e} be the formula introduced in Definition 2 and let e_1, \dots, e_m be constraints such that $\mathcal{Q} \models \forall(\hat{e} \leftrightarrow (e_1 \vee \dots \vee e_m))$. Let Y be $FVars(\hat{e} \wedge d') - Vars(H, B' \wedge R)$. Then for $i = 1, \dots, m$, we have that $\mathcal{Q} \models \forall((\exists Y(e_i \wedge d')) \rightarrow c)$.

Now we present the constraint matching procedure **CM₁** which is a non-deterministic procedure whose soundness follows from Proposition 1 and from Lemma 2. At Step 1 of the **CM₁** procedure the formula \widehat{e} is replaced by an equivalent finite disjunction $e_1 \vee \dots \vee e_n$ of constraints. At Step 2, the procedure tests whether or not there exists a constraint e_i such that $\mathcal{Q} \models \forall(c \rightarrow \exists Y (e_i \wedge d'))$ and, thus, by Lemma 2, it can be taken to be the constraint e we want to construct. Note that this technique for constructing e is very simple but, unfortunately, it leads to the incompleteness of the constraint matching procedure **CM₁** because, in general, there may be a constraint e such that $\mathcal{Q} \models \forall(c \leftrightarrow \exists Y (e \wedge d'))$, and yet, it does not exist e_i in $\{e_1, \dots, e_m\}$ such that $\mathcal{Q} \models \forall(c \rightarrow \exists Y (e_i \wedge d'))$.

Constraint Matching Procedure: CM₁

Input: two clauses $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\delta' : K' \leftarrow d' \wedge B'$ in normal form. Suppose that they are an output of the goal matching procedure **GM**.

Output: either a clause $\gamma'' : H \leftarrow e \wedge d' \wedge B' \wedge R$ such that $\gamma' \cong \gamma''$ and $EVars(\delta') \cap Vars(e) = \emptyset$, or **fail**.

Step 1. Transform the formula $\widehat{e} : \forall Z (d' \rightarrow c)$, where $Z = Vars(c) \cap Vars(d')$, into a finite disjunction of constraints as follows:

- 1.1 Transform the formula into $\neg \exists Z \neg (d' \rightarrow c)$.
- 1.2 Eliminate the negation from the formula $\neg (d' \rightarrow c)$ by eliminating \rightarrow in favor of \vee and \neg , by pushing \neg inward as much as possible, and by replacing $\neg(p_1 \geq p_2)$ by $p_2 > p_1$, $\neg(p_1 > p_2)$ by $p_2 \geq p_1$, and $\neg(p_1 = p_2)$ by $p_1 > p_2 \vee p_2 > p_1$. A formula $f_1 \vee \dots \vee f_k$, where f_1, \dots, f_k are constraints, is obtained.
- 1.3 Distribute $\exists Z$ over \vee , eliminate the existential variables of the set Z of variables from each disjunct f_i using the function *project*, and obtain the formula $g_1 \vee \dots \vee g_k$, where for $i = 1, \dots, k$, the disjunct g_i is the constraint *project*($f_i, Vars(f_i) - Z$).
- 1.4 Eliminate the negation from the formula $\neg (g_1 \vee \dots \vee g_k)$ by pushing \neg inward, and obtain a formula $e_1 \vee \dots \vee e_m$, where e_1, \dots, e_m are constraints.

Step 2. IF there exists e_i in $\{e_1, \dots, e_m\}$ such that $\mathcal{Q} \models \forall(c \rightarrow \exists Y (e_i \wedge d'))$, where Y is the set $FVars(e_i \wedge d') - Vars(H, B' \wedge R)$ THEN return $\gamma'' : H \leftarrow e_i \wedge d' \wedge B' \wedge R$ ELSE return **fail**.

In the following theorem we prove that the constraint matching procedure **CM₁** terminates and returns a correct output in the sense that if **CM₁** constructs a constraint e then the clauses $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\gamma'' : H \leftarrow e \wedge d' \wedge B' \wedge R$ are equivalent.

Theorem 3 (Termination and Soundness of the Constraint Matching Procedure CM₁). *Let the two clauses $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\delta' : K' \leftarrow d' \wedge B'$ in normal form be the input of the constraint matching procedure **CM₁**. Suppose that γ' and δ' are an output of **GM**. Then: (i) **CM₁** terminates, and (ii) if **CM₁** returns the clause $\gamma'' : H \leftarrow e \wedge d' \wedge B' \wedge R$, then $\gamma' \cong \gamma''$ and $EVars(\delta') \cap Vars(e) = \emptyset$.*

Now we introduce the folding algorithm \mathbf{FA}_1 , which makes use of the goal matching procedure \mathbf{GM} and the constraint matching procedure \mathbf{CM}_1 . In order to fold a clause γ using a clause δ , the folding algorithm \mathbf{FA}_1 : (i) applies the goal matching procedure \mathbf{GM} and computes a matching substitution ϑ and a goal R , and then (ii) it applies the constraint matching procedure \mathbf{CM}_1 and looks for a constraint e such that Conditions (1) and (2) of Definition 1 are satisfied. If \mathbf{CM}_1 fails to compute the desired constraint e , the folding algorithm executes again the \mathbf{GM} procedure so that new, alternative outputs ϑ and R are generated, until the \mathbf{CM}_1 procedure computes the desired constraint e . The folding algorithm \mathbf{FA}_1 returns **fail** whenever \mathbf{GM} fails to compute a substitution ϑ and a goal R , and whenever, for all ϑ and R computed by \mathbf{GM} , the \mathbf{CM}_1 procedure fails to compute a constraint e such that Conditions (1) and (2) of Definition 1 are satisfied.

Folding Algorithm: \mathbf{FA}_1

Input: two clauses $\gamma: H \leftarrow c \wedge G$ and $\delta: K \leftarrow d \wedge B$ in normal form and without variables in common.

Output: either a clause $\eta: H \leftarrow e \wedge K\vartheta \wedge R$ such that η is the result of folding γ using δ according to Definition 1, or **fail**.

IF there exist two clauses of the form $\gamma': H \leftarrow c \wedge B\vartheta \wedge R$ and $\delta': K\vartheta \leftarrow d\vartheta \wedge B\vartheta$, which are the output of an execution of the \mathbf{GM} procedure when the clauses γ and δ are given in input

AND there exists a clause $\gamma'': H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$ which is the output of an execution of the \mathbf{CM}_1 procedure when the clauses γ' and δ' are given in input

THEN return the clause $\eta: H \leftarrow e \wedge K\vartheta \wedge R$

ELSE return **fail**.

By using Theorems 2 and 3 we can prove termination and soundness of the folding algorithm \mathbf{FA}_1 .

Theorem 4 (Termination and Soundness of the Folding Algorithm \mathbf{FA}_1). *Let γ and δ two clauses in normal form and without variables in common, which are the input of Algorithm \mathbf{FA}_1 . Then: (i) \mathbf{FA}_1 terminates, and (ii) if the output of \mathbf{FA}_1 is a clause η , then η can be derived by folding γ using δ according to Definition 1.*

Let us now illustrate the folding algorithm \mathbf{FA}_1 by considering the example presented in the Introduction. The clauses γ and δ are already in normal form and do not have variable in common. Let p be a predicate symbol of type $\mathbf{rat} \times \mathbf{tree}$, s be of type $\mathbf{rat} \times \mathbf{rat} \times \mathbf{tree}$, q be of type $\mathbf{tree} \times \mathbf{rat}$, and r be of type \mathbf{tree} . If we apply the goal matching procedure \mathbf{GM} to the clauses γ and δ we get the substitution $\vartheta: \{A/[], W/T\}$ and the goal $R: r(Y)$. Thus, we get the following clauses:

$$\gamma': p(X, Y) \leftarrow X > 1 \wedge X > T \wedge q([], T) \wedge r(Y)$$

$$\delta': s(U, V, []) \leftarrow U > 1 \wedge V > 0 \wedge U > T \wedge q([], T)$$

Then these clauses are given in input to the constraint matching procedure \mathbf{CM}_1 . Since in that procedure we have assumed that γ' is of the form: $H \leftarrow c \wedge B' \wedge R$ and δ' is of the form: $K' \leftarrow d' \wedge B'$, we have that:

$$c \text{ is } X > 1 \wedge X > T \quad \text{and} \quad d' \text{ is } U > 1 \wedge V > 0 \wedge U > T.$$

We also have that $\text{Vars}(c) \cap \text{Vars}(d') = \{T\}$ and thus, we have to perform Step 1 starting from the formula $\forall T(d' \rightarrow c)$. At the end of Step 1 we obtain the following new formula:

$$(1 \geq U) \vee (0 \geq V) \vee (X > 1 \wedge X \geq U)$$

which is the disjunction of the following three constraints: $e_1 \equiv 1 \geq U$, $e_2 \equiv 0 \geq V$, and $e_3 \equiv X > 1 \wedge X \geq U$. Then, at Step 2 we get that:

- (i) $\mathcal{Q} \not\models \forall(c \rightarrow \exists U \exists V (1 \geq U \wedge d'))$,
- (ii) $\mathcal{Q} \not\models \forall(c \rightarrow \exists U \exists V (0 \geq V \wedge d'))$, and
- (iii) $\mathcal{Q} \models \forall(c \rightarrow \exists U \exists V (X > 1 \wedge X \geq U \wedge d'))$.

Thus, the constraint matching procedure \mathbf{CM}_1 returns the clause:

$$\gamma'': \quad p(X, Y) \leftarrow X > 1 \wedge X \geq U \wedge d' \wedge q(T) \wedge r(Y)$$

which after folding, that is, after the replacement of $d' \wedge q(T)$ by $s(U, V)$, becomes, as indicated in the Introduction:

$$\eta: \quad p(X, Y) \leftarrow X > 1 \wedge X \geq U \wedge s(U, V) \wedge r(Y).$$

Due to the the fact that in general the constraint matching procedure \mathbf{CM}_1 is not complete, the folding algorithm \mathbf{FA}_1 is not complete either. Indeed, for some input clauses γ and δ there exists a triple $\langle e, \vartheta, R \rangle$ such that Conditions (1) and (2) of Definition 1 are satisfied, and yet \mathbf{FA}_1 returns **fail**.

In the next two sections, we will present two more folding algorithms for applying the standard folding rule as introduced in Definition 1, and we will provide the conditions under which they are complete. In particular, the first algorithm can be used when there is the extra requirement that no existential variable should be present in the folded clause, while the second algorithm can be used when, before applying the folding rule, preliminary applications of the clause splitting rule are possible.

4 A Folding Rule Combined with Clause Splitting

In this section we present a rule, called \mathbf{Fold}_2 , which is an extension of the folding rule \mathbf{Fold}_1 presented in Section 3. An application of the rule \mathbf{Fold}_2 is equivalent to zero or more applications of the clause splitting rule [6] followed by some applications of the folding rule \mathbf{Fold}_1 .

Let us motivate the introduction of the rule \mathbf{Fold}_2 by means of an example. Let us consider the following two clauses:

$$\begin{aligned} \gamma: \quad & p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge 2 \geq Z \wedge q(Z) \\ \delta: \quad & s(Y) \leftarrow W + \frac{3}{4} \geq Y \wedge Y \geq W - \frac{3}{4} \wedge 4 - W \geq Y \wedge Y \geq 2 - W \wedge q(W) \end{aligned}$$

where the predicate symbols p , q , and s are of type **rat**. It is not possible to apply the rule \mathbf{Fold}_1 and fold clause γ using δ , because no constrained subgoal

of clause γ is an instance of the body of clause δ . In order to allow folding, we can use the clause splitting transformation rule. In particular, given the property $\mathcal{Q} \models \forall Z (Z \geq \frac{5}{4} \vee \frac{7}{4} \geq Z)$, we can split clause γ into two clauses:

$$\begin{aligned}\gamma_1: & p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge 2 \geq Z \wedge Z \geq \frac{5}{4} \wedge q(Z) \\ \gamma_2: & p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Z \geq 1 \wedge 2 \geq Z \wedge \frac{7}{4} \geq Z \wedge q(Z)\end{aligned}$$

Now it is possible to fold both clause γ_1 and clause γ_2 using δ .

In order to fold γ_1 using δ (by applying the rule **Fold₁**), we introduce: (i) the constraint $e_1: 2 \geq X \wedge X \geq 1 \wedge 1 \geq Y$, (ii) the substitution $\vartheta: \{W/Z\}$, and (iii) the empty conjunction R . In order to fold γ_2 using δ we introduce: (i) the constraint $e_2: 2 \geq X \wedge X \geq 1 \wedge Y \geq 2$, (ii) the substitution $\vartheta: \{W/Z\}$, and (iii) the empty conjunction R . The clauses η_1 and η_2 derived by folding γ_1 and γ_2 using δ are:

$$\begin{aligned}\eta_1: & p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge 1 \geq Y \wedge s(Y) \\ \eta_2: & p(X) \leftarrow 2 \geq X \wedge X \geq 1 \wedge Y \geq 2 \wedge s(Y)\end{aligned}$$

Note that in general it may be difficult to find a suitable formula (such as $\forall Z (Z \geq \frac{5}{4} \vee \frac{7}{4} \geq Z)$ in our example above) which after clause splitting will allow folding.

Definition 3 below introduces the folding rule **Fold₂** by which a clause $\gamma: H \leftarrow c \wedge G$ can be folded using a clause $\delta: K \leftarrow d \wedge B$ if: (i) there exist n constraints c_1, \dots, c_n such that $\mathcal{Q} \models \forall (c \leftrightarrow c_1 \vee \dots \vee c_n)$ holds, and (ii) for $i = 1, \dots, n$, by applying the folding rule **Fold₁**, the clause $H \leftarrow c_i \wedge G$ can be folded using δ , thereby deriving $H \leftarrow e_i \wedge K \vartheta \wedge R$. Note that in Definition 3 the constraints c_1, \dots, c_n are not mentioned, and we will only require to find the clauses η_1, \dots, η_n , or equivalently, the constraints e_1, \dots, e_n , the substitution ϑ , and the goal R . Indeed, in the algorithm **FA₂** for applying the rule **Fold₂** we will introduce below, we directly compute, if they exist, the clauses η_1, \dots, η_n , without finding the constraints c_1, \dots, c_n .

Definition 3 (Rule Fold₂). Let γ and δ be clauses of the form

$$\begin{aligned}\gamma: & H \leftarrow c \wedge G \\ \delta: & K \leftarrow d \wedge B\end{aligned}$$

such that γ and δ are in normal form and without variables in common. Suppose that there exist some constraints e_1, \dots, e_n , a substitution ϑ , and a goal R such that:

- (1) $\{\gamma\} \cong \{H \leftarrow e_1 \wedge d \vartheta \wedge B \vartheta \wedge R, \dots, H \leftarrow e_n \wedge d \vartheta \wedge B \vartheta \wedge R\}$;
- (2) for every variable X in $EVars(\delta)$, the following conditions hold: (i) $X \vartheta$ is a variable not occurring in $\{H, e_1, \dots, e_n, R\}$, and (ii) $X \vartheta$ does not occur in the term $Y \vartheta$, for every variable Y occurring in $d \wedge B$ and different from X .

By *folding clause γ using clause δ* we derive the clauses $\eta_1: H \leftarrow e_1 \wedge K \vartheta \wedge R, \dots, \eta_n: H \leftarrow e_n \wedge K \vartheta \wedge R$.

Now we provide the algorithm **FA₂** for applying rule **Fold₂**. Given two input clauses γ and δ , the output of the algorithm is: (i) a set of clauses derived by folding γ using δ as specified in Definition 3, if it is possible to fold, and (ii) **fail**, otherwise. The **FA₂** algorithm looks for suitable constraints e_1, \dots, e_n , substitution ϑ , and goal R such that Conditions (1) and (2) of Definition 3 hold.

Similarly to Algorithm **FA**₁ presented in Section 3, Algorithm **FA**₂ makes use of two procedures: the goal matching procedure **GM**, which is the one used in **FA**₁, and the constraint matching procedure **CM**₂, which is the following variant of **CM**₁.

Constraint Matching Procedure: CM₂

Input: two clauses $\gamma' : H \leftarrow c \wedge B' \wedge R$ and $\delta' : K' \leftarrow d' \wedge B'$ in normal form. Suppose that they are an output of the goal matching procedure **GM**.

Output: (A) a set $\{\gamma''_1, \dots, \gamma''_n\}$ of clauses, with $n \geq 0$, such that: (i) for $i = 1, \dots, n$, γ''_i is of the form $H \leftarrow e_i \wedge d' \wedge B' \wedge R$, (ii) $\{\gamma'\} \cong \{\gamma''_1, \dots, \gamma''_n\}$, and (iii) $EVars(\delta') \cap Vars(e_1, \dots, e_n) = \emptyset$, if such set exists, and (B) **fail**, otherwise.

Step 1. Starting from $\hat{e} = \forall Z(d' \rightarrow c)$ compute the disjunction $h_1 \vee \dots \vee h_m$ of constraints in the same way as it has been done at Step 1 of the procedure **CM**₁ for computing $e_1 \vee \dots \vee e_m$ starting from $\hat{e} = \forall Z(d' \rightarrow c)$.

Step 2. Let $\{e_1, \dots, e_n\}$, with $n \leq m$, be the set of those h_i , with $1 \leq i \leq m$, such that $\mathcal{Q} \models \exists(h_i \wedge d')$. IF $\mathcal{Q} \models \forall(c \rightarrow \exists Y((e_1 \vee \dots \vee e_n) \wedge d'))$, where $Y = FVars((e_1 \vee \dots \vee e_n) \wedge d') - Vars(H, B \wedge R)$ THEN return the set $\{H \leftarrow e_i \wedge d' \wedge B' \wedge R \mid i = 1, \dots, n\}$ of clauses ELSE return **fail**.

It can be shown that the constraint matching procedure **CM**₂ always terminates and it is sound and complete with respect to its specification. Thus, we slightly modify the folding algorithm **FA**₁ by using **CM**₂, instead of **CM**₁, and we get a terminating, sound, complete folding algorithm **FA**₂ to compute the result of applying rule **Fold**₂ to clause γ using clause δ .

Folding Algorithm: FA₂

Input: two clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$ in normal form and with no variables in common.

Output: n clauses $\eta_1 : H \leftarrow e_1 \wedge K\vartheta \wedge R, \dots, \eta_n : H \leftarrow e_n \wedge K\vartheta \wedge R$, with $n \geq 0$, if it is possible to fold γ using δ according to Definition 3, and **fail**, otherwise.

IF c is unsatisfiable, that is, $\mathcal{Q} \models \neg\exists(c)$ THEN return the empty set of clauses

ELSE IF there exist two clauses of the form $\gamma' : H \leftarrow c \wedge B\vartheta \wedge R$ and $\delta' : K\vartheta \leftarrow d\vartheta \wedge B\vartheta$, which are the output of an execution of the **GM** procedure when given in input clauses γ and δ

AND there exists a set $\{H \leftarrow e_1 \wedge d\vartheta \wedge B\vartheta \wedge R, \dots, H \leftarrow e_n \wedge d\vartheta \wedge B\vartheta \wedge R\}$ of clauses which is the output of an execution of the **CM**₂ procedure when given in input clauses γ' and δ'

THEN return the clauses $\eta_1 : H \leftarrow e_1 \wedge K\vartheta \wedge R, \dots, \eta_n : H \leftarrow e_n \wedge K\vartheta \wedge R$ ELSE return **fail**.

We leave it to the reader to check that the clauses η_1 and η_2 considered in the example at the beginning of this section can be computed by applying the folding algorithm **FA**₂ with clauses γ and δ as input.

Theorem 5 (Termination, Soundness, and Completeness of the Folding Algorithm \mathbf{FA}_2). *Let two clauses γ and δ , in normal form and with no variables in common, be the input of Algorithm \mathbf{FA}_2 . Then:*

- (i) \mathbf{FA}_2 terminates,
- (ii) if \mathbf{FA}_2 returns the clauses η_1, \dots, η_n , then η_1, \dots, η_n can be derived by folding γ using δ according to Definition 3, and
- (iii) if it is possible to fold γ using δ according to Definition 3, then \mathbf{FA}_2 does not return **fail**.

5 A Folding Rule for the Elimination of Existential Variables

In this section we introduce a folding rule, called \mathbf{Fold}_3 , which is a variant of the rule \mathbf{Fold}_1 . When we apply the rule \mathbf{Fold}_3 to the clauses $\gamma: H \leftarrow c \wedge G$ and $\delta: K \leftarrow d \wedge B$, we replace a subgoal of $c \wedge G$, where some existential variables may occur, by an atom $K\vartheta$ thereby getting a new clause η in which $K\vartheta$ has no existential variables.

Definition 4 (Rule \mathbf{Fold}_3). Let γ and δ be clauses of the form

$$\begin{aligned}\gamma: & H \leftarrow c \wedge G \\ \delta: & K \leftarrow d \wedge B\end{aligned}$$

such that γ and δ are in normal form and without variables in common. Suppose that there exist a constraint e , a substitution ϑ , and goal R such that:

- (1) $\gamma \cong H \leftarrow e \wedge d\vartheta \wedge B\vartheta \wedge R$;
- (2) for every variable X in $EVars(\delta)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, e, R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for every variable Y occurring in $d \wedge B$ and different from X ;
- (3) $Vars(K\vartheta) \subseteq Vars(H)$.

By *folding clause γ using clause δ* we derive the clause $\eta: H \leftarrow e \wedge K\vartheta \wedge R$.

Condition (3) ensures that no existential variable in the body of η occurs in $K\vartheta$. However, it may happen that in the folded clause η there are still some existential variables occurring in e or R , which could be eliminated by further folding steps using clause δ again or using other clauses.

Now we will present an algorithm, called \mathbf{FA}_3 , for applying rule \mathbf{Fold}_3 . The \mathbf{FA}_3 algorithm is a variant of the \mathbf{FA}_1 algorithm for applying rule \mathbf{Fold}_1 presented in Section 3. Given two clauses γ and δ , the \mathbf{FA}_3 algorithm returns a clause obtained by folding, if folding is possible, and it returns **fail**, otherwise. The \mathbf{FA}_3 algorithm makes use of: (i) the goal matching procedure \mathbf{GM} presented in Section 3 with the following additional rewriting rule:

$$(vi) \{X=t\} \cup \bar{S} \implies \mathbf{fail} \quad \text{if } X \in Vars_{\text{tree}}(K) \text{ and } Vars(t) \not\subseteq Vars_{\text{tree}}(H)$$

and (ii) a constraint matching procedure, called \mathbf{CM}_3 , we will present below.

The rewrite rule (vi) ensures that $Vars_{\text{tree}}(K\vartheta) \subseteq Vars_{\text{tree}}(H)$ and this fact will be used in the proof of Theorem 7 below for showing that Point (3) of Definition 4 holds and, thus, existential variables are eliminated.

The \mathbf{CM}_3 procedure takes as input the two clauses $\gamma': H \leftarrow c \wedge B' \wedge R$ and $\delta': K' \leftarrow d' \wedge B'$, which are the output of the goal matching procedure. If there exist a constraint e and a substitution ϑ_2 such that: (i) $\gamma' \cong H \leftarrow e \wedge d' \vartheta_2 \wedge B' \wedge R$, (ii) $B' \vartheta_2 = B'$, (iii) $\text{Vars}(K' \vartheta_2) \subseteq \text{Vars}(H)$, and (iv) $\text{Vars}(e) \subseteq \text{Vars}(H, R)$, then the constraint matching procedure \mathbf{CM}_3 returns a clause $\gamma'': H \leftarrow e \wedge d' \vartheta_2 \wedge B' \vartheta_2 \wedge R$, else it returns **fail**.

First we show (see Lemma 3) that if there exists a constraint e satisfying Points (i) and (iv) of the above paragraph, then we can always take such constraint to be $\text{project}(c, X)$, where $X = \text{Vars}(c) - \text{Vars}(B')$ (see Section 2 for the definition of the project function).

Lemma 3. *Let $\gamma': H \leftarrow c \wedge B' \wedge R$ and $\delta': K' \leftarrow d' \wedge B'$ be the two clauses in normal form which are the output clauses of the goal matching procedure. Let \tilde{e} denote the constraint $\text{project}(c, X)$, where $X = \text{Vars}(c) - \text{Vars}(B')$. There exist a constraint e and a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow \exists W(e \wedge d' \vartheta_2))$, where $W = \text{Vars}(e \wedge d' \vartheta_2) - \text{Vars}(H, B' \wedge R)$, iff $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d' \vartheta_2))$.*

By this Lemma 3 and the fact that, by Proposition 1, the equivalence $H \leftarrow c \wedge B' \wedge R \cong H \leftarrow e \wedge d' \vartheta_2 \wedge B' \wedge R$ holds iff $\mathcal{Q} \models \forall(c \leftrightarrow \exists W(e \wedge d' \vartheta_2))$, where $W = \text{Vars}(e \wedge d' \vartheta_2) - \text{Vars}(H, B' \wedge R)$, the constraint matching procedure \mathbf{CM}_3 may be reduced to the search for a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d' \vartheta_2))$.

Now we introduce some notions and we also establish some properties (see Lemma 4 and Theorem 6 below) which will be exploited by the constraint matching procedure \mathbf{CM}_3 .

We say that a conjunction $a_1 \wedge \dots \wedge a_m$ of atomic constraints is *non-redundant* iff there is no constraint $a_i \in \{a_1, \dots, a_m\}$ such that $\mathcal{Q} \models \forall((a_1 \wedge \dots \wedge a_{i-1} \wedge a_{i+1} \wedge \dots \wedge a_m) \rightarrow a_i)$. When looking for ϑ_2 satisfying the equivalence $c \leftrightarrow (\tilde{e} \wedge d' \vartheta_2)$, we may assume, without loss of generality, that c is non-redundant. However, we cannot make the same assumption for $(\tilde{e} \wedge d' \vartheta_2)$, because the application of ϑ_2 may transform a non-redundant constraint into a redundant one. The following Lemma 4 characterizes the equivalence of two constraints when one of them is non-redundant.

Lemma 4. *Let a be a non-redundant constraint of the form $a_1 \wedge \dots \wedge a_m$ and let b be a constraint of the form $b_1 \wedge \dots \wedge b_n$, where for $i = 1, \dots, m$ and $j = 1, \dots, n$, a_i and b_j are atomic constraints. Then $\mathcal{Q} \models \forall(a \leftrightarrow b)$ iff there exists an injection $\mu: \{a_1, \dots, a_m\} \rightarrow \{b_1, \dots, b_n\}$ such that for $i = 1, \dots, m$, $\mathcal{Q} \models \forall(a_i \leftrightarrow \mu(a_i))$ and for every $b' \in \{b_1, \dots, b_n\} - \{\mu(a_i) \mid 1 \leq i \leq m\}$, $\mathcal{Q} \models \forall(a \rightarrow b')$.*

In the constraint matching procedure \mathbf{CM}_3 we will use this Lemma 4 for finding, if it exists, a substitution ϑ_2 such that $\mathcal{Q} \models \forall(c \leftrightarrow (\tilde{e} \wedge d' \vartheta_2))$, where c is a non-redundant constraint and \tilde{e} is defined as in Lemma 3. Indeed, given the clauses $\gamma': H \leftarrow c \wedge B' \wedge R$ and $\delta': K' \leftarrow d' \wedge B'$ in normal form, the \mathbf{CM}_3 procedure: (1) computes an injection μ from the set of the atomic constraints occurring in c to the set of the atomic constraints occurring in $\tilde{e} \wedge d'$, and (2) computes a substitution ϑ_2 such that: (2.i) for every atomic constraint a occurring

in c , $\mathcal{Q} \models \forall(a \leftrightarrow \mu(a)\vartheta_2)$, and (2.ii) for every atomic constraint b occurring in $\tilde{e} \wedge d'$ and not in the image of μ , $\mathcal{Q} \models \forall(c \rightarrow b\vartheta_2)$.

With regard to Point (2.i), we make use of the following simple property: $\mathcal{Q} \models \forall(p \geq 0 \leftrightarrow q \geq 0)$, where p and q are linear polynomials, iff there exists a rational number $k > 0$ such that $\mathcal{Q} \models \forall(kp - q = 0)$. Analogously, $\mathcal{Q} \models \forall(p > 0 \leftrightarrow q > 0)$ holds iff there exists a rational number $k > 0$ such that $\mathcal{Q} \models \forall(kp - q = 0)$.

With regard to Point (2.ii), we make use of the following theorem [17] which is a generalization of the simple property we mentioned above.

Theorem 6. *Let $p_1 \rho_1 0, \dots, p_n \rho_n 0, p_{n+1} \rho_{n+1} 0$ be atomic constraints such that $\mathcal{Q} \models \exists(p_1 \rho_1 0 \wedge \dots \wedge p_n \rho_n 0)$, where, for $i = 1, \dots, n+1$, $\rho_i \in \{\geq, >\}$. Then $\mathcal{Q} \models \forall(p_1 \rho_1 0 \wedge \dots \wedge p_n \rho_n 0 \rightarrow p_{n+1} \rho_{n+1} 0)$ iff there exist $k_1 \geq 0, \dots, k_{n+1} \geq 0$ such that (i) $\mathcal{Q} \models \forall(k_1 p_1 + \dots + k_n p_n + k_{n+1} = p_{n+1})$, and (ii) if ρ_{n+1} is $>$ then*

$$\mathcal{Q} \models \forall\left(\left(\sum_{\substack{i=1, \dots, n \\ \rho_i \text{ is } >}} k_i\right) + k_{n+1} > 0\right)$$

Note that the constraint matching procedure **CM₃** may generate *nonlinear* polynomials. For instance, when looking for a substitution ϑ_2 such that $\mathcal{Q} \models \forall(a \leftrightarrow \mu(a)\vartheta_2)$ (see Point (2.i) above), an equivalence of the form $p \geq 0 \leftrightarrow q \geq 0$ is replaced by two constraints: (i) $nf(pV - q) = 0$ and (ii) $V > 0$, where $nf(pV - q)$ is the *normal form* of the *bilinear* polynomial $pV - q$ (see the rewrite rule (i) in **CM₃**) [12]. The bilinear polynomials and their normal forms are defined as follows.

Let p be a polynomial in the set $\{X_1, \dots, X_n\}$ of variables and let $\{P_1, P_2\}$ be a partition of $\{X_1, \dots, X_n\}$. The polynomial p is said to be *bilinear in the partition* $\{P_1, P_2\}$ if: (i) for some ground substitution σ_1 for the variables in P_1 , $p\sigma_1$ is a linear polynomial, and (ii) for some ground substitution σ_2 for the variables in P_2 , $p\sigma_2$ is a linear polynomial. In a bilinear polynomial the monomials are of the form kVX where k is a rational number, and V and X are distinct variables. Let $\langle X_1, \dots, X_m \rangle$ (with $m \leq n$) be any ordering of the variables in P_1 . A *normal form* $nf(p)$ of the bilinear polynomial p w.r.t. X_1, \dots, X_m , is a polynomial of the form $p_1 X_1 + \dots + p_m X_m + p_{m+1}$, where: (i) p_1, \dots, p_{m+1} are linear polynomials in the set P_2 of variables such that for all $k = 1, \dots, m$, $\forall(p_k \neq 0)$, and (ii) $\mathcal{Q} \models \forall(p = p_1 X_1 + \dots + p_m X_m + p_{m+1})$. Any linear polynomial p is also a bilinear polynomial in the partition $\{Vars(p), \emptyset\}$, and we can define the normal form of a linear polynomial p w.r.t. any ordering of $Vars(p)$.

Constraint Matching Procedure: **CM₃**

Input: two clauses $\gamma': H \leftarrow c \wedge B' \wedge R$ and $\delta': K' \leftarrow d' \wedge B'$ in normal form.

Output: a clause $\gamma'': H \leftarrow e \wedge d'\vartheta_2 \wedge B'\vartheta_2 \wedge R$ such that: (i) $\gamma' \cong H \leftarrow e \wedge d'\vartheta_2 \wedge B' \wedge R$, (ii) $B'\vartheta_2 = B'$, (iii) $Vars(K'\vartheta_2) \subseteq Vars(H)$, and (iv) $Vars(e) \subseteq Vars(H, R)$.

If such clause γ'' does not exist, then **fail**.

In what follows we denote the set $Vars(c) - Vars(B')$ by X , the set $Vars(d') - Vars(B')$ by Y , and the set $Vars_{\text{rat}}(B')$ by Z .

Let e be the constraint $project(c, X)$. Without loss of generality, we assume that the constraint c is of the form $p_1\rho_1 0 \wedge \dots \wedge p_m\rho_m 0$ and the constraint $e \wedge d'$ is of the form $q_1\pi_1 0 \wedge \dots \wedge q_n\pi_n 0$, where for $i = 1, \dots, m$ and $j = 1, \dots, n$, p_i and q_i are linear polynomials, and ρ_i and π_j are predicate symbols in $\{\geq, >\}$.

Consider the following rewrite rules (i)–(v) of the form:

$$\langle f_1 \leftrightarrow g_1, S_1, \sigma_1 \rangle \Longrightarrow \langle f_2 \leftrightarrow g_2, S_2, \sigma_2 \rangle$$

where: (1) f_1, g_1, f_2 , and g_2 are constraints, (2) S_1 and S_2 are sets of atomic constraints, (3) σ_1 and σ_2 are substitutions, (4) when S_1 is written as $\{f\} \cup S$ we assume that $f \notin S$, and (5) all polynomials occurring in S_1 and S_2 are bilinear in $\{X \cup Y \cup Z, W\}$, where W is the set of the new variables introduced during the rewritings (see rules (i)–(iii)), and their normal forms are computed w.r.t. the variable ordering $Z_1, \dots, Z_h, Y_1, \dots, Y_k, X_1, \dots, X_l$, where $\{Z_1, \dots, Z_h\} = Z$, $\{Y_1, \dots, Y_k\} = Y$, and $\{X_1, \dots, X_l\} = X$.

- (i) $\langle p\rho 0 \wedge f \leftrightarrow g_1 \wedge q\rho 0 \wedge g_2, S, \sigma \rangle \Longrightarrow$
 $\langle f \leftrightarrow g_1 \wedge g_2, \{nf(pV - q) = 0, V > 0\} \cup S, \sigma \rangle$
 where V is a new variable and $\rho \in \{\geq, >\}$;
- (ii) $\langle true \leftrightarrow q \geq 0 \wedge g, S, \sigma \rangle \Longrightarrow$
 $\langle true \leftrightarrow g, \{nf(p_1V_1 + \dots + p_mV_m + V_{m+1} - q) = 0,$
 $V_1 \geq 0, \dots, V_{m+1} \geq 0\} \cup S, \sigma \rangle$
 where V_1, \dots, V_{m+1} are new variables;
- (iii) $\langle true \leftrightarrow q > 0 \wedge g, S, \sigma \rangle \Longrightarrow$
 $\langle true \leftrightarrow g, \{nf(p_1V_1 + \dots + p_mV_m + V_{m+1} - q) = 0, V_1 \geq 0, \dots, V_{m+1} \geq 0,$
 $(\sum_{i=1}^m \rho_i is > V_i) + V_{m+1} > 0\} \cup S, \sigma \rangle$
 where V_1, \dots, V_{m+1} are new variables;
- (iv) $\langle f \leftrightarrow g, \{pV + q = 0\} \cup S, \sigma \rangle \Longrightarrow \langle f \leftrightarrow g, \{nf(p) = 0, q = 0\} \cup S, \sigma \rangle$
 if $V \in X \cup Z$;
- (v) $\langle f \leftrightarrow g, \{aV + q = 0\} \cup S, \sigma \rangle \Longrightarrow$
 $\langle f \leftrightarrow (g\{V/-\frac{q}{a}\}), \{nf(p\{V/-\frac{q}{a}\})\rho 0 \mid p\rho 0 \in S\}, \sigma\{V/-\frac{q}{a}\} \rangle$
 if $V \in Y$, $Vars(q) \cap Vars(R) = \emptyset$, and $a \in (\mathbb{Q} - \{0\})$;

IF $\mathcal{Q} \models \neg \exists(e)$ THEN return clause $\gamma'' : H \leftarrow e \wedge d' \vartheta_2 \wedge B' \vartheta_2 \wedge R$, where ϑ_2 is an arbitrary substitution of the form $\{V_1/a_1, \dots, V_s/a_s\}$, with $\{V_1, \dots, V_s\} = Vars_{\text{rat}}(K') - Vars(H)$, and $a_1, \dots, a_s \in \mathbb{Q}$;

ELSE IF there exists a set C of constraints and a substitution σ_Y such that:

1. $\langle c \leftrightarrow e \wedge d', \emptyset, \emptyset \rangle \Longrightarrow^* \langle true \leftrightarrow true, C, \sigma_Y \rangle$,
2. there is no triple T such that $\langle true \leftrightarrow true, C, \sigma_Y \rangle \Longrightarrow T$,
3. for all $f \in C$, $Vars(f) \subseteq W$, where W is the set of the new variables introduced when applying rules (i)–(v) at Step 1,
4. C is satisfiable and $solve(C) = \sigma_W$,

THEN let σ_C be an arbitrary substitution of the form $\{V_1/a_1, \dots, V_s/a_s\}$, where $\{V_1, \dots, V_s\} = Vars_{\text{rat}}(K' \sigma_Y \sigma_W) - Vars(H)$ and $a_1, \dots, a_s \in \mathbb{Q}$;

let ϑ_2 be $\sigma_Y \sigma_W \sigma_C$; return clause $\gamma'' : H \leftarrow e \wedge d' \vartheta_2 \wedge B' \vartheta_2 \wedge R$

ELSE return **fail**.

It can be shown that the constraint matching procedure \mathbf{CM}_3 always terminates and it is sound and complete with respect to its specification (see Algorithm \mathbf{CM}_3). Thus, by using \mathbf{CM}_3 , instead of \mathbf{CM}_1 , we get a terminating, sound, complete folding algorithm \mathbf{FA}_3 for applying the rule \mathbf{Fold}_3 .

Folding Algorithm: \mathbf{FA}_3

Input: two clauses $\gamma : H \leftarrow c \wedge G$ and $\delta : K \leftarrow d \wedge B$ in normal form and without variables in common.

Output: a clause $\eta : H \leftarrow e \wedge K\vartheta \wedge R$, if it is possible to fold γ using δ according to Definition 4, and **fail**, otherwise.

IF there exist two clauses of the form $\gamma' : H \leftarrow c \wedge B\vartheta_1 \wedge R$ and $\delta' : K\vartheta_1 \leftarrow d\vartheta_1 \wedge B\vartheta_1$, which are the output of an execution of the \mathbf{GM} procedure when the clauses γ and δ are given in input

AND there exists a clause $\gamma'' : H \leftarrow e \wedge d\vartheta_1\vartheta_2 \wedge B\vartheta_1\vartheta_2 \wedge R$ which is the output of an execution of the \mathbf{CM}_3 procedure when the clauses γ' and δ' are given in input

THEN return the clause $\eta : H \leftarrow e \wedge K\vartheta_1\vartheta_2 \wedge R$

ELSE return **fail**.

Theorem 7 (Termination, Soundness, and Completeness of the Folding Algorithm \mathbf{FA}_3). *Let two clauses γ and δ , in normal form and without variables in common, be the input of Algorithm \mathbf{FA}_3 . Then:*

- (i) \mathbf{FA}_3 terminates,
- (ii) if \mathbf{FA}_3 returns a clause η , then η can be derived by folding γ using δ according to Definition 4, and
- (iii) if it is possible to fold γ using δ according to Definition 4, then \mathbf{FA}_3 does not return **fail**.

Let us now present an example of application of the folding algorithm \mathbf{FA}_3 . Suppose we are given the following two clauses:

$$\gamma: p(X_1, X_2) \leftarrow Z > 0 \wedge X_1 - Z - 1 \geq 0 \wedge X_2 - X_1 > 0 \wedge s(Z, f(X_3))$$

$$\delta: q(Y_1, Y_2) \leftarrow U > 0 \wedge Y_1 - 3 - 2U \geq 0 \wedge 2Y_2 - Y_1 - 1 > 0 \wedge s(U, f(Y_3))$$

where p and q are predicate symbols of type $\mathbf{rat} \times \mathbf{rat}$, s of type $\mathbf{rat} \times \mathbf{tree}$, and we want to fold γ using δ . These clauses are in normal form and do not have variables in common. The output of the goal matching procedure applied to the clauses γ and δ is the following pair of clauses:

$$\gamma': p(X_1, X_2) \leftarrow Z > 0 \wedge X_1 - Z - 1 \geq 0 \wedge X_2 - X_1 > 0 \wedge s(Z, f(X_3))$$

$$\delta': q(Y_1, Y_2) \leftarrow Z > 0 \wedge Y_1 - 3 - 2Z \geq 0 \wedge 2Y_2 - Y_1 - 1 > 0 \wedge s(Z, f(X_3))$$

which are the input of the constraint matching procedure \mathbf{CM}_3 . The \mathbf{CM}_3 procedure starts off by computing the following constraint e which is defined as $\mathit{project}(Z > 0 \wedge X_1 \geq Z + 1 \wedge X_2 > X_1, \{X_1, X_2\})$:

$$e: X_1 - 1 > 0 \wedge X_2 - X_1 > 0$$

Now, starting from the triple:

$$\langle (Z > 0 \wedge X_1 - Z - 1 \geq 0 \wedge X_2 - X_1 > 0) \leftrightarrow$$

$$(X_1 - 1 > 0 \wedge X_2 - X_1 > 0 \wedge Z > 0 \wedge Y_1 - 3 - 2Z \geq 0 \wedge 2Y_2 - Y_1 - 1 > 0), \emptyset, \emptyset \rangle$$

and applying the rewrite rules (i)–(v), we derive the following set of constraints:

$$C: \{W_1 - 1 = 0, W_1 > 0, 2 - W_2 = 0, W_2 > 0, W_3 > 0\}$$

together with the following substitution:

$$\sigma_Y: \{Y_1/W_2X_1 + 3 - W_2, Y_2/(\frac{W_2}{2} - \frac{W_3}{2})X_1 + \frac{W_3}{2}X_2 + 2 - \frac{W_2}{2}\}$$

The solution $\text{solve}(C)$ is the ground substitution $\sigma_w = \{W_1/1, W_2/2, W_3/2\}$. Since $\text{Vars}_{\text{rat}}(q(Y_1, Y_2)\sigma_Y\sigma_w) \subseteq \text{Vars}(p(X_1, X_2))$, we have that σ_G is the identity substitution \emptyset . Thus, the substitution ϑ_2 is equal to $\sigma_Y\sigma_w$. Since we have to compute $q(Y_1, Y_2)\vartheta_2$, we can restrict ϑ_2 to the set $\{Y_1, Y_2\}$ and we get:

$$\vartheta_2: \{Y_1/2X_1 + 1, Y_2/X_2 + 1\}$$

Thus, an output of the folding algorithm **FA₃** is the clause $p(X_1, X_2) \leftarrow e \wedge q(Y_1, Y_2)\vartheta_2$, that is:

$$\eta: p(X_1, X_2) \leftarrow X_1 > 1 \wedge X_2 > X_1 \wedge q(2X_1 + 1, X_2 + 1)$$

Clause η has no existential variables.

When applying the rewrite rule (i), the nondeterministic procedure **CM₃** may also compute a different substitution, which we call ϑ'_2 , by selecting a different atomic constraint $q\rho 0$. In particular, the procedure **CM₃** may compute the following alternative set of constraints:

$$C': \{W_1 + W_2 + W_3 > 0, W_1 \geq 0, W_2 \geq 0, W_3 \geq 0\}$$

together with the substitution:

$$\sigma'_Y: \{Y_1/2X_1 + 1, Y_2/\frac{W_1+1}{2}X_1 + \frac{W_2}{2}X_2 + 1 + \frac{W_3}{2}\}$$

The solution $\text{solve}(C')$ is the ground substitution $\sigma'_w = \{W_1/1, W_2/0, W_3/1\}$. Since $\text{Vars}_{\text{rat}}(q(Y_1, Y_2)\sigma'_Y\sigma'_w) \subseteq \text{Vars}(p(X_1, X_2))$, we have that σ'_G is the identity substitution \emptyset . Therefore, the substitution ϑ'_2 is equal to $\sigma'_Y\sigma'_w$. Since we have to compute $q(Y_1, Y_2)\vartheta'_2$, we can restrict ϑ'_2 to the set $\{Y_1, Y_2\}$ and we get:

$$\vartheta'_2: \{Y_1/2X_1 + 1, Y_2/X_1 + \frac{3}{2}\}$$

Thus, an alternative output of the folding algorithm **FA₃** is the clause:

$$\eta': p(X_1, X_2) \leftarrow X_1 > 1 \wedge X_2 > X_1 \wedge q(2X_1 + 1, X_2 + \frac{3}{2})$$

Also this clause η' has no existential variables.

The reader can check that the **CM₃** procedure cannot generate other sets of constraints besides the two sets C and C' we have indicated above.

6 Related Work and Conclusions

The folding rule has been often considered in the papers that deal with the transformation rules for logic programs and constraints logic programs [3,5,7,10,18]. Folding is a crucial transformation rule because, besides other reasons, it allows beneficial changes of the recursive structure of the programs and, when using program transformation for inductive proofs [14], its application is basically equivalent to the use of the inductive hypothesis.

In the literature the folding rule is specified in a declarative way and no algorithm is provided to determine whether or not, given a clause γ to be folded and a clause δ for folding, one can actually fold γ using δ .

In this paper we have considered constraint logic programs with constraints that are conjunctions of linear equations and inequations over the rational numbers (or the real numbers) and we have proposed an algorithm, based on linear algebra and term rewriting techniques, for applying the folding rule.

We have also introduced two variants of the folding rule and we have presented two algorithms for applying these variants. The first variant combines the folding rule with the clause splitting rule and the second variant can be applied for eliminating the existential variables of a clause, that is, the variables which occur in the body of a clause and not in the head.

The problem of checking whether or not, given two clauses, say γ and δ , clause γ can be folded using clause δ , is similar to the problem of matching two terms modulo an equational theory [1,19], but the matching problem for deciding the applicability of folding has an extra difficulty that is due to the presence of existential variables (these variables are not taken into account in the equational theories considered in [1,19]).

In the future we plan to investigate in more detail the connections between the problem of folding and the problem of matching modulo an equational theory, by looking, in particular, at those techniques which deal with combinations of equational theories (see, for instance, [16]).

We also plan to adapt our folding algorithms to other constraint domains, such as the linear equations and inequations over the integer numbers.

One more aspect that need to be addressed is the analysis of the computational complexity of our algorithms for folding. Since our algorithms make use of the Fourier-Motzkin variable elimination procedure, they take superexponential time in the number of variables occurring in the input clauses.

Finally, an implementation of our folding algorithms is under development in the MAP transformation system [11]. This implementation will allow us to evaluate in practice the efficiency of the folding algorithms, as well as the usefulness of the various versions of the folding rule in various program derivations.

References

1. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 445–532. Elsevier Science, 2001.
2. D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1-2):203–216, 1987.
3. N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, 206:81–125, 1998.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
5. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
6. F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with clause splitting for deriving deterministic constraint logic programs. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Hammamet (Tunisia)*. IEEE Computer Society Press, 2002.

7. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer-Verlag, 2004.
8. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
9. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
10. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
11. MAP. The MAP transformation system. Available from <http://www.iasi.rm.cnr.it/~proietti/system.html>, 1995–2008.
12. E. D. Nering. *Linear Algebra and Matrix Theory*. John Wiley & Sons, 2nd edition, 1963.
13. A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
14. A. Pettorossi, M. Proietti, and V. Senni. Proving properties of constraint logic programs by eliminating existential variables. In S. Etalle and M. Truszczyński, editors, *Proceedings of the 22nd International Conference on Logic Programming (ICLP '06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 179–195. Springer-Verlag, 2006.
15. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
16. C. Ringeissen. Matching in a class of combined non-disjoint theories. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19), Miami Beach, FL, USA, July 28 - August 2, 2003*, volume 2741 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2003.
17. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.
18. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
19. Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

Graded CTL Model Checking^{*}

Alessandro Ferrante, Margherita Napoli, and Mimmo Parente

Dipartimento di Informatica ed Applicazioni “R.M. Capocelli”, University of Salerno
Via Ponte don Melillo - 84084 - Fisciano (SA) - Italy
E-mail: {ferrante,napoli,parente}@dia.unisa.it

Abstract. The use of the universal and existential quantifiers with the capability to express the concept of *at least k* or *all but k* , for a non negative integer k , has always been thoroughly studied in various kinds of logics. In classical logic there are *counting quantifiers*, in modal logics *graded modalities*, in description logics *number restrictions*.

Recently, the complexity issues related to the decidability of the μ -calculus, when the universal and existential quantifiers are augmented with graded modalities, have been investigated by Kupfermann, Sattler and Vardi. They have shown that this problem is EXPTIME-complete.

In this paper we consider another extension of modal logic, the Computational Tree Logic CTL, augmented with graded modalities generalizing standard quantifiers and investigate the complexity issues, with respect to the model-checking problem. We consider a system model represented by a pointed Kripke structure \mathcal{K} and give an algorithm to solve the model-checking problem running in time $O(|\mathcal{K}| \cdot |\varphi|)$ which is hence tight for the problem (where $|\varphi|$ is the number of temporal and boolean operators and does not include the values occurring in the graded modalities).

In this framework, the graded modalities express the ability to generate a user-defined number of counterexamples (or evidences) to a specification φ given in CTL. However these multiple counterexamples can partially overlap, that is they may share some behavior. We have hence investigated the case when all of them are completely disjoint. In this case we prove that the model-checking problem is both NP-hard and CONP-hard and give an algorithm for solving it running in polynomial space. We have thus studied a fragment of this graded-CTL logic, and have proved that in this case the model-checking problem is solvable in polynomial time.

1 Introduction

Model-checking is the process, which is now becoming widely accepted, to check whether a given model satisfies a given logical formula [CGP99, QS82], and it can be applied to all kinds of logics. In this paper we consider model-checking of formulas expressed in a logic which extends the classical Computational Tree

^{*} Work partially supported by M.I.U.R. grant ex-60%. A preliminary version of the paper is currently submitted to an international conference for publication.

Logic, CTL, with graded modalities. Classical CTL can be used for reasoning about the temporal behavior of systems considering either *all the possible futures* or *at least one possible future*. Here we use *graded* extensions on the existential and universal quantifiers.

In the literature the capability to express *at least k* and *all but k* , given a non negative integer k , has been intensively studied in various logic frameworks. In classical logics $\exists^{>k}$ and $\forall^{\leq k}$ are called *counting quantifiers*, see [GOR97,GMV99,PST00], in modal logics they are called *graded modalities*, see [Fin72,Tob01], and in description logics one speaks about *number restriction* of properties describing systems, see e.g. [HB91]. Recently the complexity issues related to the decidability of the μ -calculus when the universal and existential quantifiers are augmented with graded modalities, have been investigated in [KSV02]. They have shown that this problem is EXPTIME-complete, retaining thus the same complexity as in the case of classical μ -calculus, though strictly extending it.

In this paper we introduce the *graded-CTL*, obtained by augmenting CTL with graded modalities that generalize standard path quantifiers and this logic, here too, strictly extends classical CTL. With graded-CTL formulas we can reason about more than any constant number of futures. For example consider the formula $E^{>k}\mathcal{F}(criticSection1 \wedge criticSection2)$, which expresses that there are more than k possibilities to violate the mutual exclusion property to access critical sections of a system, and the formula $E^{>k}\mathcal{F}\neg(wait \Rightarrow AFcriticSection)$ which expresses the fact that in several cases it is possible that a waiting process never obtains the requested resource. Clearly formulas of these types cannot be expressed in CTL and not even in classical μ -calculus.

The focus in the paper is on the complexities involved in the process of model-checking system models against specifications given in this logic. In this framework the motivation in the use of these graded modalities mainly arises from the fact that during the verification of a system design, a central feature of the technique of model-checking is the generation of counterexamples. In fact the realization process for a system passes through the “Check/Analyze/Fix” cycle: model-check the design of the system against some desired properties φ , analyze the generated counterexamples to the properties, and re-design the system, trying to fix the errors. The analysis of the counterexamples usually gives clues to that part of the system model where the specification failed. Usually up-to-date model-checkers, as NuSMV and SPIN [CCGR99,Hol97], provide only one counterexample of φ , and this obliges to undergo again and again through the time-consuming three stage cycle. It is therefore highly desirable to have as many significative counterexamples as possible simultaneously, c.f. [CG07,CIW⁺01,DRS03]. Hence, the investigation of the complexities involved in the generation and the analysis of the counterexamples is a central issue, as explained also in the survey [CV03], where the role and the structure of counterexamples is nicely investigated putting an emphasis on the complexities related to the generation problem. Moreover, in [MFG04] it is shown that the set of the counterexamples (and witnesses) of a formula for a fragment of ACTL can be recognized by a finite automaton.

Given a graded-CTL formula φ and a system model represented by a pointed Kripke structure \mathcal{K} , our first result is an algorithm to solve the model-checking problem in time $O(|\mathcal{K}| \cdot |\varphi|)$, the same running time of the algorithm for classical CTL. Let us remark that this complexity does not depend at all on the values representing the grading of the modalities and the size $|\varphi|$ of the formula does not depend on the representation of these values and is simply the number of the temporal and boolean operators. However the multiple counterexamples returned by this algorithm may overlap, while it can be desirable in the analysis phase to detect independent traces where the specification fails. To deal with this case, we have introduced a semantic for temporal operators to require the edge-disjointness of the paths representing the counterexamples. The same setting can be applied also, for example, to ensure that a “correct” system behavior tolerates a given number of faults of the system. We have proved that to model-check a system model against such specifications is both NP-hard and coNP-hard. The reduction has been done from the cycle-packing problem (the problem to check whether there are k disjoint cycles in a graph). This has suggested that formulas of the type $E^{>k}\mathcal{G}\varphi$ (there exist at least $k + 1$ infinite edge-disjoint paths globally satisfying φ) are hard to verify. We have then defined the still interesting fragment of the logic obtained by dropping this kind of formulas and proved that the model-checking problem can be solved in polynomial time in this case. Clearly, unless $\text{NP} = \text{coNP}$, the problem, in the full logic, does not belong to NP. We have then given an algorithm for it, showing that however it is in PSPACE. Finally, we have considered the scenario in which only a given number of behaviors need to be disjoint and all the remaining may overlap. In this case we have proved that the problem is *fixed parameter* tractable.

The paper is organized as follows: in Section 2 we give some preliminary definitions and introduce the model-checking problem for graded-CTL; in Section 3 we prove that the graded-CTL model-checking is solvable in polynomial time; in Section 4 we study the edge-disjoint graded-CTL model-checking. Moreover we show that the same problem restricted to a fragment of graded-CTL is solvable in polynomial time, and that we can obtain a good algorithm in practical cases by relaxing the edge-disjointness requirement; finally in Section 5 we give some conclusions and open problems.

2 Graded-CTL Logic

In this section we introduce the graded-CTL logic which extends the classical CTL logic with graded quantifiers. CTL can be used for reasoning about the temporal behavior of systems considering either “all the possible futures” or “at least one possible future”. Graded extension generalizes CTL to reasoning about more than a given number of possible future behaviors.

Let us start by giving the syntax of the logic. The graded-CTL operators consist of the temporal operators \mathcal{U} (“until”), \mathcal{G} (“globally”) and \mathcal{X} (“next”), the boolean connectives \wedge and \neg , and the graded path quantifier $E^{>k}$ (“for at least $k+1$ futures”).

Given a set of atomic propositions AP , the syntax of the graded-CTL formulas is

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid E^{>k}\mathcal{X}\varphi \mid E^{>k}\varphi\mathcal{U}\varphi \mid E^{>k}\mathcal{G}\varphi$$

where $p \in AP$ and k is a non negative integer.

We define the semantics of graded-CTL with respect to *Kripke Structures*. As usual, a Kripke structure over a set of atomic propositions AP , is a tuple $\mathcal{K} = \langle S, s_{in}, R, L \rangle$, where S is a finite set of states, $s_{in} \in S$ is the initial state, $R \subseteq S \times S$ is a transition relation with the property that for each $s \in S$ there is $t \in S$ such that $(s, t) \in R$, and $L : S \rightarrow 2^{AP}$ is a labeling function.

A path in \mathcal{K} is denoted by the sequence of states $\pi = \langle s_0, s_1, \dots, s_n \rangle$ or by $\pi = \langle s_0, s_1, \dots \rangle$, if it is infinite. The length of a path, denoted by $|\pi|$, is the number of states in the sequence, and $\pi[i]$ denotes the state s_i , $0 \leq i < |\pi|$. Two paths π_1 and π_2 are *distinct* if there exists an index $0 \leq i < \min\{|\pi_1|, |\pi_2|\}$ such that $\pi_1[i] \neq \pi_2[i]$. Observe that from this definition if a path is the prefix of another path, then they are not distinct.

Let $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ be a Kripke structure and $s \in S$ be a state of \mathcal{K} . The concept of satisfiability for graded-CTL formulas is established by the relation \models , defined as follows:

- $(\mathcal{K}, s) \models p$, $p \in AP$, iff $p \in L(s)$;
- $(\mathcal{K}, s) \models \varphi_1 \wedge \varphi_2$ iff $(\mathcal{K}, s) \models \varphi_1$ and $(\mathcal{K}, s) \models \varphi_2$;
- $(\mathcal{K}, s) \models \neg\varphi$ iff $\neg((\mathcal{K}, s) \models \varphi)$ (shortly written, $(\mathcal{K}, s) \not\models \varphi$);
- $(\mathcal{K}, s) \models E^{>k}\mathcal{X}\varphi$ iff there exist $k+1$ different states s_0, \dots, s_k such that
 1. $(s, s_i) \in R$ and
 2. $(\mathcal{K}, s_i) \models \varphi$ for all $0 \leq i \leq k$;
- $(\mathcal{K}, s) \models E^{>k}\mathcal{G}\varphi$ iff there exist $k+1$ pairwise distinct infinite paths π_0, \dots, π_k such that for every $0 \leq j \leq k$,
 1. $\pi_j[0] = s$ and
 2. for all $h \geq 0$, $(\mathcal{K}, \pi_j[h]) \models \varphi$.
- $(\mathcal{K}, s) \models E^{>k}\varphi_1\mathcal{U}\varphi_2$ iff there exist $k+1$ pairwise distinct finite paths π_0, \dots, π_k of length i_0, \dots, i_k , respectively, such that for all $0 \leq j \leq k$
 1. $\pi_j[0] = s$,
 2. $(\mathcal{K}, \pi_j[i_j - 1]) \models \varphi_2$, and
 3. for every $0 \leq h < i_j - 1$, $(\mathcal{K}, \pi_j[h]) \models \varphi_1$;

The graded-CTL formulas (as in the standard non-graded CTL) are also called *state-formulas* and a state s in \mathcal{K} satisfies a state-formula φ if $(\mathcal{K}, s) \models \varphi$. On the other side, $\mathcal{X}\varphi$, $\mathcal{G}\varphi$ and $\varphi_1\mathcal{U}\varphi_2$ are called as usual *path-formulas*. In particular a path satisfying a path-formula θ is called an *evidence* of θ (note that the evidences for \mathcal{X} and \mathcal{U} are finite paths). Then, for the fulfillment of a formula $E^{>k}\theta$ in a state s , it is required the existence of $k+1$ distinct evidences of θ , starting from s .

As usual, in our logic the temporal operator \mathcal{F} (“eventually”) can be defined in terms of the operators given above: $E^{>k}\mathcal{F}\varphi \Leftrightarrow E^{>k} \text{TRUE } \mathcal{U} \varphi$. Moreover, it is easy to observe that CTL is a proper fragment of graded-CTL since the simple formula $E^{>1}\mathcal{X}p$ cannot be expressed in CTL, whereas any CTL formula is also a graded-CTL formula since the quantifier E is equivalent to $E^{>0}$ and for the universal quantifier A the standard relations hold, recalled here:

- $A\mathcal{X}\varphi \iff \neg E\mathcal{X}\neg\varphi$;
- $AG\varphi \iff \neg E\mathcal{F}\neg\varphi$;
- $A\varphi_1\mathcal{U}\varphi_2 \iff \neg E(\neg\varphi_2\mathcal{U}(\neg\varphi_1 \wedge \neg\varphi_2)) \wedge \neg E\mathcal{G}\neg\varphi_2$.

Finally, we also consider the graded extension of the quantifier A , $A^{\leq k}$, with the meaning that *all the paths starting from a node s , except for at most k , satisfy a given path-formula*. The quantifier $A^{\leq k}$ is the dual of $E^{>k}$ and can obviously be re-written in terms of $\neg E^{>k}$. We now formally define the model-checking problem.

Given a Kripke structure $\mathcal{K} = \langle S, s_{in}, R, L \rangle$, and a graded-CTL formula φ , the **graded-CTL model-checking** is the problem to verify whether $(\mathcal{K}, s_{in}) \models \varphi$.

In the next sections we study the complexity of the model-checking problem with respect to the size of the Kripke structure (expressed in terms of the number of edges, as by our definition $|R| \geq |S|$), and to the size of the CTL formula, where the size $|\varphi|$ of a CTL formula φ is the number of the temporal and the boolean operators occurring in it.

3 Graded-CTL Model-Checking

In this section we show that the graded-CTL model-checking problem can be solved in polynomial time and independently from the values, occurring in the graded modalities, involved in the formulas. Then we discuss possible applications of our result to the generation of counterexamples.

Let us recall that an algorithm to solve the model-checking problem for a given Kripke structure \mathcal{K} and a given formula φ computes the subset $\{s \in S \text{ s.t. } (\mathcal{K}, s) \models \varphi\}$.

Theorem 1. *Let $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ be a Kripke structure and φ be a graded-CTL formula. The graded-CTL model-checking problem can be solved in time $\mathcal{O}(|R| \cdot |\varphi|)$.*

Proof. To solve the problem we give an algorithm that works on the sub-formulas ψ of φ and for each state s determines whether $(\mathcal{K}, s) \models \psi$ (and sets a boolean variable $s.\psi$ to TRUE), (see Algorithm 1). The algorithm uses a primitive function $Sub(\varphi)$ which returns all the sub-formulas of φ and moreover for a path-formula θ , if $E^{>k}\theta$ is in $Sub(\varphi)$, then $E^{>0}\theta$ is in $Sub(\varphi)$ as well. In particular we assume that such formulas are returned in non decreasing order of complexity, with $E^{>0}\theta$ preceding $E^{>k}\theta$ in the sequence.

If a sub-formula ψ is of type $p \in AP, \neg\psi_1, \psi_1 \wedge \psi_2, E^{>0}\mathcal{G}\psi_1, E^{>0}\psi_1\mathcal{U}\psi_2$, then the algorithm (lines 3–13) works as the classical CTL model-checking algorithm (see e.g. [CGP99]), and, if a sub-formula is of type $E^{>k}\mathcal{X}\psi_1$, then the algorithm checks, for each state s whether $|\{t \in S \mid (s, t) \in R \text{ and } (\mathcal{K}, t) \models \psi_1\}| > k$, (lines 14–16).

Consider now a sub-formula $\psi = E^{>k}\mathcal{G}\psi_1$ with $k > 0$ (line 17). Let a *sink-cycle* be a cycle not containing *exit-nodes*, that is nodes with out-degree at least 2. The algorithm is based on the following claim, that we will prove later:

Algorithm 1: The algorithm *GradedCTL*(\mathcal{K}, φ).

Input: A Kripke Structure $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ and a graded-CTL formula φ .

Output: For each state s , $s.\varphi = \text{TRUE}$ if $(\mathcal{K}, s) \models \varphi$ and $s.\varphi = \text{FALSE}$ otherwise

```

1  Let  $s.\psi = \text{FALSE}$  for all  $s \in S$  and  $\psi \in \text{Sub}(\varphi)$ ;
2  forall  $\psi \in \text{Sub}(\varphi)$  do
3      case  $\psi = p \in AP$ : forall  $s \in S$  s.t.  $p \in L(s)$  do  $s.\psi \leftarrow \text{TRUE}$ ;
4      case  $\psi = \neg\psi_1$ : forall  $s \in S$  do  $s.\psi \leftarrow \neg s.\psi_1$ ;
5      case  $\psi = \psi_1 \wedge \psi_2$ : forall  $s \in S$  do  $s.\psi \leftarrow (s.\psi_1 \wedge s.\psi_2)$ ;
6      case  $\psi = E^{>0}\mathcal{G}\psi_1$ :
7           $S' \leftarrow \{s \in S \mid s.\psi_1 = \text{TRUE}\}$ ;  $R' \leftarrow R \cap (S' \times S')$ ;
8          forall  $s \in S'$  s.t.  $\exists$  a cycle reachable from  $s$  in  $(S', R')$  do  $s.\psi \leftarrow \text{TRUE}$ ;
9      end
10     case  $\psi = E^{>0}\psi_1\mathcal{U}\psi_2$ :
11          $S' \leftarrow \{s \in S \mid s.\psi_1 = \text{TRUE} \text{ or } s.\psi_2 = \text{TRUE}\}$ ;  $R' \leftarrow R \cap (S' \times S')$ ;
12         forall  $s \in S'$  s.t.  $\exists t$  with  $t.\psi_2 = \text{TRUE}$  reachable from  $s$  in  $(S', R')$  do
13              $s.\psi \leftarrow \text{TRUE}$ ;
14         end
15     case  $\psi = E^{>k}\mathcal{X}\psi_1$  with  $k \geq 0$ :
16         forall  $s \in S$  s.t.  $|\{(s, t) \in R \mid t.\psi_1 = \text{TRUE}\}| > k$  do  $s.\psi \leftarrow \text{TRUE}$ ;
17     end
18     case  $\psi = E^{>k}\mathcal{G}\psi_1$  with  $k > 0$ :
19          $S' \leftarrow \{s \in S \mid s.E^{>0}\mathcal{G}\psi_1 = \text{TRUE}\}$ ;  $R' \leftarrow R \cap (S' \times S')$ ;
20         forall  $s \in S'$  s.t.  $\exists$  a non-sink-cycle reachable from  $s$  in  $(S', R')$  do
21              $s.\psi \leftarrow \text{TRUE}$ ;
22         forall  $s \in S'$  s.t.  $\exists k+1$  distinct finite paths from  $s$  to sink-cycles in
23              $(S', R')$  do  $s.\psi \leftarrow \text{TRUE}$ ;
24         end
25     case  $\psi = E^{>k}\psi_1\mathcal{U}\psi_2$  with  $k > 0$ :
26          $S' \leftarrow \{s \in S \mid s.E^{>0}\psi_1\mathcal{U}\psi_2 = \text{TRUE}\}$ ;
27          $R' \leftarrow (R \cap (S' \times S')) \setminus \{(s, t) \in R \mid s.\psi_1 = \text{FALSE}\}$ ;
28         forall  $s \in S'$  s.t.  $\exists$  a non-sink-cycle reachable from  $s$  in  $(S', R')$  do
29              $s.\psi \leftarrow \text{TRUE}$ ;
30         forall  $s \in S'$  s.t.  $\exists k+1$  distinct finite paths from  $s$  to states where  $\psi_2$ 
31             holds in  $(S', R')$  do  $s.\psi \leftarrow \text{TRUE}$ ;
32         end
33     end
34 end

```

Claim 1: Let $G_\psi = (S_\psi, R_\psi)$ be the graph induced by the states where $E^{>0}\mathcal{G}\psi_1$ holds; then, given a state $s \in S$, $(\mathcal{K}, s) \models \psi$ iff $s \in S_\psi$ and either there is a non-sink-cycle reachable from s , or there are $k+1$ distinct finite paths connecting s to sink-cycles in G_ψ .

The algorithm looks for the states in G_ψ from which it is possible to reach a non-sink-cycle (line 19) and then looks for the states from which $k+1$ distinct finite paths start, each ending in sink-cycles (line 20).

Let us now consider a sub-formula $\psi = E^{>k}\psi_1\mathcal{U}\psi_2$ (line 22). In this case, the algorithm is based on the following claim:

Claim 2: Let $G_\psi = (S_\psi, R_\psi)$ be the graph induced by considering the states where $E^{>0}\psi_1\mathcal{U}\psi_2$ holds and by deleting the edges outgoing from states where ψ_1 is not satisfied; then, given a state $s \in S$, $(\mathcal{K}, s) \models \psi$ iff $s \in G_\psi$ and either there is a non-sink-cycle reachable from s , or there are $k+1$ distinct finite paths from s to states where ψ_2 holds.

Similarly to what has been done for the case of the operator \mathcal{G} , now the algorithm looks for the states in G_ψ from which it is possible to reach a non-sink-cycle (line 25), and then looks for the states from which $k+1$ distinct finite paths start, each ending in states where ψ_2 holds, (line 26). The proof of the correctness of the algorithm can be easily done by induction on the length of the formulas. To complete the proof let us first prove Claim 1.

(if): Let $s \in S_\psi$ and $C = \langle v_0, \dots, v_{h-1} \rangle$ be a cycle in G_ψ reachable from s via a finite path $\langle s, u_0, \dots, u_i, v_0 \rangle$ and containing at least one exit-node, say v_j , $0 \leq j \leq h-1$ connected to a node $w_0 \in S_\psi$ such that $w_0 \neq v_{(j+1) \bmod h}$ and $(v_j, w_0) \in R_\psi$. Since $(\mathcal{K}, w_0) \models E^{>0}\mathcal{G}\psi_1$, there is an infinite path $\langle w_0, w_1, \dots \rangle$ starting from w_0 and satisfying $\mathcal{G}\psi_1$ and there are $k+1$ pairwise distinct infinite paths π_l , $0 \leq l \leq k$, each satisfying $\mathcal{G}\psi_1$, defined as $\pi_l = \langle s, u_0, \dots, u_i, (C)^l, v_0, \dots, v_j, w_0, \dots \rangle$, where $(C)^l$ denotes the fact that π_l cycles l times on C . Thus $(\mathcal{K}, s) \models \psi$. Finally, since a finite path from s to a sink-cycle in G_ψ constitutes an infinite path satisfying $\mathcal{G}\psi_1$, if there are $k+1$ distinct finite paths connecting s to sink-cycles in G_ψ then $(\mathcal{K}, s) \models \psi$.

(only if): If $(\mathcal{K}, s) \models E^{>k}\mathcal{G}\psi_1$ then obviously $(\mathcal{K}, s) \models E^{>0}\mathcal{G}\psi_1$, therefore $s \in S_\psi$. Let us consider $k+1$ distinct infinite paths π_0, \dots, π_k starting from s and satisfying $\mathcal{G}\psi_1$. Since an infinite path on a finite Kripke structure either contains a non-sink-cycle, or ends in a sink-cycle, the claim follows from the fact that each state in π_0, \dots, π_k belongs to S_ψ .

Finally let us now prove Claim 2.

(if): Let $s \in S_\psi$ and $C = \langle v_0, \dots, v_{h-1} \rangle$ be a non-sink-cycle, reachable from s via a finite path $\langle s, u_0, \dots, u_i, v_0 \rangle$. Let v_j , for $0 \leq j \leq h-1$, be an exit-node of C connected to a node $w_0 \in S_\psi$ such that $w_0 \neq v_{(j+1) \bmod h}$ and $(v_j, w_0) \in R_\psi$. Since $(\mathcal{K}, w_0) \models E^{>0}\psi_1\mathcal{U}\psi_2$, then in G_ψ there is a finite path $\langle w_0, \dots, w_r \rangle$ starting from w_0 and ending in a w_r such that $(\mathcal{K}, w_r) \models \psi_2$. Consider the $k+1$ pairwise distinct finite paths π_l , $0 \leq l \leq k$, defined as $\pi_l = \langle s, u_0, \dots, u_i, (C)^l, v_0, \dots, v_j, w_0, \dots, w_r \rangle$, where $(C)^l$ denotes the fact that

π_l cycles l times on C . Since R_ψ does not contain edges out-going from nodes where ψ_1 is not satisfied, then $(\mathcal{K}, x) \models \psi_1$ for all x in π_l , except at most w_r , and therefore each π_l is an *evidence* of $\psi_1 \mathcal{U} \psi_2$. Thus $(\mathcal{K}, s) \models \psi$. Now, let π_0, \dots, π_k be $k + 1$ distinct finite paths connecting s to nodes where ψ_2 holds; from the definition of G_ψ , π_i is an *evidence* of $\psi_1 \mathcal{U} \psi_2$ for all $0 \leq i \leq k$, and therefore $(\mathcal{K}, s) \models \psi$, as well.

(only if): If $(\mathcal{K}, s) \models E^{>k} \psi_1 \mathcal{U} \psi_2$ then obviously $(\mathcal{K}, s) \models E^{>0} \psi_1 \mathcal{U} \psi_2$, therefore $s \in S_\psi$. On the other side, from the semantics of $E^{>k} \psi_1 \mathcal{U} \psi_2$, there are $k + 1$ distinct finite paths starting from s and ending in states satisfying ψ_2 and these are also paths in G_ψ , and this completes the proof of the claim.

Let us now evaluate the running-time of the algorithm. It is easy to see that to check a sub-formula of type $p \in AP, \neg\psi_1, \psi_1 \wedge \psi_2$, requires $\mathcal{O}(|S|)$ and for a sub-formula $E^{>k} \mathcal{X} \psi_1, E^{>0} \mathcal{G} \psi_1, E^{>0} \psi_1 \mathcal{U} \psi_2$ the algorithm requires time $\mathcal{O}(|R|)$. For a sub-formula $E^{>k} \mathcal{G} \psi_1$, note that the set of vertices from which it is possible to reach a non-sink-cycle can be globally calculated in time $\mathcal{O}(|R|)$ by using a Depth First Search algorithm and, as soon as a cycle is detected, checking whether the cycle contains an exit-node. Finally, also the set of vertices from which $k + 1$ paths leading to sink-cycles exist, can be globally calculated in time $\mathcal{O}(|R|)$ by using a standard DFS algorithm. The analysis for $E^{>k} \psi_1 \mathcal{U} \psi_2$ is essentially the same as that of the case $E^{>k} \mathcal{G} \psi_1$. Since the size of $Sub(\varphi)$ is $\mathcal{O}(|\varphi|)$, then the overall complexity of the algorithm is $\mathcal{O}(|R| \cdot |\varphi|)$. \square

An example of Claim 2 is the model in Figure 1 which satisfies the formula $E^{>k} \mathcal{F}(wait1 \wedge EG \neg critic1)$, for all $k \geq 0$, as contains reachable non-sink-cycles (one is depicted with bold-faced edges).

The graded-CTL model-checking can be used to obtain simultaneously more than one counterexample for a formula. For example, consider the formula $A\mathcal{F}p$ expressing a simple liveness property: in all behaviors something good eventually happens. Given a model \mathcal{K} , a counterexample is a path in \mathcal{K} where $\neg p$ always holds. It can be useful to detect whether there are more than a fixed number k of behaviors in which the desired property fails. To get that, we can test whether $(\mathcal{K}, s_{in}) \models E^{>k} \mathcal{G} \neg p$. Analogously, we can consider a safety property expressed by $\neg E\mathcal{F} error$: once fixed a number k , if $(\mathcal{K}, s_{in}) \models E^{>k} \mathcal{F} error$ then there are more than k wrong behaviors, each leading to an error. Note that the algorithm we introduced in Theorem 1 can be modified to return the required counterexamples.

Let us also consider the formula $AG(wait \Rightarrow A\mathcal{F}critic)$ for the access control to a critical section of a system. A counterexample for this formula is an “unfair” path which is an evidence for the formula $E\mathcal{F}(wait \wedge EG \neg critic)$. In this case, it is useful to detect whether the model can generate more bad behaviors. By using graded-CTL model-checking it is possible to analyze three bad situations: the first is to detect whether there are more “unfair” paths from the initial state, by verifying the formula $E^{>k_1} \mathcal{F}(wait \wedge EG \neg critic)$; the second is to verify whether there is a finite path from the initial state to a state where $wait$ holds, from which

more “unfair” paths stem, and this can be done by testing the formula $EF(wait \wedge E^{>k_2} \mathcal{G}\text{-critic})$, or, third, by using the formula $E^{>k_1} \mathcal{F}(wait \wedge E^{>k_2} \mathcal{G}\text{-critic})$.

The following example shows the result of running SMV Cadence and NuSMV for a system model implementing mutual exclusion and having more than one unfair path.

Example 1. Consider the model in Figure 1 which violates the graded-CTL formula $\varphi = A^{\leq 1} \mathcal{G}(wait1 \Rightarrow A \mathcal{F} \text{critic1})$.

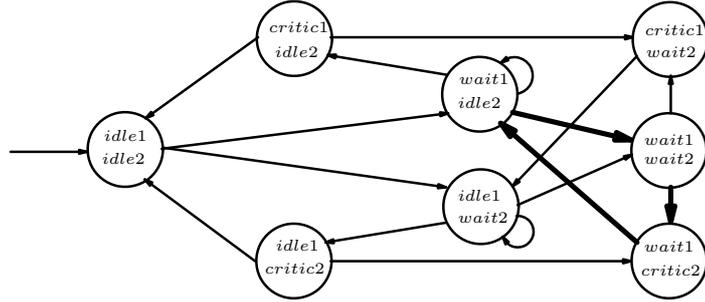


Fig. 1. A mutual exclusion system.

When SMV Cadence (or also NuSMV [McM,CCGR99]) runs on this model and on the classical CTL formula corresponding to φ , then it generates as a counterexample the path:

$$\langle (idle1, idle2), (wait1, idle2), (wait1, idle2), \dots \rangle$$

Then, if the user corrects this error by removing the self-loop on the state labeled $(wait1, idle2)$, the model-checker reports the second path

$$\langle (idle1, idle2), (wait1, idle2), (wait1, wait2), (wait1, critic2), (wait1, idle2), \dots \rangle.$$

4 Edge-Disjoint Graded-CTL Model-Checking

In this section we introduce a different semantics of graded-CTL to distinguish whether different behaviors of the system, satisfying a graded-CTL formula, are completely disjoint. This setting can be applied also to ensure that a “correct” system behavior tolerates a given number of faults of the system.

The edge-disjoint semantics of graded-CTL is given by the relation \models_{ed} , which differs from the previous \models relation only for the formulas of the following two types $E^{>k} \mathcal{G}\psi_1$ and $E^{>k} \psi_1 \mathcal{U}\psi_2$. In these two cases it is required the edge-disjointness of the *evidences*, that is of the infinite paths satisfying $\mathcal{G}\psi_1$ and of the finite paths satisfying $\psi_1 \mathcal{U}\psi_2$. Let us note that the model of Figure 1 does

no longer satisfy the formula $E^{>2}\mathcal{F}(\text{wait1} \wedge \text{EG-critic1})$ now as there are only two disjoint paths that violate the formula.

The **edge-disjoint graded-CTL model-checking** is defined as the problem of determining whether $(\mathcal{K}, s_{in}) \models_{ed} \varphi$, for a Kripke structure \mathcal{K} with initial state s_{in} and a graded-CTL formula φ .

We first prove that the problem is both NP-hard and coNP-hard, and we give an upper bound showing that it lies in PSPACE. Then we introduce a fragment of our logic for which the problem has a polynomial time solution. To show this, we use techniques which are standards for flow network problems, see e.g. [CLRS01]. Finally we give a polynomial time algorithm for the case in which only a given number of single actions of behaviors (edges) must be disjoint and all the others may overlap.

4.1 Complexity

The proof of the hardness is given by a reduction from the Cycle-Packing problem, defined as follows: given a directed graph G and an integer $n \geq 2$, check whether in G there are at least n edge-disjoint cycles. The Cycle-Packing problem is known to be NP-complete (see [CPR03]).

Theorem 2. *The edge-disjoint graded-CTL model-checking problem is both NP-hard and coNP-hard.*

Proof. We first prove that edge-disjoint model-checking problem is NP-hard for specifications in the graded-CTL fragment *FRAG* containing only formulas $E^{>k}\mathcal{G}p$, for an atomic proposition p and $k \geq 0$.

Given a graph $G = (\mathcal{V}, \mathcal{E})$ and an instance (G, n) , $n \geq 2$, of the Cycle-Packing problem, let $\mathcal{K} = (\mathcal{V} \cup \{\hat{s}\}, \hat{s}, R, L)$ be the Kripke structure obtained from G by adding an initial state $\hat{s} \notin \mathcal{V}$, connected to all the other nodes, and by labeling each state of \mathcal{K} with a single atomic proposition p . Formally, \mathcal{K} is defined on the atomic propositions $AP = \{p\}$ in such a way that $R = \mathcal{E} \cup \{(\hat{s}, s) \text{ s.t. } s \in \mathcal{V}\}$ and $L(s) = \{p\}$ for all $s \in \mathcal{V} \cup \{\hat{s}\}$. Moreover, let us consider the graded-CTL formula $\varphi = E^{>n-1}\mathcal{G}p$. Since \hat{s} is connected to each node of G and has no incoming edges, and since p holds in every node, then it follows that $(\mathcal{K}, \hat{s}) \models_{ed} \varphi$ iff G contains at least n edge-disjoint cycles. From the NP-hardness of the Cycle-Packing problem, the edge-disjoint FRAG model-checking problem is NP-hard as well. The edge-disjoint model-checking problem for specifications expressed with formulas of the type $\neg E^{>k}\mathcal{G}p$ hence turns out to be coNP-hard. Thus the theorem holds. \square

From the previous theorem, we have that the edge-disjoint graded-CTL model-checking problem is not in NP (and not in coNP as well) unless NP = coNP. Anyway we now show an upper bound for this problem. In fact let us consider the following simple algorithm to model-check formulas $E^{>k}\theta$ with either $\theta = \mathcal{G}\psi_1$ or $\theta = \psi_1\mathcal{U}\psi_2$: the Kripke structure is visited to find paths satisfying θ and, each time a path is found, a new visit is recursively started, looking for other paths in the remaining graph, until $k + 1$ edge-disjoint paths

are found. This algorithm can be easily implemented by using polynomial space, as the overall size of the $k + 1$ paths is bounded by $|R|$. Therefore we obtain the following theorem.

Theorem 3. *There is an algorithm to solve the edge-disjoint graded-CTL model-checking problem in space $\mathcal{O}(|R| \cdot |S| + |\varphi|)$.*

4.2 A fragment

One question that naturally arises from Theorem 2 is whether it is possible to define interesting fragments of graded-CTL for which the edge-disjoint graded-CTL model-checking problem can be solved in polynomial-time. In particular, the proof of Theorem 2 suggests that only formulas of the type $E^{>k}\mathcal{G}\varphi$, with $k > 0$, are “hard” to verify. In this section we introduce a fragment, called graded-RCTL, of graded-CTL not containing formulas of the type $E^{>k}\mathcal{G}\varphi$, with $k > 0$ and show that for it there is a polynomial-time algorithm for the model-checking problem. Note that the fragment is an extension of CTL and that still many significant properties can be expressed within it. For example, consider the property stating that *do not exist more than k bad behaviors such that a device does not start unless a key is pressed*: such a property can be expressed in graded-RCTL with the formula $\neg E^{>k}(\neg\text{key } \mathcal{U}(\text{start} \wedge \neg\text{key}))$.

Theorem 4. *Let $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ be a Kripke structure and φ be a graded-RCTL formula. The edge-disjoint graded-RCTL model-checking problem, for \mathcal{K} and φ , can be solved in time $\mathcal{O}(|R|^2 \cdot |S| \cdot |\varphi|)$.*

Proof. Since in the graded-RCTL there are no $E^{>k}\mathcal{G}\psi_1$ formulas, we have only to show how to check sub-formulas $E^{>k}\psi_1\mathcal{U}\psi_2$ with $k > 0$. To this aim we will use ideas from flow networks of the graph theory. Let us recall that a *flow network* is a directed graph with a *source* node, a *destination* node, and with edges having a non-negative capacity representing the amount of data that can be moved through the edge. A *maximum flow* from the source to the destination is the maximum amount of data that a network can move from the source to the destination in the time unit.

The algorithm is identical to Algorithm 1 for graded-CTL, with the lines 17–27 rewritten as follows (where $d \notin S$ is the destination node, $inDegree(s)$ returns the in-degree of a state s and $MaxFlow(S, R, c, s, d)$ returns the maximum flow from s to d on the graph (S, R) with c as the capacity function on the edges):

```

17 case  $\psi = E^{>k}\psi_1\mathcal{U}\psi_2$  with  $k > 0$ :
18    $S' \leftarrow \{s \in S \mid s.E^{>0}\psi_1\mathcal{U}\psi_2 = \text{TRUE}\} \cup \{d\}$ ;
19    $R' \leftarrow (R \cap (S' \times S')) \setminus \{(s, t) \mid s.\psi_1 = \text{FALSE}\} \cup \{(s, d) \mid s.\psi_2 = \text{TRUE}\}$ ;
20   forall  $e \in R'$  do  $c(e) = inDegree(s)$  if  $e = (s, d)$  and  $c(e) = 1$  otherwise;
21   forall  $s \in S' \setminus \{d\}$  s.t.  $MaxFlow(S', R', c, s, d) > k$  do  $s.\psi \leftarrow \text{TRUE}$ ;
22 end

```

Our algorithm considers the graph (S', R') , subgraph of \mathcal{K} , of the states where the formula $E^{>0}\psi_1\mathcal{U}\psi_2$ holds (without the edges outgoing from states where ψ_1 doesn't hold), and adds a new destination node d with incoming edges from all the nodes where ψ_2 holds (the capacity of the link (s, d) is the in-degree of s , while the remaining edges have capacity 1). It is known that in graphs with all unitary edge capacities, the maximum flow is equal to the maximum number of edge-disjoint paths from the source to the destination node, see e.g. [CLRS01]. However, it is easy to see that in our network the maximum flow from a node s to d is equal to the maximum number of edge-disjoint paths from s to the set $\{t \in S' \setminus \{d\} \mid (t, d) \in R'\}$, therefore our algorithm has only to evaluate the maximum flow from each state to d .

The running-time of the algorithm on a sub-formula $E^{>k}\psi_1\mathcal{U}\psi_2$ depends on the time required to calculate the maximum flow. Note that the total capacity of the edges entering in d is at most $|R|$, therefore the maximum flow from any state to d is upper bounded by $|R|$. Since in this case, the maximum flow can be calculated in time $\mathcal{O}(|R|^2)$, see e.g. [CLRS01], the overall time complexity of the algorithm is $\mathcal{O}(|R|^2 \cdot |S| \cdot |\varphi|)$. \square

4.3 A parameterized version of the problem

Let $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ and \hat{R} be a subset of R . We say that two paths π_1 and π_2 in \mathcal{K} are \hat{R} -edge-disjoint if there are no edges in \hat{R} belonging to both π_1 and π_2 . We introduce the relation $\models_{ed}^{\hat{R}}$ which differs from the finer relation \models_{ed} only for the formulas of the type $E^{>k}\mathcal{G}\psi_1$ and $E^{>k}\psi_1\mathcal{U}\psi_2$. In particular, we require the existence of $k + 1$ pairwise \hat{R} -edge-disjoint paths satisfying $\mathcal{G}\psi_1$ or $\psi_1\mathcal{U}\psi_2$. Then, the **subset-edge-disjoint graded-CTL model-checking** requires to verify whether $(\mathcal{K}, s_{in}) \models_{ed}^{\hat{R}} \varphi$, for a Kripke structure \mathcal{K} , a set $\hat{R} \subseteq R$, and a graded-CTL formula φ .

The lower bound to this problem obviously matches the lower bound of the edge-disjoint graded-CTL model-checking problem. However, in the following theorem we prove that the problem is *fixed parameter* tractable, in fact we solve it in time exponential only in the size of \hat{R} , obtaining thus a good algorithm for practical cases.

Theorem 5. *Let $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ be a Kripke structure, $\hat{R} \subseteq R$ and φ be a graded-CTL formula. The subset-edge-disjoint graded-CTL model-checking problem can be solved in time $\mathcal{O}((4^{|\hat{R}|} \cdot |R| + 2^{|\hat{R}|^2}) \cdot |S| \cdot |\varphi|)$ and in space $\mathcal{O}(4^{|\hat{R}|} \cdot |\hat{R}| + |R| + |\varphi|)$.*

Proof. Since the difference between graded-CTL and subset-edge-disjoint graded-CTL model-checking is only in the satisfiability of formulas $E^{>k}\theta$, with the path-formula θ being either $\theta = \mathcal{G}\psi_1$ or $\theta = \psi_1\mathcal{U}\psi_2$ and $k > 0$, the algorithm to solve our problem is identical to Algorithm 1, but for the extra input value \hat{R} , and for the lines 17–27 replaced by these:

```

17 case  $\psi = E^{>k}\theta$  with  $\theta = \mathcal{G}\psi_1$  or  $\theta = \psi_1\mathcal{U}\psi_2$  and  $k > 0$ :
18   forall  $s \in S$  do
19      $I \leftarrow \{i \in \{k+1 - |\hat{R}|, \dots, k+1\} \mid i \geq 0 \text{ and } \exists i \text{ distinct paths from } s$ 
      satisfying  $\theta$  without using edges in  $\hat{R}\}$ ;
20     if  $I \neq \emptyset$  then
21        $\hat{k} \leftarrow k+1 - \max\{i \mid i \in I\}$ ;
22       if  $\hat{k} = 0$  then  $s.\psi \leftarrow \text{TRUE}$ ; continue;
23        $\mathcal{V} \leftarrow \{T \mid T \text{ is the set of edges of } \hat{R} \text{ occurring in an evidence of } \theta\}$ ;
24        $\mathcal{E} \leftarrow \{(T, T') \in \mathcal{V}^2 \mid T \cap T' = \emptyset\}$ ;
25       if  $\exists$  a clique with size  $\hat{k}$  in  $(\mathcal{V}, \mathcal{E})$  then  $s.\psi \leftarrow \text{TRUE}$ ;
26     end
27   end
28 end

```

This part of the algorithm works as follows. Consider a state $s \in S$. As the number of \hat{R} -edge-disjoint evidences of θ which use at least one edge belonging to \hat{R} is bounded by $|\hat{R}|$ itself, the number of the remaining evidences of θ (not using edges of \hat{R}) must be greater than $k+1 - |\hat{R}|$ (otherwise $(\mathcal{K}, s) \not\models_{ed}^{\hat{R}} E^{>k}\theta$). Thus the algorithm first determines a number \hat{k} , lines 19–21, with the property that: $(\mathcal{K}, s) \models_{ed}^{\hat{R}} E^{>k}\theta$ if and only if there are \hat{k} \hat{R} -edge-disjoint evidences of θ which use at least one edge belonging to \hat{R} . Then the graph $(\mathcal{V}, \mathcal{E})$, described in lines 23 and 24, is computed, such that a vertex in \mathcal{V} is a set of edges of \hat{R} which occur in an evidence of θ in \mathcal{K} and an edge in \mathcal{E} connects two disjoint such sets. Thus, $(\mathcal{K}, s) \models_{ed}^{\hat{R}} E^{>k}\theta$ if and only if in $(\mathcal{V}, \mathcal{E})$ there is a clique of size \hat{k} .

Let us evaluate the running time and the space required by the algorithm. Since the set I described in line 19 is such that $|I| \leq |\hat{R}|$, the lines 19–21 can be easily computed in time $\mathcal{O}(|R| \cdot |\hat{R}|)$ by using a simple variation of Algorithm 1. Moreover, for a given subset T of \hat{R} , the existence of an evidence of θ which uses *all* the edges in T and possibly edges of $R \setminus \hat{R}$, can be verified in time $\mathcal{O}(|R|)$, while the set of edges outgoing from T can be computed in time $\mathcal{O}(2^{|\hat{R}|} \cdot |\hat{R}|)$; therefore the graph $(\mathcal{V}, \mathcal{E})$ can be computed in time $\mathcal{O}(4^{|\hat{R}|} \cdot |R|)$. Finally, the existence of a clique of size $\hat{k} \leq |\hat{R}|$ can be verified in time $\mathcal{O}(2^{|\hat{R}|^2})$.

The algorithm needs, to model-check a formula $E^{>k}\theta$ in a state $s \in S$, space $\mathcal{O}(4^{|\hat{R}|} \cdot |\hat{R}|)$ to store the graph $(\mathcal{V}, \mathcal{E})$ and space $\mathcal{O}(|R|)$ to calculate the path needed to verify whether a non-empty subset T of \hat{R} is in \mathcal{V} . Moreover, the algorithm globally needs only $3 \cdot |S|$ truth values for the sub-formulas (two for the operands and one for the operator in each state). Therefore the space required by the algorithm is $\mathcal{O}(4^{|\hat{R}|} \cdot |\hat{R}| + |R| + |\varphi|)$. \square

This algorithm can be modified to fit in polynomial space. Thus we have the following theorem (whose full proof is in the extended version of the paper [FNP08]).

Theorem 6. *Let $\mathcal{K} = \langle S, s_{in}, R, L \rangle$ be a Kripke structure, $\hat{R} \subseteq R$ and φ be a graded-CTL formula. The subset-edge-disjoint graded-CTL model-checking problem can be solved in time $\mathcal{O}(2^{|\hat{R}|^2} \cdot |\hat{R}| \cdot (|R| + |\hat{R}|^2) \cdot |S| \cdot |\varphi|)$ and space $\mathcal{O}(|\hat{R}|^2 + |R| + |\varphi|)$.*

5 Discussion

In this paper we have introduced the graded-CTL as a more expressive extension of classical CTL. The results presented are in the model-checking setting with specifications in this logic. We have investigated the complexities involved in various scenarios, all from a theoretical perspective. One possible future direction to work on, is to verify in practice whether an existing model-checker tool could be augmented with these grading modalities, retaining the usual performances. We believe that this framework could turn out to be useful also in the verification of fault tolerant physical properties of networks.

As said in the introduction, in [KSV02] the satisfiability problem has been studied for the graded μ -calculus obtaining the same complexity as for the non-graded logic. We have investigated the problem in our setting of graded CTL (reported in the extended version of the paper [FNP08]) and have proved that it is EXPTIME-complete, when integers are represented in unary.

Another theoretical aspect to investigate is also with respect to the Linear Temporal Logic LTL. Also here this framework is a strict extension of the standard logic, anyway a straightforward algorithm to solve the model-checking problem, seems here to involve the values representing the graded modalities.

Finally let us mention a drawback of our setting. As said in the introduction the generation of more than one counterexample is highly desirable, anyway the analyze stage (of the realization process of a system) is critical also for the size of the counterexamples and the poor human-readability of it.

References

- [CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.
- [CG07] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *Int. J. Softw. Tools Technol. Transf.*, 9(5):429–445, 2007.
- [CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
- [CIW⁺01] F. Coptly, A. Irron, O. Weissberg, N.P. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 275–292, London, UK, 2001. Springer-Verlag.
- [CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms - Second Edition*. The MIT Press, 2001.

- [CPR03] A. Caprara, A. Panconesi, and R. Rizzi. Packing cycles in undirected graphs. *Journal of Algorithms*, 48(1):239–256, 2003.
- [CV03] E.M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, pages 208–224, 2003.
- [DRS03] Y. Dong, C.R. Ramakrishnan, and S.A. Smolka. Model checking and evidence exploration. In *ECBS*, pages 214–223, 2003.
- [Fin72] K. Fine. In so many possible worlds. *Notre Dame Journal of Formal Logic*, 13(4):516–520, 1972.
- [FNP08] A. Ferrante, M. Napoli, and M. Parente. Graded CTL. *In preparation*, 2008.
- [GMV99] H. Ganzinger, C. Meyer, and M. Veanes. The two-variable guarded fragment with transitive relations. In *LICS*, pages 24–34, 1999.
- [GOR97] E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *LICS*, pages 306–317, 1997.
- [HB91] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *KR*, pages 335–346, 1991.
- [Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [KSV02] O. Kupferman, U. Sattler, and M.Y. Vardi. The complexity of the graded μ -calculus. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 423–437, London, UK, 2002. Springer-Verlag.
- [McM] K. McMillan. The Cadence smv model checker. <http://www.kenmcmil.com/psdoc.html>.
- [MFG04] R. Meolic, A. Fantechi, and S. Gnesi. Witness and counterexample automata for ACTL. In *FORTE '04: Proceedings of 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, pages 259–275, 2004.
- [PST00] L. Pacholski, W. Szwast, and L. Tendera. Complexity results for first-order two-variable logic with counting. *SIAM Journal of Computing*, 29(4):1083–1117, 2000.
- [QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
- [Tob01] Stephan Tobies. PSPACE reasoning for graded modal logics. *Journal Log. Comput.*, 11(1):85–106, 2001.

Social bugs communities and Intelligent Agents: an experimental architecture

Stefania Costantini, Mario Scotti Del Greco, and Arianna Tocchio

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{stefcost,tocchio}@di.univaq.it

Abstract. In this paper, we present an architecture composed of agents equipped with different degrees of 'intelligence' and we experiment how the distribution of intelligence can influence the global behavior of a community of agents. The architecture integrates DALI agents and IBM Aglets, two platforms that differ both in technology and in capabilities. As a case-study and experimentation scenario, we have interpreted our architecture as an artificial beehive and we have modeled some of the possible behaviors of the bees.

1 Introduction

Intelligence can be defined as the capability of an agent to adapt its own behavior to different contexts and situations. One might wonder which "degree" of intelligence is necessary in order to perform a certain task or to react to a specific stimulus. One would expect that the kind of intelligence which is required depends on the kind of task or stimulus to cope with. Human beings are able to spontaneously modulate their responses according to various parameters, among which the difficulty of a task and the relative importance attributed to its completion.

In the case of software entities put at work in dynamic environments, the problem of which is the degree of intelligence which is adequate in a context can be relevant. In fact, intelligence generally implies some kind of reasoning and reasoning requires computational resources. Intelligence modulation can be in some sense obtained by adopting several agents and by assigning them different roles according to specialization criteria. Examples of this approach are the ARCHON and YAMS industrial applications [1]. In YAMS, in particular, a manufacturing enterprise is modeled as a hierarchy of workcells. There will be, for example, workcells for milling, lathing, grinding, painting, and so on. These workcells will be further grouped into flexible manufacturing systems (FMS), each of which will provide a functionality such as assembly, paint spraying, buffering of products, and so on. This choice however is not always adequate. In fact, if e.g. agents having high reasoning and pro-active capabilities perform tasks requiring reactive abilities only, a profusion of computational power is uselessly wasted.

Starting from this consideration, we decided to explore the possibility of modulating the intelligence by adopting different kinds of agents, having different

capabilities and characteristics. In doing this, we selected two agent frameworks. The high-level framework is DALI [2],[3], [4], [5], [6], a MAS that allows for logic agents having reactive, pro-active and communicative capabilities. These agents are able to reason by exploiting the potentialities of logic. A lower-level framework is that of IBM Aglets [7], chosen for their simplicity and for the mobility of entities. The architecture resulting from the integration of these frameworks allowed us to have different 'intelligence' degrees for facing with a certain flexibility the environment challenges.

The problem of investigating how to distribute roles among agents for reaching an optimal 'intelligence' modulation, imposed us to find a scenario to test our system. Reflecting on some properties of social insect colonies, we decided to 'interpret' our MAS as a beehive in which some bees have more reasoning capabilities, some less. The roles that required more "intelligence" were mainly delegated to DALI agents due to their reasoning and learning abilities, while roles requiring communicative and reactive abilities were delegated to Aglets considering their mobility and social nature.

In particular, proactivity of DALI agents allowed us to introduce in the artificial colony an entity capable of running all activities crucial for the community life like, for example, the planning of food supplies or the generation of new individuals useful for the community. Reactivity and mobility of Aglets suggested that their ideal job was that of searching for food or defending the colony. According to this main scenario, we studied two different backgrounds concerning the distribution of roles among the two typologies of agents to see how the different distribution of the intelligence could influence the global efficiency of the community. This with the aim to find the 'best' intelligence distribution.

In this initial experimentation phase of the project we are still not able to formally assess which is the "winner" configuration. However, the second scenario, where some capabilities have been distributed to IBM Aglets, seems to behave better. We are currently performing other experiments for reaching a more clear idea about how intelligence distribution can help an agent society to reach its goals. To the best of our knowledge, no architecture exists similar to the one presented in this paper. In the view of the integration between artificial social bugs colonies and robotics, we cite the work of Bonabeau et al. in [8]. Authors simulate some aspects of the bugs life such as cooperation in transport and nests building through algorithms and robotic models.

The rest of this paper is structured as follows. In section 2 we describe some basic features of the architecture. In section 3 we present some scenarios through which we have experimented our framework. Section 4 presents conclusions and outlines future work.

2 Architecture

The integration of DALI agents and IBM Aglets has requested the development of a communication channel between the frameworks. Though SICSTUS Prolog offers among its libraries API's capable of interfacing Sicstus with JAVA, we have

chosen to implement a new interface. The reasons have been related on the one hand to reducing computational overheads (the API's are quite heavy and slow) and on the other hand to the purpose of obtaining a flexible general-purpose inter-agent communication level not linked to specific formalisms.

The architecture that we have implemented in view of the beehive case-study (figure 1) is composed of two macro systems, one called “Beehive” and the other one called “Remote Context”. In the beehive we inserted:

- DALI framework;
- IBM Aglet framework;
- communication interface.

The communication interface is composed of two communication channels, from DALI to Aglets and vice versa. It works by means of sockets, the best choice in our setting; in fact, despite this architectural choice has introduced an overhead for both agents platforms, we have guaranteed scalability and modularity. In fact, the artificial colony can run on several hosts, exploiting the cooperation of different machines located in different places.

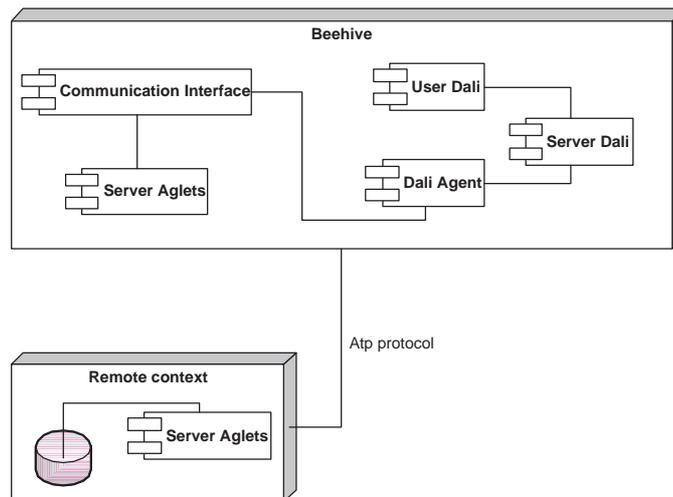


Fig. 1. The artificial colony architecture

Messages coming from one framework and directed to another one are stored in a particular structure of the communication interface and kept until the destination entity receives them. This guarantees that if a bug is in a remote context (e.g. for searching food), messages addressed to it are not lost. The remote context represents the external world and it may contain other colonies. The integra-

tion of the different DALI and Aglets contexts through such a communication interface allows the colony to get a representation of the global environment through the perception capabilities of both DALI agents and Aglets.

3 Experimentation scenarios

We have considered simplified scenarios of interaction among members of an artificial colony. In the scenarios, specific members of the colony, through a goal-reasoning process, propagate a particular need inside the colony. We identified as particular members of the beehive: the queen, implemented in DALI language; the “courtier” bee, which is a role that we have introduced in order to possibly relieve the queen from some of its responsibilities; the foragers, implemented through Aglets.

We suppose that the colony needs to produce a certain quantity of honey, for which a corresponding quantity of pollen is required. The user sets the needs of the colony: in fact, she sends a message to the DALI queen asking for some quantity of honey. Notice that “honey” can be seen as the metaphor of any product/goal of an agent society, and pollen signifies any kind of resource to be gathered in order to fulfill the needs.

The difference between the two scenarios is mainly in the degree of interaction, intelligence and sharing among the components. In the first scenario, all managerial and intelligent roles are reserved to the queen that organizes the collectivity for accomplishing the identified necessities. In the second scenario, the need for honey is faced by introducing a different role inside the society, the courtier. In this manner, the managerial role reserved only to the queen in the previous scenario is distributed. Both scenarios show that the common effort of different entities such as the DALI intelligent agents and the mobile IBM Aglets succeeds in producing coordination in order to reach a common goal.

3.1 First scenario

In this scenario, a beehive in which the role of the queen is carried out from a DALI agent is presented. The queen monitors the escorts of the beehive by means of opportune periodic controls. It has to face the consumption demands of the beehive that, in this context, represents the particular environmental demands. It manages the search and the harvest of new provisions through the collaboration with the Aglet agents (Bees).

Each single bee has therefore the goal to satisfy all of the explicit demands for honey coming from the queen. After reaching its objective, the bee sends a success message if it found the requested type of honey; else, it sends a failure message. In this way the queen, on the basis of the result, establishes whether to send other bees in quest of pollen in order to produce more honey.

In particular, tasks that we reserved for the queen in this scenario are: the management of the supplies, the management of requests of users about honey, the organization of gathering and research of pollen. To satisfy a request, the

queen generates a certain number of foragers indicating remote contexts where to find pollen. When foragers bees come back to the beehive, they send a message to the queen with the result. In case of success, the queen updates the escorts and finally satisfies the initial request.

3.2 Second scenario

In the second scenario, we force the DALI agent to share part of the responsibility with a particular Aglet agent that we call the “Courtier”. The DALI queen delegates to the Aglet courtier the task of organizing bees for producing the requested honey. In this way, the DALI queen does not directly manage the harvest and the search, but it takes advantage of some collaboration disregarding lower level aspects. In this way, the highest hierarchies of the colony can avoid the consumption of time and resources to spend in the harvest and in the search: therefore, they can devote attention to other more high-level aspects. The Courtier coordinates the supply requests through the foragers. Each bee coming from the remote context, at the end of its task sends the result to the Courtier, that will evaluate it. When all bees sent by the Courtier have come back, the Courtier notifies the DALI queen about the result. In summary, the roles of DALI queen and Courtier have been distributed as follows: DALI queen manages the escorts and, according to the users requests of honey, generates the courtier and assigns it the goal. The Aglet courtier manages the gathering via foragers and, when possible, communicates the results to the queen.

3.3 Comparison, Initial Evaluation and Perspective

The number of experiments that we have performed so far does not allow us to establish a general statement. However, the second scenario seem to behave better in the sense that the same quantity of honey is produced in less time. The results encourage us to further experiment hybrid architectures where however ‘intelligence’ is somehow distributed among levels.

In fact, our experiments will follow two directions: “horizontal” distribution, i.e., in the case-study, introducing more Courtiers. Issues of this solution are how many Courtiers to insert, and whether or not they should cooperate; “vertical” distribution, i.e., introduction of new intermediate roles. A suitable compromise should be found, so that the cost of communication does not overcome the advantages. An interesting challenge is that of identifying a metric for comparing the different solutions.

4 Conclusions and future work

We presented a framework where the integration of agents having different capabilities allowed us experiment how the roles and “intelligence” distribution can influence the global system behavior. To this aim, we used DALI intelligent

agents and mobile IBM Aglet agents, having strong social abilities. The artificial community was able to benefit from some characteristics such as, mobility, reactivity, goal-directed reasoning and learning proper of the software entities that compose it.

Future perspectives could be oriented toward the analysis of other relevant behaviors of social colonies such as, for instance, the defense of the nest or the politics of enlarging the population. In the former scenario, we could introduce new figures inside the community suitable to preserve the integrity of the nest, while in the latter one some strategy on managing the population could be implemented.

An interesting development perspective could be orientated to the involvement of other typologies of mobile agents, such as e.g. JADE agents, inside the community. Their introduction could be exploited in the sense of encouraging the specialization of the individuals inside the community, delegating or sharing out again roles of mobility between the Aglets and JADE.

An important future direction is to establish, also by means of experiments, which is the “best” mix of intelligence between high-level e low-level agents in a given context. This in order to better exploit, in perspective, the potential of such hybrid architectures.

References

1. Jennings, N.R., Wooldridge, M.J.: Applications of intelligent agents. In Jennings, N.R., Wooldridge, M.J., eds.: Agent Technology: Foundations, Applications, and Markets. Springer-Verlag: Heidelberg, Germany (1998) 3–28
2. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
3. Costantini, S., Tocchio, A.: The dali logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
4. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: Declarative Agent Languages and Technologies. LNAI 3229. Springer-Verlag, Berlin (2006) Post-Proc. of DALT 2005.
5. Costantini, S., Tocchio, A., Verticchio, A.: A game-theoretic operational semantics for the dali communication architecture. In: Proc. of WOA04, Turin, Italy, December 2004, ISBN 88-371-1533-4. (2004)
6. Tocchio, A.: Multi-agent systems in computational logic. Ph.D. Thesis, Dipartimento di Informatica, Università degli Studi di L’Aquila (2005)
7. : Ibm aglets web site (2008)
8. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm intelligence: from natural to artificial systems. Oxford University Press, Inc., New York, NY, USA (1999)

Increasing Parallelism while Instantiating ASP Programs^{*}

F. Calimeri, S. Perri, and F. Ricca

Dipartimento di Matematica, Università della Calabria, 87036 Rende (CS), Italy
{calimeri, perri, ricca}@mat.unical.it

Abstract. One of the most hard tasks performed by Answer Set Programming (ASP) systems is instantiation, which consists of generating variable-free programs equivalent to those given as input. The efficiency of this task is crucial for ASP systems performance especially in case of real-world applications where huge inputs are processed.

We recently proposed a method that exploits the capabilities of multi-processor machines for the instantiation. This method confirmed to be effective especially when dealing with programs consisting of many rules. Here, we report some preliminary results on a rewriting-based strategy that makes the existing technique exploitable even in case of programs with few rules.

1 Introduction

In the last few years, multi-core/multi-processor architectures become standard, thus making Symmetric MultiProcessing (SMP) [1] common also for entry-level systems and PCs. In SMP architectures two or more identical processors connect to a single shared main memory, enabling simultaneous multithread execution. Such technology might be profitably exploited also in the field of Answer Set Programming (ASP): indeed, recent applications of ASP in different emerging areas (see e.g., [2–8]), have evidenced the practical need for faster and scalable ASP systems.

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [9–15] which features a high declarative nature combined with a relatively high expressive power [16, 17]; unfortunately, this comes at the price of a high computational cost. The kernel modules of ASP systems work on a ground instantiation of the input program. Thus, an input program \mathcal{P} first undergoes the so-called instantiation process, which produces a program \mathcal{P}' semantically equivalent to \mathcal{P} , but not containing any variable. This phase is computationally very expensive; thus, having an efficient instantiation procedure is, in general, a key feature of ASP systems.

In [18] we have proposed a technique for the parallel instantiation of ASP programs, allowing the performance of instantiators to be improved by exploiting the power of multiprocessor computers. The technique takes advantage of some structural properties of input programs in order to reduce the usage of mutex-locks [1], and thus the

^{*} Supported by M.I.U.R. within projects “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva” and “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione.”

time spent by concurrency-control mechanisms. The strategy focuses on two different aspects of the instantiation process: on the one hand, it examines the structure of the input program \mathcal{P} , splits it into modules and, according to the interdependencies between the modules, decides which of them can be processed in parallel; on the other hand, it parallelizes the evaluation within each module. The proposed strategy has been implemented into the instantiator module of the ASP system DLV [16], thus obtaining a parallel ASP instantiator. This new system is effective especially in the evaluation of programs consisting of several rules with a large amount of input data [18].

Here we present the basic principles concerning a rewriting-based strategy that aims at improving the system performance even when dealing with programs consisting of few rules. Basically, input programs are rewritten in such a way that the instantiation of each rule is split into different jobs that can be done in parallel. Results of a preliminary experimental activity are also presented.

2 Parallel Instantiation of ASP Programs

In this Section the parallel instantiation algorithm of [18], which relies on the DLV (“standard”) instantiation procedure, is briefly described. A detailed discussion about the DLV instantiator and the details of the parallel instantiation technique are out of the scope of this short paper; for further insights we kindly refer the reader to [16, 18, 19].

Roughly, the instantiation module of DLV splits up a given program \mathcal{P} into sub-programs called *modules*. Each of these modules corresponds to a strongly connected component (SCC) of a particular graph, called *dependency graph* ($G_{\mathcal{P}}$), which, intuitively, describes how predicates depend on each other. The DLV instantiator processes them, one at a time, according to a (partial) ordering induced by $G_{\mathcal{P}}$, which ensures that all data needed for the instantiation of a module have been already generated by the instantiation of the modules preceding it. A procedure called *InstantiateComponent* is in charge of instantiating modules, while a procedure called *InstantiateRule* builds all the ground instances of a given rule r . A single call to *InstantiateRule* is sufficient for completely evaluating non-recursive rules, while recursive ones are processed several times according to a semi-naïve evaluation technique [20].

The procedure presented in [18] combines two strategies: the first one for the parallel evaluation of different modules, while the second for the concurrent instantiation of rules within a module. Both strategies avoid the use of mutex-locks: the former by properly choosing the modules to be evaluated in parallel; the latter by suitably parallelizing each iteration of the semi-naïve algorithm. The idea is that if there are no two threads in read/write (nor write/write) conflict on the same data structure, then no synchronization is needed. This allows one to drastically reduce the so-called parallel overhead.

Parallelizing the Program Instantiation. The parallel instantiation of an input program \mathcal{P} is based on classical producer-consumers pattern. A *manager* thread (producer) identifies the components that can be processed at a given time, and delegates their instantiation to a number of *instantiator* threads (consumers). The choice of the components to be processed in parallel is made according to the above-mentioned partial ordering. Intuitively, a component C from the bunch of components to be instantiated

is given by the manager to the instantiators only if all the information needed has already been computed.

Parallelizing the Instantiation of a Program Module. Within a single module, each rule is processed by one thread. First, all non-recursive rules are concurrently evaluated, then, as soon as all of them are done, recursive ones are processed. In particular, at the end of each single iteration of the semi-naïve algorithm, instantiators synchronize in such a way that common structures (like, e.g. current partial interpretation) can be safely updated by the manager, and next iteration starts.

3 Parallelization of Rule Instantiation: Ideas and Experiments

The technique described above makes parallel the execution of two different steps of the instantiation process: the evaluation of program modules and the instantiation of rules within each module. However, it is not fully exploitable in case of programs with few components and few rules. Consider, for instance, the following disjunctive encoding for the well-known 3-Colorability problem:

$$\begin{aligned} (r) \quad & col(X, red) \vee col(X, yellow) \vee col(X, green) :- node(X). \\ (c) \quad & :- col(X, C), col(Y, C), edge(X, Y). \end{aligned}$$

Predicates *node* and *edge* represent the input graph; rule (*r*) guesses the possible colorings of the graph, and the constraint (*c*) imposes that two adjacent nodes cannot have the same color. In this case, the technique proceeds by first instantiating *r*, and then by processing the constraint *c* only once the extension of *col* has been computed.

Thus, such encoding does not allow the existing technique to make the evaluation parallel at all. However, as it is easy to see, one may provide different encodings (with more rules) for the same problem, which are more amenable for the technique. In general, this would require the user to know *how* the evaluation process work, while writing a program: clearly, such a requirement is not desirable for a declarative system. Nevertheless, an automatic rewriting of the input program into an equivalent one, whose evaluation can be made more parallel, could make transparent this optimization process to the user. For instance, the following is an alternative encoding for the 3-Colorability problem which can be obtained by automatically rewriting the original one:

$$\begin{aligned} (r) \quad & col(X, red) \vee col(X, yellow) \vee col(X, green) :- node(X). \\ (c_1) \quad & :- col(X, C), col(Y, C), edge1(X, Y). \\ (c_2) \quad & :- col(X, C), col(Y, C), edge2(X, Y). \end{aligned}$$

The set of edges is *split up* into two (equally sized) subsets, represented by predicates *edge1* and *edge2*. The evaluation of constraints (*c*₁) and (*c*₂) is equivalent to the evaluation of the original constraint *c*, but the computation now can be carried out in parallel by two different instantiators. Obviously, this rewriting strategy can be straightforwardly extended for allowing more than two instantiators to work in parallel.

This rewriting technique somewhat coincides with the so-called *Or-parallelism* [21–23], which is here simulated by splitting obtained via rewriting, without a drastic and involved modification of a system implementation. In general, this idea allows one to “split” any encoding; but there are different, sometimes many, ways to rewrite a program. For instance, another possible encoding for 3-Colorability could be obtained by

splitting predicate *color* into $color_1 \dots color_n$.¹ Or, if possible, one can also consider to split also two (or more) body predicates, like both *color* and *edge*; in this case, the rewritten program requires also a number of rules, the body of each containing a join between $color_i$ and $edge_j$, $1 \leq i, j \leq n$. The choice of the most convenient is not trivial, and must be made according to several factors. For instance, the instantiation exploits, for the evaluation of each rule, clever techniques based on join ordering [20, 24] and backjumping [25]. A “bad” split might reduce or neutralize the benefits provided by these techniques, thus making the overall time consumed by the parallel evaluation not optimal (and, in some corner case, even worse than the time required to instantiate the original encoding). Intuitively, the join ordering techniques establish the body order according to several facts, as the (estimated) size (number of instances) of body predicates. While rewriting a rule r , according to the split of a body predicate p into p_1, \dots, p_n , a number of rules r_1, \dots, r_n is obtained with the same shape as r , but with body predicates p_1, \dots, p_n smaller in size w.r.t. p . Thus, the body orderings of r_1, \dots, r_n may differ from the one of r , possibly significantly affecting the instantiation time. These considerations are confirmed by some experiments reported in the following section.

Experiments. In order to check the viability of the rewriting for increasing parallelism and to evaluate the effects of different splits on performance, we have carried out some preliminary experiments. In particular, we considered three well-known problems, whose standard encodings in disjunctive ASP are not suitable for parallel evaluation with the technique of [18], namely 3-Colorability, Reachability (compute the transitive closure of a given graph), and Same Generation (given a parent-child relationship, i.e. a tree, find pairs of persons belonging to the same generation). For each problem we analyzed different splits, but for space reason we refrain from reporting the corresponding encodings here.

We assessed our encodings by exploiting the parallel instantiator of [18] on a machine equipped with two Intel Xeon HT (single core) processors clocked at 3.60GHz. In particular we compared a single-threaded grounding engine (*ST* in the table) against a multi-threaded grounding engine (*MT*).² Table 1 reports, for each problem, the average instantiation time spent by the two engines (each experiment has been repeated five times), for two different instances. For each problem we tested the standard encoding against two different rewritten programs; these are based on the split of predicates *edge* and *color*, respectively, for 3-Colorability, and *edge* and *node* for Reachability; for Same Generation we split on *edge* and considered two different body orderings. The results clearly show that *MT* always outperforms *ST* while instantiating split encodings, with a gain close to 50% (the best one can obtain from a two-processor machine), while, as expected, the standard encodings do not enjoy any gain. Moreover, as discussed before, different splits produce very different behaviors. Indeed, while the two splits for Reachability lead to comparable results, for 3-Colorability and Same Generation the scenario changes. For instance, splitting *color* instead of *edge* in 3-Colorability

¹ Note that, differently from *edge*, *color* is not an input predicate; splitting on it requires to split the predicates it depends on and generate new rules accordingly.

² The maximum number of allowed concurrent instantiator threads was set to the number of simultaneous (i.e., executed in a different CPU) threads/processes allowed by the hardware.

	3Colorability			Reachability			Same Generation		
	<i>NoSplit</i>	<i>Edge</i>	<i>Color</i>	<i>NoSplit</i>	<i>Edge</i>	<i>Node</i>	<i>NoSplit</i>	<i>Edge₁</i>	<i>Edge₂</i>
<i>ST</i>	46.4	45.3	202.5	15.8	14.3	14.7	45.11	377.9	22.9
<i>MT</i>	46.5	23.6	104.5	15.6	7.8	8.2	45.20	197.9	15.6
<i>ST</i>	124.2	128.6	544.3	375.2	272.7	281.7	2680.1	31996.8	449.5
<i>MT</i>	129.4	66.6	278.1	374.3	139.9	144.1	2674.8	16369.1	274.9

Table 1. Average Grounding Times (s).

nullifies the advantages of parallel computation, and the overall time becomes higher than the one required by the original encoding. In addition, the results for Same Generation clearly show how a split can interfere with the join ordering technique, thus leading to unexpected side-effects: splitting *edge* caused a noticeable worsening (column *Edge₁* of Table1), but a more deep analysis allowed to discover that the times can be drastically reduced by simply changing the body ordering (column *Edge₂*).

Summarizing, the splitting strategy produces encodings that fully enjoy the parallel technique (*MT* gets a 50% off against *ST*); importantly, by carefully choosing the predicate(s) to split, the technique presents noticeable gains when compared to non-parallel instantiation (*MT* with best split halves times against *ST* with original encoding).

Conclusion Preliminary experiments confirmed that the “split” rewriting is viable, but cannot be applied trivially. We plan to develop heuristics that allow for selecting the best predicate to split, in order to implement a smart rewriter that can be seamlessly integrated in our parallel instantiator. This will lead to take further advantage of multi-processing, especially in the case of program encodings containing very few rules.

References

1. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
2. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kařka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszki, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of SIGMOD 2005, Baltimore, Maryland, USA, ACM Press (2005) 915–917
3. Lembo, D., Lenzerini, M., Rosati, R.: Source Inconsistency and Incompleteness in Data Integration. In: Proceedings of KRDB-02, Toulouse, France, CEUR Electronic Workshop Proceedings <http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-54/> (2002)
4. Soininen, T., Niemelä, I.: Developing a Declarative Rule Language for Applications in Product Configuration. In Gupta, G., ed.: Proceedings of PADL’99. Volume 1551 of Lecture Notes in Computer Science., Springer (1999) 305–319
5. Aiello, L.C., Massacci, F.: Verifying security protocols as planning in logic programming. ACM Transactions on Computational Logic **2**(4) (2001) 542–580
6. Bertino, E., Mileo, A., Proveti, A.: User Preferences VS Minimality in PDDL. In Buccafurri, F., ed.: Proceedings of the Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003. (2003) 110–122

7. Buccafurri, F., Caminiti, G.: A Social Semantics for Multi-agent Systems. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: LPNMR'05, Diamante, Italy. Volume 3662 of LNCS., Springer Verlag (2005) 317–329
8. Costantini, S., Tocchio, A.: The dali logic programming agent-oriented language. In Alferes, J.J., Leite, J., eds.: Proceedings of JELIA 2004. Volume 3229 of LNAI., Springer Verlag (2004) 685–688
9. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. *ACM Transactions on Database Systems* **22**(3) (1997) 364–418
11. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers (2000) 79–103
12. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: *Proceedings of ICLP'99*, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
13. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In Apt, K.R., Marek, V.W., Truszczyński, M., Warren, D.S., eds.: *The Logic Programming Paradigm – A 25-Year Perspective*. Springer Verlag (1999) 375–398
14. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
15. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective. *Artificial Intelligence* **138**(1–2) (2002) 3–38
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* **7**(3) (2006) 499–562
17. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys* **33**(3) (2001) 374–425
18. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. *Journal of Algorithms in Cognition, Informatics and Logic* (2008) To appear. Available at <http://dx.doi.org/10.1016/j.jalgor.2008.02.003>.
19. Faber, W., Leone, N., Perri, S., Pfeifer, G.: Efficient Instantiation of Disjunctive Databases. Technical Report DBAI-TR-2001-44, Institut für Informationssysteme, Technische Universität Wien, Austria (2001) Online at <http://www.dbai.tuwien.ac.at/local/reports/dbai-tr-2001-44.pdf>.
20. Ullman, J.D.: *Principles of Database and Knowledge Base Systems*. Computer Science Press (1989)
21. Clark, K.L., Gregory, S.: Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Language Systems* **8**(1) (1986) 1–49
22. Ramakrishnan, R.: Parallelism in Logic Programs. *Annals of Mathematics and Artificial Intelligence* **3**(2–4) (1991) 295–330
23. Leone, N., Restuccia, P., Romeo, M., Rullo, P.: Expliciting Parallelism in the Semi-Naive Algorithm for the Bottom-up Evaluation of Datalog Programs. *Database Technology* **4**(4) (1993) 245–158
24. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In Eiter, T., Faber, W., Truszczyński, M., eds.: LPNMR'01, Vienna, Austria. Volume 2173 of LNAI., Springer Verlag (2001) 280–294
25. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: *Proceedings of NMR 2004*, Whistler, BC, Canada. (2004) 258–266

Experimental comparison of two tableau-based decision procedures for MLSS^{*}

Domenico Cantone, Rosario Terranova, and Pietro Ursino

*Università di Catania, Dipartimento di Matematica,
Viale A. Doria, 6, I-95125 Catania, Italy*

e-mail: cantone@dmi.unict.it
trunks@videobank.it
ursino@dmi.unict.it

Abstract. We report the experimental results relative to two tableau-based decision procedures for the fragment **MLSS** of set theory. Efficient implementations of a decision procedure for **MLSS** are particularly relevant for the proof-verifier *ÆtnaNova/Referee*, which uses it as its main inference rule.

Keywords: Decision procedures for fragments of set theory; multi-level syllogistic; proof verification.

1 Introduction

The field of *Computable Set Theory*, now about thirty years old, deals with the decision problem for fragments of set theory. It started with the seminal paper [FOS80] which provided a first decision procedure for the so-called *Multi-level syllogistic (with singleton)*, **MLSS** for short. In brief, **MLSS** is the unquantified fragment of set theory involving the set predicates = and \in , the Boolean set operators \cup, \cap, \setminus and the singleton operator $\{\bullet\}$, as well as the propositional connectives (we will review **MLSS** at length below). Partly due to the inadequacy of the computer technology available at that time, the initial emphasis on the design and implementation of a proof verifier based on the set-theoretic formalism [CS87] shifted over time to the more theoretical goal of restricting the boundary between the decidable and the undecidable in set theory. A large compendium of the results obtained up to 2001 can be found in the monographs [CFO89] and [COP01].

In the late ninety's, the reformulation in terms of decidable *tableau calculi* of the decision procedure for **MLSS** and other fragments of set theory brought a renewed interest to the more pragmatics aspects of Computable Set Theory (cf. [Can97,CZ00]; see also [CFOS03]).

Much at the same time, a completely new implementation of the “old” *ETNA* system (cf. [CF95]), called *ÆtnaNova/Referee*, was carried out by Jack Schwartz

^{*} Research partially supported by PRIN project 2006/07 “Large-scale development of certified mathematical proofs” n. 2006012773.

at New York University with some contribution from his collaborators at the Universities of Catania and Trieste; see [OS02,COSU03,OCPS06] and the URL

http://www.settheory.com/Set12/Ref_user_manual.html.

In parallel, the compilation of a large proof scenario, aimed primarily at the verification of the proof of the Cauchy's integral theorem from the barest rudiments of set theory, began to take place [SCO08].

The main inference mechanism in the *ÆtnaNova/Referee* verifier is the ELEM primitive, which is based on a decision procedure for an extension of **MLSS** with function symbols. It is therefore of great practical relevance to have a high efficient implementation of the decision test for **MLSS** (and its extensions). For such a reason, we carried out an experimental comparison of the two most efficient tableau-based decision procedures for **MLSS**, namely those presented in [Can97] (based on the interleaving of deduction and checking steps) and in [CZ00] (based on the the system KE of Mondadori-D'Agostino [DM94], which forces tableau branches to be mutually exclusive).

With some surprise, we found out that despite the frequent semantical checks characterizing the decision procedure in [Can97], this turned out to be always more efficient than the one presented in [CZ00], upon a variant of which the ELEM inference mechanism in the system *ÆtnaNova/Referee* is currently based. We therefore expect that *ÆtnaNova/Referee* will benefit very much from the reimplementaion of its ELEM rule.

The paper is organized as follows. In the next section we review the syntax and semantics of **MLSS** as well as the notion of *realization*, which is of central importance to [Can97]. Subsequently, in Section 3, we review the tableau-based decision procedures for **MLSS** presented in [Can97] and [CZ00]. Then, in Section 4, we present and comment the results of our tests. Finally, we draw our conclusions in Section 5.

2 Multi-level syllogistic (MLSS)

The unquantified set-theoretic fragment **MLSS** contains

- a denumerable infinity of variables,
- the constant \emptyset (empty set),
- the operator symbols \cup (union), \cap (intersection), \setminus (set difference), and $\{\bullet\}$ (singleton),
- the predicate symbols \in (membership) and $=$ (equality), and
- the logical connectives \neg , \wedge and \vee .

The expression $\{t_1, t_2, \dots, t_k\}$ can be regarded as a shorthand notation for the term $\{t_1\} \cup \dots \cup \{t_k\}$; likewise, we can regard the expression $s \subseteq t$ as a shorthand notation for the literal $s \cup t = t$. In other words, both the finite enumeration operator “ $\{\bullet, \bullet, \dots, \bullet\}$ ” and the predicate symbol “ \subseteq ” can be expressed in **MLSS**.

The semantics of **MLSS** is based upon the von Neumann standard cumulative hierarchy \mathcal{V} of sets defined by:

$$\begin{aligned} \mathcal{V}_0 &= \emptyset \\ \mathcal{V}_{\alpha+1} &= \mathcal{P}(\mathcal{V}_\alpha), \quad \text{for each ordinal } \alpha \\ \mathcal{V}_\lambda &= \bigcup_{\mu < \lambda} \mathcal{V}_\mu, \quad \text{for each limit ordinal } \lambda \\ \mathcal{V} &= \bigcup_{\alpha \in On} \mathcal{V}_\alpha, \end{aligned}$$

where $\mathcal{P}(S)$ is the power set of S and On denotes the class of all ordinals. It can easily be seen that there can be no membership cycle in \mathcal{V} , namely sets in \mathcal{V} are well-founded with respect to membership.

An **ASSIGNMENT** M over a collection V of variables is any function $M : V \rightarrow \mathcal{V}$. Given an assignment M over the variables of a formula φ , we denote with $M\varphi$ the truth-value obtained by interpreting each variable v in φ with the set Mv and the set symbols and logical connectives according to their standard meaning.

A **SET MODEL** for a formula φ is an assignment M over the collection of variables occurring in φ such that $M\varphi$ evaluates to true.

A formula φ is **SATISFIABLE** if it has a set model.

The **DECISION** (or **SATISFIABILITY**) **PROBLEM** for **MLSS** is the problem of establishing whether any given formula of **MLSS** has a set model.

Though not strictly necessary, it is convenient to limit our analysis to **MLSS**-formulae of a special form, namely to conjunctions of normalized literals each of which has one of the following types

$$x \in y, x \notin y, x = y, x \neq y, z = x \cup y, z = x \cap y, z = x \setminus y, x = \{y_1, \dots, y_k\}, \quad (1)$$

where x, y, z, y_1, \dots, y_k are individual variables. We call such formulae *normalized MLSS-conjunctions* (see [Can97] for a proof of the equivalence between the satisfiability problems for **MLSS**-formulae and for normalized **MLSS**-conjunctions).

2.1 Realizations

Both the tableau calculi for **MLSS**, which will be reviewed in the next section, try to derive all membership relations among terms, variables, and newly introduced parameters, which are logical consequences of the given premisses. A “minimal” assignment satisfying all the membership relations lying on a non-contradictory branch ϑ is given by the *realization* of its associated membership graph $G_\vartheta = (N, \hat{\in})$, where, up to an equivalence relation induced by equality literals,

- N is the collection of newly introduced parameters, P , and (terms or) variables, T , in the branch ϑ , and
- $\hat{\in}$ is a binary relation on N such that $x \hat{\in} y$ holds for x, y in N if and only if the literal $x \in y$ is present in ϑ .

A precise definition of realization, which plays a particularly important rôle for the tableau calculus in [Can97], is given next (we also define the useful notion of *height* of a node).

Definition 1. Let $G = (N, \widehat{\epsilon})$ be a directed acyclic graph, and let (P, T) be a bipartition of N . Also, let $\{u_x : x \in P\}$ be a family of sets.

The REALIZATION of $G = (N, \widehat{\epsilon})$ relative to $\{u_x : x \in P\}$ and to (P, T) is the assignment R over N defined recursively by:

$$\begin{aligned} Rx &= \{u_x\}, & \text{for } x \text{ in } P \\ Rt &= \{Rs : s \widehat{\epsilon} t\}, & \text{for } t \text{ in } T. \end{aligned}$$

The HEIGHT function $h : N \rightarrow \mathbb{N}$ is defined recursively by

$$h(t) = \begin{cases} 0 & \text{if } t \in P \text{ or } s \not\widehat{\epsilon} t, \text{ for all } s \in N \\ \max\{h(s) : s \widehat{\epsilon} t\} + 1 & \text{otherwise.} \end{cases}$$

Observe that since G is acyclic, the realization R and the height function h are well-defined.

3 Two Tableau Calculi for MLSS

In this section we review two tableau calculi for **MLSS** and the decision procedures derived from them, closely following [Can97] and [CZ00]. For a complete introduction to the subject of semantic tableau, the reader is referred to [Fit96].

Given a (generic) tableau calculus T_{MLSS} for (normalized conjunction of) **MLSS**, we recall the main notions related to T_{MLSS} -tableaux (these will be soon instantiated to the two tableau calculi of our interest).

Definition 2. Let φ be a normalized **MLSS**-conjunction. An INITIAL T_{MLSS} -TABLEAU for φ is a one-branch tree whose nodes are labeled by the literals in φ .

A T_{MLSS} -TABLEAU for φ is a tableau labeled with **MLSS**-literals which can be constructed from the initial tableau for φ by a finite number of applications of the rules in T_{MLSS} .

For simplification purposes, we assume that no literal can occur more than once on any given branch, namely we assume that a rule which would add a literal already present in a branch has no effect.

Definition 3. Let \mathcal{T} be a T_{MLSS} -tableau for a given normalized **MLSS**-conjunction φ . A branch ϑ of \mathcal{T} is said to be

- STRICT, if no rule has been applied more than once on ϑ to the same literal occurrences;
- SATURATED WITH RESPECT TO A GIVEN RULE, if the rule has been applied at least once to each instance of its premisses on ϑ ;
- SATURATED WITH RESPECT TO A GIVEN COLLECTION OF RULES, if ϑ is saturated with respect to each rule of the collection;
- CLOSED, if ϑ contains
 - a set of literals of the form $x \in x_1 \in \dots \in x_n \in x$, for some variables x, x_1, \dots, x_n with $n \geq 0$, or
 - a pair of complementary literals $X, \neg X$;
 - a literal of the form $t \neq t$, or

- a literal of the form $s \in \emptyset$;
- SATISFIABLE, if there exists a set model for the literals occurring on ϑ ; any such model will be called a SET MODEL FOR ϑ .

A tableau \mathcal{T} is said to be

- ANNOTATED, if some information is stored on its branches and/or nodes;
- STRICT, or SATURATED WITH RESPECT TO A SET OF RULES, or CLOSED, if such are all its branches;
- SATISFIABLE, if at least one of its branches is satisfiable.

Remark 1. We shall make use of the following notation. Let ϑ be an open branch of a tableau \mathcal{T} for a normalized **MLSS**-conjunction φ , relative to a given calculus $\mathsf{T}_{\mathbf{MLSS}}$. We define the following objects:

- V_φ : is the collection of the variables occurring in φ ;
- T_φ : is the collection of the terms occurring in the formula φ ;
- T_ϑ : is the collection of the terms occurring in the formulae in the branch ϑ ;
- P_ϑ : is the collection of parameters occurring on ϑ other than V_φ ;
- \sim_ϑ : is the equivalence relation induced on $T_\vartheta \cup P_\vartheta$ by equality literals $x = y$ in ϑ ;
- P'_ϑ : is the set $\{t \in P_\vartheta : t \not\sim_\vartheta x, \text{ for all } x \in V_\varphi\}$;
- V'_ϑ : is the set $V_\varphi \cup (P_\vartheta \setminus P'_\vartheta)$;
- $\hat{\in}_\vartheta$: is the binary relation on $V'_\vartheta \cup P'_\vartheta$ defined by

$$x \hat{\in}_\vartheta y \quad \text{iff} \quad \text{the literal } x \in y \text{ is in } \vartheta, \text{ for } x, y \in V'_\vartheta;$$
- G_ϑ : is the oriented graph $(V'_\vartheta \cup P'_\vartheta, \hat{\in}_\vartheta)$ (this is also referred to as the *membership graph* relative to ϑ);
- R_ϑ : is a realization of G_ϑ relative to the partition $(V'_\vartheta, P'_\vartheta)$ and to pairwise distinct sets u_t , for $t \in P'_\vartheta$, each having cardinality no less than $|V'_\vartheta \cup P'_\vartheta|$.
- M_ϑ : is the assignment over V_ϑ defined by $M_\vartheta v = R_\vartheta v$, for each v in V_ϑ . □

Next we review the two tableau calculi for **MLSS** of our interest. These will be referred to, respectively, as **MLSS-oracle tableau calculus** and as **MLSS-KE-tableau calculus**.

3.1 The MLSS-oracle tableau calculus

The rules of the **MLSS-oracle tableau calculus**, introduced in [Can97], are listed in Table 1. Observe that the rules (3), (9), (11), and (12) cause branch splittings. In particular, rule (12) is the only one which introduces new parameters, and therefore must be handled with particular care. This is done as follows (cf. procedures *MLSS-Oracle-Test* and *MLSS-Oracle-Saturate* in Tables 2 and 3 below).

Given a normalized **MLSS**-conjunction φ , firstly a tableau for φ is built by applying the rules (1)-(11) in all possible ways (obviously it is convenient to delay the application of the splitting rules as much as possible). This is the *deduction phase*. At this point either we have a closed tableau, in which case φ can be declared “unsatisfiable,” or there is still some open branch. In the latter

case, we switch to the *model checking phase* by picking an open branch ϑ and testing whether the realization R_ϑ associated to its membership graph satisfies the branch itself (and therefore our input formula φ). If this is the case, the formula φ can plainly be declared “satisfiable.” Otherwise there will exist two literals of the form $x \in z$ and $y \notin z$ on ϑ such that $R_\vartheta x = R_\vartheta y$ and therefore either $R_\vartheta x \notin R_\vartheta z$ or $R_\vartheta y \in R_\vartheta z$. In this case we apply the splitting rule (12) with the variables x and y , and go back to the *deduction phase* (notice that the realization R_ϑ has been used as an *oracle* to pinpoint among all possible pairs of variables to which one could apply rule (12) one which is guaranteed to give a nonnull contribution towards the search for a proof or a countermodel).

The above process continues until one ends either with a closed tableau or with a satisfiable branch. The interleaving of deduction and checking steps has the overall effect to speed up the saturation process. Termination, completeness, and soundness of the test just outlined can be found in the original source [Can97].

Remark 2. Each branch ϑ in the tableau \mathcal{T} constructed by procedures *MLSS-Oracle-Test*(φ) and *MLSS-Oracle-Saturate*(\mathcal{T}, V) is annotated with the attribute *Diversified* $_\vartheta$. Roughly speaking, the set *Diversified* $_\vartheta$ collects all pairs (x_1, x_2) of variables in V_φ for which there exists another variable $u \in V_\varphi \cup P_\vartheta$ such that $u \in x_i$ and $u \notin x_{3-i}$ are in ϑ , for some $i \in \{1, 2\}$. Notice that if all variables of a lesser height than that of x_i are “diversified”, then it can be shown that for any realization R_ϑ of G_ϑ relative to sufficiently large and pairwise disjoint sets u_t , for $t \in P'_\vartheta$, one has $R_\vartheta x_1 \neq R_\vartheta x_2$.

In view of what has just been said, it follows that the set Δ_ϑ defined in procedure *MLSS-Oracle-Test* consists of all those pairs (x_1, x_2) of variables which are mapped into the same set by the realization R_ϑ , though they would need to be diversified. Therefore, the subsequent assertion affirms that if at any moment the branch ϑ is not closed and R_ϑ does not satisfy ϑ , then there must exist a pair of variables (x_1, x_2) which need to be diversified but which do not yet belong to the set *Diversified* $_\vartheta$. \square

3.2 The MLSS-KE-tableau calculus

The **MLSS-KE**-tableau calculus, introduced in [CZ00], is based upon the KE system of Mondadori and D’Agostino [DM94], which forces tableau branches to be mutually exclusive, getting in favorable cases an exponential speed-up over Smullyan tableau-based calculi.

The **MLSS-KE**-tableau calculus has two kinds of rules: *saturation* and *fulfilling* rules.

The collection of saturation rules is given in Table 4. We impose the restriction that *no new term will be created by any application of a saturation rule*. Thus, for instance, the rule

$$s \in t_1 \implies s \in t_1 \cup t_2$$

$\frac{z = y \cup y' \quad x \in y}{x \in z} \quad (1)$	$\frac{z = y \cup y' \quad x \in y'}{x \in z} \quad (2)$	$\frac{z = y \cup y' \quad x \in z}{x \in y \mid x \in y'} \quad (3)$
$\frac{z = y \cap y' \quad x \in z}{x \in y \quad x \in y'} \quad (4)$	$\frac{z = y \cap y' \quad x \in y \quad x \in y'}{x \in z} \quad (5)$	$\frac{x = y \setminus y' \quad w \in x}{w \in y \quad w \notin y'} \quad (6)$
$\frac{x = y \setminus y' \quad w \in y \quad w \notin y'}{w \in x} \quad (7)$	$\frac{y = \{x_1, \dots, x_k\}}{x_1 \in y \quad \vdots \quad x_k \in y} \quad (8)$	$\frac{y = \{x_1, \dots, x_k\} \quad z \in y}{z = x_1 \mid \dots \mid z = x_k} \quad (9)$
$\frac{x = y \quad \psi}{\psi\{y/x} \quad \psi\{x/y}} \quad (10)^a$	$\frac{x = y \setminus y' \quad w \in y}{w \in y' \mid w \notin y'} \quad (11)$	$\frac{}{w \in x \mid w \notin x \quad w \notin y \mid w \in y} \quad (12)^b$
<hr style="width: 20%; margin-left: 0;"/> ^a $\psi\{y/x\}$ is obtained by replacing in ψ all occurrences of y by x . ^b w must be a new variable not occurring on the branch to which the rule is applied.		

Table 1. MLSS-oracle tableau calculus

can be applied to a branch ϑ of a tableau for φ *only* if the term $t_1 \cup t_2$ is already in T_φ . Notice also that in the first two rules for equality, ℓ stands for a literal, and that in order to prevent the search space from exploding the following further restriction applies: $\ell\{t/u\}$ is the result of substituting any *top level* occurrence of t by the term u .

A branch is said to be LINEARLY SATURATED if no saturation rule can produce any new formula.

A fulfilling rule can be applied to an open linearly saturated branch, provided that its associated *precondition* and *subsumption requirement* are, respectively, true and false. Table 5 summarizes the fulfilling rules and their associated preconditions and subsumption requirements. Notice that even fulfilling rules are not allowed to introduce new terms, with the exception of the last one, which introduces a fresh parameter x not occurring in the branch to which it is applied.

```

Procedure MLSS-Oracle-Test( $\varphi$ );
  Comment:  $\varphi$  is a normalized MLSS-conjunction.
  let  $V_\varphi$  be the collection of variables occurring in  $\varphi$ ;
  let  $\mathcal{T}$  be a one branch tableau induced by  $\varphi$ ;
  for  $\vartheta \in \mathcal{T}$  do
     $Diversified_\vartheta := \emptyset$ ;
  end for;
   $\mathcal{T} := \text{MLSS-Oracle-Saturate}(\mathcal{T}, V_\varphi)$ ;
  while there exists a non-closed branch  $\vartheta$  in  $\mathcal{T}$  do
    let  $P_\vartheta, \sim_\vartheta, P'_\vartheta, \hat{\in}_\vartheta, G_\vartheta, R_\vartheta$  be defined as in Remark 1;
    if  $R_\vartheta$  satisfies  $\vartheta$  then
      return "input formula is satisfied by  $R_\vartheta$ ";
    else
      put  $\Delta_\vartheta = \{(w, w') \in V_\varphi \times V_\varphi : R_\vartheta w = R_\vartheta w' \text{ and } w \in y, w' \notin y \text{ are in } \vartheta, \text{ for some } y \in V_\varphi\}$ ;

      Assert: the set  $\Delta_\vartheta \setminus Diversified_\vartheta$  is non-empty;

      pick a pair  $(x, y) \in \Delta_\vartheta \setminus Diversified_\vartheta$ ;
      apply rule (12) to the pair of variables  $x, y$ , namely
      - let  $u$  be the next unused variable;
      - split the branch  $\vartheta$  in two branches  $\vartheta_1$  and  $\vartheta_2$ , where
         $\vartheta_1 = \vartheta; u \in x, u \notin y$ ;
         $\vartheta_2 = \vartheta; u \notin x, u \in y$ ;
      - and put:
         $Diversified_{\vartheta_1} := Diversified_{\vartheta_2}$ 
         $:= Diversified_\vartheta \cup \{(x', y'), (y', x') \in V_\varphi \times V_\varphi : x' \sim_\vartheta x \text{ and } y' \sim_\vartheta y\}$ ;
      assign to  $\mathcal{T}$  the resulting tableau;
       $\mathcal{T} := \text{MLSS-Oracle-Saturate}(\mathcal{T}, V_\varphi)$ ;
    end if;
  end while;
  return "input formula is unsatisfiable, as proved by the closed tableau  $\mathcal{T}$ ";
end procedure.

```

Table 2. Procedure *MLSS-Oracle-Test*

Table 6 shows a terminating saturation strategy for **MLSS-KE**-tableaux, which results in a decision procedure for **MLSS**. Notice that the normalization process is taken care directly by the rules of the **MLSS-KE**-tableau calculus.

The reader will find complete details on the soundness, completeness, and termination of the decision procedure outlined in Table 6 in the original paper [CZ00].

4 Experimental results

Using the integrated development environment **NetBeans**, we have developed a tool in **Java**, which allows to compare from an experimental point of view the space and time complexities of the decision procedures **MLSS-Oracle-Test** and **MLSS-KE-Test** for **MLSS** described in the preceding section.

Our tool, which is about 7,800 lines of code, includes a user-friendly interface and a random generator of **MLSS**-formulae and propositional formulae as well¹ of assigned sizes and variable densities, based on the **Java** method `Math.Random`.

¹ In fact, the inferential core of our tool comprises also a **KE**-tableau based decision procedure for propositional logic.

<p>Procedure <i>MLSS-Oracle-Saturate</i>(T, V); <i>Comment:</i> T is an annotated MLSS-tableau, with attributes <i>To_be_diversified</i>$_{\vartheta}$ and <i>Diversified</i>$_{\vartheta}$, for each of its branches ϑ. V is a subset of the variables occurring in T.</p> <ul style="list-style-type: none"> - strictly saturate the non-closed branches of T with respect to rules (1)–(11) of Table 1; - let T be the resulting annotated tableau, where it is supposed that during saturation the attributes are inherited by the new branches; <p>for each branch ϑ of T do <i>To_be_diversified</i>$_{\vartheta} :=$ $\{(x_1, x_2) \in V \times V : x_i \in z \text{ and } x_{3-i} \notin z \text{ are in } \vartheta, \text{ for some } z \text{ in } V$ and for some $i \in \{1, 2\}$, or $x_1 \neq x_2$ is in $\vartheta\}$; <i>Diversified</i>$_{\vartheta} := \{(x, y) \in V \times V : (x', y') \in \text{Diversified}_{\vartheta}, \text{ for some } x' \sim_{\vartheta} x, y' \sim_{\vartheta} y\}$; end for; return T;</p> <p>end procedure.</p>
--

Table 3. Procedure *MLSS-Oracle-Saturate*

We chose to carry on our experimental comparisons using *normalized MLSS*-conjunctions, rather than general **MLSS**-formulae, since the normalization phase would have involved just propositional transformations, whereas our focus was on the complexity of pure set reasoning. To this purpose, we notice that even the satisfiability problem for normalized **MLSS**-conjunctions is NP-complete (see [COP90]).

We have performed a few hundred tests on **MLSS**-formulae of various densities (but most tests have been done on **MLSS**-formulae with 7 literals and 10 distinct variables). For each test, we collected the number of branches, of nodes and the height of the two tableaux constructed, as well as the time needed for their saturation.

Remarkably, the **MLSS-Oracle-Test** decision procedure turned out to perform better than the **MLSS-KE-Test** decision procedure at *each* single test and *both* in space and time.

In particular, the decision procedure **MLSS-Oracle-Test** produced tableaux whose average number of branches, number of nodes, and height were, respectively, about 1/2, 1/9, and 1/5 than in the tableaux produced by the decision procedure **MLSS-KE-Test**. And the difference in running time was even more remarkable: the decision procedure **MLSS-Oracle-Test** turned out to be about 37 times faster than **MLSS-KE-Test** on our random tests.

In the case of unsatisfiable formulae containing memberships chains, the superiority of the **MLSS-Oracle-Test** turned out to be even more impressive, always characterized by running times about one thousand times smaller than those of the **MLSS-KE-Test**.

Though such tests have no statistical relevance, due to their fewness, nevertheless the uniformity of their results strongly hints at a clear superiority of the **MLSS-Oracle-Test** over the **MLSS-KE-Test**.

It comes with no surprise that the decision procedure **MLSS-Oracle-Test** generates smaller tableaux, since its “oracle” (based on the realization of the branch’s membership graph, cf. Definition 1) hints always at rules whose application is “necessary” in some sense.

propositional rules		rules for \cup
$p \wedge q \implies p, q$		$s \notin t_1 \cup t_2 \implies s \notin t_1, s \notin t_2$
$\neg(p \vee q) \implies \neg p, \neg q$		$s \in t_1 \implies s \in t_1 \cup t_2$
$p \vee q, \neg p \implies q$		$s \in t_2 \implies s \in t_1 \cup t_2$
$p \vee q, \neg q \implies p$		$s \in t_1 \cup t_2, s \notin t_1 \implies s \in t_2$
$\neg(p \wedge q), p \implies \neg q$		$s \in t_1 \cup t_2, s \notin t_2 \implies s \in t_1$
$\neg(p \wedge q), q \implies \neg p$		$s \notin t_1, s \notin t_2 \implies s \notin t_1 \cup t_2$
rules for \cap		rules for \setminus
$s \in t_1 \cap t_2 \implies s \in t_1, s \in t_2$		$s \in t_1 \setminus t_2 \implies s \in t_1, s \notin t_2$
$s \notin t_1 \implies s \notin t_1 \cap t_2$		$s \notin t_1 \implies s \notin t_1 \setminus t_2$
$s \notin t_2 \implies s \notin t_1 \cap t_2$		$s \in t_2 \implies s \notin t_1 \setminus t_2$
$s \notin t_1 \cap t_2, s \in t_1 \implies s \notin t_2$		$s \notin t_1 \setminus t_2, s \in t_1 \implies s \in t_2$
$s \notin t_1 \cap t_2, s \in t_2 \implies s \notin t_1$		$s \notin t_1 \setminus t_2, s \notin t_2 \implies s \notin t_1$
$s \in t_1, s \in t_2 \implies s \in t_1 \cap t_2$		$s \in t_1, s \notin t_2 \implies s \in t_1 \setminus t_2$
rules for $\{\bullet\}$		rules for equality
$\implies t_1 \in \{t_1\}$		$t_1 = t_2, \ell \implies \ell\{t_2/t_1\}$
$s \in \{t_1\} \implies s = t_1$		$t_1 = t_2, \ell \implies \ell\{t_1/t_2\}$
$s \notin \{t_1\} \implies s \neq t_1$		$s \in t, s' \notin t \implies s \neq s'$

Table 4. Saturation rules.

fulfilling rule	precondition	subsumption requirement
$\frac{}{p \mid \neg p}$	$p \vee q$ is in ϑ	p is in ϑ or $\neg p$ is in ϑ
$\frac{}{\neg p \mid p}$	$\neg(p \wedge q)$ is in ϑ	$\neg p$ is in ϑ or p is in ϑ
$\frac{}{s \in t_1 \mid s \notin t_1}$	$s \in t_1 \cup t_2$ is in ϑ	$s \in t_1$ is in ϑ or $s \notin t_1$ is in ϑ
$\frac{}{s \in t_2 \mid s \notin t_2}$	$t_1 \cap t_2 \in T_\varphi$ $s \in t_1$ is in ϑ	$s \in t_2$ is in ϑ or $s \notin t_2$ is in ϑ
$\frac{}{s \in t_2 \mid s \notin t_2}$	$t_1 \setminus t_2 \in T_\varphi$ $s \in t_1$ is in ϑ	$s \in t_2$ is in ϑ or $s \notin t_2$ is in ϑ
$\frac{}{x \in t_1 \mid x \notin t_1 \mid x \notin t_2 \mid x \in t_2}$	$t_1, t_2 \in T_\varphi$ $t_1 \neq t_2$ is in ϑ	$\exists y : (y \in t_1 \text{ is in } \vartheta \text{ and } y \notin t_2 \text{ is in } \vartheta)$ or $\exists y : (y \notin t_1 \text{ is in } \vartheta \text{ and } y \in t_2 \text{ is in } \vartheta)$

Table 5. Fulfilling rules.

<p>Procedure <i>MLSS-KE-Test</i>(φ); <i>Comment:</i> φ is an <i>MLSS</i>-formula.</p> <ol style="list-style-type: none"> 1. let \mathcal{T} be the initial tableau for φ; 2. linearly saturate \mathcal{T} by strictly applying to it all possible saturation rules from Table 4 until either \mathcal{T} is closed or no new formula can be produced; 3. if \mathcal{T} is closed, announce that φ is unsatisfiable; 4. otherwise, if there exists an open branch ϑ in \mathcal{T} whose subsumption requirements are all fulfilled, announce that φ is satisfied by the model M_ϑ; 5. otherwise, let ϑ be an open branch; apply to ϑ any fulfilling rule from Table 5 whose subsumption requirement is false and go to step 2.

Table 6. Procedure *MLSS-KE-Test*

However, prior to performing the tests reported in the present note, we thought that the time needed for constructing a new realization by far would overcome the time needed by procedure *MLSS-KE-Test* to apply its deduction rules. It turns out that the syntactic checks that must be performed especially in the case of fulfilling rules (see Table 5) have a great impact on the overall running time of the procedure *MLSS-KE-Test*. In fact, we believe that such an observation could be established in a theoretical setting.

5 Conclusions

We have compared experimentally two tableau-based decision procedures for the unquantified fragment of set theory *MLSS*, upon which the main inference rule *ELEM* of the verifier system *ÆtnaNova/Referee* is based.

It turned out that the procedure *MLSS-Oracle-Test*, which interleaves checking steps with a deduction phase, in practice is much more efficient than the procedure *MLSS-KE-Test*, which is based on the tableau calculus *KE*, despite the fact that the realization of the membership graph of a branch must be constructed at each checking step.

We believe that the procedure *MLSS-Oracle-Test* can be further speeded up by suitably importing in it some tableau rules from the *KE*-approach to force mutual exclusion among branches.

Also, we expect that additional efficiency can be gained by using appropriate data structures which allow, among other things, dynamic computations of acyclicity tests (cf. [Rod03]) and dynamic updates of the membership graph realizations on growing branches.

References

- [Can97] D. Cantone. A fast saturation strategy for set-theoretic tableaux. In D. Galmiche, editor, *Proceedings of the International Conference on Theorem Proving with Analytic Tableaux and Related Methods - TABLEAUX '97 (Abbaye des Prémontrés, Pont-à-Mousson, France, May 13–16, 1997)*, volume 1227 of *Lecture Notes in Artificial Intelligence*, pages 122–137, Berlin, May 13-16 1997. Springer-Verlag.

- [CF95] D. Cantone and A. Ferro. Techniques of computable set theory with applications to proof verification. *Comm. Pure App. Math.*, 48(9-10):1–45, 1995. Special Issue in honor of J. T. Schwartz.
- [CFO89] D. Cantone, A. Ferro, and E. G. Omodeo. *Computable Set Theory. Vol. 1*. Oxford University Press, 1989. Int. Series of Monographs on Computer Science.
- [CFOS03] D. Cantone, A. Formisano, E. G. Omodeo, and J. T. Schwartz. Various commonly occurring decidable extensions of multi-level syllogistic. In S. Ranise and C. Tinelli, editors, *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning 2003 - PDPAR'03 (Miami, USA, July 29, 2003)*, pages 2–14, 2003.
- [COP90] Domenico Cantone, Eugenio G. Omodeo, and Alberto Policriti. The automation of syllogistic. II: Optimization and complexity issues. *Journal of Automated Reasoning*, 6(2):173–187, June 1990.
- [COP01] D. Cantone, E. G. Omodeo, and A. Policriti. *Set theory for computing - From decision procedures to declarative programming with sets*. Monographs in Computer Science. Springer-Verlag, New York, 2001.
- [COSU03] D. Cantone, E. G. Omodeo, J. T. Schwartz, and Pietro Ursino. Notes from the logbook of a proof-checker's project. In Nachum Dershowitz, editor, *Proceedings of the International Symposium on Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 182–207, Berlin, 2003. Springer-Verlag.
- [CS87] D. Cantone and J. T. Schwartz. *A set-theoretically based proof verifier*. Proposal to the National Science Foundation, 1987.
- [CZ00] D. Cantone and C. G. Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *Lecture Notes in Artificial Intelligence*, pages 127–137. Springer-Verlag, 2000.
- [DM94] M. D'Agostino and M. Mondadori. The taming of the cut. Classical refutations with analytic cut. *Journal of Logic and Computation*, 4(3):285–319, June 1994.
- [FOS80] A. Ferro, E. G. Omodeo, and J. T. Schwartz. Decision Procedures for Elementary Sublanguages of Set Theory I. Multilevel Syllogistic and Some Extensions. *Communications on Pure and Applied Mathematics*, 33:599–608, 1980.
- [Fit96] M. C. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 2nd edition, 1996. 1st ed., 1990.
- [OCPS06] E. G. Omodeo, D. Cantone, A. Policriti, and J. T. Schwartz. A Computerized Referee. In M. Schaerf and O. Stock, editors, *Reasoning, Action and Interaction in AI Theories and Systems – Essays dedicated to Luigia Carlucci Aiello*, volume 4155 of *Lecture Notes in Artificial Intelligence*, pages 117–139. Springer Berlin/Heidelberg, 2006.
- [OS02] E. G. Omodeo and J. T. Schwartz. A 'theory' mechanism for a proof-verifier based on first-order set theory. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond – Essays in honour of Bob Kowalski, Part II*, volume 2408 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2002.

- [Rod03] Liam Roditty. A faster and simpler fully dynamic transitive closure. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 404–412, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
- [SCO08] J. T. Schwartz, D. Cantone, and E. G. Omodeo. *Computational logic and set theory: The systematic application of formalized logic to the foundations of analysis*. Texts in Computer Science. Springer-Verlag, 2008. To appear.

DECIDABILITY OF THE $\exists^*\forall^*$ CLASS FOR SET THEORY

EUGENIO OMODEO AND ALBERTO POLICRITI

ABSTRACT. As is well-known, the Bernays-Schönfinkel-Ramsey class of all prenex $\exists^*\forall^*$ -sentences which are valid in classical first-order logic is decidable. This paper shows that an analogous result holds when the only available predicate symbols are \in and $=$, no constants or function symbols are available, and one moves inside a (rather generic) set theory whose axioms yield the well-foundedness of membership and the existence of infinite sets.

Key words: Satisfiability decision algorithms, extended multilevel syllogistics, computable set theory.

INTRODUCTION

In this paper we prove the decidability of the satisfiability problem for the celebrated *Bernays-Schönfinkel-Ramsey class* (*BSR-class*) in a (rather generic) set-theoretic context. The BSR-class has a history for inspiring deep combinatorial issues and results—Ramsey theorem above all, see [Ram28]. The set theoretic context in which we are tackling the satisfiability problem is one more case in which a non-trivial (infinitary) combinatorial treatment turns out to be necessary to prove decidability.

Definition 1. A prenex sentence Φ belongs to the BERNAYS-SCHÖNFINKEL-RAMSEY CLASS (BSR-CLASS) if its quantificational prefix has the form $\exists^*\forall^*$.

An open formula $\varphi(x_1, \dots, x_n)$ all of whose quantifiers are grouped at the beginning belongs to the BSR-CLASS if its quantificational prefix has the form \forall^* .¹ \square

The reader is referred to [BGG97, DG79, Lew79] for basic notions on the taxonomy of quantificational classes and key results on the Classical Decision Problem.

The basic language we consider contains equality and one binary relational symbol \in to be interpreted as the *membership* relation. The set-theoretic satisfiability issue we will consider concerns the existence of an algorithm which can establish, given a formula $\varphi(x_1, \dots, x_n)$ in the BSR-class, whether or not there are n sets satisfying φ in the *standard* (von Neumann) universe \mathbb{V} . This is the class of all sets inductively defined on all ordinals by the recursion

$$\mathbf{V}_\alpha = \bigcup_{\beta < \alpha} \mathcal{P}(\mathbf{V}_\beta),$$

Date: June 10, 2008.

This research has been partially funded by PRIN project 2006/2007 ‘*Large-scale development of certified mathematical proofs*’.

This paper has been submitted for publication to a scientific journal.

¹Occasionally, e.g. when speaking of *restricted* BSR-formulae, we will slightly abuse terminology and ascribe to the BSR-class also formulae which are trivially equivalent to formulae in the BSR-class proper, as can be shown by elementary syntactic transformations.

where $\mathcal{P}(\cdot)$ designates the power-set operation and α, β range over ordinals. Obviously, $\mathbb{V}_0 = \emptyset$. As for \mathbb{V} , it is the union of all sets \mathbb{V}_α , taken over all ordinals α .

Our answer to the said decision problem will be affirmative: actually, we will single out a specific algorithm solving the satisfiability problem addressed above. Related results were presented in [OPP93, OPP96], which treated a decision algorithm for the subclass $\exists^*\forall$ (with only one universal quantifier) of the BSR-class, and in [BP06], which solved the decision problem for the subclass $\exists^*\forall\forall$ (with two universal quantifiers).

With our approach, decidability will ensue from the following observations:

- Within the BSR-class, we can associate with any formula φ an equi-satisfiable formula φ' whose quantifiers are of the restricted form $\forall y \in z$.
- This restricted form of quantification enables one, when the set values for the free variables of φ' are drawn from a family \mathcal{F} , to evaluate φ' within a confined domain of discourse: the *transitive closure* (see below) T of \mathcal{F} .
- There is a subset W of T which retains enough of the structure of T to enable one to determine the truth value of φ' in \mathcal{F} ; despite not being finite in general, this W can be adequately *represented* by a finite graph.
- Indicating by ν the overall number of distinct variables in φ , we can set a computable bound $f(\nu)$ on the size of the finite representation of W . Indirectly, this sets a bound on the amount of time needed to explore the search space within which a “witness” W of the satisfiability of φ' can lie.

The construction of W and the last two points represent a novelty with respect to previously mentioned approaches.

As will emerge from the ongoing, although we have cast it in semantic terms, our decision problem could easily be referred to the classical axiomatic theory of sets ZF, or to a weaker theory postulating among others the existence of infinite sets and the well-foundedness of membership.

1. BASICS AND PRELIMINARY RESULTS

We begin with a few simple notions (the reader can refer to [Lev79] for a more detailed treatment) and basic results.

It is worth recalling that, as a consequence of the hierarchical construction of the class \mathbb{V} of all sets, the membership relation is *well-founded* (i.e., every non-empty subset of \mathbb{V} has a \in -minimal element) and one can rely on the usual notion of *rank*:

Definition 2. The RANK function rk , taking values on the ordinal numbers, is recursively defined over \mathbb{V} as follows:²

$$\text{rk}(x) = \sup\{\text{rk}(y) + 1 : y \in x\}.$$

²Incidentally, note that our syntax for the set abstraction terms is $\{s(x_1, \dots, x_n) : x_1 \in t_1, \dots, x_n \in t_n \mid P(x_1, \dots, x_n)\}$, where the x_i 's are variables, s and the t_i 's are terms, and P is a condition. In this kind of setformer s and P can involve some or all of the x_i 's, and each t_i can involve some or all of the variables x_1, \dots, x_{i-1} (but none of x_{i+1}, \dots, x_n). We will not write down the part ' $\mid P(x_1, \dots, x_n)$ ' explicitly when P is true, and will omit the part ' $s(x_1, \dots, x_n) :$ ' when s is the same variable x_1 which occurs as left-hand side of the leftmost iterator $x_i \in t_i$.

In \mathbb{V} the only element of rank 0 is \emptyset ; a set is said to be HEREDITARILY FINITE whenever its rank is a natural number. We often use the abbreviation

$$x^{\mathfrak{R}\alpha} = \{y \in x \mid \text{rk}(y)\mathfrak{R}\alpha\},$$

where x is a set, α is an ordinal, and \mathfrak{R} is a binary relation: thus, e.g.,

$$x^{>\alpha} = \{y \in x \mid \text{rk}(y) > \alpha\}, x^{<\alpha} = x \cap \mathbf{V}_\alpha, \text{ and } x^{\geq\alpha} = x \setminus \mathbf{V}_\alpha.$$

Definition 3. Given a set \mathbf{v} , we denote by $\text{TrCl}(\mathbf{v})$ the TRANSITIVE CLOSURE of \mathbf{v} , defined through the recursion

$$\text{TrCl}(\mathbf{v}) = \mathbf{v} \cup \bigcup_{\mathbf{u} \in \mathbf{v}} \text{TrCl}(\mathbf{u}).$$

Definition 4. Given an acyclic graph $G = \langle V, E \rangle$, we define the MOSTOWSKI COLLAPSE OF G to be the family $\mathbf{M}(G)$ of sets defined as

$$\mathbf{M}(v, G) = \{\mathbf{M}(u, G) : \langle v, u \rangle \in E\}.$$

Moreover, given $Z \subseteq V$, we define the MOSTOWSKI COLLAPSE OF G WITH PARAMETER Z to be the family $\mathbf{M}(G; Z)$ of sets defined as

$$\mathbf{M}(v, G; Z) = \begin{cases} \{\mathbf{M}(u, G; Z) : \langle v, u \rangle \in E\}; & \text{if } v \notin Z \\ \mathbf{A}_Z(v) & \text{if } v \in Z, \end{cases}$$

where \mathbf{A}_Z is an arbitrary bijection between Z and a set none of whose elements comes to equal any $\mathbf{M}(v, G; Z)$ —this can be achieved simply by requiring that all values of \mathbf{A}_Z have rank $|V| + 1$. \square

When using the above definition we will omit the parameter G in the notation whenever this is clear from the context.

Definition 5. Given a set \mathbf{S} , we denote by $G_{\mathbf{S}}$ the graph $\langle \mathbf{S}, E_{\mathbf{S}} \rangle$ whose edge relation $E_{\mathbf{S}}$ is

$$\{\langle v, w \rangle : v, w \in \mathbf{S} \mid w \in v\}.$$

\square

Let \mathcal{F} be a finite family of sets. Given $\mathbf{z} \in \bigcup \mathcal{F}$, we denote by $\mathcal{F}(\mathbf{z})$ the set $\{\mathbf{v} \setminus \{\mathbf{z}\} : \mathbf{v} \in \mathcal{F}\}$.

Definition 6. A $\mathbf{z} \in \bigcup \mathcal{F}$ is said to be REDUNDANT if $|\mathcal{F}| = |\mathcal{F}(\mathbf{z})|$. We say that \mathcal{F} is IRREDUNDANT if no element of $\bigcup \mathcal{F}$ is redundant. \square

If \mathcal{F} is irredundant then $\bigcup \mathcal{F}$ is also called a *minimal differentiating set* for \mathcal{F} (see Fig. 1). What is important for our combinatorial problem, is the size of such a minimal differentiating set as well as a technique to determine it. The following lemma, whose proof (see [PPR97, Bol86]) relies on *extensionality*, namely on the set-theoretic assumption $\forall u \forall v (\forall w (w \in u \leftrightarrow w \in v) \rightarrow u = v)$, gives us indications on those two issues.

Lemma 1.1 (Discrimination lemma). *Given an n -element nonempty family $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$, we can determine $\mathbf{z}_1, \dots, \mathbf{z}_k \in \bigcup \mathcal{F}$, with $k \leq n - 1$, so that the family*

$$\{\mathbf{v}_i \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\} : \mathbf{v}_i \in \mathcal{F}\}$$

is irredundant and has cardinality n .

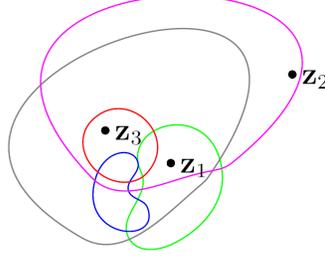


FIGURE 1. A minimal differentiating set of three elements for a family \mathcal{F} of five sets.

A useful variant of the above lemma is stated below.

Lemma 1.2 (Élite discrimination lemma). *Given an n -element family $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$, such that $\text{rk}(\mathcal{F}) - 1 = \varrho + 1$ for some ordinal ϱ (i.e., the maximum rank of an element of \mathcal{F} is a successor ordinal), we can determine $\mathbf{z}_1, \dots, \mathbf{z}_k \in \bigcup \mathcal{F}$, with $k \leq n$, so that the family*

$$\{\mathbf{v}_i \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\} : \mathbf{v}_i \in \mathcal{F}\}$$

has cardinality n and, for every $i \in \{1, \dots, n\}$,

$$\text{rk}(\mathbf{v}_i) = \varrho + 1 \quad \text{implies} \quad \text{rk}(\mathbf{v}_i \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\}) = \varrho + 1.$$

Proof. The proof proceeds by induction on the number $m = |\{v \in \mathcal{F} \mid \text{rk}(v) = \varrho + 1\}|$ of sets of maximum rank in \mathcal{F} . We will assume in the ongoing that $z \in w \in \mathcal{F}$ and $\text{rk}(z) = \varrho$. Observe that the claim of this lemma trivially holds when $m = n = 1$ (just take $k = 1$ and $\mathbf{z}_1 = z$); if this is not the case, then we can determine $\mathbf{z}_2, \dots, \mathbf{z}_k \in \bigcup(\mathcal{F} \setminus \{w\})$ with $k \leq n$ so that the family $\{\mathbf{v}_i \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\} : \mathbf{v}_i \in \mathcal{F} \setminus \{w\}\}$ has cardinality $n - 1$, by exploiting either Lemma 1.1 (if $m = 1$) or the induction hypothesis. When $m = 1 < n$, we get the desired $\mathbf{z}_1, \dots, \mathbf{z}_k \in \bigcup \mathcal{F}$ by simply adding $\mathbf{z}_1 = z$, because then $w \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\} \supseteq \{\mathbf{z}_1\} \not\subseteq v \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\}$ for any $v \in \mathcal{F} \setminus \{w\}$.

When $m > 1$, we can inductively assume that

$$\text{rk}(\mathbf{v}_i) = \varrho + 1 \quad \text{implies} \quad \text{rk}(\mathbf{v}_i \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}) = \varrho + 1$$

for every $\mathbf{v}_i \in \mathcal{F}$ other than $\mathbf{v}_i = w$. If $\text{rk}(w \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}) \neq \varrho + 1$, then we put $\mathbf{z}_1 = z$, so that $\text{rk}(w \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\}) = \varrho + 1$ and the set $w \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\}$ differs from any set $v \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\}$ with $\text{rk}(v) \leq \varrho$ (since $\mathbf{z}_1 \in w$ and $\text{rk}(\mathbf{z}_1) = \varrho$) and it also differs from any set $\mathbf{v}_i \cap \{\mathbf{z}_1, \dots, \mathbf{z}_k\}$ with $\mathbf{v}_i \in \mathcal{F} \setminus \{w\}$ and $\text{rk}(\mathbf{v}_i) = \varrho + 1$ (since $\mathbf{v}_i \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}$ has an element \mathbf{z}_j of rank ϱ not belonging to w). If, on the contrary, $\text{rk}(w \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}) = \varrho + 1$, then either $w \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}$ already differs from any $\mathbf{v}_i \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}$ with $\mathbf{v}_i \in \mathcal{F} \setminus \{w\}$, in which case we do not need a *new* \mathbf{z}_1 (and we can put $\mathbf{z}_1 = \mathbf{z}_2$), or else we can draw \mathbf{z}_1 from the symmetric difference $w \triangle \mathbf{v}_i$, where $\mathbf{v}_i \in \mathcal{F} \setminus \{w\}$ is such that $w \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\} = \mathbf{v}_i \cap \{\mathbf{z}_2, \dots, \mathbf{z}_k\}$. \square

The decidability of restricted unnested BSR-formulae. Let us start by giving a rather simplified and intuitive satisfiability test for the subclass of *restricted unnested BSR-formulae* defined below.

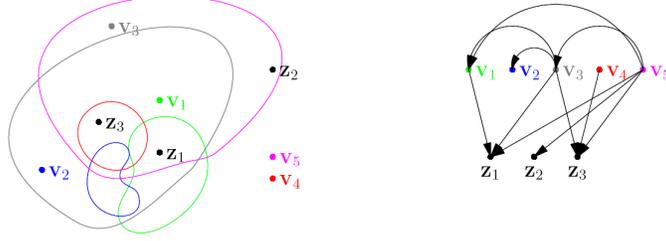


FIGURE 2. A family \mathcal{F} of five sets with a minimal differentiating set and the associated membership graph (without edges among the elements of the differentiating set).

Definition 7. A BSR-formula $\varphi(x_1, \dots, x_n)$ is said to be **RESTRICTED** if all of its universal quantifiers are bounded. A restricted formula is **UNNESTED** if for any bounded quantifier $\forall y \in x$ the variable x is free in φ . \square

Given a restricted unnested formula $\varphi(x_1, \dots, x_n)$, if there are sets $\mathbf{v}_1, \dots, \mathbf{v}_n$ satisfying it, consider the graph $G_{\{\mathbf{v}_1, \dots, \mathbf{v}_n\}}$ (see Definition (5)). Such graph is a finite structure that can simply be “guessed” as a non deterministic step of our satisfiability procedure. The problem is that we are generally unable, given such a structure, to reconstruct the tuple of sets—or even build one *ex novo*—satisfying the formula. The natural move would be trying to use the Mostowski collapse of $G_{\{\mathbf{v}_1, \dots, \mathbf{v}_n\}}$, namely the family

$$\mathbf{M}(\mathbf{v}_j) = \{\mathbf{M}(\mathbf{v}_i) : \langle \mathbf{v}_j, \mathbf{v}_i \rangle \in E_{\{\mathbf{v}_1, \dots, \mathbf{v}_n\}}\}$$

with $j = 1, \dots, n$; however, such sets have the obvious problem that it can easily be the case that $\mathbf{v}_h \neq \mathbf{v}_k$ while $\mathbf{M}(\mathbf{v}_h) = \mathbf{M}(\mathbf{v}_k)$. To avoid this problem, we should enrich the graph with further nodes acting as witnesses and use the parametric version of the Mostowski collapse.

Such an enrichment is always possible. In fact, recalling Lemma 1.1 we have that $k < n$ new elements of $\mathbf{v}_1 \cup \dots \cup \mathbf{v}_n$ are sufficient as witnesses of the differences among $\mathbf{v}_1, \dots, \mathbf{v}_n$. These extra nodes $\mathbf{z}_1, \dots, \mathbf{z}_k$ can be “guessed” from the outset by our algorithm.

At this point—as the reader can easily verify—the Mostowski collapse with parameter $\mathbf{Z} = \{\mathbf{z}_1, \dots, \mathbf{z}_k\}$ ensures that all equalities and membership relations among $\mathbf{M}(\mathbf{v}_1; \mathbf{Z}), \dots, \mathbf{M}(\mathbf{v}_n; \mathbf{Z})$ and their elements mimic exactly the ones among $\mathbf{v}_1, \dots, \mathbf{v}_n$ and their elements.

Notice that it is the universal character of φ that allows us to conclude. In fact, if the sets $\mathbf{M}(\mathbf{v}_1; \mathbf{Z}), \dots, \mathbf{M}(\mathbf{v}_n; \mathbf{Z})$ were not to satisfy φ , then a counterexample to the satisfiability of φ by $\mathbf{v}_1, \dots, \mathbf{v}_n$ (using \mathbf{v} ’s and witnesses only) could easily be built. This is the key point in the construction where the assumption of un-nestedness plays its role: a counterexample to a purely universal unnested formula could never use an element which does not correspond to a node in $G_{\{\mathbf{v}_1, \dots, \mathbf{v}_n, \mathbf{z}_1, \dots, \mathbf{z}_k\}}$.

An algorithm which solves the satisfiability problem for restricted un-nested formulae was proposed in [BFOS81], as one of the first results in Computable Set Theory. The argument used in [BFOS81] was different from the one presented above and the decidability result was also extended to cover some extra constructors (notably arithmetic ones) that were shown to maintain both the property of reflection into the (hereditarily) finite sets and decidability.

The absence of nested quantifiers is crucial in showing the property of reflection into hereditarily finite sets: below we examine a restricted and nested formula which is satisfiable without being finitely satisfiable (cf. [PP88, PP90, PP91b]). Consider the formula in two free variables—to be denoted ω_0 and ω_1 , respectively—which is the conjunction of the following sub-formulae where $b = 0, 1$:

- a) $\forall y \in \omega_b \forall x \in y (x \in \omega_{1-b})$, that is $\omega_b \subseteq \mathcal{P}(\omega_{1-b})$;
- b) $\omega_b \notin \omega_{1-b}$;
- c) $\omega_0 \neq \omega_1$.

Two sets satisfying the above conditions are not necessarily infinite; in fact $\omega_0 = \{\emptyset, \{\{\emptyset\}\}\}$, $\omega_1 = \{\{\emptyset\}, \{\{\{\emptyset\}\}\}\}$ would satisfy our formula. However, the above interpretation suggests the beginning of a back-and-forth process that will eventually force ω_0 and ω_1 to take infinite values

$$\omega_0 = \{\emptyset, \{\{\emptyset\}\}, \dots\}, \quad \omega_1 = \{\{\emptyset\}, \{\emptyset, \{\{\emptyset\}\}\}, \dots\}$$

(see Fig. 3) when a suitable extra condition is added.

Assuming that \in does not form either cycles or infinite descending chains, either one of the following conditions will do to the case (cf. [PP88, PP90]):

- d) $\forall x_1, x_2 \in \omega_0 \forall y_1, y_2 \in \omega_1 (x_1 \in y_1 \in x_2 \in y_2 \rightarrow x_1 \in y_2)$,
- e) $\forall x \in \omega_0 \forall y \in \omega_1 (x \in y \vee y \in x)$.

When membership is allowed to be a non well-founded relation, these two must be added to the conjunction together (see [PP91b]); but here, since we are assuming membership to be well-founded, we can state:

Lemma 1.3. *The conjunction of conditions a), b), c), and e) force both ω_0 and ω_1 to have the same limit rank (and hence to be infinite).*

The above result can be seen pictorially: two sets satisfying conditions a), b) and c) on ω_0 and ω_1 must have elements of increasing rank $\omega_{0,i}$ and $\omega_{1,i}$, respectively, as in Fig. 3. Stopping at any given finite even rank i forces ω_0 to become $\omega_{1,i}$ and ω_1 to become $\omega_{0,i-1}$, from which $\omega_1 \in \omega_0$ follows. For odd ranks the situation is symmetric.

There is nothing special about the fact that *two* sets are involved in the above example. A more general situation which, for any nonnegative n , analogously forces $n + 2$ sets ω_i to have limit rank, is described by the formula appearing in the following proposition, easily rewritable as a BSR-formula:³

Lemma 1.4. *The formula*

$$\emptyset \neq \omega_0 \wedge \bigwedge_{i=0}^{n+1} (\omega_i \notin \omega_{(i+1) \bmod (n+2)}) \wedge \bigwedge_{i=0}^{n+1} (\bigcup \omega_{(i+1) \bmod (n+2)} \subseteq \omega_i) \wedge$$

$$(\forall x_0 \in \omega_0, \dots, x_{n+1} \in \omega_{n+1}) \left(\bigvee_{i=0}^{n+1} x_i \in x_{(i+1) \bmod (n+2)} \right)$$

is satisfiable but is not finitely satisfiable.

³Here \bmod designates the integer remainder operation.

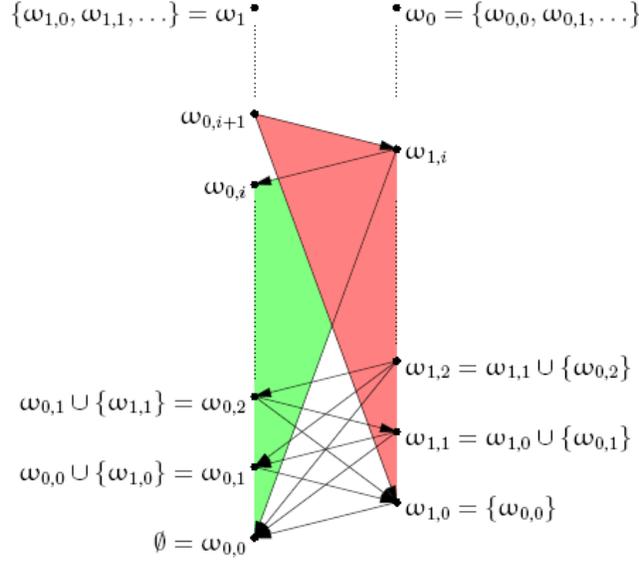


FIGURE 3. A pictorial proof of Lemma 1.3.

2. THE SYNTACTIC REDUCTION

In this section we show that the (satisfiability) decision problem for the entire BSR-class can be reduced to the subclass of restricted BSR-formulae.

To see this we begin by proving that every BSR-sentence can always be cast in the following simple format:

Proposition 2.1. *Every prenex sentence*

$$\Phi \equiv (\exists x_1, \dots, x_n)(\forall y_1, \dots, y_m)\phi(x_1, \dots, x_n, y_1, \dots, y_m)$$

in the BSR-class is logically equivalent to a sentence

$$\Phi \equiv (\exists x_1, \dots, x_n) \bigwedge_{i=1}^k (\forall y_1, \dots, y_m)\phi_i(x_1, \dots, x_n, y_1, \dots, y_m),$$

where each ϕ_i is a disjunction of literals of the forms

$$z \in w, z \notin w, z = w, z \neq w,$$

with $z, w \in \{x_1, \dots, x_n, y_1, \dots, y_m\}$.

To see that the above proposition is true, it suffices to bring the matrix ϕ of Φ to conjunctive normal form, then interchanging universal quantifiers and conjunction.

Our next step will be to perform a transformation, rather common in decision procedures for fragments of set theory, which does not apply to *sentences* but to the corresponding formulae obtained by withdrawing the existential quantifiers (see Definition (1)). Here the notion of *injective* satisfiability, meaning “satisfiability by a tuple of pairwise distinct sets”, enters into play. We are about to observe that,

within first order logic (extensionality being the only set-theoretic assumption which enters into play), we can restrain our analysis to equality-free formulae:

Lemma 2.2. *For any given formula φ in n free variables of the BSR-class, one can determine a finite collection of conjunctions*

$$\varphi_h = \bigwedge_{i=1}^{k_h} (\forall y_1, \dots, y_m) \phi_i(x_{j_1}, \dots, x_{j_{n_h}}, y_1, \dots, y_m)$$

of BSR-formulae, where $n_h \leq n$ and every ϕ_i is a disjunction of literals of the forms

$$z \in w, z \notin w,$$

with $z, w \in \{x_{j_1}, \dots, x_{j_{n_h}}, y_1, \dots, y_m\}$, so that φ is satisfiable if and only if some φ_h is injectively satisfiable.

(Sets satisfying the original φ will correspond to sets injectively satisfying one of the φ_h 's.)

Example 1. The satisfiability problem regarding the BSR-formula

$$\forall y_1 \forall y_2 ((y_1 \neq y_2 \wedge y_1 \in x_1) \rightarrow y_2 \notin x_1)$$

(stating that x_1 is either void or singleton) reduces to the injective satisfiability problem regarding the BSR-formula

$$\begin{aligned} & \forall y_1 \forall y_2 \forall y_3 ((y_1 \in x_1 \wedge y_3 \in y_1 \wedge y_3 \notin y_2) \rightarrow y_2 \notin x_1) \wedge \\ & \forall y_1 \forall y_2 \forall y_3 ((y_1 \in x_1 \wedge y_3 \notin y_1 \wedge y_3 \in y_2) \rightarrow y_2 \notin x_1). \end{aligned}$$

□

On the ground of the above result, from now on we will assume that no equalities/inequalities appear in the matrix of BSR-formulae/sentences and we will intend “satisfiability” as actually meaning “injective satisfiability”.

Our last result in this section proves that we can further restrict our consideration to the bounded-quantifier case:

Lemma 2.3. *Any conjunction*

$$\bigwedge_{i=1}^k (\forall y_1, \dots, y_m) \phi_i(x_1, \dots, x_n, y_1, \dots, y_m)$$

of BSR-formulae, where each ϕ_i is a disjunction of literals of the forms $z \in w$ and $z \notin w$, is equivalent to an alike formula in which every ϕ_i involving a variable y_j comprises at least one literal of the form

$$y_j \notin z,$$

with $z \in \{x_1, \dots, x_n, y_1, \dots, y_{j-1}\}$.

Proof. Assume first that there exist a y_j and a ϕ_i within which y_j , but no literal $y_j \notin z$ with

$$z \in \{x_1, \dots, x_n, y_1, \dots, y_{j-1}, y_{j+1}, \dots, y_m\},$$

occurs. Hence, all literals involving y_j within ϕ_i are of the forms

$$z \in y_j, z \notin y_j, y_j \in z.$$

If $\phi_i(x_1, \dots, x_n, y_1, \dots, y_m)$ is true for some n -tuple X_1, \dots, X_n of sets and for every m -tuple Y_1, \dots, Y_m of sets, then consider any specific such m -tuple Y_1, \dots, Y_m , and replace Y_j in such a tuple by a set Y'_j which satisfies all literals

$$Y'_j \notin X_1, \dots, Y'_j \notin X_n, Y'_j \notin Y_1, \dots, Y'_j \notin Y_m$$

and also satisfies all biimplications

$$Z \in Y'_j \quad \text{if and only if} \quad \phi_i \text{ does not comprise the literal } z \in y_j$$

with $z \in \{x_1, \dots, x_n, y_1, \dots, y_{j-1}, y_{j+1}, \dots, y_m\}$. It is easy to see that one such Y'_j can always be found; therefore truth (under the assignment $x_k \mapsto X_k$ is preserved if ϕ_i is replaced by its sub-formula ϕ'_i obtained by withdrawing all literals which involve y_j .

At this point, if a chain of the form

$$y_j \notin z_1, \dots, z_h \notin y_j$$

appears in a conjunct ϕ_i , we can discard the conjunct inasmuch as blatantly true. Hence, we can rename the universally quantified variables in every conjunct, so as to meet the claim of this theorem. \square

On the ground of the above result we can assume the following *restricted* format for formulae of the BSR-class:

$$\bigwedge_{i=1}^k (\forall y_1 \in z_1, \dots, y_{m_i} \in z_{m_i}) \phi_i(x_1, \dots, x_n, y_1, \dots, y_{m_i}),$$

where $z_h \in \{x_1, \dots, x_n, y_1, \dots, y_{h-1}\}$ for $h \in \{1, \dots, m_i\}$.

3. THE SEMANTIC REDUCTION

In this section we prove that a finite family satisfying a given restricted BSR-formula always admits a finite description, for whose size we will assess a bound in Section (4). In rough outline, our strategy will be to adapt the proof given for the unnested case (see Section (1)), by producing a family of witnesses that will not need the parametric version of the Mostowski collapse.

Let us begin by stating what kind of self-sustaining witness sets—typically infinite in the case before us, in spite of the finite description at which we are aiming—will interest us:

Definition 8. Given a family \mathcal{F} , we say that $\mathbf{W} \subseteq \text{TrCl}(\mathcal{F})$ REPRESENTS \mathcal{F} if:

- $\mathcal{F} \subseteq \mathbf{W}$, and
- for all $\mathbf{u}, \mathbf{v} \in \mathbf{W}$, if $\mathbf{u} \neq \mathbf{v}$ then $\mathbf{M}(\mathbf{u}, G_{\mathbf{W}}) \neq \mathbf{M}(\mathbf{v}, G_{\mathbf{W}})$. \square

The latter of the above conditions, requiring the Mostowski collapse of \mathbf{W} to be injective, is equivalent to the condition that distinct elements \mathbf{u}, \mathbf{v} of \mathbf{W} never satisfy $\mathbf{u} \cap \mathbf{W} = \mathbf{v} \cap \mathbf{W}$. We could state this by saying that membership is extensional on the set \mathbf{W} —as well as by saying that the graph $G_{\mathbf{W}}$ has no two nodes with the same immediate descendants.

Proposition 3.1. *If $\varphi(x_1, \dots, x_n)$ is a restricted BSR-formula satisfied by $\mathbf{v}_1, \dots, \mathbf{v}_n$ and \mathbf{W} represents $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$, then also the n -tuple*

$$\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}})$$

satisfies $\varphi(x_1, \dots, x_n)$.

Proof. The proof is entirely analogous to the one given in the unnested case (see Section (1)). \square

Assuming that a given BSR-formula φ has a model, we seek a finitely describable representation of the set \mathcal{F} of values assigned by the model to the free variables of φ . The sought description will not refer directly to \mathcal{F} , but to the simplified model which results from the Mostowski collapse of a set \mathbf{W} which besides representing \mathcal{F} (in the sense of Def. (8)) also enjoys various convenient features. Of such features, which will enable us to encode \mathbf{W} by means of a finite structure, a few are introduced with our next lemma, others will enter into play with Lemma (3.3) into which we will soon refine this lemma.

From now on when two elements $\mathbf{u}, \mathbf{v} \in \mathbf{W}$ are such that $\mathbf{u} \neq \mathbf{v}$ and $\mathbf{M}(\mathbf{u}, G_{\mathbf{W}}) = \mathbf{M}(\mathbf{v}, G_{\mathbf{W}})$, we say that \mathbf{u} and \mathbf{v} *collide*.

Lemma 3.2. *Given a finite family \mathcal{F} , a set \mathbf{W} can be determined so that:*

- (1) \mathbf{W} represents \mathcal{F} ;
- (2) \mathbf{W} owns at most $|\mathcal{F}|$ members of rank α , for each ordinal α ;
- (3) $|\{\mathbf{w} \in \mathbf{W} \mid \text{rk}(\mathbf{M}(\mathbf{w}, G_{\mathbf{W}})) \text{ is a limit ordinal}\}| < \omega$.

Proof. We proceed by induction on $\text{rk}(\mathcal{F})$.

If $\text{rk}(\mathcal{F}) \leq 1$, then the claim trivially holds, since either \mathcal{F} is empty or $|\mathcal{F}| = 1$. As a matter of fact, if $|\mathcal{F}| = 1$ then for any value of $\text{rk}(\mathcal{F})$ it suffices to take $\mathbf{W} = \mathcal{F}$.

For the inductive step, let $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$. We will consider two cases: either $\text{rk}(\mathcal{F}) - 1$ (i.e. the maximum rank of a \mathbf{v}_i) is a successor, or is a limit ordinal.

In case $\text{rk}(\mathcal{F}) - 1$ is a successor, say $\varrho + 1$ for some ϱ , we can apply the elite discrimination lemma (Lemma (1.2)) to the set $\{\mathbf{v} \in \mathcal{F} \mid \text{rk}(\mathbf{v}) = \varrho + 1\}$ thus determining elements $\mathbf{z}_1, \dots, \mathbf{z}_k$ complying with the claim of that lemma. We can then apply the inductive hypothesis to get a set \mathbf{W}' satisfying analogs of the conditions (1), (2), (3) relative to the family

$$\mathcal{F}' = \{\mathbf{v} \in \mathcal{F} \mid \text{rk}(\mathbf{v}) \leq \varrho\} \cup \{\mathbf{z}_1, \dots, \mathbf{z}_k\},$$

(notice that $\text{rk}(\mathcal{F}') = \text{rk}(\mathcal{F}) - 1$ and $|\mathcal{F}'| \leq n$). Next we show that we achieve our goal if we put

$$\mathbf{W} = \mathbf{W}' \cup \mathcal{F} (= \mathbf{W}' \cup \{\mathbf{v} \in \mathcal{F} \mid \text{rk}(\mathbf{v}) = \varrho + 1\}).$$

To see that \mathbf{W} represents \mathcal{F} (claim (1)), observe that:

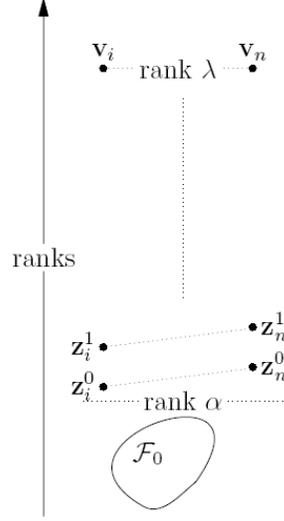
- two elements of \mathbf{W}' cannot collide, by the induction hypothesis;
- two elements of $\mathcal{F} \setminus \mathbf{W}' (= \{\mathbf{v} \in \mathcal{F} \mid \text{rk}(\mathbf{v}) = \varrho + 1\})$ cannot collide, because the differentiating set $\{\mathbf{z}_1, \dots, \mathbf{z}_k\}$ for $\mathcal{F} \setminus \mathbf{W}'$ has been included in \mathbf{W}' ;
- there can be no collision between an element of $\mathcal{F} \setminus \mathbf{W}'$ and an element of \mathbf{W}' , as by Lemma (1.2) every element of $\mathcal{F} \setminus \mathbf{W}'$ owns an element \mathbf{z}_i of rank ϱ , which cannot be the case for any element in \mathbf{W}' .

Claims (2) and (3) are easily seen to follow from the inductive hypothesis.

In case $\text{rk}(\mathcal{F}) - 1$ is a limit ordinal λ , assume that $\text{rk}(\mathbf{v}_1) \leq \text{rk}(\mathbf{v}_2) \leq \dots \leq \text{rk}(\mathbf{v}_n)$ and that $\{\mathbf{v}_i, \dots, \mathbf{v}_n\}$ are the elements of rank λ in \mathcal{F} . Let

$$\alpha = \text{rk}(\{\mathbf{v}_1, \dots, \mathbf{v}_{i-1}\}) \cup \text{rk}(\{\mathbf{z}_1, \dots, \mathbf{z}_{n'}\}),$$

where $\{\mathbf{z}_1, \dots, \mathbf{z}_{n'}\}$ is a differentiating set for $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$; actually, it would suffice here to choose α to be any ordinal (strictly below λ) which exceeds all of the ranks

FIGURE 4. Positions of $\mathbf{z}_i^u, \dots, \mathbf{z}_n^u$ relative to $\mathbf{v}_i, \dots, \mathbf{v}_n$, for $u = 0, 1, \dots$

$\text{rk}(\mathbf{v}_1), \dots, \text{rk}(\mathbf{v}_{i-1})$ and is high enough to ensure that $\mathbf{v}_1, \dots, \mathbf{v}_{i-1}, \mathbf{v}_i^{<\alpha}, \dots, \mathbf{v}_n^{<\alpha}$ are n distinct sets.

Let $\mathcal{F}^0 = \{\mathbf{v}_1, \dots, \mathbf{v}_{i-1}\} \cup \{\mathbf{v}_i^{<\alpha}, \dots, \mathbf{v}_n^{<\alpha}\}$, and apply the inductive hypothesis to get a set \mathbf{W}^0 satisfying analogs of (1), (2), and (3) relative to \mathcal{F}^0 .

For every $u \in \omega$, let $\mathbf{z}_i^u, \dots, \mathbf{z}_n^u$ be $n - i + 1$ sets belonging to $\mathbf{v}_i, \dots, \mathbf{v}_n$, respectively (see Figure (4)), such that:

- $\alpha < \text{rk}(\mathbf{z}_i^0)$,
- $\text{rk}(\mathbf{z}_i^u) < \dots < \text{rk}(\mathbf{z}_n^u) < \text{rk}(\mathbf{z}_i^{u+1})$ for all $u \in \omega$.

Now, for every $u \in \omega$, define the $(n - i + 1)$ -tuple $t_u = \langle t_u(i), \dots, t_u(n) \rangle$ by putting

$$t_u(\ell) = \begin{cases} \ell' & \text{if } \mathbf{z}_\ell^u = \mathbf{v}_{\ell'}^{<\text{rk}(\mathbf{z}_\ell^u)} = \{x \in \mathbf{v}_{\ell'} \mid \text{rk}(x) < \text{rk}(\mathbf{z}_\ell^u)\}, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that the above definition is well posed, as \mathbf{z}_ℓ^u can be at most one among $\mathbf{v}_i^{<\text{rk}(\mathbf{z}_\ell^u)}, \dots, \mathbf{v}_n^{<\text{rk}(\mathbf{z}_\ell^u)}$, since otherwise two elements among $\mathbf{v}_i^{<\text{rk}(\mathbf{z}_\ell^u)}, \dots, \mathbf{v}_n^{<\text{rk}(\mathbf{z}_\ell^u)}$ would collide, contradicting the fact that there are witnesses of their difference at a rank smaller than $\alpha < \text{rk}(\mathbf{z}_\ell^u)$.

Clearly every t_u can be seen as a character in a finite alphabet, hence the infinite sequence t_0, t_1, \dots must contain an infinitely repeating character. Let t_{r_0}, t_{r_1}, \dots be a subsequence of t_0, t_1, \dots consisting of one such infinitely repeating character. Moreover, let $\bar{t}_{r_0}, \bar{t}_{r_1}, \dots$ be the sequence resulting from t_{r_0}, t_{r_1}, \dots through replacement of each t_{r_j} by the \bar{t}_{r_j} defined as

$$\bar{t}_{r_j}(\ell) = \begin{cases} t_{r_j}(\ell) & \text{if } \exists v(\ell \neq t_{r_j}(\ell) = \ell_1 \wedge t_{r_j}(\ell_1) = \ell_2 \wedge \dots \wedge t_{r_j}(\ell_v) = \ell), \\ 0 & \text{otherwise.} \end{cases}$$

We claim that by putting

$$\mathbf{W} = (\mathbf{W}^0 \cap \text{TrCl}(\mathcal{F})) \cup \{\mathbf{z}_\ell^{r_j} : j \in \omega, \ell \in \{i, \dots, n\} \mid \bar{t}_{r_0}(\ell) \neq 0\} \cup \{\mathbf{v}_i, \dots, \mathbf{v}_n\}$$

we meet the conditions (1), (2), and (3).

Concerning claim (1) we proceed by contradiction, assuming that $\mathbf{x}, \mathbf{y} \in \mathbf{W}$, despite being different, own the same successors in $G_{\mathbf{W}}$.

Consider the following three cases:

- i) $\text{rk}(\mathbf{x}) \geq \text{rk}(\mathbf{y}) > \alpha$;
- ii) $\text{rk}(\mathbf{x}) > \alpha > \text{rk}(\mathbf{y})$;
- iii) $\alpha > \text{rk}(\mathbf{x}) \geq \text{rk}(\mathbf{y})$.

Case i) splits into three subcases:

- i.1) $\mathbf{x}, \mathbf{y} \in \{\mathbf{v}_i, \dots, \mathbf{v}_n\}$. This case cannot hold, because \mathbf{W}^0 includes a differentiating set for $\{\mathbf{v}_i^{<\alpha}, \dots, \mathbf{v}_n^{<\alpha}\}$ (and therefore for $\{\mathbf{v}_i, \dots, \mathbf{v}_n\}$).
- i.2) $\mathbf{x} \in \{\mathbf{v}_i, \dots, \mathbf{v}_n\}$ and $\mathbf{y} \notin \{\mathbf{v}_i, \dots, \mathbf{v}_n\}$. In this case the equality $\mathbf{y} = \mathbf{z}_\ell^{r_j}$ must hold for some j and ℓ . Let $\bar{t}_{r_j}(\ell) = \ell' \notin \{0, \ell\}$.

It cannot be the case that $\mathbf{x} = \mathbf{v}_{\ell'}$ as otherwise, by definition of \bar{t}_{r_j} , it would be the case that $\exists v(t_{r_j}(\ell) = \ell' = \ell_1 \wedge t_{r_j}(\ell_1) = \ell_2 \wedge \dots \wedge t_{r_j}(\ell_v) = \ell)$. From this it would follow that $\bar{t}_{r_j}(\ell') = \bar{t}_{r_j}(\ell_1) \neq 0$ (as well as $\bar{t}_{r_j}(\ell_2) \neq 0, \dots, \bar{t}_{r_j}(\ell_v) \neq 0$). As $\bar{t}_{r_j}(\ell') \neq 0$, there would be infinitely many elements of rank greater than α witnessing the difference between \mathbf{x} and \mathbf{y} .

Moreover, it cannot be the case that $\mathbf{x} = \mathbf{v}_{\ell''} \neq \mathbf{v}_{\ell'}$, since by definition $\mathbf{z}_\ell^{r_j} = \mathbf{v}_{\ell'}^{<\text{rk}(\mathbf{z}_\ell^{r_j})}$, and therefore a witness of the difference between $\mathbf{v}_{\ell''}^{<\alpha}$ and $\mathbf{v}_{\ell'}^{<\alpha}$ is also a witness of the difference between \mathbf{x} and \mathbf{y} .

- i.3) $\mathbf{x}, \mathbf{y} \notin \{\mathbf{v}_i, \dots, \mathbf{v}_n\}$. This case is entirely analogous to the previous one. Details are left to the reader.

Case ii) cannot occur, as if \mathbf{x} and \mathbf{y} could collide, their respective ranks being greater and smaller than α , then $\mathbf{v}_{\ell'}^{<\alpha}$ and \mathbf{y} would also collide, for some $\ell' \in \{i, \dots, n\}$, contradicting the inductive hypothesis on \mathbf{W}^0 .

Case iii) cannot occur, due to the inductive hypothesis, as both \mathbf{x} and \mathbf{y} belong to \mathbf{W}^0 .

Claim (2) follows from the definition of the $\mathbf{z}_\ell^{r_j}$'s and from the inductive hypothesis.

Claim (3) is a plain consequence of the above construction. \square

Given \mathcal{F} , the proof of the above lemma implicitly defines a procedure to determine a (countable) \mathbf{W} representing it. Such a procedure—to be specified in a more “algorithmic” format in appendix—inserts elements of decreasing ranks in \mathbf{W} and, for a finite number

$$h = |\{\mathbf{w} \in \mathbf{W} \mid \text{rk}(\mathbf{M}(\mathbf{w}, G_{\mathbf{W}})) \text{ is a limit ordinal}\}|$$

of times, introduces elements of the form $\mathbf{z}_\ell^{r_j}$. We will denote such elements in \mathbf{W} as $\mathbf{z}_{k, i_k}^j, \dots, \mathbf{z}_{k, n_k}^j$ for $j \in \omega$ and $k \leq h$, and call them the *rotors* of \mathbf{W} . The (extra) index k in $\mathbf{z}_{k, \ell}^j$ indicates that the rotor was introduced while dealing with the k -th limit ordinal.

We will work with the following definition.

Definition 9. Let \mathbf{W} be a countable set representing the family \mathcal{F} so that

$$\{\text{rk}(\mathbf{w}) : \mathbf{w} \in \mathbf{W} \mid \text{rk}(\mathbf{M}(\mathbf{w}, G_{\mathbf{W}})) \text{ is a limit ordinal}\} = \{\lambda_1, \dots, \lambda_h\},$$

has cardinality $h < \omega$. Assume $\lambda_1 < \dots < \lambda_h$ for definiteness.⁴

A ROTOR SYSTEM for \mathbf{W} is a decomposition

$$\mathbf{W} = c(\mathbf{W}) \cup R_1 \cup \dots \cup R_h$$

of \mathbf{W} into disjoint sets such that

- $c(\mathbf{W})$, to be called the CORE of \mathbf{W} , is finite;
- each R_k , to be called the k -th LEVEL of \mathbf{W} , consists of distinct elements

$$\mathbf{z}_{k,i_k}^j, \dots, \mathbf{z}_{k,n_k}^j \quad (j \in \omega)$$

of rank less than λ_k , to be called the ROTORS ASCRIBED TO λ_k . \square

The rotors determined as in the proof of Lemma (3.2) are *downward uniform*, in the sense that of two rotors having the same subscripts and different superscripts the one with higher superscript always includes the other. Formally: if $j < j'$, then $\mathbf{z}_{k,\ell}^j \subsetneq \mathbf{z}_{k,\ell}^{j'}$. Indeed, this property follows directly from the fact that rotors with the same subscripts, by definition, collide with the same element (of the core) at different stages of our procedure.

The definition below captures also the symmetrical property of *upward uniformity* whose fulfillment, as we will see, can be achieved jointly with downward uniformity.

Definition 10. Let \mathbf{W} represent \mathcal{F} and satisfy the claims of Lemma (3.2). We will say that \mathbf{W} is UPWARD UNIFORM (respectively, DOWNWARD UNIFORM) if for $k \leq h$ and $j, j' \in \omega$, any pair $\mathbf{z}_{k,\ell}^j, \mathbf{z}_{k,\ell}^{j'}$ of rotors belong to the same elements of \mathbf{W} (respectively, $\mathbf{z}_{k,\ell}^j \subsetneq \mathbf{z}_{k,\ell}^{j'}$, if $j < j'$). \square

In order to fulfill upward uniformity, we can proceed from higher to lower λ_k , erasing some of the rotors ascribed to λ_k , but preserving infinitely many of them along each “track” ℓ , and managing things in such a way that all rotors $\mathbf{z}_{k,\ell}^j$ left on the same track belong to the same elements above. More precisely:

Lemma 3.3. *Given a finite family \mathcal{F} , a set \mathbf{W} can be determined so that:*

- (1) \mathbf{W} represents \mathcal{F} ;
- (2) \mathbf{W} owns at most $|\mathcal{F}|$ members of rank α , for each ordinal α ;
- (3) $|\{\mathbf{w} \in \mathbf{W} \mid \text{rk}(\mathbf{M}(\mathbf{w}, G_{\mathbf{W}})) \text{ is a limit ordinal}\}| < \omega$;
- (4) \mathbf{W} is downward and upward uniform.

Proof. On the ground of Lemma 3.2, one needs only to concentrate on the upper uniformity part of (4). In appendix, we will indicate the refinements to the construction used to prove Lemma 3.2 in order to meet this new requirement. \square

A rotor system \mathbf{W} as above, representing a family \mathcal{F} which satisfies a given restricted BSR-formula, can be encoded as follows by a finite graph:

⁴Notice that the λ 's are limit ordinal in our applications throughout, even though their present definition does not suffice to ensure this. Limit λ 's result, in particular, from the construction of Lemma (3.2).

Definition 11. Let \mathbf{W} represent \mathcal{F} in downward and upward uniform fashion, with h levels of rotors. We define the ENCODING of \mathbf{W} to be the graph $G_{\mathbf{W}}^{\mathcal{F}} = \langle V, E \rangle$ whose nodes V result from the disjoint union of sets V_c and V_r defined as follows:

$$\begin{aligned} V_c &= \{v_i : \mathbf{w}_i \in c(\mathbf{W})\}, \\ V_r &= \{z_{k,i_k}^b, \dots, z_{k,n_k}^b : (\mathbf{z}_{k,i_k}^b, \dots, \mathbf{z}_{k,n_k}^b \in \mathbf{W}), k \in \{1, \dots, h\}, b \in \{0, 1\}\}, \end{aligned}$$

and whose set E of arcs results from the disjoint union of four sets defined as follows:

$$\begin{aligned} E_{cc} &= \{\langle v_j, v_i \rangle : v_i, v_j \in V_c \mid \mathbf{w}_i \in \mathbf{w}_j\}, \\ E_{rr} &= \{\langle z_{k,\ell}^b, z_{k',\ell'}^b \rangle : (z_{k',\ell'}^b, z_{k,\ell}^b \in V_r), b \in \{0, 1\} \mid \\ &\quad (k = k' \wedge \mathbf{z}_{k',\ell'}^0 \in \mathbf{z}_{k,\ell}^1) \vee (k \neq k' \wedge \mathbf{z}_{k',\ell'}^b \in \mathbf{z}_{k,\ell}^b)\}, \\ E_{rc} &= \{\langle z_{k,\ell}^b, v_i \rangle : z_{k,\ell}^b \in V_r, v_i \in V_c, b \in \{0, 1\} \mid \mathbf{v}_i \in \mathbf{z}_{k,\ell}^b\}, \\ E_{cr} &= \{\langle v_i, z_{k,\ell}^b \rangle : z_{k,\ell}^b \in V_r, v_i \in V_c, b \in \{0, 1\} \mid \mathbf{z}_{k,\ell}^b \in \mathbf{v}_i\}. \end{aligned}$$

□

The encoding $G_{\mathbf{W}}^{\mathcal{F}}$ retains the overall information necessary to obtain a set $\mathbf{W}' = \mathbf{W}(G_{\mathbf{W}}^{\mathcal{F}})$ mimicking \mathcal{F} in the sense to be clarified by Propositions (3.4,3.5) below.

Definition 12. The ω -UNROLLING of $G_{\mathbf{W}}^{\mathcal{F}} = \langle V, E \rangle$ is defined to be the set

$$\mathbf{W}(G_{\mathbf{W}}^{\mathcal{F}}) = \mathbf{W}_c(G_{\mathbf{W}}^{\mathcal{F}}) \cup \mathbf{W}_r(G_{\mathbf{W}}^{\mathcal{F}}),$$

where

$$\mathbf{W}_c(G_{\mathbf{W}}^{\mathcal{F}}) = \{\mathbf{v} : v \in V_c\}, \quad \mathbf{W}_r(G_{\mathbf{W}}^{\mathcal{F}}) = \{\mathbf{z}^j : z \in V_r, j \in \omega\},$$

and the following equalities are recursively satisfied for all $v \in V_c$, $z = z_{k,\ell} \in V_r$, and $j \in \omega$:

$$\begin{aligned} \mathbf{v} &= \{\mathbf{w} : \langle v, w \rangle \in E_{cc}\} \cup \{\tilde{\mathbf{z}}^j : \langle v, \tilde{z} \rangle \in E_{cr}, j \in \omega\}, \\ \mathbf{z}^j &= \{\mathbf{w} : \langle z, w \rangle \in E_{rc}\} \cup \\ &\quad \{\tilde{\mathbf{z}}^{j'} : \tilde{z} = z_{k',\ell'}^b \in V_r, \langle z, \tilde{z} \rangle \in E_{rr}, b \in \{0, 1\} \mid k' < k \vee j' < j \vee (j' = j \wedge \ell' < \ell)\}. \end{aligned}$$

We get a fully analogous definition of m -UNROLLING by putting $m + 1$ in place of ω in the above characterizations. □

To achieve full rigor we should prove that the above definition is well given, as no cycles or infinite descending chains can appear in an ω -unrolling. In fact, as the reader can easily check, a cycle or an infinite descending chain would produce a corresponding cycle or infinite descending chain of either of the indices k or ℓ in the \mathbf{z}^j 's involved.

Although not essential to our decidability result, our next proposition clarifies the notion of ω -unrolling introduced above. The proof of the fact reported below it relates to our goals more directly.

Proposition 3.4. *Given \mathbf{W} representing \mathcal{F} , we have that:*

$$\mathbf{W}(G_{\mathbf{W}}^{\mathcal{F}}) = \mathbf{M}(\mathbf{W}, G_{\mathbf{W}}).$$

Proposition 3.5. *Given a restricted BSR-formula $\varphi = \varphi(x_1, \dots, x_n)$ and a family $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ represented by \mathbf{W} in downward and upward uniform fashion, one can establish whether or not $\mathbf{v}_1, \dots, \mathbf{v}_n$ satisfy φ on the basis of the encoding $G_{\mathbf{W}}^{\mathcal{F}}$.*

Proof. Proposition (3.1) implies that if $\mathbf{v}_1, \dots, \mathbf{v}_n$ satisfy $\varphi(x_1, \dots, x_n)$, then by taking $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ and by choosing a \mathbf{W} representing \mathcal{F} , we will have $\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}})$ satisfying $\varphi(x_1, \dots, x_n)$. We will now show that testing $\varphi(x_1, \dots, x_n)$ for satisfaction under the assignment $x_i \mapsto \mathbf{M}(\mathbf{v}_i, G_{\mathbf{W}})$ is possible by mere inspection of the (finite) graph $G_{\mathbf{W}}^{\mathcal{F}}$.

To this end notice that if the formula

$$\varphi(x_1, \dots, x_n) = \bigwedge_{i=1}^k (\forall y_{i,1} \in z_{i,1}, \dots, y_{i,m_i} \in z_{i,m_i}) \phi_i(x_1, \dots, x_n, y_{i,1}, \dots, y_{i,m_i})$$

is not satisfied by $\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}})$, then there is a conjunct in φ whose negation is satisfied by such sets. Namely, for some $i \in \{1, \dots, k\}$ the formula

$$(\exists y_{i,1} \in z_{i,1}, \dots, y_{i,m_i} \in z_{i,m_i}) \neg \phi_i(x_1, \dots, x_n, y_{i,1}, \dots, y_{i,m_i}),$$

is satisfied by $\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}})$. In this case, let $\mathbf{y}_{i,1}, \dots, \mathbf{y}_{i,m_i}$ be m_i elements that, together with $\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}})$, satisfy

$$\neg \phi_i(x_1, \dots, x_n, y_{i,1}, \dots, y_{i,m_i}).$$

It can easily be seen that since \mathbf{W} is downward and upward uniform, all the \mathbf{M} -images of rotors among $\mathbf{M}^{-1}(\mathbf{y}_{i,1}, G_{\mathbf{W}}), \dots, \mathbf{M}^{-1}(\mathbf{y}_{i,m_i}, G_{\mathbf{W}})$ can be assumed to be of level less than m_i . From this it follows that the m_i -unrolling of $G_{\mathbf{W}}^{\mathcal{F}}$ would be sufficient to check whether φ is satisfied by $\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}})$ and hence by $\mathbf{v}_1, \dots, \mathbf{v}_n$. \square

4. SMALL-MODEL PROPERTY

In consequence of the above series of results, the decidability result announced at the beginning of this paper will ensue from our ability to place a bound on the size of the core of a set representing a family potentially satisfying a restricted BSR-formula $\varphi(x_1, \dots, x_n)$: once that bound is known, from the given φ we can determine a list of all candidate encodings, and testing φ for satisfiability will amount to checking that one of them actually encodes a satisfying tuple.

Lemma 4.1. *Given $\mathcal{F} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ and an integer m , there exist a computable function $f(n, m)$ and a set $\mathbf{W}^* \subseteq \text{TrCl}(\mathcal{F})$, together with a rotor system, such that:*

- (1) \mathbf{W}^* represents \mathcal{F} in upward and downward uniform fashion;
- (2) $\{\mathbf{M}(\mathbf{v}_1, G_{\mathbf{W}^*}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\mathbf{W}^*})\}$ satisfy any BSR-formula in n free variables and m universally quantified variables satisfied by \mathcal{F} ;
- (3) the size $\|G_{\mathbf{W}^*}^{\mathcal{F}}\|$ of $G_{\mathbf{W}^*}^{\mathcal{F}}$ (the number of nodes plus the number of edges) meets the inequality $\|G_{\mathbf{W}^*}^{\mathcal{F}}\| \leq f(n, m)$.

Proof. We assume, without loss of generality, that $\mathbf{v}_1 = \emptyset$. Recalling Lemma (3.3), assume that \mathbf{W} is a downward and upward uniform set representing \mathcal{F} whose core (finite, as ever) has least possible cardinality. To prove our claim by induction on the number n of free (understood as existentially bound) variables, we add the following extra condition to the above clauses (1)–(3):

- (4) for all $\mathbf{u} \in \mathbf{W}^*$,

- if $\text{rk}(\mathbf{M}(\mathbf{u}, \mathbf{W}))$ is a successor,⁵ ϱ , then $\exists \mathbf{u}' \in \mathbf{u}(\mathbf{u}' \in \mathbf{W}^* \wedge \text{rk}(\mathbf{M}(\mathbf{u}', \mathbf{W})) = \varrho - 1)$;
- if $\text{rk}(\mathbf{M}(\mathbf{u}, \mathbf{W}))$ is a limit ordinal,⁶ then all \mathbf{W} -rotors in the transitive closure of \mathbf{u} and ascribed to $\text{rk}(\mathbf{u})$ are in \mathbf{W}^* .

Case $n = 1$ is trivial, as in this case $\mathbf{W}^* = \{\mathbf{v}_1\} (= \mathcal{F})$ will satisfy (1)–(4).

In case $n > 1$, after isolating as \mathbf{v}_n an element of maximum rank in \mathcal{F} , consider the family

$$\mathcal{F}' = \{\mathbf{M}(\mathbf{v}_1, \mathbf{W}), \dots, \mathbf{M}(\mathbf{v}_{n-1}, \mathbf{W})\},$$

and let \mathbf{W}' satisfy analogs of (1)–(4) relative to \mathcal{F}' .

To obtain a bounded-core \mathbf{W}^* representing \mathcal{F} and satisfying (1)–(4), we will construct a list $\mathbf{W}^0, \mathbf{W}^1, \dots, \mathbf{W}^k, \dots$ issuing from $\mathbf{W}^0 = \mathbf{M}^{-1}(\mathbf{W}')$ and $\mathbf{W}^1 = \mathbf{W}^0 \cup \{\mathbf{v}_n\}$, where the inclusions $\mathbf{W}^k \subsetneq \mathbf{W}^{k+1} \subsetneq \mathbf{W}$ hold as long as the component \mathbf{W}^{k+1} must be introduced, namely as long as putting $\mathbf{W}^* = \mathbf{W}^k$ would not suffice to ensure injectivity of the Mostowski collapse (associated with $G_{\mathbf{W}^*}$) and the fulfillment of condition (4). After we discuss the rules for progressively prolonging this list, it will be clear that its overall length must be finite; hence we can define \mathbf{W}^* to be its last component. Since, by Lemma (3.3), at most $n - 1$ element per rank can be present in \mathbf{W}' , the construction can be depicted as adding a sequence of elements on the n -th track (see Figure (5)).

If no collisions take place in \mathbf{W}^k , which however fails to meet condition (4), then choose an element \mathbf{y} of $\mathbf{M}(\mathbf{W}^k)$ not belonging to $\mathbf{M}(\mathbf{W}^{k-1})$, so that $\text{rk}(\mathbf{y})$ is as small as possible. If $\text{rk}(\mathbf{y})$ is a successor, then put $\mathbf{W}^{k+1} = \mathbf{W}^k \cup \{\mathbf{w}\}$, where \mathbf{w} is such that $\mathbf{M}(\mathbf{w}, \mathbf{W}) \in \mathbf{y}$ and $\text{rk}(\mathbf{M}(\mathbf{w}, \mathbf{W})) + 1 = \text{rk}(\mathbf{y})$; otherwise, let $\mathbf{W}^{k+1} \setminus \mathbf{W}^k$ be the set of all rotors ascribed to $\text{rk}(\mathbf{M}^{-1}(\mathbf{y}, \mathbf{W}))$ which belong to $\text{TrCl}(\mathbf{M}^{-1}(\mathbf{y}, \mathbf{W}))$.

If a collision takes place in \mathbf{W}^k , the reader can verify that only one element \mathbf{x} in $\mathbf{W}^k \setminus \mathbf{W}^{k-1}$ can collide with an \mathbf{x}' in \mathbf{W}^{k-1} : either a rotor of least rank, or the unique element of $\mathbf{W}^k \setminus \mathbf{W}^{k-1}$, when $|\mathbf{W}^k \setminus \mathbf{W}^{k-1}| = 1$. Then pick an element $\mathbf{w} \notin \mathbf{W}^k$ in the symmetric difference $\mathbf{x} \triangle \mathbf{x}' = (\mathbf{x} \setminus \mathbf{x}') \cup (\mathbf{x}' \setminus \mathbf{x})$.

If $\mathbf{M}(\mathbf{x}, \mathbf{W})$ has limit rank, then let $\mathbf{W}^{k+1} \setminus \mathbf{W}^k$ be the set of all rotors ascribed to $\text{rk}(\mathbf{x})$ which belong to $\text{TrCl}(\mathbf{x})$. Otherwise we act as described in the following, depending on whether

- i) $\text{rk}(\mathbf{x}') > \text{rk}(\mathbf{x})$,
 - ii) $\text{rk}(\mathbf{x}') = \text{rk}(\mathbf{x})$, or
 - iii) $\text{rk}(\mathbf{x}) > \text{rk}(\mathbf{x}')$.
- i) This case cannot occur as otherwise, by (4), in \mathbf{W}^{k-1} there would be an element of \mathbf{x}' endowed with rank greater than or equal to $\text{rk}(\mathbf{x})$, and hence no collision could take place.
 - ii) Let $\mathbf{W}^{k+1} \setminus \mathbf{W}^k = \{\mathbf{w}\}$ (Notice that (4) ensures that \mathbf{W}^k owns an element \mathbf{x} of maximum rank in this case.)
 - iii) Let $\mathbf{W}^{k+1} \setminus \mathbf{W}^k = \{\mathbf{w}\}$, with \mathbf{w} an element of \mathbf{x} of rank $\text{rk}(\mathbf{x}) - 1$.

It is straightforward to check that the proposed action will enforce injectivity, save possibly on the element of least rank in $\mathbf{W}^{k+1} \setminus \mathbf{W}^k$, without disrupting (4).

Moreover, at the end, it can easily be seen that the elements of $\mathbf{W}^* \setminus \mathbf{W}^0$ have pairwise distinct ranks.

⁵Notice that this does not mean that $\text{rk}(\mathbf{u})$ is a successor.

⁶Notice that this does not mean that $\text{rk}(\mathbf{u})$ is a limit ordinal.

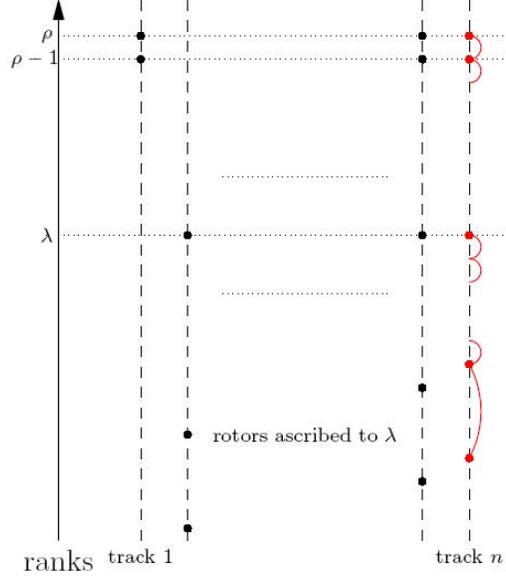


FIGURE 5. Elements added to \mathbf{W}' in order to obtain \mathbf{W}^* can be thought as “placed on the the n -th track”.

Consider any BSR-formula φ in n existentially and m universally quantified variables satisfied by \mathcal{F} . As for the core of \mathbf{W}^* , the minimality of $|c(\mathbf{W})|$ guarantees that $|c(\mathbf{W})| = |c(\mathbf{W}^*)|$. Its size can be determined as the sum of the following two values:

$$a = |\{\mathbf{w} \in c(\mathbf{W}^*) \mid |c(\mathbf{W}^*)^{\text{rk}(\mathbf{w})}| = 1\}|;$$

$$b = |\{\mathbf{w} \in c(\mathbf{W}^*) \mid |c(\mathbf{W}^*)^{\text{rk}(\mathbf{w})}| > 1\}|.$$

Clearly $b \leq 2 \cdot |\mathbf{M}^{-1}(c(\mathbf{W}'))| = 2 \cdot |c(\mathbf{W}')|$ (see Figure (5)) and hence, by the inductive hypothesis, b is bounded by $2 \cdot f(n-1, m)$.

In order to put a bound on a , we prove that any collection of elements $\mathbf{u}_1, \dots, \mathbf{u}_k \in \mathbf{W}^*$ such that for $i \in \{1, \dots, k\}$:

$$|\{\mathbf{w} \in c(\mathbf{W}^*) \mid \text{rk}(\mathbf{w}) = \text{rk}(\mathbf{u}_i)\}| = 1,$$

and $\mathbf{u}_1 \ni \mathbf{u}_2 \ni \dots \ni \mathbf{u}_k$, can be limited in length.

We begin by treating the case when $\mathbf{u}_i \not\ni \mathbf{u}_j$ whenever $j > i + 1$. In this case, let Σ be the alphabet whose characters are the subsets of the finite set $c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)}$. Consider then the string of sets:

$$\{\mathbf{w} \in c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)} \mid \mathbf{u}_1 \in \mathbf{w}\} \dots \{\mathbf{w} \in c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)} \mid \mathbf{u}_k \in \mathbf{w}\}.$$

that we will denote by $u_1 \dots u_k$, with $u_i \in \Sigma$ for $i \in \{1, \dots, k\}$.

We claim that the minimality of $|c(\mathbf{W}^*)|$ implies that the above string cannot own a repeated substring of length exceeding m . In order to prove the claim, arguing by contradiction let us assume that this is not the case and let $j < j' < k$ be such that

$$u_j \dots u_{j+m} = u_{j'} \dots u_{j'+m}.$$

Let $\overline{\mathbf{W}}$ be the set resulting from \mathbf{W}^* through the replacement of \mathbf{u}_{j+h} by $\mathbf{u}_{j'+h}$, for $0 \leq h \leq m$, in any $\mathbf{w} \in \mathbf{W}^*$ having \mathbf{u}_{j+h} as an element. Formally:

$$\overline{\mathbf{W}} = \{(\mathbf{w} \setminus \{\mathbf{u}_{j+h}\}) \cup \{\mathbf{u}_{j'+h}\} : \mathbf{w} \in \mathbf{W}^*, 0 \leq h \leq m \mid \mathbf{u}_{j+h} \in \mathbf{w}\}.$$

We want to prove that the Mostowski collapse of $\overline{\mathbf{W}}$, still satisfies φ . As $|c(\overline{\mathbf{W}})| < |c(\mathbf{W}^*)|$, this would contradict the minimality of $|c(\mathbf{W}^*)|$.

First of all notice that the Mostowski collapse of $G_{\overline{\mathbf{W}}}$ is injective (details are left to the reader). To see that if \mathcal{F} satisfies φ then also $\mathbf{M}(G_{\overline{\mathbf{W}}})$ satisfies it, we proceed again by contradiction. Reasoning as in the proof of Proposition (3.1), we assume that $\mathbf{y}_{i,1}, \dots, \mathbf{y}_{i,m_i}$ are m_i elements which, together with $\mathbf{M}(\mathbf{v}_1, G_{\overline{\mathbf{W}}}), \dots, \mathbf{M}(\mathbf{v}_n, G_{\overline{\mathbf{W}}})$, satisfy

$$\neg\phi_i(x_1, \dots, x_n, y_{i,1}, \dots, y_{i,m_i}),$$

with ϕ_i conjunct of φ .

If none of $\mathbf{y}_{i,1}, \dots, \mathbf{y}_{i,m_i}$ is among $\mathbf{u}_{j'}, \dots, \mathbf{u}_{j'+m}$, then $\mathbf{y}_{i,1}, \dots, \mathbf{y}_{i,m_i}$ would prove the unsatisfiability of φ by \mathcal{F} .

Otherwise, let $\mathbf{y}_{i,h} = \mathbf{u}_{j'+r}$. Since $m \geq m_i$, the following two cases cannot occur together:

- (1) $\mathbf{u}_{j-1}, \mathbf{u}_{j'}, \dots, \mathbf{u}_{j'+r} \in \{\mathbf{y}_{i,1}, \dots, \mathbf{y}_{i,m_i}\}$;
- (2) $\mathbf{u}_{j'+r}, \dots, \mathbf{u}_{j'+m}, \mathbf{u}_{j'+m+1} \in \{\mathbf{y}_{i,1}, \dots, \mathbf{y}_{i,m_i}\}$.

By replacing every element $\mathbf{y}_{i,h} = \mathbf{u}_{j'+r}$ by \mathbf{u}_{j+r} whenever the first case occurs, that an m_i -tuple of sets satisfying $\neg\phi_i$ together with \mathcal{F} can be obtained, as the reader can verify. This contradicts the satisfiability of φ by \mathcal{F} .

The last remaining case is the one in which in the list

$$\mathbf{u}_1 \ni \mathbf{u}_2 \ni \dots \ni \mathbf{u}_k,$$

there exists some index i for which $\mathbf{u}_i \ni \mathbf{u}_j$ and $j > i + 1$. We claim that there are at most $|c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)}|$ such indexes and, moreover, that the overall number of \mathbf{u}_h 's such that $i > h > j$ is again limited by $|c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)}|$. To see this notice that, given $\mathbf{u}_i, \mathbf{u}_j$ such that $\mathbf{u}_i \ni \mathbf{u}_j$ with $j > i + 1$, either \mathbf{u}_h with $i > h > j$ differentiates a pair of elements in $c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)}$ or it can be eliminated without causing any collision and thereby contradicting the minimality of $c(\mathbf{W}^*)$. This, by Proposition (1.1), bounds to $|c(\mathbf{W}^*)^{>\text{rk}(\mathbf{u}_1)}|$ the overall number of possible h 's; from such a bound also the bound on the number of possible i 's follows.

The above two bounds on the length of chains of elements in

$$\{\mathbf{w} \in c(\mathbf{W}^*) \mid |c(\mathbf{W}^*)^{\text{rk}(\mathbf{w})}| = 1\},$$

allow us to recursively specify the function $f(n, m)$ and conclude our proof of point (3). □

We can now conclude with the following:

Theorem 4.2. *The BSR-class is decidable in Set Theory.*

Proof. By Lemma (2.3) we have that the satisfiability problem for the BSR-class is equivalent to the satisfiability problem for the restricted BSR-class.

By Lemmas (3.3) and (4.1), if a formula in n variables is satisfied by a family \mathcal{F} , then there exists a downward and upward uniform set \mathbf{W} representing \mathcal{F} ; moreover, under these hypotheses, the size of an encoding $G_{\mathbf{W}}^{\mathcal{F}}$ of \mathbf{W} is bounded by a computable value g .

Accordingly, we can list all possible encodings of size less than g and apply Proposition (3.5) to test whether any of them is the encoding of a satisfying tuple. If this is not the case, we can declare the formula to be unsatisfiable. \square

REFERENCES

- [BFOS81] M. Breban, A. Ferro, E. G. Omodeo, and J. T. Schwartz. Decision Procedures for Elementary Sublanguages of Set Theory. II: Formulas involving Restricted Quantifiers, together with Ordinal, Integer, Map, and Domain Notions. *Comm. Pure Appl. Math.*, 34:177–195, 1981.
- [BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.
- [Bol86] B. Bollobás. *Combinatorics*. Cambridge University Press, 1986.
- [BP06] D. Bellè and F. Parlamento. Truth in \mathbf{V} for $\exists^*\forall$ -sentences is decidable. *The Journal of Symbolic Logic*, 71(4):1200–1222, 2006.
- [DG79] B. Dreben and W.D. Goldfarb. *The Decision Problem. Solvable Classes of Quantificational Formulas*. Addison-Wesley, Reading, MA, 1979.
- [Lev79] A. Levy. *Basic Set Theory*. Perspectives in Mathematical Logic. Springer-Verlag, 1979.
- [Lew79] H.R. Lewis. *Unsolvable Classes of Quantificational Formulas*. Addison-Wesley, Reading, MA, 1979.
- [OP95] E. G. Omodeo and A. Policriti. Solvable set/hyperset contexts: I. Some decision procedures for the pure, finite case. *Comm. Pure Appl. Math.*, 48(9-10):1123–1155, 1995. Special issue in honor of Jacob T. Schwartz.
- [OPP93] E. G. Omodeo, F. Parlamento, and A. Policriti. A derived algorithm for evaluating ε -expressions over abstract sets. *Journal of Symbolic Computation*, 15(5-6):673–704, 1993. Special issue on Automatic Programming, A. W. Biermann and W. Bibel (eds.).
- [OPP96] E. G. Omodeo, F. Parlamento, and A. Policriti. Decidability of $\exists^*\forall$ -sentences in membership theories. *Mathematical Logic Quarterly*, 42(1):41–58, 1996.
- [PP88] F. Parlamento and A. Policriti. The Logically Simplest Form of the Infinity Axiom. *Proceedings of the American Mathematical Society*, 103(1):274–276, May 1988.
- [PP90] F. Parlamento and A. Policriti. Note on: The Logically Simplest Form of the Infinity Axiom. *Proceedings of the American Mathematical Society*, 108(1), 1990.
- [PP91a] F. Parlamento and A. Policriti. Decision procedures for elementary sublanguages of set theory: XIII. Model graphs, reflection and decidability. 7:271–284, 1991.
- [PP91b] F. Parlamento and A. Policriti. Expressing Infinity without Foundation. *Journal of Symbolic Logic*, 56(4):1230–1235, 1991.
- [PPR97] F. Parlamento, A. Policriti, and K. P. S. B. Rao. Witnessing Differences Without Redundancies. *Proceedings of the American Mathematical Society*, 125(2):587–594, February 1997.
- [Ram28] F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30(2):338–384, 1928.

APPENDIX: ALGORITHMIC SPECIFICATIONS

The procedures ‘represents’ (with its subordinate ‘findRotors’) and ‘reprUniformly’ (with its subordinate ‘findUniformRotors’) in this appendix specify in detail the (perpetual) construction of a set W representing a given finite family \mathcal{F} , as carried out within the proof of Lemma (3.2) and within the proof of Lemma (3.3), respectively. Both procedures receive \mathcal{F} as an input parameter and return W (along with additional information concerning the rotors) as a result; the second input parameter of ‘reprUniformly’, \mathcal{A} , should be actualized as the empty set at top level and will grow progressively across recursion, inside the core of W .

```

procedure represents( $\mathcal{F}$ );
  assert  $|\mathcal{F}| < \omega$ ;
   $\mu := \bigcup \{ \text{rk}(x) : x \in \mathcal{F} \}$ ;
   $\mathcal{T} := \{ x \in \mathcal{F} \mid \text{rk}(x) = \mu \}$ ;
  if  $|\mathcal{F}| \leq 1$  then
    return  $[\mathcal{F}, \emptyset, \mathcal{F}]$ ;
  elseif  $\exists \varrho \in \mu \mid \mu = \varrho + 1$  then
     $[\mathbf{W}', \text{rotors}', \text{core}'] := \text{represents}((\mathcal{F} \setminus \mathcal{T}) \cup \text{elite}(\mathcal{T}))$ ;
    return  $[\mathbf{W}' \cup \mathcal{T}, \text{rotors}', \text{core}' \cup \mathcal{T}]$ ;
  else
     $\alpha := \text{rk}((\mathcal{F} \setminus \mathcal{T}) \cup \text{differentiators}(\mathcal{F}))$ ;
     $[\mathbf{W}_0, \text{rotors}_0, \text{core}_0] := \text{represents}((\mathcal{F} \setminus \mathcal{T}) \cup \{ v^{<\alpha} : v \in \mathcal{T} \})$ ;
     $\text{rotors}_1 := \text{findRotors}(\alpha, \mathcal{T})$ ;
    return  $[(\mathbf{W}_0 \cap \text{TrCl}(\mathcal{F})) \cup \mathcal{T} \cup \bigcup \text{rotors}_1,$ 
       $\text{rotors}_0 \cup \text{rotors}_1, (\text{core}_0 \cap \text{TrCl}(\mathcal{F})) \cup \mathcal{T}]$ ;
  end if;
end represents;

procedure findRotors( $\alpha, \mathcal{T}$ );
  assert  $|\mathcal{T}| < \omega$  &
     $(\exists \lambda \mid \alpha < \lambda \ \& \ \{ \text{rk}(v) : v \in \mathcal{T} \} = \{ \lambda \} \ \& \ \lambda \neq \emptyset \ \& \ (\forall \varrho \in \lambda \mid \lambda \neq \varrho + 1))$ ;
   $\vec{v} := [v : v \in \mathcal{T}]$ ;  $\beta := \alpha$ ;  $\mathcal{Z} := []$ ;
  for  $j \in \omega$  loop
     $\vec{z} := []$ ;
    for  $v = \vec{v}(j)$  loop
       $\vec{z} := \vec{z}$  with  $(z := \text{arb}(v^{>\beta}))$ ;  $\beta := \text{rk}(z)$ ;
    end loop;
     $\mathcal{Z} := \mathcal{Z}$  with  $\vec{z}$ ;
  end loop;
   $j := 1$ ;
  repeat  $\vec{t} := \text{template}(\mathcal{Z}(j))$ ;  $j := j + 1$ ;
  until  $|\{ r \in \omega \setminus j \mid \text{template}(\mathcal{Z}(r)) = \vec{t} \}| = \omega$ ;
  return  $\{ \{ \mathcal{Z}(r)(\ell) : r \in \omega \setminus 1 \mid \text{template}(\mathcal{Z}(r)) = \vec{t} \} :$ 
     $\ell \in \{1, \dots, |\mathcal{T}|\} \mid \text{inCycle}(\ell, \vec{t}) \}$ ;

  procedure template( $\vec{z}$ );
    return  $[\text{if } \exists v = \vec{v}(p) \mid z = v^{<\text{rk}(z)} \text{ then } p \text{ else } 0 \text{ end if} : z = \vec{z}(\ell)]$ ;
  end template;

end findRotors;

procedure inCycle( $\ell, \vec{t}$ );
   $\text{seen} := \{0\}$ ;  $p := \ell$ ;
  while  $p \notin \text{seen}$  loop  $\text{seen} := \text{seen}$  with  $p$ ;  $p := \vec{t}(p)$  end loop;
  return  $\vec{t}(\ell) \neq \ell \ \& \ \ell \in \text{seen}$ ;
end inCycle;

```

```

procedure reprUniformly( $\mathcal{F}, \mathcal{A}$ );
  assert  $|\mathcal{F}| < \omega$ ;
   $\mu := \bigcup \{ \text{rk}(x) : x \in \mathcal{F} \}$ ;
   $\mathcal{T} := \{ x \in \mathcal{F} \mid \text{rk}(x) = \mu \}$ ;
  if  $|\mathcal{F}| \leq 1$  then
    return  $[\mathcal{F}, \emptyset]$ ;
  elseif  $\exists \varrho \in \mu \mid \mu = \varrho + 1$  then
     $[\mathbf{W}', \text{rotors}'] := \text{reprUniformly}((\mathcal{F} \setminus \mathcal{T}) \cup \text{elite}(\mathcal{T}), \mathcal{A} \cup \mathcal{T})$ ;
    return  $[\mathbf{W}' \cup \mathcal{T}, \text{rotors}']$ ;
  else
     $\alpha := \text{rk}((\mathcal{F} \setminus \mathcal{T}) \cup \text{differentiators}(\mathcal{F}))$ ;
     $[\mathbf{W}_0, \text{rotors}_0] := \text{reprUniformly}((\mathcal{F} \setminus \mathcal{T}) \cup \{ v^{<\alpha} : v \in \mathcal{T} \}, \mathcal{A} \cup \mathcal{T})$ ;
     $\text{rotors}_1 := \text{findUniformRotors}(\alpha, \mathcal{T}, \mathcal{A})$ ;
    return  $[(\mathbf{W}_0 \cap \text{TrCl}(\mathcal{F})) \cup \mathcal{T} \cup \bigcup \text{rotors}_1, \text{rotors}_0 \cup \text{rotors}_1]$ ;
  end if;
end reprUniformly;

procedure findUniformRotors( $\alpha, \mathcal{T}, \mathcal{A}$ );
  assert  $|\mathcal{T}| < \omega$  &
     $(\exists \lambda \mid \alpha < \lambda \ \& \ \{ \text{rk}(v) : v \in \mathcal{T} \} = \{ \lambda \} \ \& \ \lambda \neq \emptyset \ \& \ (\forall \varrho \in \lambda \mid \lambda \neq \varrho + 1))$ ;
   $\vec{v} := [v : v \in \mathcal{T}]$ ;    $\beta := \alpha$ ;    $\mathcal{Z} := []$ ;
  for  $j \in \omega$  loop
     $\vec{z} := []$ ;
    for  $v = \vec{v}(j)$  loop
       $\vec{z} := \vec{z}$  with  $(z := \text{arb}(v^{>\beta}))$ ;    $\beta := \text{rk}(z)$ ;
    end loop;
     $\mathcal{Z} := \mathcal{Z}$  with  $\vec{z}$ ;
  end loop;
   $j := 1$ ;
  repeat  $\vec{t} := \text{template}(\mathcal{Z}(j))$ ;    $j := j + 1$ ;
  until  $|\{ r \in \omega \setminus j \mid \text{template}(\mathcal{Z}(r)) = \vec{t} \}| = \omega$ ;
   $\mathcal{Z} := \text{shrink}([\mathcal{Z}(r) : r \in \omega \setminus 1 \mid \text{template}(\mathcal{Z}(r)) = \vec{t}],$ 
     $\vec{m} := [\ell \in \{1, \dots, |\mathcal{T}|\} \mid \text{inCycle}(\ell, \vec{t})], \quad \mathcal{A} \cup \mathcal{T})$ ;
  return  $\{ \{ \mathcal{Z}(r)(\ell) : r \in \omega \setminus 1 \} : \ell = \vec{m}(k) \}$ ;

procedure shrink( $\mathcal{Z}, \vec{m}, \mathcal{A}$ );
  assert  $|\mathcal{A}| < \omega$  &  $(\forall j \in \omega \setminus 1, a \in \mathcal{A}, \ell = \vec{m}(k) \mid \text{rk}(\mathcal{Z}(j)(\ell)) < \text{rk}(a))$ ;
   $j := 1$ ;
  repeat  $\vec{s} := [\{ a \in \mathcal{A} \mid \mathcal{Z}(j)(\ell) \in a \} : \ell = \vec{m}(k)]$ ;    $j := j + 1$ ;
   $\mathcal{S} := \{ q \in \omega \setminus j \mid [\{ a \in \mathcal{A} \mid \mathcal{Z}(q)(\ell) \in a \} : \ell = \vec{m}(k)] = \vec{s} \}$ ;
  until  $|\mathcal{S}| = \omega$ ;
  return  $\mathcal{S}$ ;
end shrink;
procedure template( $\vec{z}$ );
  return  $[\text{if } \exists v = \vec{v}(p) \mid z = v^{<\text{rk}(z)} \text{ then } p \text{ else } 0 \text{ end if} : z = \vec{z}(\ell)]$ ;
end template;
end findUniformRotors;

```

E. OMODEO: DIPARTIMENTO DI MATEMATICA E INFORMATICA, UNIVERSITÀ DI TRIESTE, 34127,
VIA VALERIO 12/B, ITALY.

E-mail address: `eomodeo@units.it`

A. POLICRITI: DIPARTIMENTO DI MATEMATICA E INFORMATICA, UNIVERSITÀ DI UDINE, 33100,
VIA DELLE SCIENZE 206, ITALY.

E-mail address: `policriti@dimi.uniud.it`

A refined calculus for Intuitionistic Propositional Logic

Mauro Ferrari¹, Camillo Fiorentini², Guido Fiorino³

¹ Dipartimento di Informatica e Comunicazione, Università degli Studi dell’Insubria
Via Mazzini 5, 21100, Varese, Italy

² Dipartimento di Scienze dell’Informazione, Università degli Studi di Milano
Via Comelico, 39, 20135 Milano, Italy

³ Dipartimento di Metodi Quantitativi per le Scienze Economiche Aziendali,
Università degli Studi di Milano-Bicocca
P.zza dell’Ateneo Nuovo 1, 20126 Milano, Italy[†]

Abstract. Since 1993, when Hudelmaier developed a $O(n \log n)$ -space decision procedure for propositional intuitionistic logic, a lot of work has been done to improve the efficiency of the related proof-search algorithms. This has been done working on proof-search strategies and on implementation structures more than on logical properties of the calculi. In this paper we provide a tableau calculus using the signs **T**, **F** and **F_e**, with a new set of rules to treat signed formulas of the kind $\mathbf{T}((A \rightarrow B) \rightarrow C)$. The main feature of the calculus is to reduce both the non-determinism in proof-search and the width of proofs with respect to Hudelmaier’s one. The new rules come out from a deep semantic analysis of nested implications and their behavior in the counter-model construction.

1 Introduction

In this paper we present a tableau calculus for propositional Intuitionistic Logic **Int** with a new set of rules to treat signed formulas of the kind $\mathbf{T}((A \rightarrow B) \rightarrow C)$. This calculus collocates itself in a long history of researches on the design of efficient decision procedures for **Int**. In this context, the main concern is the treatment of “positive” implicative formulas, namely implicative formulas having sign **T** in a tableau deduction or occurring in the left-side of a sequent. Indeed, differently from Classical Logic, intuitionistic implication cannot be handled by invertible rules and this makes the decision procedures for Intuitionistic Logic PSPACE-complete (see [10]).

The early calculi for **Int** were based on the re-use of implicative formulas (see, e.g., [7]). The major drawback of this solution is that deductions may have infinite depth, hence some loop-checking mechanism is needed to guarantee termination. To avoid this, Vorob’ev introduced in [11] (in the context of sequent

[†] An extended version of this paper has been submitted to Logical Methods in Computer Science.

calculi) rules to treat signed formulas of the kind $\mathbf{T}(A \rightarrow B)$ according to the main connective of A ; see also [3, 9], where calculi with analogous properties are given. In these cases, the re-use of formulas is avoided by replacing $\mathbf{T}(A \rightarrow B)$ with "simpler" formulas built up from the subformulas of $A \rightarrow B$; moreover, suitable measures on formulas are defined, which guarantee the termination of derivations. But, if on the one hand decision procedures for these calculi do not need loop-checking mechanisms, on the other hand the rules to treat formulas of the kind $\mathbf{T}(A \vee B \rightarrow C)$ and $\mathbf{T}((A \rightarrow B) \rightarrow C)$ give rise to proofs that may be of exponential length in the size of the formula to be proved. This problem is overcome in Hudelmaier's sequent calculi described in [8], where proofs have linear length and the related decision procedures require $O(n \log n)$ -space. In this paper we refer to the calculus LG of [8], which has a natural translation in the tableau setting. The novelties of LG essentially regard the treatment of formulas of the kind $\mathbf{T}(A \rightarrow B)$. To save space, in some rules of LG the repetitions of formulas is avoided by introducing new propositional variables. Moreover, LG provides rules to handle the sets containing both $\mathbf{T}(A \rightarrow B)$ and $\mathbf{F}A$, providing a rule for every main connective of A . We remark that in [8], the $O(n \log n)$ -space result is proved for the calculus LE , which improves LG by providing a compact notation to represent the pairs of formulas $\mathbf{F}A, \mathbf{T}(A \rightarrow B)$.

The calculus $\mathcal{T}_{\mathbf{Int}}$ we introduce in this paper is a refinement of Hudelmaier calculus LG . Here, we improve Hudelmaier's rules by giving rules to treat formulas of the kind $\mathbf{T}((A \rightarrow B) \rightarrow C)$, for *all* the main connectives of B , without introducing rules treating pairs of signed formulas. As discussed in the paper, even if $\mathcal{T}_{\mathbf{Int}}$ has the same computational performances of Hudelmaier's calculi, it allows us to define a "better" decision procedure due to the following facts: (i) in general $\mathcal{T}_{\mathbf{Int}}$ proofs have width which is less than that of the corresponding LG proofs; (ii) $\mathcal{T}_{\mathbf{Int}}$ rules introduce a lower degree of non determinism. Thus, both the search space and the dimension of the proofs of $\mathcal{T}_{\mathbf{Int}}$ are narrower than LG . The new rules of $\mathcal{T}_{\mathbf{Int}}$ exploit the ideas used in [4] to get a calculus having proofs of depth bounded by $3n$, which gives rise to an $O(n \log n)$ -space decision procedure.

Finally, we point out that our reasoning is based on semantic tools, whereas [8] uses syntactic techniques (to prove the equivalence between LG and Gentzen calculus, the author has to introduce some intermediate calculi and prove their equivalence). As a by-product, our decision procedure allows us to build a counter-model of A whenever A is not intuitionistically valid.

To conclude, we remark that since [3, 8, 9] the main works on the improvement of decision procedures for Intuitionistic Logic were faced with the definition of "efficient" implementations, see e.g. [6, 1]. Here we reconsider the problem from a purely logical point of view.

The paper is structured as follows: in the next section we introduce the notation and the preliminary definitions. In Section 3 we describe $\mathcal{T}_{\mathbf{Int}}$ and we discuss the main differences with respect to Hudelmaier's calculus LG . In Section 4 we prove $\mathcal{T}_{\mathbf{Int}}$ is sound, while in Section 5 we prove the completeness and some computational complexity properties.

2 Notation and Preliminaries

We consider the propositional language \mathcal{L} based on a denumerable set of propositional variables (atoms) \mathcal{PV} and the logical connectives $\neg, \wedge, \vee, \rightarrow$. We write $A \in \mathcal{L}$ to mean that A is a formula of \mathcal{L} .

Kripke models are the main tool to semantically characterize propositional intuitionistic logic **Int** (see e.g. [2, 5] for the details). A Kripke model for \mathcal{L} is a structure $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$, where $\langle P, \leq, \rho \rangle$ is a finite poset with minimum element ρ , and \Vdash is the *forcing relation*, namely a binary relation on $P \times \mathcal{PV}$ satisfying the monotonicity condition: $\alpha \Vdash p$ and $\alpha \leq \beta$ implies $\beta \Vdash p$. The forcing relation is extended to arbitrary formulas of \mathcal{L} as follows:

1. $\alpha \Vdash A \wedge B$ iff $\alpha \Vdash A$ and $\alpha \Vdash B$;
2. $\alpha \Vdash A \vee B$ iff $\alpha \Vdash A$ or $\alpha \Vdash B$;
3. $\alpha \Vdash A \rightarrow B$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash A$ implies $\beta \Vdash B$;
4. $\alpha \Vdash \neg A$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash A$ does not hold.

We write $\alpha \not\Vdash A$ to mean that $\alpha \Vdash A$ does not hold. It is easy to check that the monotonicity property holds for arbitrary formulas, i.e., for every formula $A \in \mathcal{L}$, $\alpha \Vdash A$ and $\alpha \leq \beta$ implies $\beta \Vdash A$. A formula A is *valid in a Kripke model* $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ iff $\rho \Vdash A$ (by monotonicity property, this means that $\alpha \Vdash A$ for every $\alpha \in P$). It is well-known [2, 5] that propositional intuitionistic logic **Int** coincides with the set of formulas valid in all Kripke models.

3 The Tableau Calculus

The tableau calculus $\mathcal{T}_{\mathbf{Int}}$ we introduce in this section, is a refinement of the one introduced in [9] and [4]. It works on *signed formulas*, namely expressions of the kind \mathbf{TA} , \mathbf{FA} or $\mathbf{F}_c A$, where $A \in \mathcal{L}$. Signed formulas have a natural interpretation in Kripke semantics. Given a Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$, an element $\alpha \in P$ and a signed formula H , we say that α *realizes* H in \underline{K} , and we write $\underline{K}, \alpha \triangleright H$, iff one of the following conditions holds:

- $H = \mathbf{TA}$ and $\alpha \Vdash A$;
- $H = \mathbf{FA}$ and $\alpha \not\Vdash A$;
- $H = \mathbf{F}_c A$ and $\alpha \Vdash \neg A$.

$\underline{K}, \alpha \not\triangleright H$ means that $\underline{K}, \alpha \triangleright H$ does not hold. Given a set S of signed formulas, we write $\underline{K}, \alpha \triangleright S$ to mean that $\underline{K}, \alpha \triangleright H$ for every $H \in S$; we say that S is *realizable* if $\underline{K}, \alpha \triangleright S$ for some \underline{K} and α . We call the *certain* part of S the set

$$S_c = \{\mathbf{TA} \mid \mathbf{TA} \in S\} \cup \{\mathbf{F}_c A \mid \mathbf{F}_c A \in S\}$$

We remark that, by the monotonicity property, $\underline{K}, \alpha \triangleright S$ and $\alpha \leq \beta$ implies $\underline{K}, \beta \triangleright S_c$.

The rules of the tableau calculus $\mathcal{T}_{\mathbf{Int}}$ are shown in Tables 1-3. In the rules we write S, H as a shorthand for $S \cup \{H\}$. Every rule applies to a set of signed

$\frac{S, \mathbf{T}(A \wedge B)}{S, \mathbf{TA}, \mathbf{TB}} \mathbf{T}\wedge$	$\frac{S, \mathbf{F}(A \wedge B)}{S, \mathbf{FA} \mid S, \mathbf{FB}} \mathbf{F}\wedge$	$\frac{S, \mathbf{F}_c(A \wedge B)}{S, \mathbf{F}_cA \mid S, \mathbf{F}_cB} \mathbf{F}_c\wedge$
$\frac{S, \mathbf{T}(A \vee B)}{S, \mathbf{TA} \mid S, \mathbf{TB}} \mathbf{T}\vee$	$\frac{S, \mathbf{F}(A \vee B)}{S, \mathbf{FA}, \mathbf{FB}} \mathbf{F}\vee$	$\frac{S, \mathbf{F}_c(A \vee B)}{S, \mathbf{F}_cA, \mathbf{F}_cB} \mathbf{F}_c\vee$
Tables 2 and 3	$\frac{S, \mathbf{F}(A \rightarrow B)}{S_c, \mathbf{TA}, \mathbf{FB}} \mathbf{F}\rightarrow$	$\frac{S, \mathbf{F}_c(A \rightarrow B)}{S_c, \mathbf{TA}, \mathbf{F}_cB} \mathbf{F}_c\rightarrow$
$\frac{S, \mathbf{T}(\neg A)}{S, \mathbf{F}_cA} \mathbf{T}\neg$	$\frac{S, \mathbf{F}(\neg A)}{S_c, \mathbf{TA}} \mathbf{F}\neg$	$\frac{S, \mathbf{F}_c(\neg A)}{S_c, \mathbf{TA}} \mathbf{F}_c\neg$
$S_c = \{\mathbf{TA} \mid \mathbf{TA} \in S\} \cup \{\mathbf{F}_cA \mid \mathbf{F}_cA \in S\}$		

Table 1. The $\mathcal{T}_{\mathbf{Int}}$ calculus

formulas, but only acts on the signed formula H explicitly indicated in the premise; we call H the *major premise* of the rule, whereas we call all the other signed formulas *minor premises* of the rule. The sets in the consequence are obtained by decomposing in some way the major premise of the rule and either copying all the minor premises (see e.g. the rule $\mathbf{T}\wedge$ of Table 1) or only copying the certain part of the minor premises (see e.g. the rule $\mathbf{F}\rightarrow$ of Table 1). We call *splitting rules* the rules of $\mathcal{T}_{\mathbf{Int}}$ having two sets in the consequence.

Some rules require additional conditions in order to be applied. The rule $\mathbf{T}\rightarrow$ *certain* of Table 2 can be applied only if $S = S_c$ (that is, the set S of minor premises does not contain \mathbf{F} -signed formulas). The rule MP (*modus ponens*) of Table 2, having $\mathbf{T}(A \rightarrow B)$ as major premise, requires the presence of \mathbf{TA} among the minor premises. We point out that in [8] this rule is given in the restricted form where A is a propositional variable.

Finally, we notice that some rules of Tables 2 and 3 require the introduction of a new atom q , namely a propositional variable q not occurring in the premises of the rule. This expedient goes back to [8] and avoids duplications of non atomic formulas in the consequence of a rule (namely, repetitions of subformulas of the major premise); for instance, without the introduction of q , the consequence of $\mathbf{T}\rightarrow\vee$ should be $S, \mathbf{T}(A \rightarrow C), \mathbf{T}(B \rightarrow C)$, where C occurs twice, and this prevents the definition of a linear complexity measure on sets of signed formulas.

A set S of signed formulas is *contradictory* if either $\{\mathbf{TA}, \mathbf{FA}\} \subseteq S$ or $\{\mathbf{TA}, \mathbf{F}_cA\} \subseteq S$, for some formula A . Clearly, contradictory sets cannot be realized. A *proof table* (or proof tree) for S is a finite tree τ with S as root and such that all the children of a node S' of τ are the sets in the consequence of a rule applied to S' . If all the leaves of τ are contradictory sets, we say that τ is a

$$\frac{S, \mathbf{T}A, \mathbf{T}(A \rightarrow B)}{S, \mathbf{T}A, \mathbf{T}B} MP$$

$$\frac{S, \mathbf{T}(A \rightarrow B)}{S, \mathbf{F}_c A \mid S, \mathbf{T}B} \mathbf{T} \rightarrow \text{certain} \quad \text{if } S = S_c$$

$$\frac{S, \mathbf{T}((A \wedge B) \rightarrow C)}{S, \mathbf{T}(A \rightarrow (B \rightarrow C))} \mathbf{T} \rightarrow \wedge$$

$$\frac{S, \mathbf{T}(\neg A \rightarrow B)}{S_c, \mathbf{T}A \mid S, \mathbf{T}B} \mathbf{T} \rightarrow \neg$$

$$\frac{S, \mathbf{T}((A \vee B) \rightarrow C)}{S, \mathbf{T}(A \rightarrow q), \mathbf{T}(B \rightarrow q), \mathbf{T}(q \rightarrow C)} \mathbf{T} \rightarrow \vee \quad \text{with } q \text{ a new atom}$$

Table 2. Rules for $\mathbf{T} \rightarrow$

closed proof table for S . A closed proof table is a proof of the calculus: a formula A is provable in $\mathcal{T}_{\mathbf{Int}}$ iff there exists a closed proof table for $\{\mathbf{F}A\}$.

To conclude this section we discuss the main novelties of our calculus; in particular we consider the differences among $\mathcal{T}_{\mathbf{Int}}$ and the tableau calculi of [9, 4] and the sequent calculi introduced in [8]. For sequent calculi we present the rules adopting the standard translation into tableau rules.

First of all we notice that the rules of Tables 1 and 2 essentially coincide with those described in [9], where the sign \mathbf{F}_c is introduced to characterize intuitionistic negation. As for the rules of Table 3, they replace the rule

$$\frac{S, \mathbf{T}((A \rightarrow B) \rightarrow C)}{S_c, \mathbf{T}A, \mathbf{F}B, \mathbf{T}(B \rightarrow C) \mid S, \mathbf{T}C} \mathbf{T} \rightarrow \rightarrow$$

of [9], that goes back to [3] and [11] (given in a sequent calculus style), and the rule *Fio* $\rightarrow \rightarrow$ of [4] shown at the end of this section.

The rule $\mathbf{T} \rightarrow \rightarrow$ has been introduced by Vorob'ev to avoid loop-checking in the decision procedure. On the other hand, the double occurrence of formula B in the leftmost conclusion of $\mathbf{T} \rightarrow \rightarrow$ gives rise to deductions that may be of exponential depth in the length of the formula to be proved (see [8, 6] for a detailed discussion). In [8] the problem is solved by introducing, beside the rule $\mathbf{T} \rightarrow \rightarrow$, some rules to treat the leftmost conclusion of $\mathbf{T} \rightarrow \rightarrow$, according to the main connective of B . Moreover, the calculus *LG* of [8] provides rules to handle the pairs of formulas $\mathbf{F}B, \mathbf{T}(B \rightarrow C)$, according to the main connective of B .

$\frac{S, \mathbf{T}((A \rightarrow p) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}p, \mathbf{T}(p \rightarrow C) \mid S, \mathbf{TC}} \mathbf{T} \rightarrow \rightarrow \text{Atom} \quad \text{where } p \in \mathcal{PV}$
$\frac{S, \mathbf{T}((A \rightarrow \neg B) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{TB} \mid S, \mathbf{TC}} \mathbf{T} \rightarrow \rightarrow \neg$
$\frac{S, \mathbf{T}((A \rightarrow (X \wedge Y)) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(X \rightarrow (Y \rightarrow q)), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}} \mathbf{T} \rightarrow \rightarrow \wedge \quad \text{with } q \text{ a new atom}$
$\frac{S, \mathbf{T}((A \rightarrow (X \vee Y)) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(X \rightarrow q), \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}} \mathbf{T} \rightarrow \rightarrow \vee \quad \text{with } q \text{ a new atom}$
$\frac{S, \mathbf{T}((A \rightarrow (X \rightarrow Y)) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{TX}, \mathbf{F}q, \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}} \mathbf{T} \rightarrow \rightarrow \rightarrow \quad \text{with } q \text{ a new atom}$

Table 3. Rules for $\mathbf{T} \rightarrow \rightarrow$

The translation in the tableau setting of the rules of *LG* for $B = X \vee Y$ are:

$$\frac{S, \mathbf{F}(X \vee Y), \mathbf{T}(X \vee Y \rightarrow C)}{S, \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C), \mathbf{FX}, \mathbf{T}(X \rightarrow q)} \text{Hud} \rightarrow \vee_1$$

$$\frac{S, \mathbf{F}(X \vee Y), \mathbf{T}(X \vee Y \rightarrow C)}{S, \mathbf{T}(X \rightarrow q), \mathbf{T}(q \rightarrow C), \mathbf{FY}, \mathbf{T}(Y \rightarrow q)} \text{Hud} \rightarrow \vee_2$$

where q is a propositional variable not occurring in the premises. We remark that to get the completeness both the rules are required. Indeed, to build a proof for $S, \mathbf{T}((A \rightarrow X \vee Y) \rightarrow C)$ (working on the signed formula $\mathbf{T}((A \rightarrow X \vee Y) \rightarrow C)$), in *LG* we firstly have to apply $\mathbf{T} \rightarrow \rightarrow$ rule:

$$\frac{S, \mathbf{T}((A \rightarrow X \vee Y) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}(X \vee Y), \mathbf{T}(X \vee Y \rightarrow C) \mid S, \mathbf{TC}} \mathbf{T} \rightarrow \rightarrow$$

At this point we have to non-deterministically choose which rule to apply between *Hud* $\rightarrow \vee_1$ and *Hud* $\rightarrow \vee_2$. In the former case we get

$$S_c, \mathbf{TA}, \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C), \mathbf{FX}, \mathbf{T}(X \rightarrow q) \mid S, \mathbf{TC}$$

in the latter we get

$$S_c, \mathbf{TA}, \mathbf{T}(X \rightarrow q), \mathbf{T}(q \rightarrow C), \mathbf{FY}, \mathbf{T}(Y \rightarrow q) \mid S, \mathbf{TC}$$

Obviously, to build up a closed proof table it may be necessary to try both rules. In contrast, in \mathcal{T}_{Int} only the application of the $\mathbf{T} \rightarrow \rightarrow \vee$ -rule is required, indeed:

$$\frac{S, \mathbf{T}((A \rightarrow X \vee Y) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(X \rightarrow q), \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}}^{\mathbf{T} \rightarrow \rightarrow \vee}$$

In this case, our rule allows us to reduce the non-determinism in proof-search.

Now, let us consider the rule of *LG* for the case $B = X \wedge Y$

$$\frac{S, \mathbf{F}(X \wedge Y), \mathbf{T}(X \wedge Y \rightarrow C)}{S, \mathbf{FX}, \mathbf{T}(X \rightarrow (Y \rightarrow C)) \mid S, \mathbf{FY}, \mathbf{T}(Y \rightarrow (X \rightarrow C))}^{\text{Hud} \rightarrow \wedge}$$

and let us consider the tableau

$$\frac{\frac{S, \mathbf{T}((A \rightarrow X \wedge Y) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}(X \wedge Y), \mathbf{T}(X \wedge Y \rightarrow C) \mid S, \mathbf{TC}}^{\mathbf{T} \rightarrow \rightarrow}}{S_c, \mathbf{TA}, \mathbf{FX}, \mathbf{T}(X \rightarrow (Y \rightarrow C)) \mid S_c, \mathbf{TA}, \mathbf{FY}, \mathbf{T}(Y \rightarrow (X \rightarrow C)) \mid S, \mathbf{TC}}^{\text{Hud} \rightarrow \wedge}$$

In our calculus, for the same initial set we get:

$$\frac{S, \mathbf{T}((A \rightarrow X \wedge Y) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(X \rightarrow (Y \rightarrow q)), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}}^{\mathbf{T} \rightarrow \rightarrow \wedge}$$

where q is a new propositional variable. We point out that our rule decreases the width of the proof tree. As a matter of fact, to decide the realizability of the initial set, with \mathcal{T}_{Int} two sets have to be decided, instead of three sets as in *LG*.

Finally, let us consider the *LG* rule for the case $B = X \rightarrow Y$:

$$\frac{S, \mathbf{F}(X \rightarrow Y), \mathbf{T}((X \rightarrow Y) \rightarrow C)}{S_c, \mathbf{TX}, \mathbf{FY}, \mathbf{T}(Y \rightarrow C)}^{\text{Hud} \rightarrow \rightarrow}$$

and let us consider the tableau

$$\frac{\frac{S, \mathbf{T}((A \rightarrow (X \rightarrow Y)) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}(X \rightarrow Y), \mathbf{T}((X \rightarrow Y) \rightarrow C) \mid S, \mathbf{TC}}^{\mathbf{T} \rightarrow \rightarrow}}{S_c, \mathbf{TA}, \mathbf{TX}, \mathbf{FY}, \mathbf{T}(Y \rightarrow C) \mid S, \mathbf{TC}}^{\text{Hud} \rightarrow \rightarrow}$$

In our calculus the corresponding tableau is

$$\frac{S, \mathbf{T}((A \rightarrow (X \rightarrow Y)) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{TX}, \mathbf{F}q, \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}}^{\mathbf{T} \rightarrow \rightarrow \rightarrow}$$

with q a new propositional variable. We point out that we apply one non-invertible rule, whereas in the previous proof tree two non-invertible rules are

required. A deeper discussion about the proof-search strategy is given after the Completeness Theorem (Section 5).

We emphasize that rules of Table 3 are a refinement of the rule

$$\frac{S, \mathbf{T}((A \rightarrow B) \rightarrow C)}{S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(B \rightarrow q), \mathbf{T}(q \rightarrow C) \mid S, \mathbf{TC}}^{Fio \rightarrow \rightarrow} \quad \text{with } q \text{ a new atom}$$

introduced in [4]. The calculus of [4] gives rise to proof trees having depth bounded by $6n$, where n is the length of the formula to be proved, and this yields a $O(n \log n)$ -space decision procedure for **Int**. Rules of Table 3 are obtained by specializing rule $Fio \rightarrow \rightarrow$ according to the main connective of B . How we discuss later, the new rules allow us to get proof trees having depth $3n$ at most.

4 Soundness

In order to prove the soundness \mathcal{T}_{Int} we show that every rule of \mathcal{T}_{Int} preserves realizability. The following lemma is helpful to treat the rules of Table 3.

Lemma 1. *Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model and let $\alpha \in P$ such that*

$$\underline{K}, \alpha \triangleright S, \mathbf{T}((A \rightarrow B) \rightarrow C) \quad \text{and} \quad \underline{K}, \alpha \not\triangleright \mathbf{TC}$$

Let \mathcal{V} be the set of propositional variables occurring in $S \cup \{\mathbf{T}((A \rightarrow B) \rightarrow C)\}$ and let q be a propositional variable such that $q \notin \mathcal{V}$. Then, there exists a Kripke model $\underline{K}' = \langle P', \leq', \rho', \Vdash' \rangle$ and $\alpha' \in P'$ such that

$$\underline{K}', \alpha' \triangleright S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(B \rightarrow q), \mathbf{T}(q \rightarrow B), \mathbf{T}(q \rightarrow C)$$

Proof. Let $\underline{K}' = \langle P, \leq, \rho, \Vdash' \rangle$ be the Kripke model based on the same poset $\langle P, \rho, \leq \rangle$ of \underline{K} with \Vdash' defined as follows:

- if $p \in \mathcal{V}$, then, for every $\gamma \in P$, $\gamma \Vdash' p$ iff $\gamma \Vdash p$;
- for every $\gamma \in P$, $\gamma \Vdash' q$ iff $\gamma \Vdash B$;
- if $p \notin \mathcal{V} \cup \{q\}$, then, for every $\gamma \in P$, $\gamma \not\Vdash' p$.

It is easy to check that, if H is a formula whose propositional variables belong to \mathcal{V} and $\gamma \in P$, $\gamma \Vdash H$ iff $\gamma \Vdash' H$. In particular, by the hypothesis $\alpha \Vdash (A \rightarrow B) \rightarrow C$ and $\alpha \not\Vdash C$, we get $\alpha \Vdash' (A \rightarrow B) \rightarrow C$ and $\alpha \not\Vdash' C$. This implies $\alpha \not\Vdash' A \rightarrow B$, therefore there exists $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash' A$ and $\beta \not\Vdash' B$. We get:

1. $\beta \Vdash' B \rightarrow q$ and $\beta \Vdash' q \rightarrow B$ (by definition of \Vdash' on q);
2. $\beta \Vdash' A$ and $\beta \not\Vdash' q$ (by (1) and by the fact that $\beta \not\Vdash' B$);
3. $\beta \Vdash' q \rightarrow C$ (by the fact that $\beta \Vdash' (A \rightarrow B) \rightarrow C$ and $\beta \Vdash' q \rightarrow B$).

Summarizing, by Points (1)-(3) we conclude

$$\underline{K}', \beta \triangleright S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(B \rightarrow q), \mathbf{T}(q \rightarrow B), \mathbf{T}(q \rightarrow C)$$

which proves the assertion. \square

Now we prove that the rules of $\mathcal{T}_{\mathbf{Int}}$ preserve realizability:

Lemma 2. *Let S be a set of signed formulas, let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model, let $\alpha \in P$ such that $\underline{K}, \alpha \triangleright S$ and let R be a rule of $\mathcal{T}_{\mathbf{Int}}$ applicable to S . Then, there exist a set S' in the consequence of the rule R , a Kripke model $\underline{K}' = \langle P', \leq', \rho', \Vdash' \rangle$ and $\alpha' \in P'$ such that $\underline{K}', \alpha' \triangleright S'$.*

Proof. We have to analyze the rules R of $\mathcal{T}_{\mathbf{Int}}$; we only discuss the most relevant cases of Tables 2 and 3.

Rule $\mathbf{T} \rightarrow$ certain: let us assume $\underline{K}, \alpha \triangleright S_c, \mathbf{T}(A \rightarrow B)$. By finiteness of P , there is $\phi \in P$ such that $\alpha \leq \phi$ and ϕ is a final element of \underline{K} (that is, for every $\psi \in P$, $\phi \leq \psi$ implies $\phi = \psi$). By the monotonicity property, $\underline{K}, \phi \triangleright S_c, \mathbf{T}(A \rightarrow B)$. If $\phi \Vdash B$, we immediately get $\underline{K}, \alpha \triangleright S_c, \mathbf{TB}$; otherwise $\phi \not\Vdash A$ and, being ϕ a final element, this implies $\phi \Vdash \neg A$, hence $\underline{K}, \phi \triangleright S_c, \mathbf{F}_c A$.

Rule $\mathbf{T} \rightarrow \rightarrow$ Atom: if $\underline{K}, \alpha \triangleright S, \mathbf{T}((A \rightarrow p) \rightarrow C)$, then $\alpha \Vdash (A \rightarrow p) \rightarrow C$, thus either $\alpha \Vdash C$ or $\alpha \not\Vdash A \rightarrow p$. In the first case we immediately deduce that $\underline{K}, \alpha \triangleright S, \mathbf{TC}$. In the second case, there exists $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash A$ and $\beta \not\Vdash p$. Moreover, since $\beta \Vdash (A \rightarrow p) \rightarrow C$, we also have $\beta \Vdash p \rightarrow C$. We conclude that $\underline{K}, \beta \triangleright S_c, \mathbf{TA}, \mathbf{F}p, \mathbf{T}(p \rightarrow C)$.

Rule $\mathbf{T} \rightarrow \rightarrow \vee$: if $\underline{K}, \alpha \triangleright S, \mathbf{T}((A \rightarrow (X \vee Y)) \rightarrow C)$, then $\alpha \Vdash (A \rightarrow (X \vee Y)) \rightarrow C$. If $\alpha \Vdash C$, we immediately get $\underline{K}, \alpha \triangleright S, \mathbf{TC}$. Otherwise, by Lemma 1 there exist a Kripke model $\underline{K}' = \langle P', \leq', \rho' \Vdash' \rangle$ and $\alpha' \in P'$ such that

$$\underline{K}', \alpha' \triangleright S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}((X \vee Y) \rightarrow q), \mathbf{T}(q \rightarrow (X \vee Y)), \mathbf{T}(q \rightarrow C)$$

Since $\alpha' \Vdash' (X \vee Y) \rightarrow q$ implies both $\alpha' \Vdash' X \rightarrow q$ and $\alpha' \Vdash' Y \rightarrow q$, we get $\underline{K}', \alpha' \triangleright S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}(X \rightarrow q), \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C)$.

Rule $\mathbf{T} \rightarrow \rightarrow \rightarrow$: if $\underline{K}, \alpha \triangleright S, \mathbf{T}((A \rightarrow (X \rightarrow Y)) \rightarrow C)$, then $\alpha \Vdash (A \rightarrow (X \rightarrow Y)) \rightarrow C$. If $\alpha \Vdash C$, we immediately get $\underline{K}, \alpha \triangleright S, \mathbf{TC}$. Otherwise, by Lemma 1 there exist a Kripke model $\underline{K}' = \langle P', \leq', \rho' \Vdash' \rangle$ and $\alpha' \in P'$ such that

$$\underline{K}', \alpha' \triangleright S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{T}((X \rightarrow Y) \rightarrow q), \mathbf{T}(q \rightarrow (X \rightarrow Y)), \mathbf{T}(q \rightarrow C)$$

Since $\alpha' \Vdash' (X \rightarrow Y) \rightarrow q$ and $\alpha' \not\Vdash' q$, there exists $\beta' \in P'$ such that $\alpha' \leq' \beta'$, $\beta' \Vdash' X$ and $\beta' \not\Vdash' Y$. Since $\beta' \Vdash' q \rightarrow (X \rightarrow Y)$, we have $\beta' \not\Vdash' q$. Moreover, since $\beta' \Vdash' (X \rightarrow Y) \rightarrow q$, it holds that $\beta' \Vdash' Y \rightarrow q$. Summarizing, we get

$$\underline{K}', \beta' \triangleright S_c, \mathbf{TA}, \mathbf{F}q, \mathbf{TX}, \mathbf{T}(Y \rightarrow q), \mathbf{T}(q \rightarrow C)$$

\square

The previous lemma is the main step to prove the soundness of the calculus $\mathcal{T}_{\mathbf{Int}}$. Indeed, let us assume that there exists a closed proof table τ of $\mathcal{T}_{\mathbf{Int}}$ for $\{\mathbf{F}A\}$. If $\{\mathbf{F}A\}$ is realizable, by the previous lemma there exists a leaf S of τ such that S is realizable, a contradiction (recall that S is a contradictory set). It follows that $\{\mathbf{F}A\}$ is not realizable, and this means that A is valid in all the Kripke models, that is $A \in \mathbf{Int}$. Thus:

Theorem 1 (Soundness). *If there exists a closed proof table for $\{\mathbf{F}A\}$, then A is intuitionistically valid.*

5 Completeness

To prove the completeness we need to introduce the following complexity measure \deg on formulas:

- if p is a propositional variable, then $\deg(p) = 0$;
- $\deg(A \wedge B) = \deg(A) + \deg(B) + 2$;
- $\deg(A \vee B) = \deg(A) + \deg(B) + 3$;
- $\deg(A \rightarrow B) = \deg(A) + \deg(B) + 1$;
- $\deg(\neg A) = \deg(A) + 1$.

We extend the function \deg to signed formulas as follows:

- For a signed formula $\mathcal{S}A$ ($\mathcal{S} \in \{\mathbf{T}, \mathbf{F}, \mathbf{F}_c\}$), $\deg(\mathcal{S}A) = \deg(A)$.
- For a finite set S of signed formulas, $\deg(S) = \sum_{H \in S} \deg(H)$.

It is easy to check that, if S' is a set in the consequence of a rule of $\mathcal{T}_{\mathbf{Int}}$ applied to a finite set of signed formulas S , then $\deg(S') < \deg(S)$.

To describe our proof search strategy, we introduce the notion of *rule related* to S, H , where S is a set of signed formulas and H a signed formula.

- If H has not the form $\mathbf{T}(A \rightarrow B)$, the rule related to S, H is the only rule of Table 1 having H as major premise and $S \setminus \{H\}$ as set of minor premises.
- If $H = \mathbf{T}(A \rightarrow B)$ and $\mathbf{T}A \in S$, the rule related to S, H is the rule *MP* of Table 2 having H as major premise and $S \setminus \{H\}$ as set of minor premises.
- If $H = \mathbf{T}(A \rightarrow B)$, $\mathbf{T}A \notin S$ and $S = S_c$, the rule related to S, H is the rule $\mathbf{T} \rightarrow$ *certain* of Table 2 having H as major premise and $S \setminus \{H\}$ as set of minor premises.
- If $H = \mathbf{T}(A \rightarrow B)$, $\mathbf{T}A \notin S$ and $S \neq S_c$, the rule related to S, H is one of the rules of Table 2 and 3 having H as major premise and $S \setminus \{H\}$ as set of minor premises (there exists only one applicable rule).

We notice that given S and H there exists at most a rule R of $\mathcal{T}_{\mathbf{Int}}$ related to S, H . If R is a splitting rule, we denote with $\mathcal{R}_{S,H}^1$ and $\mathcal{R}_{S,H}^2$ the leftmost set and the rightmost set in the consequence of R respectively; for non-splitting rules we denote with $\mathcal{R}_{S,H}^1$ the only set in the consequence of R . A set of signed formulas S is *consistent* if there exists no closed proof table for S .

Lemma 3. *Let S be a finite set of signed formulas. If S is consistent, then S is realizable.*

Proof. We prove the assertion by complete induction on $\deg(S)$. Let us assume the assertion holds for all S' such that $\deg(S') < \deg(S)$; we prove it for S . Let $\bar{S} \subseteq S$ be the set of signed formulas H of S satisfying one of the following conditions:

- (i). $H = SA$, where $S \in \{\mathbf{T}, \mathbf{F}, \mathbf{F}_c\}$, and $A = B \wedge C$ or $A = B \vee C$.
- (ii). $H = \mathbf{T}(\neg A)$ or $H = \mathbf{T}(A \wedge B \rightarrow C)$ or $H = \mathbf{T}(A \vee B \rightarrow C)$.
- (iii). $H = \mathbf{T}(A \rightarrow B)$ and $\mathbf{T}A \in S$.
- (iv). $H = \mathbf{T}(A \rightarrow B)$ and $S = S_c$.
- (v). $H = \mathbf{T}(\neg A \rightarrow C)$ and $(S \setminus \{H\}) \cup \{\mathbf{TC}\}$ is consistent.
- (vi). $H = \mathbf{T}((A \rightarrow p) \rightarrow C)$, p is a propositional variable, and $(S \setminus \{H\}) \cup \{\mathbf{TC}\}$ is consistent.
- (vii). $H = \mathbf{T}((A \rightarrow \neg B) \rightarrow C)$ and $(S \setminus \{H\}) \cup \{\mathbf{TC}\}$ is consistent.
- (viii). $H = \mathbf{T}((A \rightarrow (X \wedge Y)) \rightarrow C)$ and $(S \setminus \{H\}) \cup \{\mathbf{TC}\}$ is consistent.
- (ix). $H = \mathbf{T}((A \rightarrow (X \vee Y)) \rightarrow C)$ and $(S \setminus \{H\}) \cup \{\mathbf{TC}\}$ is consistent.
- (x). $H = \mathbf{T}((A \rightarrow (X \rightarrow Y)) \rightarrow C)$ and $(S \setminus \{H\}) \cup \{\mathbf{TC}\}$ is consistent.

Let us assume that $\bar{S} \neq \emptyset$ and let H be any formula of \bar{S} . Since S is consistent, there exists $k \in \{1, 2\}$ such that the set $S' = \mathcal{R}_{\bar{S}, H}^k$ is consistent; moreover, if H is one of the signed formulas in cases (v), (vi), (vii), (viii), (ix) and (x), we take $S' = \mathcal{R}_{\bar{S}, H}^2$ (i.e. $S' = (S \setminus \{H\}) \cup \{\mathbf{TC}\}$). Since $\deg(S') < \deg(S)$, by induction hypothesis there exists a Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ such that $\underline{K}, \rho \triangleright S'$. It is easy to check that $\underline{K}, \rho \triangleright S$, and this proves the assertion.

Now, let us assume that $\bar{S} = \emptyset$. Let $S_1 \subseteq S$ be the set of formulas $H \in S$ satisfying one of the following conditions:

- 1. $H = \mathbf{T}p$ or $H = \mathbf{F}_c p$ or $H = \mathbf{F}p$, with p a propositional variable.
- 2. $H = \mathbf{T}(p \rightarrow B)$, with p a propositional variable and $\mathbf{T}p \notin S$.

Let $S_2 \subseteq S$ be the set of formulas $H \in S$ satisfying one of the following conditions:

- 3. $H = \mathbf{F}(A \rightarrow B)$, or $H = \mathbf{F}_c(A \rightarrow B)$, or $H = \mathbf{F}(\neg A)$ or $H = \mathbf{F}_c(\neg A)$.
- 4. $H = \mathbf{T}(\neg A \rightarrow C)$ and $\mathcal{R}_{\bar{S}, H}^1$ is consistent.
- 5. $H = \mathbf{T}((A \rightarrow B) \rightarrow C)$ and $\mathcal{R}_{\bar{S}, H}^1$ is consistent.

Since S is consistent and \bar{S} is empty, $S_1 \cup S_2 = S$. If $S_2 = \emptyset$, then $S = S_1$ can be realized in the Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ where $P = \{\rho\}$ and, for every propositional variable p , $\rho \Vdash p$ iff $\mathbf{T}p \in S$.

Now, let us suppose that $S_2 = \{H_1, \dots, H_n\}$ and let $j \in \{1, \dots, n\}$. By definition of S_2 , the set $T_j = \mathcal{R}_{\bar{S}, H_j}^1$ is consistent. Since $\deg(T_j) < \deg(S)$, by induction hypothesis there exists a Kripke model $\underline{K}_j = \langle P_j, \leq_j, \rho_j, \Vdash_j \rangle$ such that $\underline{K}_j, \rho_j \triangleright T_j$ (we assume that the P_j 's are pairwise disjoint). We build the

Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ where ρ is a new element ($\rho \notin \bigcup_{1 \leq j \leq n} P_j$) and the immediate successors of ρ are the elements ρ_1, \dots, ρ_n , formally:

$$P = \left(\bigcup_{1 \leq j \leq n} P_j \right) \cup \{\rho\} \quad \leq = \left(\bigcup_{1 \leq j \leq n} \leq_j \right) \cup \{(\rho, \alpha) \mid \alpha \in P\}$$

Finally, for every $\alpha \in P$ and every propositional variable p , $\alpha \Vdash p$ iff one of the following conditions holds:

- $\alpha \in P_j$ and $\alpha \Vdash_j p$.
- $\alpha = \rho$ and $\mathbf{T}p \in S$.

We point out that, if $\alpha \in P_j$, then $\alpha \Vdash H$ iff $\alpha \Vdash_j H$; in particular, $\underline{K}, \rho_j \triangleright T_j$ for every $1 \leq j \leq n$. We prove that $\underline{K}, \rho \triangleright H$ for every $H \in S$.

If $H = \mathbf{T}p$, by definition $\rho \Vdash p$. If $H = \mathbf{F}p$, then, by consistency, $\mathbf{T}p \notin S$, hence $\rho \not\Vdash p$. If $H = \mathbf{F}_c p$, then $\mathbf{F}_c p \in T_j$ for every $1 \leq j \leq n$; it follows that $\rho_j \Vdash \neg p$ for every $1 \leq j \leq n$, hence $\rho \Vdash \neg p$.

Let $H = \mathbf{T}(p \rightarrow B)$ and let $\alpha \in P$ such that $\alpha \Vdash p$. Since $\mathbf{T}p \notin S$ (by Point (2)), by definition $\rho \not\Vdash p$. Let k be such that $\alpha \in P_k$. Since $\rho_k \Vdash p \rightarrow B$ and $\rho_k \leq \alpha$, it follows that $\alpha \Vdash B$.

Let $H = \mathbf{F}(A \rightarrow B)$, then there exists k such that $T_k = (S_c \setminus \{H\}) \cup \{\mathbf{T}A, \mathbf{F}B\}$ and $\underline{K}, \rho_k \triangleright T_k$. It follows that $\rho_k \Vdash A$ and $\rho_k \not\Vdash B$, hence $\rho \not\Vdash A \rightarrow B$.

Let $H = \mathbf{T}((A \rightarrow X \wedge Y) \rightarrow C)$, then there exists k such that

$$T_k = (S_c \setminus \{H\}) \cup \{\mathbf{T}A, \mathbf{F}p, \mathbf{T}(X \rightarrow (Y \rightarrow p)), \mathbf{T}(p \rightarrow C)\}$$

and $\underline{K}, \rho_k \triangleright T_k$. Let $\alpha \in P$ such that $\alpha \Vdash A \rightarrow X \wedge Y$. Since $\rho_k \Vdash A$ and $\rho_k \not\Vdash X \wedge Y$ (otherwise, $\rho_k \Vdash p$ would follow), $\alpha \neq \rho$. Let j be such that $\alpha \in P_j$. If $j = k$, we have $\rho_k \leq \alpha$, which implies $\alpha \Vdash C$. If $j \neq k$, since $H \in T_j$, $\underline{K}, \rho_j \triangleright T_j$ and $\rho_j \leq \alpha$, and we get $\alpha \Vdash C$. The remaining cases are similar. \square

By the above lemma, it follows that, if $\{\mathbf{F}A\}$ is not realizable then there exists a closed proof table for $\{\mathbf{F}A\}$. Since $A \in \mathbf{Int}$ implies that $\{\mathbf{F}A\}$ is not realizable, we get:

Theorem 2 (Completeness). *If $A \in \mathbf{Int}$, then there exists a closed proof table for $\{\mathbf{F}A\}$.*

The proof of Lemma 3 implicitly defines a decision procedure for Intuitionistic Logic; indeed, starting from a finite set S of signed formulas, either a closed proof table or a counter-model for S is built. In the following we give some insights on the strategy we apply in the decision procedure.

As usual, applying invertible rules before non-invertible ones reduces the search-space. In our decision procedure, cases (i)-(iv) in the definition of \bar{S} correspond to the application of invertible rules. Accordingly, if there exists $H \in S$ satisfying one of cases (i)-(iv), we firstly apply *the* rule related to S, H ; if the search for a closed proof table fails, we conclude that S is not provable (there is no need to backtrack and try the application of another rule to S).

Otherwise, let us assume that no formula $H \in S$ satisfies cases (i)-(iv) and that there exists $H = \mathbf{T}(A \rightarrow B)$ in S . In this hypothesis, we firstly try to build a proof table for the “invertible” consequence $\mathcal{R}_{S,H}^2 = (S \setminus \{H\}) \cup \{\mathbf{TC}\}$; if such a proof does not exist, we get a counter-model for S and S is not provable. On the other hand, if we find a proof for $\mathcal{R}_{S,H}^2$ but $\mathcal{R}_{S,H}^1$ is not provable, one of the cases (4)-(5) holds: nothing can be concluded and we have to try the application of another rule to S (this corresponds to the fact that, to build the counter-model for S , we have to find out, for *all* $H' \in S$, a counter-model for $\mathcal{R}_{S,H'}^1$). In all the other cases, either non-invertible rules are applicable to S (case (3)), or no rules at all (cases (1)-(2)).

Finally, we remark that a proof table for a set S not containing \mathbf{F} -signed formulas is essentially a classical derivation. Indeed, in the proof we can always apply one of the rules of Table 1, *MP* and $\mathbf{T} \rightarrow$ *certain*, which are classical rules and do not generate \mathbf{F} -signed formulas.

We conclude with some remarks about the complexity of our calculus. First of all we notice that, for a finite set S of signed formulas, $\deg(S) \leq 3|S|$, where $|S|$ is the number of symbols occurring in S . Since in a proof the complexity of a set decreases after the application of a rule, the depth of every proof tree for S is linearly bounded by $|S|$. More precisely, one can prove:

Proposition 1. *Let S be a finite set of signed formulas. Then, the depth of every proof table for S is at most $3|S|$.*

By inspecting the rules of $\mathcal{T}_{\mathbf{Int}}$, it is easy to check that the number of symbols in every consequence of a rule of $\mathcal{T}_{\mathbf{Int}}$ increases of a constant value at most. As a consequence, a depth-first decision procedure for S requires at most $O(n \log n)$ bits to store the data structures.

References

1. A. Avellone, G. Fiorino, and U. Moscato. A new $O(n \log n)$ -space decision procedure for propositional intuitionistic logic. In Andrei Voronkov Matthias Baaz, Johann Makowsky, editor, *LPAR 2002: Short Contributions, CSL 2003: Extended Posters*, volume VIII of *Kurt Gödel Society, Collegium Logicum*, pages 17–33, 2004.
2. A. Chagrov and M. Zakharyashev. *Modal Logic*. Oxford University Press, 1997.
3. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.
4. G. Fiorino. *Decision procedures for propositional intermediate logics*. PhD thesis, Dipartimento di Scienze dell’Informazione, Università degli Studi di Milano, Italy, 2001.
5. M.C. Fitting. *Intuitionistic Logic, Model Theory and Forcing*. North-Holland, 1969.
6. D. Galmiche and D. Larchey-Wendling. Structural sharing and efficient proof-search in propositional intuitionistic logic. In *ASIAN’99*, volume 1742 of *LNCS*, pages 101–112, 1999.
7. G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Works of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

8. J. Hudelmaier. An $O(n \log n)$ -SPACE decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, 3(1):63–75, 1993.
9. P. Miglioli, U. Moscato, and M. Ornaghi. Avoiding duplications in tableau systems for intuitionistic logic and Kuroda logic. *Logic Journal of the IGPL*, 5(1):145–167, 1997.
10. R. Statman. Intuitionistic logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.
11. N. N. Vorob'ev. A new algorithm of derivability in a constructive calculus of statements. In *Sixteen papers on logic and algebra*, volume 94 of *American Mathematical Society Translations, Series 2*, pages 37–71. American Mathematical Society, Providence, R.I., 1970.

Actions over a constructive semantics for description logics

Loris Bozzato, Mauro Ferrari, and Paola Villa

Dipartimento di Informatica e Comunicazione
Università degli Studi dell'Insubria
Via Mazzini 5, 21100, Varese, Italy**

Abstract. Following the approaches and motivations given in recent works about action languages over description logics, we propose an action formalism based on a constructive semantics for \mathcal{ALC} . We address the problems to determine executability of an action, to build the state obtained by an action application and to check its consistency: we present an algorithm to solve the latter two problems.

1 Introduction

In [3] we have introduced the *information terms semantics* for the basic description logic \mathcal{ALC} . This semantics is related to the BHK interpretation, the well known constructive explanation of logical connectives (see, e.g., [16]). Specifically, in our semantics the truth of a formula in a given interpretation is explained by a mathematical object that we call *information term*. For the time being, the latter can be visualized as a sort of *proof term* inhabiting a type/formula. One of the main features of information terms semantics is that it supports a natural notion of *state* which has already been used to define the semantics of CooML (Constructive Object Oriented Modeling Language) [8, 14], a modeling language based on a constructive first-order logic.

In recent papers [2, 6, 7] different approaches to the definition of action languages over description logics have been addressed and different solutions have been proposed. In this paper we describe an approach based on the notion of state given by information terms. In our context an action is an expression $\mathcal{P} \Rightarrow \mathcal{Q}$ where \mathcal{P} and \mathcal{Q} are sets of \mathcal{ALC} formulas. An action can be informally understood as follows: if in a given state the formulas in \mathcal{P} (*preconditions*) are true, then the action can be applied and in the resulting state the formulas in \mathcal{Q} (*postconditions*) will be true. In this paper we address the problems to determine executability of an action, to build the state obtained by an action application and to check its consistency. As discussed in [8, 9], these problems are strictly connected with *snapshot generation* in CooML and with model generation in SAT [11] and in Answer Set Programming [13].

For expository reasons, in this paper we restrict our attention to the simple case where the formulas occurring in $\mathcal{P} \Rightarrow \mathcal{Q}$ are literals over the terminological

** A previous version of this paper was published in the proceedings of DL2008 [4].

alphabet. However, our approach can be extended to more complex actions. The paper is organized as follows: in Section 2, we present a formalization for \mathcal{ALC} syntax and classical semantics and in Section 3 we introduce the *information terms* semantics. In Sections 4 and 5 we present the definition of action and an algorithm to build the state corresponding to an action application. We conclude outlining some future works.

2 \mathcal{ALC} language and semantics

We begin introducing the language \mathcal{L} for \mathcal{ALC} [1, 15], based on the following denumerable sets: the set NR of *role names*, the set NC of *concept names* and the set NI of *individual names*. A *concept* H is an expression of the kind:

$$H ::= C \mid \neg H \mid H \sqcap H \mid H \sqcup H \mid \exists R.H \mid \forall R.H$$

where $C \in \text{NC}$ and $R \in \text{NR}$. Let Var be a denumerable set of *individual variables*; the *formulas* K of \mathcal{L} are defined according to the following grammar:

$$K ::= \perp \mid (s, t) : R \mid (s, t) : \neg R \mid t : H \mid \forall H$$

where $s, t \in \text{NI} \cup \text{Var}$, $R \in \text{NR}$ and H is a concept. An *atomic formula* of \mathcal{L} is a formula of the kind \perp , $(s, t) : R$ or $t : C$ where $R \in \text{NR}$ and $C \in \text{NC}$; a *negated formula* is a formula of the kind $(s, t) : \neg R$ or $t : \neg H$. A formula is *closed* if it does not contain variables. We write $\neg((s, t) : R)$, $\neg((s, t) : \neg R)$, $\neg(t : H)$ as abbreviations for $(s, t) : \neg R$, $(s, t) : R$, $t : \neg H$ respectively; $A \sqsubseteq B$ stands for $\forall(\neg A \sqcup B)$.

Let \mathcal{N} be a finite subset of NI . By $\mathcal{L}_{\mathcal{N}}$ we denote the set of formulas K of \mathcal{L} such that all the individual names occurring in K belong to \mathcal{N} .

A *substitution (over \mathcal{N})* is a function $\sigma : \text{Var} \rightarrow \mathcal{N}$; we extend σ to $\mathcal{L}_{\mathcal{N}}$ so that for every $c \in \mathcal{N}$, $\sigma(c) = c$. Given a set of formulas Γ of $\mathcal{L}_{\mathcal{N}}$ and a substitution σ over \mathcal{N} , $\sigma\Gamma$ denotes the set of closed formulas obtained replacing every variable x occurring in Γ with $\sigma(x)$.

A *model (interpretation)* \mathcal{M} for \mathcal{L} is a pair $(\mathcal{D}^{\mathcal{M}}, \cdot^{\mathcal{M}})$, where $\mathcal{D}^{\mathcal{M}}$ is a non-empty set (the *domain* of \mathcal{M}) and $\cdot^{\mathcal{M}}$ is a *valuation* map such that: for every $c \in \text{NI}$, $c^{\mathcal{M}} \in \mathcal{D}^{\mathcal{M}}$; for every $C \in \text{NC}$, $C^{\mathcal{M}} \subseteq \mathcal{D}^{\mathcal{M}}$; for every $R \in \text{NR}$, $R^{\mathcal{M}} \subseteq \mathcal{D}^{\mathcal{M}} \times \mathcal{D}^{\mathcal{M}}$. A non atomic concept H is interpreted by a subset $H^{\mathcal{M}}$ of $\mathcal{D}^{\mathcal{M}}$ as usual:

- $(\neg A)^{\mathcal{M}} = \mathcal{D}^{\mathcal{M}} \setminus A^{\mathcal{M}}$
- $(A \sqcap B)^{\mathcal{M}} = A^{\mathcal{M}} \cap B^{\mathcal{M}}$
- $(A \sqcup B)^{\mathcal{M}} = A^{\mathcal{M}} \cup B^{\mathcal{M}}$
- $(\exists R.A)^{\mathcal{M}} = \{d \in \mathcal{D}^{\mathcal{M}} \mid \exists d' \in \mathcal{D}^{\mathcal{M}} \text{ s.t. } (d, d') \in R^{\mathcal{M}} \text{ and } d' \in A^{\mathcal{M}}\}$
- $(\forall R.A)^{\mathcal{M}} = \{d \in \mathcal{D}^{\mathcal{M}} \mid \forall d' \in \mathcal{D}^{\mathcal{M}}, (d, d') \in R^{\mathcal{M}} \text{ implies } d' \in A^{\mathcal{M}}\}$

We say that a model \mathcal{M} of $\mathcal{L}_{\mathcal{N}}$ is *reachable* if, for every $d \in \mathcal{D}^{\mathcal{M}}$, there exists $c \in \text{NI}$ such that $c^{\mathcal{M}} = d$.

An *assignment* on a model \mathcal{M} is a map $\theta : \mathbf{Var} \rightarrow \mathcal{D}^{\mathcal{M}}$. If $t \in \mathbf{NI} \cup \mathbf{Var}$, $t^{\mathcal{M},\theta}$ is the element of $\mathcal{D}^{\mathcal{M}}$ denoting t in \mathcal{M} w.r.t. θ , namely: $t^{\mathcal{M},\theta} = \theta(t)$ if $t \in \mathbf{Var}$; $t^{\mathcal{M},\theta} = t^{\mathcal{M}}$ if $t \in \mathbf{NI}$. A formula K is *valid* in \mathcal{M} w.r.t. θ , and we write $\mathcal{M}, \theta \models K$, if $K \neq \perp$ and one of the following conditions holds:

- $\mathcal{M}, \theta \models (s, t) : R$ iff $(s^{\mathcal{M},\theta}, t^{\mathcal{M},\theta}) \in R^{\mathcal{M}}$
- $\mathcal{M}, \theta \models (s, t) : \neg R$ iff $(s^{\mathcal{M},\theta}, t^{\mathcal{M},\theta}) \notin R^{\mathcal{M}}$
- $\mathcal{M}, \theta \models t : H$ iff $t^{\mathcal{M},\theta} \in H^{\mathcal{M}}$
- $\mathcal{M}, \theta \models \forall H$ iff $H^{\mathcal{M}} = \mathcal{D}^{\mathcal{M}}$

We write $\mathcal{M} \models K$ iff $\mathcal{M}, \theta \models K$ for every assignment θ . Note that $\mathcal{M} \models \forall H$ iff $\mathcal{M} \models x : H$, with x any variable. If Γ is a set of formulas, $\mathcal{M} \models \Gamma$ means that $\mathcal{M} \models K$ for every $K \in \Gamma$. We say that K is a *logical consequence* of Γ , and we write $\Gamma \models K$, iff, for every \mathcal{M} and every θ , $\mathcal{M}, \theta \models \Gamma$ implies $\mathcal{M}, \theta \models K$.

3 Information terms semantics

We introduce *information terms* that will be the base structure of our constructive semantics. Given a finite subset \mathcal{N} of \mathbf{NI} and a closed formula K of $\mathcal{L}_{\mathcal{N}}$, we define the set of information terms $\text{IT}_{\mathcal{N}}(K)$ by induction on K as follows.

$$\begin{aligned} \text{IT}_{\mathcal{N}}(K) &= \{\mathbf{tt}\}, \text{ if } K \text{ is an atomic or negated formula} \\ \text{IT}_{\mathcal{N}}(c : A_1 \sqcap A_2) &= \{(\alpha, \beta) \mid \alpha \in \text{IT}_{\mathcal{N}}(c : A_1) \text{ and } \beta \in \text{IT}_{\mathcal{N}}(c : A_2)\} \\ \text{IT}_{\mathcal{N}}(c : A_1 \sqcup A_2) &= \{(k, \alpha) \mid k \in \{1, 2\} \text{ and } \alpha \in \text{IT}_{\mathcal{N}}(c : A_k)\} \\ \text{IT}_{\mathcal{N}}(c : \exists R.A) &= \{(d, \alpha) \mid d \in \mathcal{N} \text{ and } \alpha \in \text{IT}_{\mathcal{N}}(d : A)\} \\ \text{IT}_{\mathcal{N}}(c : \forall R.A) &= \text{IT}_{\mathcal{N}}(\forall A) = \{\phi : \mathcal{N} \rightarrow \bigcup_{d \in \mathcal{N}} \text{IT}_{\mathcal{N}}(d : A) \mid \phi(d) \in \text{IT}_{\mathcal{N}}(d : A)\} \end{aligned}$$

As we will discuss later, information terms for a formula K provide possible justifications for the validity of K in a classical model.

Formally, let \mathcal{M} be a model for \mathcal{L} , K a closed formula of $\mathcal{L}_{\mathcal{N}}$ and $\eta \in \text{IT}_{\mathcal{N}}(K)$. We define the *realizability relation* $\mathcal{M} \triangleright \langle \eta \rangle K$ as follows:

$$\begin{aligned} \mathcal{M} \triangleright \langle \mathbf{tt} \rangle K &\text{ iff } \mathcal{M} \models K, \text{ where } K \text{ is an atomic or negated formula} \\ \mathcal{M} \triangleright \langle (\alpha, \beta) \rangle c : A_1 \sqcap A_2 &\text{ iff } \mathcal{M} \triangleright \langle \alpha \rangle c : A_1 \text{ and } \mathcal{M} \triangleright \langle \beta \rangle c : A_2 \\ \mathcal{M} \triangleright \langle (k, \alpha) \rangle c : A_1 \sqcup A_2 &\text{ iff } \mathcal{M} \triangleright \langle \alpha \rangle c : A_k \\ \mathcal{M} \triangleright \langle (d, \alpha) \rangle c : \exists R.A &\text{ iff } \mathcal{M} \models (c, d) : R \text{ and } \mathcal{M} \triangleright \langle \alpha \rangle d : A \\ \mathcal{M} \triangleright \langle \phi \rangle c : \forall R.A &\text{ iff } \mathcal{M} \models c : \forall R.A \text{ and, for every } d \in \mathcal{N}, \\ &\quad \mathcal{M} \models (c, d) : R \text{ implies } \mathcal{M} \triangleright \langle \phi(d) \rangle d : A \\ \mathcal{M} \triangleright \langle \phi \rangle \forall A &\text{ iff } \mathcal{M} \models \forall A \text{ and, for every } d \in \mathcal{N}, \mathcal{M} \triangleright \langle \phi(d) \rangle d : A \end{aligned}$$

If $\Gamma = \{K_1, \dots, K_n\}$ is a finite set of closed formulas of $\mathcal{L}_{\mathcal{N}}$, $\text{IT}_{\mathcal{N}}(\Gamma)$ denotes the set of n -tuples $\bar{\eta} = (\eta_1, \dots, \eta_n)$ such that, for every $1 \leq j \leq n$, $\eta_j \in \text{IT}_{\mathcal{N}}(K_j)$; $\mathcal{M} \triangleright \langle \bar{\eta} \rangle \Gamma$ iff, for every $1 \leq j \leq n$, $\mathcal{M} \triangleright \langle \eta_j \rangle K_j$.

We remark that, according to the above definition, an information term provides a justification for the validity of a formula in a classical model. For instance, if $\mathcal{M} \triangleright \langle (d, \alpha) \rangle c : \exists R.C$, the information term (d, α) explicitly provides the witness d such that $(c^{\mathcal{M}}, d^{\mathcal{M}}) \in R^{\mathcal{M}}$ and $d^{\mathcal{M}} \in C^{\mathcal{M}}$; moreover, the information term α recursively explains why $d^{\mathcal{M}} \in C^{\mathcal{M}}$. We remark that the validity of an atomic formula in a classical model does not need further justifications; indeed, the information term for an atomic formula is simply the constant \mathbf{tt} . As for the negated formulas, they are treated in a classical way and are not constructively explained by an information term (the only information term for a negated formula is \mathbf{tt}). In this paper we have chosen to treat negation in this way for expository reasons, indeed, we can extend information term semantics to treat various kinds of constructive negation as those discussed in [10].

The relation between classical and constructive semantics is established by the following propositions.

Proposition 1. *Let \mathcal{N} be a finite subset of \mathbf{NI} , K a closed formula of $\mathcal{L}_{\mathcal{N}}$, and $\eta \in \text{IT}_{\mathcal{N}}(K)$. For every model \mathcal{M} , if $\mathcal{M} \triangleright \langle \eta \rangle K$, then $\mathcal{M} \models K$. \square*

The above proposition states that the constructive semantics is compatible with the classical one: as a consequence of this relationship, we can conveniently read \mathcal{ALC} formulas in the classical way. The converse in general does not hold and stronger conditions are required:

Proposition 2. *Let K be a closed formula of \mathcal{L} and let \mathcal{M} be a finite model for \mathcal{L} . If $\mathcal{M} \models K$, there exists a finite subset \mathcal{N} of \mathbf{NI} and $\eta \in \text{IT}_{\mathcal{N}}(K)$ such that $\mathcal{M} \triangleright \langle \eta \rangle K$. \square*

In the following we will indicate with the term *theory* a set \mathbf{T} of closed formulas of $\mathcal{L}_{\mathcal{N}}$ consisting of a TBox and an ABox. The TBox is a set of formulas of the kind $\forall A$, representing the constraints of our system. The ABox is a set of concept and role assertions that represent our knowledge about the current state of the system. A *state* of the system is any $\bar{\gamma} \in \text{IT}_{\mathcal{N}}(\mathbf{T})$.

Example 1 (The alert system). In this example we model a simple home alert system. The system has two kinds of sensors, namely to detect fire and flood events: whenever a sensor activates and signals the occurrence of one of these events, a corresponding alert goes off. When a sensor stops signaling an event, the alarm must stop.

The theory that models our system consists of the TBox \mathbf{TAS} , representing the constraints of the model:

$$\begin{aligned} (Ax_1) : & \forall (\neg \mathbf{CurrentAlert} \sqcup (\exists \mathbf{hasReason.CurrentSignal} \sqcap \mathbf{Alert})) \\ (Ax_2) : & \forall (\neg \mathbf{CurrentSignal} \sqcup (\mathbf{Active} \sqcap (\mathbf{Fire} \sqcup \mathbf{Flood}))) \end{aligned}$$

that can be restated as:

$$\begin{aligned} (Ax_1) : & \mathbf{CurrentAlert} \sqsubseteq \exists \mathbf{hasReason.CurrentSignal} \sqcap \mathbf{Alert} \\ (Ax_2) : & \mathbf{CurrentSignal} \sqsubseteq \mathbf{Active} \sqcap (\mathbf{Fire} \sqcup \mathbf{Flood}) \end{aligned}$$

and an ABox \mathbf{AAS}_0 :

alert1:Alert	fire_s1:Fire	flood_s1:Flood
alert2:Alert	fire_s2:Fire	flood_s2:Flood

asserting that our system has two sensors for every kind. Moreover, in the initial state none of the signals is active.

Let $\mathbf{As}_0 = \text{TAS} \cup \text{AAS}_0$ and let W be the set of the individual names occurring in AAS_0 . As an example, an element of $\text{IT}_W(Ax_1)$ is a function ϕ mapping each $c \in W$ to an element

$$\delta \in \text{IT}_W(c : \neg \text{CurrentAlert} \sqcup (\exists \text{hasReason}.\text{CurrentSignal} \sqcap \text{Alert}))$$

where, δ is either $(1, \mathbf{tt})$ (meaning that c is not a current alert) or $(2, ((d, \mathbf{tt}), \mathbf{tt}))$ (i.e., c is a current alert and his reason is the current signal d).

Now, we have to select an initial state of the system consistent with our knowledge contained in AAS_0 . To this aim let $\gamma_1 \in \text{IT}_W(Ax_1)$ and $\gamma_2 \in \text{IT}_W(Ax_2)$ the functions mapping every $c \in W$ in $(1, \mathbf{tt})$. The initial state of our system is the information term $\bar{\gamma}_0 \in \text{IT}_W(\mathbf{As}_0)$ associating γ_1 to Ax_1 , γ_2 to Ax_2 and \mathbf{tt} to every formula of the ABox AAS_0 . It is easy to define a model \mathcal{M} of \mathbf{As}_0 such that $\mathcal{M} \triangleright \langle \bar{\gamma}_0 \rangle \mathbf{As}_0$. We remark that $\bar{\gamma}_0$ is the only state that can justify our theory only assuming the information contained in AAS_0 . We also notice that $\bar{\gamma}_0$ assumes a sort of closed world assumption about the current state, in the sense that what is not true in the current state is assumed as false. \diamond

According to the above definitions an information term is a structured data whose correct reading is provided by the related formula. According to this interpretation we call *piece of information* over $\mathcal{L}_{\mathcal{N}}$ a pair $\langle \alpha \rangle F$, where F is a closed formula of $\mathcal{L}_{\mathcal{N}}$ and $\alpha \in \text{IT}_{\mathcal{N}}(F)$. Given a theory \mathbf{T} over $\mathcal{L}_{\mathcal{N}}$, we introduce two notions of consistency:

- A state (information term) $\bar{\gamma} \in \text{IT}_{\mathcal{N}}(\mathbf{T})$ is *consistent* if there exists a model \mathcal{M} such that $\mathcal{M} \triangleright \langle \bar{\gamma} \rangle \mathbf{T}$.
- \mathbf{T} is *state consistent* if there exists $\bar{\gamma} \in \text{IT}_{\mathcal{N}}(\mathbf{T})$ such that $\bar{\gamma}$ is consistent.

Now, we will see how, introducing the notion of information content, we can reduce the problem to check consistency of a state to the problem to check classical consistency of a set of atomic and negated formulas.

Given a finite subset \mathcal{N} of NI , let $\mathcal{R}_{\mathcal{N}} = \{(s, t) : R \mid R \in \text{NR and } s, t \in \mathcal{N}\}$. This set intuitively represents the set of all the possible role assertions that we can express over \mathcal{N} . Given a finite subset \mathcal{R} of $\mathcal{R}_{\mathcal{N}}$, we define the *information content* (w.r.t. \mathcal{R}) of a piece of information as follows:

- $\text{IC}_{\mathcal{R}}(\langle \mathbf{tt} \rangle c : H) = \{c : H\}$, if $c : H$ is an atomic or negated formula
- $\text{IC}_{\mathcal{R}}(\langle (\alpha, \beta) \rangle c : A_1 \sqcap A_2) = \text{IC}_{\mathcal{R}}(\langle \alpha \rangle c : A_1) \cup \text{IC}_{\mathcal{R}}(\langle \beta \rangle c : A_2)$
- $\text{IC}_{\mathcal{R}}(\langle (k, \alpha) \rangle c : A_1 \sqcup A_2) = \text{IC}_{\mathcal{R}}(\langle \alpha \rangle c : A_k)$ ($k = 1$ or $k = 2$)
- $\text{IC}_{\mathcal{R}}(\langle (d, \alpha) \rangle c : \exists R.A) = \text{IC}_{\mathcal{R}}(\langle \alpha \rangle d : A) \cup \{(c, d) : R\}$
- $\text{IC}_{\mathcal{R}}(\langle \phi \rangle c : \forall R.A) = \bigcup_{(c,d):R \in \mathcal{R}} \text{IC}_{\mathcal{R}}(\langle \phi(d) \rangle d : A) \cup \{(c, d) : R \mid (c, d) : R \in \mathcal{R}\}$
- $\text{IC}_{\mathcal{R}}(\langle \phi \rangle \forall A) = \bigcup_{d \in \mathcal{N}} \text{IC}_{\mathcal{R}}(\langle \phi(d) \rangle d : A)$

If $\Gamma = \{K_1, \dots, K_n\}$ is a set of closed formulas of $\mathcal{L}_{\mathcal{N}}$ and $\bar{\gamma} \in \text{IT}_{\mathcal{N}}(\Gamma)$, $\text{IC}_{\mathcal{R}}(\langle \bar{\gamma} \rangle \Gamma) = \bigcup_{K_i \in \Gamma} (\langle \gamma_i \rangle K_i)$.

We remark that the information content of a piece of information is a set of atomic and negated formulas. The following relation holds between a piece of information and its information content:

Theorem 1. *Let \mathcal{N} be a finite subset of NI , let K be a closed formula of $\mathcal{L}_{\mathcal{N}}$, let \mathcal{R} be a finite subset of $\mathcal{R}_{\mathcal{N}}$ and let \mathcal{M} be a reachable model of $\mathcal{L}_{\mathcal{N}}$ such that $\mathcal{M} \models (c, d) : R$ iff $(c, d) : R \in \mathcal{R}$. Then $\mathcal{M} \triangleright \langle \alpha \rangle K$ iff $\mathcal{M} \models \text{IC}_{\mathcal{R}}(\langle \alpha \rangle K)$. \square*

The proof of this theorem easily follows by induction on the structure of the formula K . According with this result, $\text{IC}_{\mathcal{R}}(\langle \alpha \rangle K)$ intuitively represents the minimum¹ amount of information needed to get evidence for K according to the information term α , assuming the roles in \mathcal{R} . We remark that, by Theorem 1, checking consistency of a state $\bar{\gamma}$ of \mathbf{T} is equivalent to check consistency of $\text{IC}_{\mathcal{R}}(\langle \bar{\gamma} \rangle \mathbf{T})$ for the given \mathcal{R} .

Example 2 (Information content). Let us consider the theory \mathbf{AS}_0 and the information terms γ_1 and γ_2 defined in Example 1. Let $\mathcal{R} = \emptyset$, corresponding to the initial state of our system where no role is specified. Then the information content of the initial state of our system is:

$$\begin{aligned} \text{IC}_{\mathcal{R}}(\langle \gamma_1 \rangle Ax_1) &= \{a : \neg \text{CurrentAlert} \mid a \in W\} \\ \text{IC}_{\mathcal{R}}(\langle \gamma_2 \rangle Ax_2) &= \{a : \neg \text{CurrentSignal} \mid a \in W\} \end{aligned}$$

Moreover, since every formula in the \mathbf{AAS}_0 is atomic, its information content is \mathbf{AAS}_0 itself. So we have that:

$$\begin{aligned} \text{IC}_{\mathcal{R}}(\langle \bar{\gamma}_0 \rangle \mathbf{AS}_0) &= \{a : \neg \text{CurrentAlert} \mid a \in W\} \cup \\ &\quad \{a : \neg \text{CurrentSignal} \mid a \in W\} \cup \mathbf{AAS}_0 \end{aligned}$$

Now, let \mathcal{M}_0 be the model having the set of individual names W as domain and mapping every individual name in itself, every concept so to satisfy \mathbf{AAS}_0 and every role in the empty set. \mathcal{M}_0 satisfies the hypothesis of Theorem 1 and hence the state $\bar{\gamma}_0$ is consistent. \diamond

4 Actions

We call *literal* a formula either of the kind $t : C$, $t : \neg C$, $(s, t) : R$ or $(s, t) : \neg R$ where $C \in \text{NC}$ and $R \in \text{NR}$. Given a set of literals L , we denote with $\bar{L} = \{\neg K \mid K \in L\} \cup \{K \mid \neg K \in L\}$.

An *action* over $\mathcal{L}_{\mathcal{N}}$ is an expression of the kind $\mathcal{P} \Rightarrow \mathcal{Q}$ where \mathcal{P} and \mathcal{Q} are sets of literals over $\mathcal{L}_{\mathcal{N}}$ and every individual variable occurring in \mathcal{Q} also occurs in \mathcal{P} . Informally an action can be understood as follows: if in a given state the formulas in \mathcal{P} (*preconditions*) are true, then the action can be applied and in the

¹ Minimality of $\text{IC}_{\mathcal{R}}(\langle \alpha \rangle K)$ can be formalised in model-theoretic terms (see [8]).

resulting state the formulas in \mathcal{Q} (*postconditions*) will be true. Given an action $\alpha \equiv \mathcal{P} \Rightarrow \mathcal{Q}$ we denote with $\text{Pre}(\alpha)$ the preconditions of α and with $\text{Post}(\alpha)$ the postconditions of α .

Now, let \mathbf{T} be a theory with ABox \mathcal{A} over $\mathcal{L}_{\mathcal{N}}$ and let $\bar{\gamma} \in \text{IT}_{\mathcal{N}}(\mathbf{T})$. The action $\alpha \equiv \mathcal{P} \Rightarrow \mathcal{Q}$ is *active in the state $\bar{\gamma}$ w.r.t. a substitution σ* if $\sigma\mathcal{P} \subseteq \text{IC}_{\mathcal{R}}(\langle \bar{\gamma} \rangle \mathbf{T})$. An active action can be applied and its application in state $\bar{\gamma}$ has two effects:

- We get a new ABox $\mathcal{A}' = (\mathcal{A} \setminus \overline{\sigma\mathcal{Q}}) \cup \sigma\mathcal{Q}$ (*ABox update*);
- We get the set $\text{Out}(\alpha) = ((\text{IC}_{\mathcal{R}}(\langle \bar{\gamma} \rangle \mathbf{T}) \cup \mathcal{R}) \setminus \overline{\sigma\mathcal{Q}}) \cup \sigma\mathcal{Q}$ (*action output*).

An action application changes both the ABox of the theory and the state of the system. Obviously, given a consistent state of our system, an action application could lead the system to an inconsistent state. According to Theorem 1, to guarantee that action application is consistent we must prove that the action output is consistent and that we can construct an information term for \mathbf{T} from the formulas in the action output. Before discussing the consistency issues let us define the actions governing the behaviour of our alert system.

Example 3 (Actions of the alert system). For every fire and flood sensor, we define the following actions (here defined for `fire_s1`) that model the beginning and the end of a signal from the sensor:

$$\text{SignalFireS1}() : \emptyset \Rightarrow \{\text{fire_s1} : \text{Active}\}$$

$$\text{UnSignalFireS1}() : \emptyset \Rightarrow \{\text{fire_s1} : \neg\text{Active}\}$$

We remark that the above actions have an empty set of preconditions, thus they are active in every state and they simply update the knowledge base. The system reacts to these signals with the following actions:

$$\begin{aligned} \text{StartFireAlert}(x) : \{x : \text{Fire}, x : \text{Active}\} \Rightarrow \\ \{\text{alert1} : \text{CurrentAlert}, x : \text{CurrentSignal}, \\ (\text{alert1}, x) : \text{hasReason}\} \end{aligned}$$

$$\begin{aligned} \text{StartFloodAlert}(x) : \{x : \text{Flood}, x : \text{Active}\} \Rightarrow \\ \{\text{alert2} : \text{CurrentAlert}, x : \text{CurrentSignal}, \\ (\text{alert2}, x) : \text{hasReason}\} \end{aligned}$$

$$\begin{aligned} \text{StopAlert}(x, y) : \{x : \neg\text{Active}, (y, x) : \text{hasReason}\} \Rightarrow \\ \{y : \neg\text{CurrentAlert}, x : \neg\text{CurrentSignal}, \\ (y, x) : \neg\text{hasReason}\} \end{aligned}$$

We write actions in a function-like form to emphasise the free variables and how they are instantiated. Note that the above three actions are not active in the initial state $\bar{\gamma}_0$.

Now, let us consider the situation where a fire is detected by `fire_s1`. This raises the action $\text{SignalFireS1}()$. Let $\mathbf{As}_1 = \text{TAS} \cup \text{AAS}_1$ where $\text{AAS}_1 = \text{AAS}_0 \cup \{\text{fire_s1} : \text{Active}\}$ and let $\bar{\gamma}_1$ be the information term for \mathbf{As}_1 associating γ_1 and γ_2 of Example 1 to Ax_1 and Ax_2 respectively, and associating `tt` to every

formula in \mathbf{AAS}_1 . It is easy to check that $\text{IC}_{\mathcal{R}}(\langle \bar{\gamma}_1 \rangle \mathbf{AS}_1)$ is exactly the result of applying this action to the state $\bar{\gamma}_0$.

In the state $\bar{\gamma}_1$ the action $\text{StartFireAlert}(\text{fire_s1})$ can be activated, and the result of its application is the output

$$\begin{aligned} \text{Out} = & \mathbf{AAS}_0 \cup \{a : \neg \text{CurrentAlert} \mid a \in (W \setminus \{\text{alert1}\})\} \cup \\ & \{a : \neg \text{CurrentSignal} \mid a \in (W \setminus \{\text{fire_s1}\})\} \cup \\ & \{\text{fire_s1}:\text{Active}, \text{alert1}:\text{CurrentAlert}, \text{fire_s1}:\text{CurrentSignal}, \\ & (\text{alert1}, \text{fire_s1}):\text{hasReason}\} \end{aligned}$$

Now, let $\mathbf{AAS}_2 = \mathbf{AAS}_1 \cup \text{Post}(\text{StartFireAlert}(\text{fire_s1}))$, and let $\mathbf{AS}_2 = \text{TAS} \cup \mathbf{AAS}_2$. Let \mathcal{M}_2 be a model of Out , it is easy to check that \mathcal{M}_2 is also a model of \mathbf{AS}_2 . \diamond

5 An algorithm to build up information terms

We remark that we have concluded the above example without giving the state corresponding to the action application. To build up the output state of an action we will use the algorithm $\text{GENIT}(X, F)$ of Figure 1. This algorithm takes as input a set X of closed atomic and negated formulas of $\mathcal{L}_{\mathcal{N}}$ and a closed formula F of $\mathcal{L}_{\mathcal{N}}$ and generates as output a (possibly empty) set of information terms in $\text{IT}_{\mathcal{N}}(F)$. GENIT invokes the function OPENIT of Figure 2 to compute the information terms for compound and open formulas. OPENIT takes as input the set X and a formula F and returns a (possibly empty) set of pairs (α, c) where $\alpha \in \text{IT}_{\mathcal{N}}(F)$ and $c \in \mathcal{N}$ (intuitively, if the pair (α, c) is built, α justifies the formula F with respect to the individual name c).

```

GENIT(X, F){
  if (F is either (c, d) : R or (c, d) : ¬R) then
    if (F ∈ X) then return {tt};
    else return ∅;
  else if (F = ∀A) then begin
    let Z = OPENIT(X, x : A);
    let Γ = {c | (α, c) ∈ Z};
    if (Γ ≠ N) then return ∅;
    else return {ϕ | ϕ(c) = α with (α, c) ∈ Z};
  end
  else return {α | (α, c) ∈ OPENIT(X, F)};
}

```

Fig. 1. The GENIT algorithm

It is easy to prove, by induction on the structure of the formula F , the following result:

```

OPENIT( $X, F$ ) {
  if ( $F = t : A$  with  $A \in \text{NC}$  or  $A = \neg H$ ) then
    if ( $t \in \mathcal{N}$ ) then
      if ( $t : A \in X$ ) then return  $\{(tt, t)\}$ ;
      else return  $\emptyset$ ;
    else return  $\{(tt, c) \mid c : A \in X\}$ ;
  if ( $F = t : A \sqcap B$ ) then
    return  $\{((\alpha, \beta), c) \mid (\alpha, c) \in \text{OPENIT}(X, t : A) \text{ and } (\beta, c) \in \text{OPENIT}(X, t : B)\}$ ;
  if ( $F = t : A \sqcup B$ ) then
    return  $\{(1, \alpha), c \mid (\alpha, c) \in \text{OPENIT}(X, t : A)\} \cup$ 
       $\{(2, \beta), c \mid (\beta, c) \in \text{OPENIT}(X, t : B)\}$ ;
  if ( $F = t : \exists R.A$ ) then
    if ( $t \in \mathcal{N}$ ) then let  $D = \{t\}$ ;
    else let  $D = \{c \mid (c, d) : R \in X\}$ ;
    return  $\{(d, \alpha), c \mid c \in D \text{ and } (c, d) : R \in X \text{ and } (\alpha, d) \in \text{OPENIT}(X, x : A)\}$ ;
  if ( $F = t : \forall R.A$ ) then begin
    if ( $t \in \mathcal{N}$ ) then begin
      let  $D = \{t\}$ ;
      let  $Z = \{d \mid (t, d) : R \in X \text{ and } (\alpha, d) \in \text{OPENIT}(X, x : A)\}$ ;
    else begin
      let  $D = \{c \mid (c, d) : R \in X\}$ ;
      let  $Z = \{d \mid (\alpha, d) \in \text{OPENIT}(X, x : A)\}$ ;
    end let  $\Phi = \emptyset$ ;
    for all ( $c \in D$ ) do begin
      let  $C = \{d \mid (c, d) : R \in X\}$ ;
      if ( $C \subseteq Z$ ) then
         $\Phi = \Phi \cup \left\{ (\phi, c) \mid \phi(d) = \begin{cases} \alpha & \text{if } d \in C \text{ and } (a, d) \in Z \\ \text{any } \eta^+ \text{ of } \text{IT}_{\mathcal{N}}(d : A) & \text{otherwise} \end{cases} \right\}$ 
      end;
    end return  $\Phi$ ;
  end
}

```

Fig. 2. The OPENIT function

Theorem 2. *Let X be a set of closed atomic and negated formulas of $\mathcal{L}_{\mathcal{N}}$, let $\mathcal{R} = \{(t, t') : R \mid (t, t') : R \in X\}$ and let F be a closed formula of $\mathcal{L}_{\mathcal{N}}$. Then, for every $\tau \in \text{GENIT}(X, F)$, $\text{IC}_{\mathcal{R}}(\langle \tau \rangle F) \subseteq X$. \square*

Example 4 (State generation). The execution of $\text{GENIT}(\text{Out}, Ax_1)$ provides the following information term $\gamma'_1 \in \text{IT}_{\mathcal{W}}(Ax_1)$, where we enclose between square brackets the pairs $(c, \gamma'_1(c))$ belonging to the function:

[(alert1, (2, ((fire_s1, tt), tt))), (alert2, (1, tt)), (fire_s1, (1, tt)),
 (fire_s2, (1, tt)), (flood_s1, (1, tt)), (flood_s2, (1, tt))]

while the execution of $\text{GENIT}(\text{Out}, Ax_2)$ provides the information term $\gamma'_2 \in \text{IT}_{\mathcal{W}}(Ax_2)$:

[(alert1,(1,tt)), (alert2,(1,tt)), (fire_s1,(2,(tt,(1,tt)))),
 (fire_s2,(1,tt)), (flood_s1,(1,tt)), (flood_s2,(1,tt))]

Consider AAS_2 as defined in the previous example. If $\bar{\gamma}_2$ is the information term associating γ'_1 to Ax_1 , γ'_2 to Ax_2 and \mathbf{tt} to every formula of AAS_2 and if \mathcal{M}_2 is a model of Out , then it follows that $\mathcal{M}_2 \triangleright \langle \bar{\gamma}_2 \rangle \mathbf{As}_2$. Hence the action application leads to a consistent state and thus \mathbf{As}_2 is state consistent.

If we consider $\mathcal{R} = \{(\mathbf{alert1}, \mathbf{fire_s1}) : \mathbf{hasReason}\}$, which is the set of role formulas asserted by AAS_2 , then the information content of \mathbf{As}_2 for the axioms in \mathbf{TAS} is defined as:

$$\begin{aligned} IC_{\mathcal{R}}(\langle \gamma'_1 \rangle Ax_1) &= \{\mathbf{alert1:Alert}, \mathbf{fire_s1:CurrentSignal}, \\ &\quad (\mathbf{alert1}, \mathbf{fire_s1}) : \mathbf{hasReason}\} \cup \\ &\quad \{a : \neg \mathbf{CurrentAlert} \mid a \in W \setminus \{\mathbf{alert1}\}\} \\ IC_{\mathcal{R}}(\langle \gamma'_2 \rangle Ax_2) &= \{\mathbf{fire_s1:Active}, \mathbf{fire_s1:Fire}\} \cup \\ &\quad \{a : \neg \mathbf{CurrentSignal} \mid a \in W \setminus \{\mathbf{fire_s1}\}\} \end{aligned}$$

So we have that $IC_{\mathcal{R}}(\langle \bar{\gamma}_2 \rangle \mathbf{As}_2) = IC_{\mathcal{R}}(\langle \gamma'_1 \rangle Ax_1) \cup IC_{\mathcal{R}}(\langle \gamma'_2 \rangle Ax_2) \cup AAS_2$. As in Example 2, if \mathcal{M}_2 satisfies the hypotheses, then Theorem 1 holds and $\mathcal{M}_2 \triangleright \langle \bar{\gamma}_2 \rangle \mathbf{As}_2$ iff $\mathcal{M}_2 \models IC_{\mathcal{R}}(\langle \bar{\gamma}_2 \rangle \mathbf{As}_2)$.

Note that, if $\mathbf{fire_s1}$ stops being active and $UnSignalFireS1()$ is executed, in the new state it does not hold that $\mathbf{fire_s1:Active}$ and so the action $StopAlert(\mathbf{fire_s1}, \mathbf{alert1})$ can be fired. If that is the case, it is easy to verify that the system returns in the initial state of our example.

Moreover, it is possible to show that, for example, since $GENIT(Out, Ax_1) = \{\gamma'_1\}$ and $IC_{\mathcal{R}}(\langle \gamma'_1 \rangle Ax_1) \subseteq Out$, then Theorem 2 holds. \diamond

We remark that in general $GENIT$ generates more than one state. In this case the action could lead the system to different states and a non deterministic choice has to be done. Moreover, given a state generated by $GENIT$, one has to show that this state is consistent. As for the latter point, the usual considerations about checking consistency hold (see, e.g., [8]). In relation with this problem we plan to study connections of our semantics with SAT [11] and ASP [13].

An important point of our approach is that we can use $GENIT$ as a first consistency check for an action application. Indeed, if X is the output of an action application over $IC_{\mathcal{R}}(\langle \bar{\gamma} \rangle \mathbf{T})$ and $GENIT(X, F)$ is empty for some $F \in \mathbf{T}$ then \mathbf{T} is not state consistent. This usually means that the action does not provide enough information to justify the system constraints as shown in the following example.

Example 5 (Action consistency check). Suppose that, in the development of our system, we write the following (wrong) version of $StartFireAlert(x)$:

$$\begin{aligned} StartFireAlert2(x) : \{x:Fire, x:Active\} \Rightarrow \\ \{\mathbf{alert1:CurrentAlert}, (\mathbf{alert1}, x) : \mathbf{hasReason}\} \end{aligned}$$

As can be noted, we forgot to set $x:CurrentSignal$ in $Post(StartFireAlert2(x))$. This action is “wrong” in the sense that its execution leads to an inconsistent

action output: if that is the case, GENIT will find this inconsistency as it is unable to generate a set of information terms for an axiom.

For example, consider the set Out' obtained applying the above action to the state $\bar{\gamma}_1$ of Example 3 (namely, $Out' = Out \setminus \{\text{fire_s1} : \text{CurrentSignal}\}$). The inconsistency of Out' is verified with the execution of $\text{GENIT}(Out', Ax_1)$: this execution leads to the following recursive executions of OPENIT on the subformulas of Ax_1

$$\begin{aligned} \text{OPENIT}(Out', x' : \text{CurrentSignal}) &= \emptyset \\ \text{OPENIT}(Out', x : \exists \text{hasReason.CurrentSignal}) &= \emptyset \\ \text{OPENIT}(Out', x : \exists \text{hasReason.CurrentSignal} \sqcap \text{Alert}) &= \emptyset \\ \text{OPENIT}(Out', x : \neg \text{CurrentAlert} \sqcup (\exists \text{hasReason.CurrentSignal} \sqcap \text{Alert})) &= H \end{aligned}$$

with $H = \{((1, \text{tt}), a) \mid a \in W \setminus \{\text{alert1}\}\}$. Since no information term in H can be associated to alert1 , $\text{GENIT}(Out', Ax_1) = \emptyset$. This means that the corresponding theory is not state consistent. We also remark that the execution of $\text{GENIT}(Out', x' : \text{CurrentSignal})$ traces the reason of state inconsistency. \diamond

We conclude this section noting that GENIT is exponential in the size of \mathcal{N} : this complexity is due to the case of $F = \forall A$, where the algorithm must generate all the possible functions ϕ such that $\phi : \mathcal{N} \rightarrow \bigcup_{d \in \mathcal{N}} \text{IT}_{\mathcal{N}}(d : A)$. The number of such functions is obviously exponential in the number of elements of \mathcal{N} , hence the complexity of GENIT. Without considering universal quantified formulas, the complexity of GENIT remains polynomial in the size of the input set X and the number of recursive calls of OPENIT only depends on the structure of the formula F .

6 Conclusion and future works

In this paper we have presented an action formalism based on the information terms semantics. We have shown how our semantics supports a natural notion of state and how an action language can be defined on the top of this notion. The problem to determine the consistency of an action is reduced to the problem to study the information terms generated by the application of GENIT. We have shown, by means of an example, how GENIT can be used, in some cases, to debug inconsistent actions. For lack of space we have not treated in details the problem to study the consistency of an action when the output of GENIT is not empty. However, we remark that this problem is similar to the problem to check *snapshot* consistency in CooML [8].

Compared with [2, 6, 7], our approach is still incomplete with respect to the study of typical action problems, such as planning [12], and relationships with classical action formalisms such as Situation Calculus [2] and Fluent Calculus [7]. However, our constructive approach already shows its advantages in checking consistency of an action application.

As for the future works we plan to investigate this question also considering its relations with model generation in SAT [11] and ASP [13] and with the planning problem. Moreover, we plan to study the projection problem (see,

e.g., [2]), that needs to be restated in our setting. We are also working on an implementation of our action language.

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
2. F. Baader, M. Milicic, C. Lutz, U. Sattler, and F. Wolter. Integrating description logics and action formalisms: First results. In I. Horrocks, U. Sattler, and F. Wolter, editors, *Proceedings of the 2005 International Workshop on Description Logics (DL2005)*, volume 147 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.
3. L. Bozzato, M. Ferrari, C. Fiorentini, and G. Fiorino. A constructive semantics for \mathcal{ALC} . In Calvanese et al. [5], pages 219–226.
4. Loris Bozzato, Mauro Ferrari, and Paola Villa. Actions over a constructive semantics for \mathcal{ALC} . Accepted at DL2008 - 21st International Workshop on Description Logics, 2008.
5. D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, S. Tessaris, and A. Turhan, editors. *Proceedings of the 20th International Workshop on Description Logics (DL2007)*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
6. D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Actions and programs over description logic ontologies. In Calvanese et al. [5], pages 29–40.
7. C. Drescher and M. Thielscher. Integrating action calculi and description logics. In J. Hertzberg, M. Beetz, and R. Englert, editors, *KI*, volume 4667 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2007.
8. M. Ferrari, C. Fiorentini, A. Momigliano, and M. Ornaghi. Snapshot generation in a constructive object-oriented modeling language. In A. King, editor, *Logic Based Program Synthesis and Transformation, LOPSTR 2007, Selected Papers*, volume 4915 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, 2008.
9. C. Fiorentini and M. Ornaghi. Answer set semantics vs. information term semantics. In *ASP2007: Answer Set Programming, Advances in Theory and Implementation*. <http://cooml.dsi.unimi.it/papers/asp.pdf>, 2007.
10. K. Kaneiwa. Negations in description logic - contraries, contradictories, and sub-contraries. In *Proceedings of the 13th International Conference on Conceptual Structures (ICCS '05)*, pages 66–79. Kassel University Press, 2005.
11. F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
12. M. Miličić. Planning in Action Formalisms based on DLs: First Results. In Calvanese et al. [5], pages 112–122.
13. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In *LPNMR*, pages 421–430, 1997.
14. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A Constructive Modeling Language for Object Oriented Information Systems. In *Constructive Logic for Automated Software Engineering*, volume 153 of *Electronic Notes in Theoretical Computer Science*, pages 55–75, 2006.
15. M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.
16. A. S. Troelstra. From constructivism to computer science. *TCS*, 211(1-2):233–252, 1999.

Lifting Databases to Ontologies^{*}

Gisella Bennardo, Giovanni Grasso, Salvatore Maria Ielpa, Nicola Leone,
Francesco Ricca

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{lastname}@mat.unical.it

Abstract. Nowadays it is widely recognized that ontologies are a fundamental tool for knowledge representation and reasoning; and, in particular, they have been recently exploited for setting out business enterprise information (obtaining the so-called *enterprise/corporate ontologies*). Enterprise ontologies offer a clean view of the enterprise knowledge, simplifying the retrieval of information and the discovery of new knowledge through powerful reasoning mechanisms.

However, enterprise ontologies are not widely used yet, mainly because of two major obstacles: (i) the specification of a real-world enterprise ontology is an hard task, developing an enterprise ontology by scratch would be a time-consuming and expensive task; and, (ii) usually, enterprises already store their relevant information in large database systems, and do not want to load the information again in the ontologies; moreover, these databases have to keep their autonomy since many applications work on them. In this paper we propose a solution that combines the advantages of an ontology representation language (i.e., high expressive power and clean representation of data) having powerful reasoning capabilities, with the capability to efficiently exploit a large (and, often already existent) enterprise database. In particular, we allow to “lift” an existing database to an ontology. The database is kept and the existing applications can still work on it, but the user can take profit of the new ontological view of the data, and exploit powerful reasoning mechanisms for consistency checking, knowledge discovery, and other advanced knowledge-based tasks.

Keywords: Logic Programming, Answer Set Programming, Ontology Languages, Enterprise Ontologies, Databases.

1 Introduction

Nowadays, the need for knowledge-based technologies is emerging in several application areas and, in particular, both enterprises and large organizations are looking for powerful instruments for knowledge-representation and reasoning.

^{*} Supported by M.I.U.R. within projects “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva” and “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione.”

In this field, *ontologies* [1] have been recognized to be a fundamental tool. Indeed, ontologies are well-suited formal tools to provide a clean abstract specification of the entities of a given domain and powerful reasoning capabilities. In particular, they have been recently exploited for specifying terms and definitions relevant to business enterprises, obtaining the so-called *enterprise/corporate ontologies*.

Enterprise/Corporate ontologies can be used to share/manipulate the information already present in a company. Indeed, they provide for a “conceptual view” expressing at the intensional level complex relationships among the entities of enterprise domains. In this way, they can offer a convenient access to the enterprise knowledge, simplifying the retrieval of information and the discovery of new knowledge through powerful reasoning mechanisms.

However, enterprise ontologies are not widely used yet, mainly because of two major obstacles:

- (i) the specification of a real-world enterprise ontology is an hard task;
- (ii) usually, enterprises already store their relevant information in large database systems.

As far as point (i) is concerned, it can be easily seen that developing an enterprise ontology by scratch would be a time-consuming and expensive task, requiring the cooperation of knowledge engineers with domain experts. Moreover, (ii) the obtained specification must incorporate the knowledge (mainly regarding concept instances) already present in the enterprise information systems. This knowledge is often stored in large (relational) database systems, and loading it again in the ontologies may be unpractical or even unfeasible. This happens because of the large amount of data to deal with, but also because these databases have to keep their autonomy (considering that many applications work on them).

In this paper we propose a solution that combines the advantages of an ontology representation language (i.e., high expressive power and clean representation of data) having powerful reasoning features, with the capability to efficiently exploit a large (and, often already existent) enterprise database. In particular, we allow to “lift” an existing database to an ontology in the spirit of [2]. The database is kept and the existing applications can still work on it, but the user can take profit of the new ontological view of the data, and exploit powerful reasoning mechanisms for consistency checking, knowledge discovery, and other advanced knowledge-based tasks.

This has been done by properly extending the OntoDLV system [3, 4] language by suitable constructs, called *virtual class* and *virtual relation*, that allows one to specify the instances of an ontology concept/relation when they “already exist” autonomously in a relational database.

More in detail, OntoDLV implements a powerful logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) Answer Set Programming [5–7] (ASP) with all the main ontology constructs including classes, inheritance, relations, and axioms. OntoDLP is strongly typed,

and it combines in a natural way the modeling power of ontologies with a powerful “rule-based” language.¹

Suppose now that it is given an existing database; then, we can analyze its schema and comfortably recognize both entities and relationships that the database engineer stored on it, and we can represent it by means of an OntoDLP ontology. This gives us a clean and high level specification of the knowledge present in the given database, but the obtained intensional specification is not linked with it. Since, up to now, both ontology specification and logic programs have to be specified directly in the OntoDLP syntax in order to exploit the ontology, then the database should be loaded into the OntoDLV system (which, as previously pointed out, is inconvenient). To overcome this limitation, we purposely extended the OntoDLV language in order to directly specify how the instances of an OntoDLP class have to be obtained from the given database. In particular, we introduced *virtual classes* (and *virtual relations*), that are classes (and relations) whose instances are specified by means of special logic rules. Those rules admit a special kind of logic predicates which represent database tables/views or even SQL queries results. Basically, those rules define a mapping among the data in the database and the corresponding instances of the ontology. Given that, the obtained ontology specification can be exploited as usual, and all the powerful features on OntoDLP (from advanced type-checking to complex reasoning) can be exploited on existing data.

Moreover, we extended the OntoDLV system in order to implement *virtual classes* and *virtual relations*, in such a way that it seamlessly provide to the users both abstract browsing and query facility on the ontology and efficient query processing on existing data sources.

The remainder of the paper is organized as follows. In the next section, we provide a brief overview of the original OntoDLP language. In Section 3 we show, by means of an example, how an existing database can be lift to an ontology by exploiting virtual classes. Section 4 overviews the architecture and the implementation of virtual classes in the OntoDLV system. Eventually, in Section 5 we draw conclusions and compare related works.

2 The OntoDLP language

In this section we briefly describe OntoDLP, an ontology representation and reasoning language which provides the most important ontological constructs, namely classes, attributes, relations, inheritance and axioms, and combines them with the reasoning capabilities of ASP. For a detailed description refer to [3, 4].

Hereafter, we assume the reader to be familiar with ASP syntax and semantics, for further details refer to [5, 9].

Classes. A *class* can be thought of as a collection of individuals that belong together because they share some properties. Classes can be defined in OntoDLP

¹ In general, disjunctive ASP, and thus OntoDLP, can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [8].

by using the keyword **class** followed by its name. Class attributes can be specified by means of pairs (*attribute-name* : *attribute-type*), where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

For instance, *person*, *food*, and *place* are classes of individuals, that can be defined in OntoDLV as follows:

```
class place(name : string).
class food(name : string, origin : place).
class person(name : string, father : person, mother : person, birthplace : place).
```

Class attributes in OntoDLP model the properties that *must* be present in all class instances; properties that *might* be present or not might be modeled, for instance, by using relations. Moreover, class definitions can be recursive (e.g., in class *person* both *father* and *mother* are of type *person*), and attribute types can exploit the built-in classes *string* and *integer* (respectively representing the class of all alphanumeric strings and the class of non-negative integers).

Objects. Domains contain individuals which are called *objects* or *instances*. Each individual in OntoDLP belongs to a class and is uniquely identified by a constant called *object identifier* (oid). Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, with the fact:

```
john : person(name : "John", father : jack, mother : ann, birthplace : rome).
```

we declare that *john* is an instances of the class *person*. Note that, when we declare an instance, we immediately give an oid to the instance which may be used to fill an attribute of another object. In the example above, the attribute *birthplace* is filled with the oid *rome* (identifying an instance of *Place*) modeling the fact that *john* is born in Rome; in the same way, *jack* and *ann* are suitable oids respectively filling the attributes *father*, *mother* (both of type *person*).

Oids are proper of a given base class, i.e., base classes cannot share individuals. However, an individual may belong to different classes when other two modeling tools are employed (that will be described later), namely: inheritance and collection classes.

Inheritance. Concepts in an ontology are usually organized in taxonomies by using the *specialization/generalization* mechanism (which is called *inheritance* in object-oriented languages). For instance, employees are a special category of person having extra attributes, like *salary* and *company*. OntoDLV supports inheritance by means of the special binary relation *isa*. In particular, the above-mentioned employee class can be declared as follows:

```
class employee isa {person}(salary : integer, boss : person).
```

In this case, *person* is a more generic concept or *superclass* and *employee* is a specialization (or *subclass*) of *person*. Moreover, an instance of *employee* will have the local attributes *salary* and *boss*, in addition to those that are defined in *person*. We say that the latter are “inherited” from the superclass *person*. Hence, each proper instance of *employee* will also be automatically considered an instance of *person* (the opposite does not hold!).

Collection Classes The notions of base class and base relation introduced above correspond, from a database point of view, to the *extensional* part of the OntoDLP language. However, there are many cases in which some property or some class of individuals can be “derived” (or inferred) from the information already stated in an ontology. In particular, OntoDLP allows one to specify the instances of a class by means of logic rules, thus obtaining a *Collection class*.

For instance, the class *richEmployee* can be defined as follows:

```
collection class richEmployee(name: string){
  E : richEmployee(name: N) :- E : employee(name: N, salary: S), S > 1000000.}
```

Basically, this class *collects* instances defined by another class (i.e., *person*) and performs a re-classification based on some information which is already present in the ontology.

Importantly, the programs (set of rules) defining collection classes must be normal and stratified (see e.g., [10, 11]).

Relations. Another important feature of an ontology language is the ability to model relationships among individuals. Relations are declared like classes: the keyword *relation* (instead of *class*) precedes a list of attributes.

As an example, we model a relationship between person and living place as follows:

```
relation personLivesIn(individual: person, location: place).
```

The instances of a relation are called *tuples*; for instance, we can assert that *john* lives in *rome* by writing a logic fact as follows:

```
personLivesIn(individual: john, location: rome).
```

Contrary to class instances, tuples are not equipped with an oid.

As for classes, also relations can be defined via rules, obtaining the so-called *intensional relations*.

We complete the description of relations observing that OntoDLP allows one also to organize them in taxonomies. Basically, attributes and tuples are inherited by following the same criterions defined above for classes.

Axioms and Consistency. *Axioms* are a consistency-control construct modeling sentences that are always true. For example, we may enforce that a person cannot be father of itself as follows:

```
:- X : person(father: X).
```

If an axiom is violated, we say that the ontology is inconsistent (i.e., it contains information which is contradictory or not compliant with the domain’s intended perception).

Reasoning modules and queries. In addition to the ontology specification, OntoDLP provides powerful reasoning and querying capabilities by means of the language components *reasoning modules* and *queries*.

In practice, a *reasoning module* is a disjunctive logic program conceived to reason about the data described in an ontology. Reasoning modules are identified

by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints; clearly, the rules of a module can access the information present in the ontology.

An important feature of the language is the possibility of asking queries, on both the ontology and the predicates defined in reasoning modules. Queries offer the possibility of extract knowledge implicitly contained in the ontology. As an example, we ask for the list of person having a father who is born in Rome as follows:

$$X : person(father : person(birthplace : place(name : "Rome")))?$$

3 Virtual Classes and Virtual Relations

In this section we show how an existing database database can be “lift” to an OntoDLP ontology. In particular, the new features of the language, namely, *virtual classes* and *virtual relations*, which have been conceived for dealing with this problem, will be illustrated by exploiting the following example.

Suppose that a Banking Enterprise asks for building an ontology of its domain of interest. This request has arisen from the need of an uniform view of the knowledge stored in the enterprise information system, which is shared among all the enterprise branches. Indeed, ontologies offer a clear high-level perspective of a domain, which can be, thus, analyzed by exploiting more expressive and powerful querying/reasoning methods.

The schema of the existing database exploited by the information system of the banking enterprise is reported in Table 1.

The first step that must be done is to reconstruct the semantics of data stored in this database.

It is worth noting that, in general, a database schema is the product of a previously-done modeling step on the domain of interest. Usually, the result of this conceptual-design phase is a semantic data model which describes the structure of the entities stored in the database. Likely, the database engineers exploited the Entity-Relationship Model (ER-model) [12], that consists of a set of basic objects (called entities), and of relationships among these objects. The ER-model underlying a database can be reconstructed by reverse-engineering (there are few well-known rules for obtaining a database schema from an ER-model) or can be directly obtained from the documentation of the original project.

Suppose now that, we obtained the ER-model corresponding to the database of Table 1. In particular, the corresponding ER diagram is shown in Figure 1. From this diagram it is easy to recognize that the enterprise is organized into *branches*, which are located into a given place and also have an asset and an unique name. A bank *customer* is identified by its social-security number and, in addition, the bank stores information about customer’s name, street and living place. Moreover, customers may have *accounts* and can take out *loans*. The bank offers two types of *accounts*: *saving-accounts* with an interest-rate, and *checking-accounts* with a overdraft-amount. To each account is assigned an unique account-number, and maintains last access date. Moreover, accounts can

Table name	attributes
Branch	<u>branch-name</u> branch-city assets
Customer	customer-name <u>social-security</u> customer-street customer-city
Depositor	<u>customer-social-sec</u> <u>account-number</u> access-date
Saving-account	<u>account-number</u> balance interest-rate
Checking-account	<u>account-number</u> balance overdraft-amount
Loan	<u>loan-number</u> amount branch-name
Borrower	<u>customer-social-sec</u> <u>loan-number</u>
Payment	<u>loan-number</u> <u>payment-number</u> payment-date payment-amount

Table 1. The Banking Enterprise database.

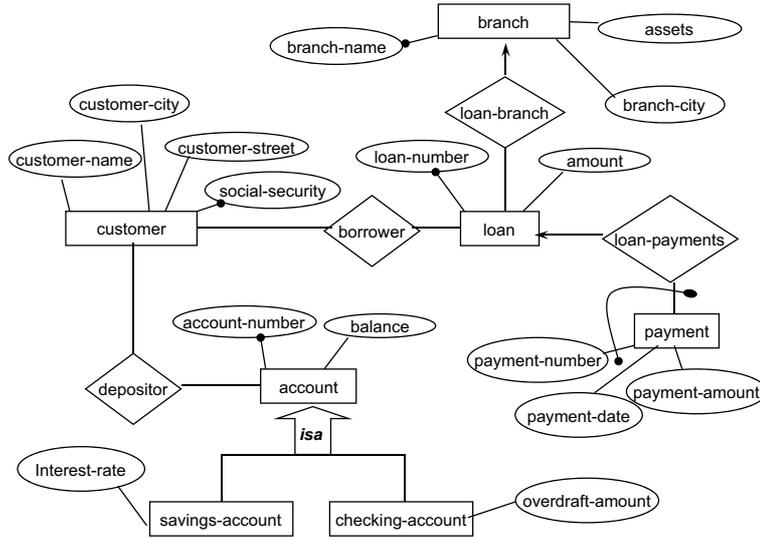


Fig. 1. The Banking Enterprise ER diagram

be held by more than one customer, and obviously one customer can have various accounts (*depositors*). Note that, in the case of *accounts*, the ER-model exploits specialization/generalization construct. A *loan* is identified by a unique loan-number and, as well as accounts, can be held by several customers (*borrowers*). In addition, the bank keeps track of the loan amount and *payments* and also of the branch at the loan originates. For each payment the bank records the date and the amount; for a specific loan a payment-number univocally identifies a particular payment.

All this information we obtained so-far, represents a good starting point for defining an ontology that describes the banking enterprise domain. Indeed, we can easily exploit it for identifying ontology concepts and for detecting which tables store the information regarding their instances.

At this point, what we have to do for “lifting” the banking database to a banking ontology is to create an OntoDLP (base) class, with name c , for each concept c in the domain, and exploit special logic rules to specify a mapping between class c and its instances “stored” in the database. A class c defined by means of such kind of mapping rules is called *virtual*, because its instances come from an external source but, as far as reasoning and querying are concerned, they appear directly specified in OntoDLP.

Thus, in more detail, a *virtual class* is defined by using the keywords **virtual class** followed by its name, and specifying a list of attributes by means of pairs (*attribute-name : attribute-type*); while, as previously pointed out, its instances are defined by means of rules, which contain some special predicates that allows one to access the original database tables. For example, we model the *branch*

entity as follows:

```
virtual class branch(name: string, city: string, assets: integer){
  f(BN) : branch(name: BN, city: BC, assets: A) :-
    [db1, "SELECT branch-name AS BN, branch-city AS BC, assets AS A
      FROM branch"]}
```

Note how the rule acts as mapping between the data contained in table *branch* and the instances of class *branch* by exploiting a new type of atom (called *SQL atom*) which contains an SQL query. More in detail, a *SQL atom* consists of a pair [db object identifier, sql query] enclosed in square brackets. The db object identifier plays a crucial role, because identifies the database on which the sql query will be performed. As a matter of fact, data sources are specified directly in OntoDLP as instances of the built-in class *dbSource* as follows:

```
db1 : dbSource(connectionURI: "http://mydb.mysite.com", user: "myUser",
  password: "myPsw").
```

This statement is automatically recognized, since the system automatically provides those declarations:

```
class externalSource .
class dbSource isa {externalSource}(connectionURI: string, user: string,
  password: string).
```

Note that such a mechanism allows to build an ontology starting from one or more databases, just specifying more *dbSources*. Moreover, this scheme is sufficiently general to be (in the future) extended also to access other kind of sources beside databases.

An important issue to be better described in the above example regards the way how object identifiers for virtual class instances are built. First of all, note that while the database stores values, ontologies manage instances each of which is uniquely identified by an object identifier, which is not merely a value. This is the well-known impedance mismatch problem. We provide a specific mechanisms for facing this problem, in which values appearing in the databases are kept, somehow, distinct from object identifiers appearing in the ontology. To this end, all the instances of a *virtual class* are identified by means of a *functional object identifier* that is suitably built from data values stored at the databases. In our example, the head of the mapping rule contains the functional term $f(BN)$, that builds, for each instance of *branch*, a *functional object identifier* composed of the functor f containing the value of the *name* attribute stored at the table *branch*.²

² Note that *name* is a key for table *branch*. Since object identifiers in OntoDLP uniquely identify instances, it is preferable to exploit only keys for defining functional object identifiers. This simple policy ensures that we will obtain an admissible ontology; however, in order to obtain the maximum flexibility, the responsibility of writing a “right” ontology mapping is left to the ontology engineer.

In practice, if the *branch* table stores a tuple (*Spagna, Rome, 1000000*), then the associated instance in the ontology will be:

$$f(\textit{Spagna}) : \textit{branch}(\textit{name} : \textit{Veneto}, \textit{city} : \textit{Rome}, \textit{assets} : 1000000)$$

In this way we build the *functional object identifier* $f(\textit{Spagna})$ starting from the data value *Spagna*, keeping data values alphabet distinct from the one of *functional object identifiers*.

We say that a *virtual class* declared by means of *SQL atoms* is in *sql notation*, but we provided, in addition, a more direct notation for accessing database tables, called *logical notation*.

In particular, the *virtual class* *branch* can be equivalently defined as follows:

$$\begin{aligned} &\mathbf{virtual\ class}\ \textit{branch}(\textit{name} : \textit{string}, \textit{city} : \textit{string}, \textit{assets} : \textit{integer})\{ \\ &\quad f(\textit{BN}) : \textit{branch}(\textit{name} : \textit{BN}, \textit{city} : \textit{BC}, \textit{assets} : \textit{A}) :- \\ &\quad \quad \textit{branch@db1}(\textit{branch-name} : \textit{BN}, \textit{branch-city} : \textit{BC}, \textit{assets} : \textit{A}). \} \end{aligned}$$

The *logical notation* differs from the *sql* one by using *sourced atoms* in replacing of *SQL atoms*. A *sourced atoms* consist of a name (*branch*) that identifies a table "at" (@) a specific database source (*db1*), in addition to a list of attribute-names (that must match those in the table) linked to values or variables.

Hereafter, we will use the *logical notation* in all the examples of *virtual classes* and *virtual relations*.

The next entity we focus on is *customer*, for dealing with it we define another virtual class as follows:

$$\begin{aligned} &\mathbf{virtual\ class}\ \textit{customer}(\textit{ssn} : \textit{string}, \textit{name} : \textit{string}, \textit{street} : \textit{string}, \textit{city} : \textit{string})\{ \\ &\quad c(\textit{SSN}) : \textit{customer}(\textit{ssn} : \textit{SSN}, \textit{name} : \textit{N}, \textit{street} : \textit{S}, \textit{city} : \textit{C}) :- \\ &\quad \quad \textit{customer@db1}(\textit{social-security} : \textit{SSN}, \textit{customer-name} : \textit{N}, \textit{customer-street} : \textit{S}, \\ &\quad \quad \quad \textit{customer-city} : \textit{C}). \} \end{aligned}$$

Note that, in this case we used the functional term $c(\textit{SSN})$ in order to assign to each instance a suitable *functional object identifier* built on the *social-security* attribute value. Actually, we use one fresh functor for each virtual class; in this way, we are sure that functional object identifiers, belonging to different classes, are distinct. In our example, the *customer* and the *branch* class instances, are thus made disjoint. In fact, the former uses the functor f , while the latter uses the functor c .

Following the same methodology, we can define a virtual class for the *loan* entity:

$$\begin{aligned} &\mathbf{virtual\ class}\ \textit{loan}(\textit{number} : \textit{integer}, \textit{loaner} : \textit{branch}, \textit{amount} : \textit{integer})\{ \\ &\quad l(\textit{N}) : \textit{loan}(\textit{number} : \textit{N}, \textit{loaner} : f(\textit{L}), \textit{amount} : \textit{A}) :- \\ &\quad \quad \textit{loan@db1}(\textit{loan-number} : \textit{N}, \textit{branch-name} : \textit{L}, \textit{amount} : \textit{A}). \} \end{aligned}$$

The above examples is slightly different from the ones so far illustrated. In fact, the *loan* class has an attribute (*loaner*) of type *branch*, which is a virtual class too. In this case we have to carefully deal with functional terms in order to ensure referential integrity. As shown above in our example, we face this conditions by properly using *functional object identifiers*. Note in fact that the

mapping uses the functional term $f(L)$ to build values for the *loaner* attribute (rule's head of the *loan* virtual class).

Basically, since the *branch* class use the functor f to build its object identifiers, then we also use the same functor where an object identifier of *branch* is expected. In this way, we maintain referential integrity constraints unchanged at the ontology level, achieving a consistency-safe results.

In the next example we stress the above idea while modeling the *payment* entity:

```
virtual class payment(ref-loan : loan, number : integer, payDate : date,
                    amount : integer){
  p(l(L), N) : payment(ref-loan : l(L), number : N, payDate : D, amount : A) :-
  payment@db1(loan-number : L, payment-number : N, payment-date : D,
             payment-amount : A).}
```

Also in this case we deal with referential integrity constraints by using a proper functional term $l(L)$ where a *loan* object identifier is expected (*ref-loan* attribute); moreover, since payments are identified by a pair (payment-number, relative loan) each instance of *payment* will be identified by a functional object identifier with two arguments: one of these is a functional object identifier of type *loan*; and, the other is the loan number.

As far as *accounts* are concerned, we know from the ER-model that they are specialized in two types: *saving-accounts* and *checking-accounts*. This situation can be easily dealt with in OntoDLP by exploiting inheritance (see Section 2). Thus, we introduce a *virtual class* named *account* as follows:

```
virtual class account(number : integer, balance : integer).
```

and, in addition, we provide two *virtual classes* *savingAccount* and *checkingAccount* both subclasses of *account* which contain the mappings with the corresponding database tables:

```
virtual class savingAccount isa {account}(interestRate : integer){
  acc(N) : savingAccount(number : N, balance : B, interestRate : I) :-
  saving-account @db1(account-number : N, balance : L, interest-rate : I).}
```

```
virtual class checkingAccount isa {account}(overdraft : integer){
  acc(N) : checkingAccount(number : N, balance : B, overdraft : I) :-
  checking-account @db1(account-number : N, balance : L, overdraft-amount : I).}
```

Up to now, we specified all the concepts in the banking domain, but we miss model relationship between them. For instance, the ER diagram clearly shows that *customers* and *loans* are in relationship through relations *borrower* and *depositor*. To deal with this problem, OntoDLP allows to define also *virtual relations* besides virtual classes. Hence, we can directly model both *borrower* and *depositor* as follows:

```
virtual relation borrower(cust : customer, loan : loan){
  borrower(cust : c(C), loan : l(L)) :-
  borrower@db1(customer-social-sec : C, loan-number : L).}
```

virtual relation $depositor(cust: customer, account: account, , lastAccess: date)\{$
 $depositor(cust: c(C), account: acc(A), lastAccess: D) :-$
 $depositor@db1(customer-social-sec: C, account-number: A, access-date: d).\}$

It is worth noting that a *virtual relation* differs from a *virtual class* mainly because the latter does not specify object identifiers for its instances (tuples). In fact, *virtual relations* represent some properties that link individuals already present in the ontology. However, as previously pointed out, we have to carefully take into account integrity constraints by properly using functional object identifiers.

4 Virtual Entities Implementation

In this Section we describe how *virtual classes* and *virtual relations* have been implemented into the OntoDLV system. To this end, we first describe the general architecture of the system, and then we detail the modules that have been introduced/extended for dealing with the new features. We refrain from giving an in-depth description of all technical details underlying the implementation of OntoDLV, rather we present the main new features of the system.

4.1 OntoDLV Architecture

OntoDLV is a complete framework that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies.

The system architecture of OntoDLV, depicted in Figure 2, can be divided in three abstraction layers. The lowest layer, named *OntoDLV core* contains the components implementing the main functionalities of the system; above it, the *Application Programming Interface* (API) act as a facade for supporting for the development of applications based on the core; while the Graphical User Interface (GUI) is the end-user interface of the system.

In turn, the OntoDLV core is made of three submodules, namely: *Persistency Manager*, *Type Checker*, and *Rewriter*. The Persistency Manager provides all the methods needed to store and manipulate the ontology components. In particular, this module of the system is able to deal with large distributed ontologies. Indeed, ontologies can be stored transparently in a number of text files and/or database management systems, possibly distributed across several machines.

Text files in OntoDLP format are analyzed by the *Parser* module that builds in main memory an image of the ontology components it recognizes; while, the *DB Manager* module is able to manipulate ontology entities that are imported into the system and stored in mass-memory by exploiting relational databases. In order to deal with virtual classes and virtual relations, we introduced a new submodule of the persistency manager, called *Virtual Entity Manager*, which will be described more in detail in the next subsection.

The *Persistency Manager* builds a global view of the distributed ontology which can be then exploited by the other components of the kernel, namely: *Type Checker* and *Rewriter*.

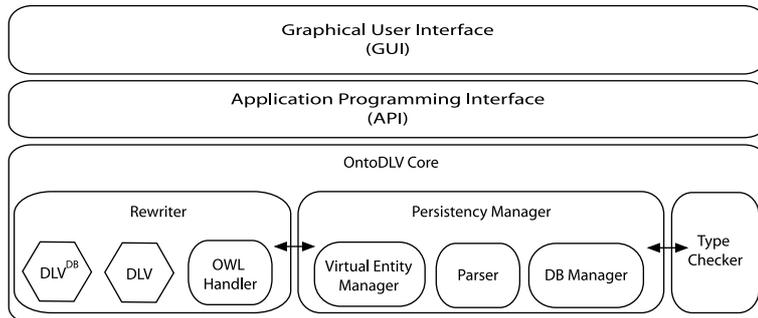


Fig. 2. The OntoDLV architecture.

The *Type Checker* module verifies the admissibility of an ontology by exploiting a number of type checking routines. It is important to say that, if the loaded ontology contains some admissibility problem (e.g., a class is declared twice) the type checker builds a precise description of the problem. This information can be exploited by external applications, and in particular the user interface of OntoDLV relies on this feature for helping the ontology design during the development process.

The *Rewriter* module translates OntoDLP ontologies, axioms, reasoning modules and queries to an equivalent ASP program [3] which, in the general case runs on the DLV system [9]. DLV is a state-of-the art ASP system that has been shown to perform efficiently on both hard and “easy” (having polynomial complexity) problems. Moreover, if the rewritten program is stratified and non disjunctive, then the evaluation is efficiently carried out on mass memory by exploiting a specialized version of DLV, called DLV^{DB} [13]. This feature has been purposely introduced to implement in an efficient way virtual classes and virtual relations. To this end, the original rewriter of OntoDLV has been extended to deal with the new features as described in the next subsection.

Finally, we recall that third parties are allowed for developing their own knowledge-based applications on top of OntoDLV by exploiting a rich Application Programming Interface: the OntoDLV API [14]. Moreover, the end user exploits the system through an easy-to-use and intuitive visual development environment called *GUI* (Graphical User Interface), which is built on top of the *OntoDLV API*. The OntoDLV GUI was designed to be simple for a novice to understand and use, and powerful enough to support experienced users.

4.2 Extension of the OntoDLV core: Virtual Entity Manager and New Rewriter

The implementation of *virtual classes* and *virtual relation* has been carried out by properly improving the *Rewriter* module and by adding a new submodule of the *Persistency Manager*, namely: the *Virtual Entity Manager*. The latter is in charge to make available suitable methods form defining, manipulating and storing both virtual classes and relations definitions.

More in detail, the *Virtual Entity Manager* implements two different usage modalities for virtual entities, that we call *off-line* and *on-line* modes. The first consists of materializing in OntoDLV the instances of virtual entities according with their respective mapping rules. Basically, a suitable routine performs the SQL queries on the proper database and each tuples of the result set is stored into the internal data structures (basically, instances are stored into the existing *DB manager* module). The *off-line* mode is preferable when one wants to migrate the database into an ontology, or when parts of a proprietary database are one-time granted to third parties. In fact, once the materialization is obtained, the source database can be disconnected, since the data are stored into the OntoDLV persistency manager. Obviously, depending on database size, instance materialization could be time-consuming or even unpractical and, in addition, one may want keep the information in the database (which is accessed by legacy applications) in order to deal always with “updated” information. In this case the *on-line* mode is preferable, in which queries are performed directly at the sources. In fact, our implementation allows to efficiently perform reasoning/querying tasks directly on databases, with a very limited usage of main-memory. This can be achieved by exploiting DLV^{DB} [13], a specialized version of DLV that addresses the problem of reasoning on massive amounts of (possibly distributed) data.

In order to integrate DLV^{DB} in OntoDLV we extended the Rewriter module to generate the mapping statements that DLV^{DB} requires. In fact, to properly carry out the program evaluation, it is necessary to specify the mappings between input and output data and program predicates. For a better understanding, in the following we show which mappings are needed to rewrite the virtual class *branch*. We recall that the *branch* definition is:

```
virtual class branch(name: string, city: string, assets: integer){
  f(BN) : branch(name: BN, city: BC, assets: A) :-
    branch@db1(branch-name: BN, branch-city: BC, assets: A).
```

Then the following directives for DLV^{DB} are generated by the new rewriter:

```
USEDDB "http://mydb.mysite.com":myUser:myPsw.
USE branch (branch-name, branch-city, assets)
MAPTO branchPredicate (varchar,varchar,integer).
```

The above directive specifies the database source (**USEDDB**) on which the SQL query will be performed. Moreover, the listed attributes of the table *branch* (**USE**) are mapped (**MAPTO**) on the logic predicate *branchPredicate*. In this case, *branchPredicate* is the predicate name used internally to rewrite in standard ASP the class *branch*.

Note that, for obtaining the integration of a legacy database system we dealt with several non-trivial technical problems that are mainly due to the different

ways in which different DBMSs store/represent data.³ However, we refrain from reporting them here, mainly because of their inherent technical nature.

5 Conclusion and Related Work

In this paper we proposed a solution that allows one to “lift” an existing database to an ontology. The result is the natural combination of the advantages of an ontology language (clean high-level view of the information and powerful reasoning capabilities) with the efficient exploitation of a large already-existent databases.

This has been obtained by properly extending the OntoDLV and system by means of *virtual classes* and *virtual relations*. The new modeling constructs allow the knowledge engineer for defining the instances of classes and relations of an enterprise ontology by means of special logic rules, which act as a mapping from the information stored in database tables to concept instances. In this way, the database is kept and the possibly already-existing applications can still work on it, but the user can take profit of the new ontological view of the data, and he/she can exploit the powerful reasoning mechanisms of the OntoDLV system for consistency checking, knowledge discovery, and other advanced knowledge-based tasks.

As a matter of fact, the problem of linking ontology to databases is not new [2] and has been studied also for other ontology languages. For instance, in [15] a set of pre-existing data sources is linked to the description logic DL-Lite_A. In this approach, a very similar solution for creating object identifiers from database values (which exploits function symbols) is used and, query answering on the obtained ontology is very efficient/scalable (it can be performed in LogSpace in the size of the original database). This makes the solution proposed in [15] very effective when dealing with large databases, and complexity-wise cheaper than our approach. However, the language of OntoDLV is much more expressive than DL-Lite_A. Indeed, OntoDLP can express in a natural way more complex reasoning tasks on the ontology, which can be very useful for an enterprise, like e.g. solving an instance of the team building problem.⁴

References

1. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition* **5** (1993) 199–220
2. Lenzerini, M.: Data integration: a theoretical perspective. In Popa, L., ed.: PODS '02: Proceedings of the twenty-first ACM SIGMOD- SIGACT-SIGART symposium on Principles of database systems, New York, USA, ACM (2002) 233–246

³ For example, there are several strings representation formats (quoted, unquoted, system reserved chars, etc.) or several different *date* types. Not all DBMS vendors adopt the same representation, so our routine must “understand” different formats and convert them properly according with the OntoDLV data-type system.

⁴ Which amounts to finding a team of employees which satisfies some constraints on a project, like the overall budget, the maximum number of members, and so on.

3. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV⁺ System. *Journal of Applied Logics* **5** (2007) 545–573
4. Dell’Armi, T., Gallucci, L., Leone, N., Ricca, F., Schindlauer, R.: OntoDLV: an ASP-based System for Enterprise Ontologies. In: *Proceedings ASP07 - Answer Set Programming: Advances in Theory and Implementation*. (2007) Invited for publication in *Journal of Logic and Computation*.
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *NGC* **9** (1991) 365–385
6. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence* **138** (2002) 3–38
7. Minker, J.: Overview of Disjunctive Logic Programming. *AMAI* **12** (1994) 1–24
8. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. *ACM TODS* **22** (1997) 364–418
9. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* **7** (2006) 499–562
10. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In Minker, J., ed.: *Foundations of Deductive Databases and Logic Programming*. Washington DC (1988) 89–148
11. Przymusiński, T.C.: On the Declarative Semantics of Deductive Databases and Logic Programs. In Minker, J., ed.: *Foundations of Deductive Databases and Logic Programming*. (1988) 193–216
12. Ullman, J.D.: *Principles of Database and Knowledge Base Systems*. Computer Science Press (1989)
13. Giorgio, T., Leone, N., Vincenzino, L., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming* **7** (2007) 1–37
14. Gallucci, L., Ricca, F.: Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In: *Proc. of SEA’07 Workshop*, Arizona, USA, 2007. (1998) 56–70
15. Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Linking Ontologies to Data. *Journal of Data Semantics* (2008) 133–173

A graphical representation of relational formulae with complementation

Domenico Cantone¹, Andrea Formisano²,
Marianna Nicolosi Asmundo¹, and Eugenio G. Omodeo³

¹ Università di Catania, email: cantone@dmi.unict.it, nicolosi@dmi.unict.it

² Università di Perugia, email: formis@dipmat.unipg.it

³ Università di Trieste, email: eomodeo@units.it

Abstract. We improve existing techniques for graphical representation of relational expressions and for exploitation of this representation in the translation of (hybrid) dyadic first-order sentences into equalities between relational expressions. The enhanced technique can cope with the relational complement construct and the negation connective.

Complementation is handled by adopting a Smullyan-like uniform notation to classify and decompose map expressions, whereas negation is treated by generalizing the notion of graph for a formula in \mathcal{L}^+ and by introducing a series of graph transformation rules which reflect the meaning of the connectives and quantifiers occurring in the formula.

Key words: Algebraic logic, quantifier elimination, graph rewriting.

Introduction

The possibility to exploit map calculus for mechanical reasoning can be caught from [11], where Tarski and Givant show how to recast many axiomatic systems of Set Theory as equational theories based on a relational language devoid of quantifiers.

Map calculus [6] cannot represent *per se* an alternative to predicate calculus. As for expressive power, it corresponds in fact to a fragment of first order logic endowed with only three individual variables and with binary predicates only. As for deductive power, it is semantically incomplete; that is, there are semantically valid equations which are not derivable within it. Moreover, predicate logic has acquired such an unquestioned status of *de facto* standard as to make one reluctant to adopt the map formalism in its stead, in spite of the greater conciseness of the latter.

Nonetheless, map calculus can be applied in synergy with predicate calculus, inside theorem provers or proof assistants, as an inferential engine to be used in the activities of proof-search and model building. This calls for translation algorithms that can bridge across predicate logic and map calculus. Moreover, to increase readability by exploiting the immediate perspicuity of graphics, it is useful to design algorithms allowing one to represent map formulae in a visually alluring way.

In [1, 2, 5] both problems have been addressed by presenting an algorithm for translating formulae of dyadic predicate logic into map algebra, and an algorithm for converting map expressions into a graphical representation. Both algorithms are based on suitably defined graphs; one of them is designed to treat existentially quantified conjunctions of literals, the other to treat map expressions containing the constructs of relational intersection, composition and conversion.

In this paper the techniques introduced in [1, 2, 5] are further extended to the treatment of formulae which involve the negation connective and of expressions involving the relational complement construct. This allows us to get a graphical representation of any map expression, and to process any formula of dyadic predicate logic with the aim of getting an equivalent map equation. In the latter case, the algorithm which we will present sometimes fails to find the sought translation even if it exists. This apparent drawback, which also affected the earlier versions of the algorithm, stems from an unsurmountable limiting result [9], namely the fact that no algorithm can establish in full generality whether a given sentence in $n + 1$ variables is logically equivalent to some other sentence in n variables, for any integer number $n > 1$.

The enhanced techniques in this paper have been obtained by extending Smullyan's unifying notation for formulae of predicate logic to map expressions, by enriching the notion of directed multigraph associated with formulae (map expressions) given in [1, 2, 5] so that nodes are labeled with sets of variables (instead of with single variables), and so that a relation is induced between the components of the multigraph containing an edge labeled with a disjunctive formula (map expression) and the components representing the complementary formula (map expression).

In particular, in the *graph fattening* algorithm, which translates map expressions into directed multigraphs whose edges are labeled with map literals (map letters or their complements), Smullyan's unifying notation allows us to decompose map expressions by pushing the complement construct inward till map literals are reached. When a disjunctive expression (viz. one of the form $P \cup Q$) is reached during the decomposition process, a new component of the multigraph representing its complement is created, and the first component is linked with the second one.

In the *graph thinning* algorithm, which translates formulae of dyadic predicate logic into map expressions, disjunctive formulae are handled in analogy to disjunctive expressions in the graph fattening algorithm, by resorting to the relationship among components of the graphs, and to the new labeling of nodes. Existential and universal quantifiers are treated by classifying bound variables in such a way that it is always possible to distinguish between existentially and universally quantified variables.

1 The languages \mathcal{L}^\times and \mathcal{L}^+

\mathcal{L}^\times is an equational language devoid of variables where one can state properties of dyadic relations, *maps*, over an unspecified, yet fixed, *domain* \mathcal{U} of *discourse*.

Its basic ingredients are three *constants* $\mathbf{0}$, $\mathbf{1}$, ι ; infinitely many *map letters* P_1, P_2, P_3, \dots ; dyadic constructs \cap , \cup , $;$ of map *intersection*, map *union*, and map *composition*; and the monadic constructs $\overline{}$ and \smile of map *complementation* and *conversion*. (Further defined constructs, such as relational sum \dagger can be introduced, e.g. by putting $P \dagger Q =_{\text{Def}} \overline{\overline{P; Q}}$.) A *map expression* is any term built up from this signature in the usual manner. A *map equality* is a writing of the form $Q=R$, where both Q and R are map expressions.

Once a non-empty domain \mathcal{U} has been fixed, the map constants $\mathbf{0}$, $\mathbf{1}$, and ι are interpreted by putting: $\mathbf{0}^{\mathfrak{S}} =_{\text{Def}} \emptyset$, $\mathbf{1}^{\mathfrak{S}} =_{\text{Def}} \mathcal{U}^2 =_{\text{Def}} \mathcal{U} \times \mathcal{U}$, and $\iota^{\mathfrak{S}} =_{\text{Def}} \{[a, a] : a \in \mathcal{U}\}$. On the basis of the usual evaluation rules, by putting subsets $P_1^{\mathfrak{S}}, P_2^{\mathfrak{S}}, P_3^{\mathfrak{S}}, \dots$ of \mathcal{U}^2 in correspondence with the P_i s, each map expression P comes to designate a specific map $P^{\mathfrak{S}}$, and each equality $Q=R$ turns out to be either true or false.

The language \mathcal{L}^+ is a variant version of a first-order dyadic predicate language: an *atomic formula* of \mathcal{L}^+ has either the form xQy or the form $Q=R$, where x, y stand for individual variables (ranging over \mathcal{U}) and Q, R stand for map expressions. Here propositional connectives and existential/universal quantifiers are employed as usual. We assume known the notions of: syntax tree of a *well-formed expression* of \mathcal{L}^+ (in short *wfe*), literal, (immediate) subformulae of a given formula, sentence, and so on.⁴ Precise definitions can be found in [3, 4]. It is convenient to assume that the individual variables Var are arranged in a sequence $\langle \dots, \mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots \rangle$ whose two subsequences

$$Var^- = \langle \mathbf{x}_{-1}, \mathbf{x}_{-2}, \dots \rangle \quad \text{and} \quad Var^+ = \langle \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots \rangle$$

are used as repositories of bound and free variables in formulae of \mathcal{L}^+ , respectively. Variables indexed by an odd (resp., even) negative number will play the role of existentially (resp., universally) quantified variables.

1.1 Occurrences

A wfe E *occurs* within another wfe F at *position* ν , where ν is a node in the syntax tree \mathbb{T}_F for F , if the subtree of \mathbb{T}_F rooted at the node ν is identical to the syntax tree for E . In such a case, we refer to the node ν as an *occurrence* of E in F and to the path from the root of \mathbb{T}_F to ν as its *occurrence path*.

In what follows we show how an occurrence of E within F can be conveniently coded by a sequence of positive integers, representing the position, among its siblings, of each node in the occurrence path.

The set Pos of *positions* (of all nodes) in a wfe F can be defined as follows.

1. The empty word ε is in $Pos(F)$;
2. if F is an atomic formula xRy where $x, y \in Var$, then $Pos(F) = \{\varepsilon, 1, 2\}$;
3. if $F = \varphi_1 \wedge \dots \wedge \varphi_n$ or $F = \varphi_1 \vee \dots \vee \varphi_n$ and $\pi \in Pos(\varphi_i)$, for some $i \in \{1, \dots, n\}$, then $i.\pi \in Pos(F)$;

⁴ The definitions of syntax tree, occurrence, and position adopted in this paper are based on the connectives and quantifiers of predicate logic. Map expressions occurring in formulae of \mathcal{L}^+ are considered as meta-expressions and their internal structure is ignored.

4. if $F = \neg\psi$, or $F = (\forall x)\psi$, or $F = (\exists x)\psi$ and $\pi \in Pos(\psi)$, then $1.\pi \in Pos(F)$.

Given any wfe F , the occurrences of subformulae or subterms of F in F are determined as follows. We put $F|_\varepsilon = F$. In case F is an atomic formula xRy we put $F|_1 = x$ and $F|_2 = y$. If $F = \varphi_1 \circ \varphi_2$, with $\circ \in \{\wedge, \vee\}$, we put $F|_{i.\pi} = \varphi_i|_\pi$, for $i \in \{1, 2\}$. Finally, if $F = \neg\psi$, or $F = (\forall x)\psi$, or $F = (\exists x)\psi$, we put $F|_{1.\pi} = \psi|_\pi$. We indicate by P_E^F the collection of all positions $\pi \in Pos(F)$ such that $F|_\pi = E$. If $|P_E^F| = 1$, where $|\cdot|$ denotes the cardinality operator, we may use π_E^F to denote the position of the unique occurrence of E in F . We write P_E and π_E in case F is clear from the context. With $Lab(F, n)$ we indicate the symbol labeling the node of position n in the syntax tree Γ_F .

It is possible to establish a lexicographic ordering \prec over the set $Pos(\varphi)$ of positions in a formula φ . For any $\pi_1, \pi_2 \in Pos(\varphi)$ we put $\pi_1 \prec \pi_2$ if either

- $\pi_1 = \pi_2.\eta$, where η is a non-null word in \mathbb{N}^+ , or
- $\pi_1 = n_1 \dots n_i n_{i+1} \dots n_k$, $\pi_2 = n_1 \dots n_i n'_{i+1} \dots n'_j$, and $n_{i+1} < n'_{i+1}$.

Plainly, \prec is a well-ordering. Therefore, we can define an operation \min which selects from any non-empty subset X of $Pos(\varphi)$ its \prec -minimum.

An occurrence of a wfe E within a formula F is *positive* if its occurrence path deprived by its last node contains an even number of nodes labeled by the negation symbol \neg . Otherwise, the occurrence is said to be *negative*.

Let F be a wfe, π a position in F , and let E be a formula if $F|_\pi$ is a formula, and a term otherwise. We indicate with $F[\pi/E]$ the wfe obtained from F by replacing $F|_\pi$ with E at the node indicated by π .

Given two wfes E and F , we write $F = F(E)$ to stress that the occurrences of E in F play a significant rôle. Moreover, if E' is another formula, by $F(E')$ we denote the wfe resulting from F when each occurrence of E in F is replaced by a distinct copy of E' .

1.2 Uniform notation for formulae and relational expressions

For the sake of simplicity, we adopt Smullyan's unifying notation [10] to classify and decompose formulae of \mathcal{L}^+ . Hence, the formulae of the language are partitioned into four categories: conjunctive, disjunctive, universal, and existential formulae (called α -, β -, γ -, and δ -formulae, respectively). In particular, δ -formulae are those of the form $(\exists x)\varphi$ and $\neg(\forall x)\varphi$, whereas γ -formulae are those of the form $(\forall x)\varphi$ and $\neg(\exists x)\varphi$.

Given a δ -formula δ , the notation $\delta_0(x)$ will be used to denote the formula φ , if δ is of the form $(\exists x)\varphi$, or to denote the formula $\neg\varphi$, if δ is of the form $\neg(\forall x)\varphi$. We will refer to $\delta_0(x)$ as *the instance of δ* and to x as *the quantified variable of δ* . Likewise, for any γ -formula γ , $\gamma_0(x)$ denotes the formula φ or $\neg\varphi$, according to whether γ has the form $(\forall x)\varphi$ or $\neg(\exists x)\varphi$, respectively.

We allow generalized n -ary α - and β -formulae. To each of them, one can associate its components as shown in Table 1.

Map expressions occurring as atomic formulae of \mathcal{L}^+ , possess an internal structure. For the purpose of representing them with graphs, it is helpful to extend Smullyan's notation to relational constructs. By exploiting the following

α	α_1	\dots	α_n	β	β_1	\dots	β_2
$\varphi_1 \wedge \dots \wedge \varphi_n$	φ_1	\dots	φ_n	$\varphi_1 \vee \dots \vee \varphi_n$	φ_1	\dots	φ_1
$\neg(\varphi_1 \vee \dots \vee \varphi_n)$	$\neg\varphi_1$	\dots	$\neg\varphi_n$	$\neg(\varphi_1 \wedge \dots \wedge \varphi_n)$	$\neg\varphi_1$	\dots	$\neg\varphi_n$
$\neg\neg\varphi$	φ	$-$	$-$				

Table 1. α - and β -components

Conjunctive atoms,	$x\alpha y = xR \cap S y$	$x\alpha_1 y = xR y$	$x\alpha_2 y = xS y$	(\wedge)
α -atoms	$x\alpha y = x\overline{R \cup S} y$	$x\alpha_1 y = x\overline{R} y$	$x\alpha_2 y = x\overline{S} y$	$(\neg\vee)$
Disjunctive atoms,	$x\beta y = xR \cup S y$	$x\beta_1 y = xR y$	$x\beta_2 y = xS y$	(\vee)
β -atoms	$x\beta y = x\overline{R \cap S} y$	$x\beta_1 y = x\overline{R} y$	$x\beta_2 y = x\overline{S} y$	$(\neg\wedge)$
Atoms of type δ^α	$x\delta^\alpha y = x\overline{R}; S y$	$x\delta_0^{\alpha_1} z = xR z$	$z\delta_0^{\alpha_2} y = zS y$	$(\exists\wedge)$
	$x\delta^\alpha y = x\overline{R} \dagger S y$	$x\delta_0^{\alpha_1} z = x\overline{R} z$	$z\delta_0^{\alpha_2} y = z\overline{S} y$	$(\neg\forall\vee)$
	where z is existentially quantified ($\delta^\alpha \equiv (\exists z)\delta_0^\alpha(z) \equiv (\exists z)(\delta_0^{\alpha_1}(z) \wedge \delta_0^{\alpha_2}(z))$)			
Atoms of type γ^β	$x\gamma^\beta y = x\overline{R}; S y$	$x\gamma_0^{\beta_1} z = x\overline{R} z$	$z\gamma_0^{\beta_2} y = z\overline{S} y$	$(\neg\exists\wedge)$
	$x\gamma^\beta y = x\overline{R} \dagger S y$	$x\gamma_0^{\beta_1} z = xR z$	$z\gamma_0^{\beta_2} y = zS y$	$(\forall\vee)$
	where z is universally quantified ($\gamma^\beta \equiv (\forall z)\gamma_0^\beta(z) \equiv (\forall z)(\gamma_0^{\beta_1}(z) \vee \gamma_0^{\beta_2}(z))$)			
Atoms of type κ	$x\kappa y = x\overline{R} \smile y$	$y\kappa_1 x = yR x$		
	$x\kappa y = x\overline{R} \smile y$	$y\kappa_1 x = y\overline{R} x$		

Table 2. Classification of atomic formulae of \mathcal{L}^+

axiom schemata [11], we classify and decompose atomic formulae of \mathcal{L}^+ as shown in Table 2.

- (1) $(\forall x)(\forall y)(xA \cup By \equiv xAy \vee xBy)$
- (2) $(\forall x)(\forall y)(x\overline{A}y \equiv \neg xAy)$
- (3) $(\forall x)(\forall y)(xA; By \equiv (\exists z)(xAz \wedge zBy))$
- (4) $(\forall x)(\forall y)(xA \smile y \equiv yAx)$

2 Graphical representation of formulae of \mathcal{L}^+

In this section we extend the techniques of [1, 5, 2] for representing map expressions as well as identities of the form $P = \mathbf{1}$ and, more generally, formulae of \mathcal{L}^+ , by means of directed multigraphs. Our extension calls into play the negation connective (\neg) and the relational complement construct ($\overline{\quad}$), which lie well beyond the scope of the original proposals.

We will make use of *directed multigraphs* allowing multiple edges and self-loops. Let $G = (V, (E, m))$, $G_1 = (V_1, (E_1, m_1))$, \dots , $G_n = (V_n, (E_n, m_n))$ be directed multigraphs such that V, V_1, \dots, V_n are not empty, $V_1 \cup \dots \cup V_n = V$, $V_i \cap V_j = \emptyset$ when $i \neq j$, and $E_1 \cup \dots \cup E_n = E$. Then $\mathcal{S} = \{G_1, \dots, G_n\}$ is said to be a *partition of G*, and G_1, \dots, G_n are said to be the *components* of G . In case none of the components G_i admits a partition other than itself, \mathcal{S} is said to

be the *most refined partition of G into components*, and G_1, \dots, G_n are said to be the *most refined components* of G .

Let φ be a formula of \mathcal{L}^+ . We can assume φ constructed out of atomic formulae of the form xPy . In fact, any equality atom $\overline{Q} = R$ can be rewritten as $x\mathbf{1}; ((Q \cup R) \cap \overline{Q} \cap \overline{R}); \mathbf{1}y$. Negative literals in the form $x\overline{Q}y$ and free variables may occur in φ together with the (existentially and universally) quantified ones. Then φ can be represented by means of a directed multigraph $G_\varphi = (V_\varphi, (E_\varphi, m_\varphi))$ which is constructed in such a way that:

1. G_φ is provided with a function $lNode : V_\varphi \rightarrow \text{pow}(Var(\varphi))$ labeling its nodes, and with a function $lEdge : E_\varphi \rightarrow (\text{pow}(\mathcal{L}^\times) \cup \mathcal{L}^+)$ labeling its edges either with map expressions or with formulae of \mathcal{L}^+ devoid of quantifiers. The multiplicity function $m_\varphi : E_\varphi \rightarrow \mathbb{N}$ is defined $m_\varphi((u, v)) =_{\text{Def}} |lEdge((u, v))|$, for every $(u, v) \in E_\varphi$.
2. G_φ has at least one node, for every distinct variable in φ .
3. Nodes of G_φ are divided into two sets: the ones labeled with a subset of $Var(\varphi)$ containing at least one variable of Var^- (occurring bound in φ , existentially or universally quantified), called *bound nodes*, and the nodes labeled with subsets of $Var(\varphi)$ containing only variables of Var^+ (free in φ).

Let (u, v) be an element of E_φ . If $|lEdge((u, v))| > 1$, there are at least two edges from u to v in G_φ . Each of them is labeled with one of the relational expressions in $lEdge((u, v))$. If $|lEdge((u, v))| = 1$ the unique edge from u to v may be labeled either with a map expression or with a β -formula.

$lNode$ and $lEdge$ are defined so that for every occurrence of a literal xPy in φ , there is a subgraph of G_φ representing either P or \overline{P} . This subgraph is often constituted by an edge (u, v) such that $lNode(u) = \{x\}$, $lNode(v) = \{y\}$ and, either P or \overline{P} is an element of $lEdge((u, v))$. In case $lEdge((u, v))$ contains a non-atomic formula of \mathcal{L}^+ , $lNode(u)$ and $lNode(v)$ may have more than one element: $lNode(u)$ contains the left arguments of the atoms occurring in the formula labeling (u, v) , whereas $lNode(v)$ contains the right arguments.

Example 1. Let $\varphi_1 = x\overline{P} \cap Qy$. G_{φ_1} is the direct multigraph depicted in Fig. 1 having $V_{\varphi_1} = \{u_0, v_0\}$, $E_{\varphi_1} = \{(u_0, v_0)\}$, $m_{\varphi_1}((u_0, v_0)) = 2$, $lEdge((u_0, v_0)) = \{\overline{P}, Q\}$, $lNode(u_0) = \{x\}$ and $lNode(v_0) = \{y\}$.

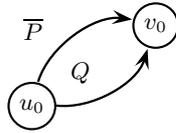


Fig. 1. The multigraph G_{φ_1} associated with $\varphi_1 = x\overline{P} \cap Qy$ (see Example 1).

Let G_i and G_j be two components of a multigraph G_φ . G_i is said to be *in relation \rightsquigarrow with G_j* , and we write $G_i \rightsquigarrow G_j$, if there is an edge (u, u') of G_i and

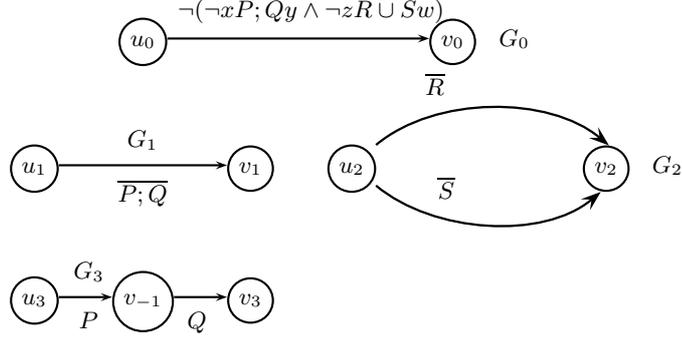


Fig. 2. The multigraph G_{φ_2} associated with $\varphi_2 = xP; Qy \vee zR \cup Sw$ (see Example 2).

two vertices v, v' in V_j such that $lNode(v) \subseteq lNode(u)$ and $lNode(v') \subseteq lNode(u')$, and if G_j represents a map expression (respectively, a formula of \mathcal{L}^+) that is a conjunct of the complement (the negation) of an element of $lEdge((u, u'))$.

The intuition behind the relation \rightsquigarrow is that edges labeled with a β -formula, a β -atom or with a γ^β -atom “call” a graph component representing the dual formula (an α -formula, an α -atom or a δ^α -atom).

Example 2. Let $\varphi_2 = xP; Qy \vee zR \cup Sw$. G_{φ_2} is the direct multigraph with components G_0, G_1, G_2 and G_3 illustrated in Fig. 2. The component G_0 is in relation \rightsquigarrow with both G_1 and G_2 , whereas G_1 is in relation \rightsquigarrow with G_3 . In particular, $lNode(u_0) = \{x, z\}$ includes both $lNode(u_1) = \{x\}$ and $lNode(u_2) = \{z\}$. Analogously, $lNode(v_0) = \{y, w\}$ includes both $lNode(v_1) = \{y\}$ and $lNode(v_2) = \{w\}$.

We say that a graph G_φ , representing a formula φ , is in *simple form* if all its edges, with the exception of the ones labeled with β -formulae, β -atoms and γ^β -atoms, are labeled with a map letter or with the complement of a map letter.

2.1 Graph transformation rules

In [5, 2] some meaning preserving rules of transformation for such graphs have been introduced. In the following we slightly modify them to treat expressions of the extended relational language.

1. An edge (ν, ν') with $lEdge((\nu, \nu')) = \{\mathbf{1}\}$ may be removed or created between nodes ν, ν' .
2. An edge (ν, ν') with $lEdge((\nu, \nu')) = \{P\}$ can be converted into an edge (ν', ν) with $lEdge((\nu', \nu)) = \{Q\}$, where either $P \equiv \overline{Q}$, or $Q \equiv \overline{P}$, or $P \equiv Q \equiv \iota$.
3. If there are two edges from ν to ν' labeled with the relational expressions α_1 and α_2 , respectively (that is $lEdge((\nu, \nu')) \supseteq \{\alpha_1, \alpha_2\}$), it is possible to replace them with a single edge from ν to ν' labeled with the relational expression $\alpha \equiv \alpha_1 \cap \alpha_2$, updating $lEdge((\nu, \nu'))$ to $(lEdge((\nu, \nu')) \setminus \{\alpha_1, \alpha_2\}) \cup \{\alpha\}$. The converse transformation is also possible.

4. Let (ν, ν') be an edge labeled with a formula φ such that $lNode(\nu) \varphi lNode(\nu')$ is either a β -atom or a γ^β -atom. Then, a component representing $lNode(\nu) \overline{\varphi} lNode(\nu')$ may be added or removed.
5. If there is an edge from ν to ν'' labeled with a relational expression δ^α (that is $lEdge((\nu, \nu'')) \supseteq \{\delta^\alpha\}$), it is possible to replace it with two edges (ν, ν') , (ν', ν'') , where ν' is a new bound node labeled with a new existentially quantified variable, $lEdge((\nu, \nu')) = \{\delta_1^\alpha\}$, $lEdge((\nu', \nu'')) = \{\delta_2^\alpha\}$, and $lEdge((\nu, \nu''))$ is updated to $lEdge((\nu, \nu'')) \setminus \{\delta^\alpha\}$. Conversely, if (ν, ν') , (ν', ν'') are the only edges involving the node ν' , $lEdge((\nu, \nu')) = \{\delta_1^\alpha\}$ and $lEdge((\nu', \nu'')) = \{\delta_2^\alpha\}$, these edges can be substituted with a single edge from ν to ν'' labeled with δ^α updating $lEdge((\nu, \nu''))$ to $lEdge((\nu, \nu'')) \cup \{\delta^\alpha\}$.
6. If $lEdge((\nu, \nu')) \supseteq \{\iota\}$, where either ν' is a bound node of degree 1 or $\nu' \equiv \nu$, then $lEdge((\nu, \nu'))$ may be updated to $lEdge((\nu, \nu')) \setminus \{\iota\}$. On the other hand, an edge (ν, ν') labeled with ι , where ν' is a new bound node or $\nu' \equiv \nu$, may be created, and $lEdge((\nu, \nu'))$ updated to $lEdge((\nu, \nu')) \cup \{\iota\}$.
7. An isolated node may be removed.

2.2 The graph fattening algorithm

We present now an algorithm to construct a multigraph G associated to a map expression P . This improves the one presented in [5, 2], unable to deal with map complementation.

Two nodes s_0 and s_1 (called source and sink) represent the two arguments of P (seen as atomic formula of \mathcal{L}^+), and every other distinct node with a different label will be considered bound. A multigraph G associated to a map expression has every node labeled with a singleton. Furthermore, every edge of G labeled with a β -atom or with a γ^β -atom is in relation \rightsquigarrow with one of the most refined components of G .

Given a map expression P , we proceed non-deterministically in the construction of G , s_0 , and s_1 , by selecting one of the following tactics:

- G consists of a single edge from s_0 to s_1 , such that $lEdge((s_0, s_1)) = \{P\}$;
- P is of type κ and G, s_1, s_0 (with source and sink exchanged) represents κ_1 ;
- P is of type δ^α , the disjoint graphs G', s_0, s'_2 and G'', s'_2, s_1 , represent $\delta_0^{\alpha_1}$ and $\delta_0^{\alpha_2}$, respectively. Then G, s_0, s_1 representing P is obtained from G' and G'' by ‘gluing’ together s'_2 and s'_2 to form a single node;
- P is of type α , the disjoint graphs G', s'_0, s'_1 and G'', s''_0, s''_1 , represent α_1 and α_2 , respectively. Then G is obtained from G' and from G'' by gluing s''_0 to s'_0 to form s_0 , and s''_1 to s'_1 to form s_1 ;
- P is either of type β or γ^β , the graph G', s_0, s_1 consists of a single edge from s_0 to s_1 such that $lEdge((s_0, s_1)) = \{P\}$, the graph G'', s''_0, s''_1 represents an atomic formula of type α or δ^α whose complement is equal to P , and $G' \rightsquigarrow G''$. Then G, s_0, s_1 , representing P , is the graph with components G' and G'' .

2.3 The graph thinning algorithm

Our aim in what follows is to determine, out of a given formula φ of \mathcal{L}^+ , an equivalent formula ψ of \mathcal{L}^+ devoid of quantifiers.

We will assume, without loss of generality, that distinct occurrences of quantifiers in φ bind different variables according to the classification of variables given in Section 1. Following our conventions about the subscripts of variable, no variable of Var^- can occur in the output formula ψ more than once. In case $\psi = xRy$, where $x, y \in Var^+$, we say that R is a map translation of φ .

This problem has been already analyzed in [5, 2], by designing an algorithm (called *graph thinning algorithm*) that seeks for a quantifier free formula of \mathcal{L}^+ logically equivalent to a given existentially quantified conjunction of literals of the form xPy . According to its original specification, such an algorithm contains as a preliminary step the construction of a labeled multigraph G_φ , its normalization (i.e. elimination of loop edges), the fusion of multiple edges, and the application, up to stabilization, of the rules of bypass and bigamy.

The extension of the graph thinning algorithm we introduce in this paper does no longer resort to a preliminary construction of G_φ . It transforms directly the input formula φ , by operating on its positions with the purpose of deriving a formula ψ of \mathcal{L}^+ devoid of quantifiers (that is, a formula each of whose variables belonging to Var^- , if any, occurs in it only once), while constructing the graph G_ψ associated with ψ . Notice that our algorithm may fail in this task. Namely it may happen that some variables in Var^- occur more than once in the output formula ψ . This does not mean that φ does not admit a quantifier free translation in \mathcal{L}^+ , it simply witnesses the incompleteness of the algorithm in solving a problem which is, in fact, undecidable.

Let us denote by \mathcal{P} the set of all the positions that have to be analyzed to derive the formula ψ and to construct the graph G_ψ . \mathcal{P} is initially set equal to $Pos(\varphi)$, the set of all the positions in φ . We construct a sequence of formulae $\psi^{(0)}, \psi^{(1)}, \dots$ and a sequence of multigraphs

$$G_\psi^{(0)} = (V_\psi^{(0)}, (E_\psi^{(0)}, m_\psi^{(0)})), G_\psi^{(1)} = (V_\psi^{(1)}, (E_\psi^{(1)}, m_\psi^{(1)})), \dots$$

such that for a certain index $k \in \mathbb{N}$, $\psi^{(k)} = \psi$ and $G_\psi^{(k)} = G_\psi$. In case variables of Var^- occur only once in ψ , ψ is a formula of \mathcal{L}^+ devoid of quantifiers logically equivalent to φ . The two sequences are constructed as follows. We put $\psi^{(0)} = \varphi$, whereas $G_\psi^{(0)}$ is the graph having $V_\psi^{(0)} = \{\}$ and $E_\psi^{(0)} = \{\}$. For $i = 0, 1, \dots$, $\psi^{(i+1)}$ is obtained from $\psi^{(i)}$ and $G_\psi^{(i+1)}$ is obtained from $G_\psi^{(i)}$ as follows. Extract the minimal position n from \mathcal{P} and distinguish the following cases.

1. If $\psi^{(i)}|_n$ is a variable, $\psi^{(i+1)} = \psi^{(i)}$ and $G_\psi^{(i+1)} = G_\psi^{(i)}$.
2. If $\psi^{(i)}|_n$ is an atomic formula xRy , we have $\psi^{(i+1)} = \psi^{(i)}$. Then, we construct the graph G_n , with distinguished nodes u_n, v_n , where $G_n = (E_n, V_n)$ with $E_n = \{(u_n, v_n)\}$ and $V_n = \{u_n, v_n\}$, by putting $lNode(u_n) = \{\psi^{(i)}|_{n.1}\}$, $lNode(v_n) = \{\psi^{(i)}|_{n.2}\}$ and $lEdge((u_n, v_n)) = \{R\}$. $G_\psi^{(i+1)}$ is obtained from $G_\psi^{(i)}$ by introducing in $G_\psi^{(i)}$ the component G_n , that is $V_\psi^{(i+1)} = V_\psi^{(i)} \cup V_n$ and $E_\psi^{(i+1)} = E_\psi^{(i)} \cup E_n$.

```

procedure Collapse( $n$ )
1.  $\Phi_n := \{j \in \mathbb{N} : j \in Pos(\psi^{(i)}|_n)\}$ ; // Positions of the conjuncts of  $Pos(\psi^{(i)}|_n$ 
2.  $m := |\Phi_n|$ ;
3. while ( $\Phi_n \neq \emptyset$ ) do
4.    $j := extractMin(\Phi_n)$ ;
5.   if (isAlpha( $\psi^{(i)}|_{n.j}$ )) then // if it is an  $\alpha$ -formula
6.      $\Psi_n := \{k \in \mathbb{N} : k \in Pos(\psi^{(i)}|_{n.j})\}$ ;
       //replace it in  $\psi^{(i)}|_n$  with its first conjunct
7.      $\psi^{(i)} := \psi^{(i)}[n.j/\psi^{(i)}|_{n.j.1}]$ ;
8.      $\Psi_n := \Psi_n \setminus \{1\}$ ;
9.      $G_\psi^{(i)} := G_\psi^{(i)} \setminus \{G_{n.j}\}$ ; // modify the multigraph
10.     $G_{n.j} := Rename(G_{n.j.1}, n.j)$ ;
11.     $G_\psi^{(i)} := G_\psi^{(i)} \cup \{G_{n.j}\}$ ;
12.    while ( $\Psi_n \neq \emptyset$ ) do
       // enlarge  $\psi^{(i)}|_n$  adding, as new conjuncts,
       // the other conjuncts of  $\psi^{(i)}|_{n.j}$ , and modify the multigraphs
13.       $k := extractMin(\Psi_n)$ ;
14.       $\psi^{(i)} := \psi^{(i)}[n.[(m-1)+k]/\psi^{(i)}|_{n.j.k}]$ ;
15.       $G_{n.k} := Rename(G_{n.j.k}, n.k)$ ;
16.       $G_\psi^{(i)} := (G_\psi^{(i)} \setminus G_{n.j.k}) \cup G_{n.k}$ ;
17.    endwhile;
18.  endif;
19. endwhile;
end procedure

```

Fig. 3. The procedure *Collapse*.

3. Let $\psi^{(i)}|_n = \chi_1 \wedge \dots \wedge \chi_k$, where $k > 1$ and each ψ_i may be an atomic formula, an α -formula or a β -formula. $\psi^{(i+1)}$ and $G_\psi^{(i+1)}$ are obtained as follows:
- For every χ_j which is a conjunction (α -formula), add all the conjuncts of χ_j directly to $\psi^{(i)}|_n$ (and rename the labels of the relative graphs and distinguished nodes according to the new positions). We call such an operation a *Collapse* of χ_j in $\psi^{(i)}|_n$ (cf., Fig. 3). The process of renaming the distinguished nodes of every component of $G_{n.j}$ (and of their sub-components) is performed by the recursive procedure *Rename* of Fig. 4. Once the procedure *Collapse* is applied to every conjunct of $\psi^{(i)}|_n$, $\psi^{(i)}|_n$ results in a conjunction of formulae χ_1, \dots, χ_m that may be atoms or β -formulae. The graphs $G_{n.1}, \dots, G_{n.m}$ representing χ_1, \dots, χ_m respectively, are components of $G_\psi^{(i)}$ and, by the ordering \prec of the set \mathcal{P} , have already been processed. G_n is the graph having as components $G_{n.1}, \dots, G_{n.m}$.
 - Proceed with the normalization step, consisting in eliminating edge loops from G_n and atoms with same left and right argument from $\psi^{(i)}|_n$. The normalization step is illustrated in Fig. 5 and works as follows. For every conjunct χ_j in $\psi^{(i)}|_n$ (line 1) that is an atom of type xRx (lines 4 and 5), substitute in $\psi^{(i)}|_n$ the occurrence of xRx with $x(R \cap \iota)x'$, where x' is a

```

procedure Rename( $G_n, k$ )
1. if ( $u_n, v_n \in V_n$ ) then
    //Rename the distinguished nodes of  $G_n$ 
2.    $u_n := u_k; v_n := v_k;$ 
3. endif;
4. if ( $\neg isAtom(\psi^{(i)}|_n)$ ) then
5.    $\Phi_n := \{j \in \mathbb{N} : j \in Pos(\psi^{(i)}|_n)\};$ 
6.   while ( $\Phi_n \neq \emptyset$ ) do
7.      $j := extractMin(\Phi_n);$ 
8.      $G_{k,j} := Rename(G_{n,j}, k.j);$ 
9.      $G_\psi^{(i)} := (G_\psi^{(i)} \setminus \{G_{n,j}\}) \cup G_{k,j};$ 
10.  endwhile;
11. endif;
end procedure

```

Fig. 4. The procedure *Rename*.

```

procedure Normalize( $n$ )
1.  $\Phi_n := \{j \in \mathbb{N} : j \in Pos(\psi^{(i)}|_n)\};$ 
2. while  $\Phi_n \neq \emptyset$  do
3.    $j := extractMin(\Phi_n);$ 
4.   if ( $isAtom(\psi^{(i)}|_{n.j})$ ) then
5.     if ( $\psi^{(i)}|_{n.j.1} = \psi^{(i)}|_{n.j.2}$ ) then
6.        $\psi^{(i)}|_{n.j.2} := newOddVar(\psi^{(i)});$ 
7.        $Lab(\psi^{(i)}, n.j) := Lab(\psi^{(i)}, n.j) \cap \iota;$ 
8.        $lEdge(u_{n.j}, v_{n.j}) := Lab(\psi^{(i)}, n.j);$ 
9.        $lNode(v_{n.j}) := \{\psi^{(i)}|_{n.j.2}\};$ 
10.    endif;
11.  endif;
12. endwhile;
end procedure

```

Fig. 5. The procedure of normalization.

new existentially quantified variable introduced by means of the function *newOddVar*. Transform the component $G_{n,j}$ of G_n (lines 8 and 9) into an edge labeled with $(R \cap \iota)$, whose nodes are labeled with $\{x\}$ and $\{x'\}$, respectively.

- Merge all the components of G_n , $G_{n,j}, G_{n,k}$ with $j \neq k$, having only one edge labeled with a map expression and such that $lNode(u_{n.j}) \cup lNode(v_{n.j}) = lNode(u_{n.k}) \cup lNode(v_{n.k})$, by combining them into a unique component labeled with the relational intersection of the map expressions labeling their edges. Such kind of components correspond in $\psi^{(i)}|_n$ to atomic formulae involving the same variables (as in xRy and xSy , for

```

procedure Fusion( $n$ )
1.  $\Phi_n := \{j \in \mathbb{N} : j \in Pos(\psi^{(i)}|_n)\}$ ;
2.  $m := 1$ ;
3.  $\chi := \Lambda$ ;
4. while  $\Phi_n \neq \emptyset$  do
5.    $j := extractMin(\Phi_n)$ ;
6.   if isAtom( $\psi^{(i)}|_{n,j}$ ) then
7.      $\bar{j} := n.j$ ;
8.      $Expr(\bar{j}) := Lab(\psi^{(i)}, \bar{j})$ ;
9.      $S_n := \Phi_n \setminus \{\bar{j}\}$ ;
10.    while  $S_n \neq \emptyset$  do
11.       $k := extractMin(S_n)$ ;
12.      if ( $Var(\psi^{(i)}|_{n,k}) = Var(\psi^{(i)}|_{\bar{j}})$ ) then
13.         $\bar{k} := n.k$ ;
14.        if ( $\psi^{(i)}|_{\bar{k}.1} = \psi^{(i)}|_{\bar{j}.1}$ ) then
15.           $Expr(\bar{j}) := Expr(\bar{j}) \cap Lab(\psi^{(i)}, \bar{k})$ ;
16.        else
17.           $Expr(\bar{j}) := Expr(\bar{j}) \cap Lab(\psi^{(i)}, \bar{k})^\smile$ ;
18.        endif;
19.         $lEdge((u_{\bar{j}}, v_{\bar{j}})) := \{Expr(\bar{j})\}$ ;
20.         $\Phi_n := \Phi_n \setminus \{k\}$ ;
21.         $G_n := G_n \setminus \{G_{\bar{k}}\}$ ;
22.      endif;
23.    endwhile;
24.     $\bar{m} := n.m$ ;
25.     $G_{\bar{m}} := Rename(G_{\bar{j}}, \bar{m})$ ;
26.     $G_n := (G_n \setminus \{G_{\bar{j}}\}) \cup \{G_{\bar{m}}\}$ ;
27.     $\chi_{\bar{m}} := lNode(u_{\bar{m}})Expr(\bar{j})lNode(v_{\bar{m}})$ ;
28.     $\chi := \chi \wedge \chi_{\bar{m}}$ ;
29.  else
30.     $\chi := \chi \wedge \psi^{(i)}|_{\bar{j}}$ ;
31.  endif;
32.   $m := m + 1$ ;
33. endwhile;
34.  $\psi^{(i)} := \psi^{(i)}[n/\chi]$ ;
end procedure

```

Fig. 6. The procedure of elimination of multiple edges.

instance). The fusion operation in $\psi^{(i)}|_n$ is performed by substituting the two atoms with an atomic formula having as a map expression the intersection of the relational expressions of the two atoms.

The *Fusion* procedure is depicted in Fig. 6. For every atomic conjunct of $\psi^{(i)}|_n$, the procedure searches within $\psi^{(i)}|_n$ for all the atoms sharing the same variables as arguments and merges them (lines 14-19). The formula resulting from the fusion process (line 34) is stored in the auxiliary for-

mula χ . Note that in the procedure χ is initialized, in line 3, to a “void formula” Λ . Such a Λ is not interpreted in any particular way. The only requirements it must satisfy are that $\neg\Lambda$, and $(\forall y)\Lambda$, and $(\exists y)\Lambda$ are to be considered as syntactic variations of Λ itself. Moreover, $\Lambda \oplus \chi$, $\chi \oplus \Lambda$ (where \oplus stands for any connective) is to be considered as a syntactic variations of χ , for any formula χ .

- Repeatedly apply the bypass rule and the bigamy rule to G_n and to $\psi^{(i)}|_n$ (Figures 7 and 8, respectively). Applying again *Normalize* and *Fusion* till stability is reached, that is, until $\psi^{(i)}|_n$ and the graph G_n cannot be modified anymore. Then $\psi^{(i+1)} = \psi^{(i)}$ and $G_{\psi}^{(i+1)} = G_{\psi}^{(i)}$.

For every atomic conjunct χ_j of $\psi^{(i)}|_n$, the *Bypass* procedure (Fig. 7) constructs the set $BoundVar(\psi^{(i)}|_{n,j})$ of all the variables in $Var(\psi^{(i)}|_{n,j}) \cap Var^-$ that occur exactly twice in $\psi^{(i)}$ and only in $\psi^{(i)}|_n$ (lines 4-7).

For every variable x in $BoundVar(\psi^{(i)}|_{n,j})$, let q be the position of the quantifier Qx binding x in $\psi^{(i)}$. Such position is uniquely determined since every quantifier binds a distinct variable.

If x is an existentially quantified variable (line 10), if $\psi^{(i)}|_n$ occurs positively (negatively) on the syntax tree with root the node labeled with Qx , and Qx is an existential quantifier (universal quantifier), the other conjunct of $\psi^{(i)}|_n$ having x as argument is determined, and a bypass operation performed (lines 14-34).⁵

For instance, let us consider formulae $\xi_1 = (\exists x)(yRx \wedge xQz)$ and $\xi_2 = (\forall x)\neg(yRx \wedge xQz)$. Suppose that ξ_1 occurs positively in a formula φ_1 , whereas ξ_2 negatively in a formula φ_2 (ξ_1 and ξ_2 occur uniquely in φ_1 and φ_2 because every quantifier in a formula binds a different variable). In both cases the subformula $yRx \wedge xQz$, is transformed by the Bypass procedure into $yR;Qz$ and the components of G_{n_1} and G_{n_2} (where G_{n_1} and G_{n_2} are the graphs associated to $yRx \wedge xQz$ in φ_1 and in φ_2 , respectively) representing yRx and xQz are combined into a multigraph representing $yR;Qz$ indexed by the position of the occurrence of $yR;Qz$ in φ_1 and φ_2 , respectively.

Note that in Fig. 7 we omitted the code corresponding to the case in which x is universally quantified. Actually, such a case can be treated similarly to the previous one. If $\psi^{(i)}|_n$ is a positive (negative) occurrence in the syntax tree having as root the node labeled with Qx , and Qx is an existential quantifier (universal quantifier), the other conjunct of $\psi^{(i)}|_n$ having x as argument is localized and the Bypass procedure executed.

Consider again ξ_1 and ξ_2 , but assuming that ξ_1 occurs negatively in a formula φ'_1 and ξ_2 positively in φ'_2 . In both cases the subformula $yRx \wedge xQz$, is transformed by the Bypass procedure into $y\overline{R} \dagger \overline{Q}z$. Let G'_{n_1} and G'_{n_2} be the graphs associated to $yRx \wedge xQz$ in φ_1 and in φ_2 , respectively. The components of G'_{n_1} and G'_{n_2} representing yRx and xQz are combined

⁵ Here the existential quantifier is intended as occurring positively in $\psi^{(i)}$, whereas the universal quantifier occurs negatively.

into a unique component representing $y\overline{R} \dagger \overline{Q}z$, indexed by the position of the occurrence of $y\overline{R} \dagger \overline{Q}z$ in φ_1 and φ_2 , respectively.

The *Bigamy* procedure is applied to every bound node of G_n with just one adjacent edge. In terms of $\psi^{(i)}|_n$, it applies to every variable x that is existentially quantified, in case $\psi^{(i)}|_n$ occurs positively in $\psi^{(i)}|_q$ (where q is the index of the position of Qx in $\psi^{(i)}$), or universally quantified, in case $\psi^{(i)}|_n$ is a negative occurrence of $\psi^{(i)}|_q$, that occurs only once in ψ^i . Again we provide a simplified procedure: Fig. 8 shows a procedure dealing only with the case in which x is an existentially quantified variable occurring in $\psi^{(i)}|_n$ as a left arguments of an atomic formula. The other cases are treated in a similar way.

4. If $\psi^{(i)}|_n = (\exists x)\chi$ then $\psi^{(i)}|_n = \psi^{(i)}|_{n.1}$, $\psi^{(i+1)} = \psi^{(i)}$ and $G_\psi^{(i+1)} = (G_\psi^{(i)} \setminus \{G_{n.1}\}) \cup \{G_n\}$, where G_n is obtained from $G_{n.1}$ by calling the procedure *Rename*($G_{n.1}, n$).
5. If $\psi^{(i)}|_n = \neg\chi$, we can distinguish the following cases:
 - If χ is an atomic formula xRy , the negation in front of χ is removed and the map expression R is complemented: χ is set equal to $x\overline{R}y$, and $\psi^{(i)}|_n$ to χ . Then we put $\psi^{(i+1)} = \psi^{(i)}$.
If R is a map letter, the graph G_n is obtained from $G_{n.1}$ calling the procedure *Rename*($G_{n.1}, n$), and $G_\psi^{(i+1)} = (G_\psi^{(i)} \setminus \{G_{n.1}\}) \cup \{G_n\}$. If R is a compound map expression not complemented, the graph G_n with $E_n = \{(u_n, v_n)\}$, $V_n = \{u_n, v_n\}$, $lNode(u_n) = \{x\}$, $lNode(v_n) = \{y\}$, and $lEdge((u_n, v_n)) = \{\overline{R}\}$ is constructed such that $G_n \rightsquigarrow G_{n.1}$, where $G_{n.1}$ represents xRy . Then we put $G_\psi^{(i+1)} = G_\psi^{(i)} \cup \{G_n\}$.
 - If χ is a conjunction, $\psi^{(i+1)} = \psi^{(i)}$ and $G_\psi^{(i+1)} = G_\psi^{(i)} \cup \{G_n\}$, where G_n is obtained as follows. $V_n = \{u_n, v_n\}$ and $E_n = \{(u_n, v_n)\}$, where u_n and v_n are nodes new to $G^{(i)}$, $lNode(u_n) = \{\varphi|_{i.1} : i \in \Phi_{n.1}\}$ and $lNode(v_n) = \{\varphi|_{i.2} : i \in \Phi_{n.1}\}$, with $\Phi_{n.1} = \{i \in \mathbb{N}^+ : \varphi|_{n.1.i} \in Pos(\varphi|_{n.1})\}$, and $lEdge((u_n, v_n)) = \{\overline{\varphi|_{n.1}}\}$. Moreover, we put $G_n \rightsquigarrow G_{n.1}$.
 - If χ is a complemented atomic formula or a negated formula (a double negation), $\psi^{(i)}|_n = \psi^{(i)}|_{n.1.1}$, and $\psi^{(i+1)} = \psi^{(i)}$. $G_\psi^{(i+1)} = (G_\psi^{(i)} \setminus \{G_{n.1}, G_{n.1.1}\}) \cup \{G_n\}$ where G_n is obtained from $G_{n.1.1}$ by calling the procedure *Rename*($G_{n.1.1}, n$).
6. If $\psi^{(i)}|_n = \chi_1 \vee \dots \vee \chi_k$, we put $\psi^{(i)}|_n = \neg(\neg\chi_1 \wedge \dots \wedge \neg\chi_k)$ and $\mathcal{P} := \mathcal{P} \cup \{n, n.1, n.1.1, \dots, n.1.k\}$. Now the minimal position to be processed is $n.1.1$ (a negation, thus falling in the previous cases).
7. If $\psi^{(i)}|_n = (\forall x)\chi$, we put $\psi|_n = \neg(\exists x)\neg\chi$ and $\mathcal{P} := \mathcal{P} \cup \{n, n.1, n.1.1, n.1.1\}$. The minimal position to be processed is $m = n.1.1$, $\psi^{(i)}|_m$ is a negation and therefore is treated as outlined in case 5.

Example 3. We illustrate the graph thinning algorithm through an example. Let us consider the formula $\varphi = (\forall x_{-2})((x_1Ex_{-2} \wedge x_{-2}Ex_2) \rightarrow x_1Ex_2)$ of \mathcal{L}^+ . Put $\psi^{(0)} = \varphi$, whereas $G_\psi^{(0)}$ is the multigraph with $V_\psi^{(0)} = \{\}$ and $E_\psi^{(0)} = \{\}$. At first, the positions of φ corresponding to the atomic formulae x_1Ex_{-2} and $x_{-2}Ex_2$, i.e., 111 and 112, are considered. As a result, $\psi^{(2)} = \psi^{(1)} = \psi^{(0)}$, and $G_\psi^{(2)}$ is the direct

```

procedure Bypass( $n$ )
1.  $\Phi_n := \{j \in \mathbb{N} : j \in Pos(\psi^{(i)}|_n)\}$ ;
2.  $m := 1$ ;
3.  $\chi := \Lambda$ ;
4. while  $\Phi_n \neq \emptyset$  do
5.    $l := extractMin(\Phi_n)$ ;
6.   if isAtom( $\psi^{(i)}|_{n,l}$ ) then
7.      $BoundVar(\psi^{(i)}|_{n,j}) := \{x \in Var(\psi^{(i)}|_{n,j}) \cap Var^- : |P_x^{\psi^{(i)}}| = |P_x^{\psi^{(i)}|_n| = 2\}$ ;
8.     for every  $x \in BoundVar(\psi^{(i)}|_{n,j})$  do
9.        $q := \pi_{Qx}$ ; //  $q$  is the position of the quantifier binding  $x$ 
10.      if  $(\exists j \in \mathbb{N} : x = x_{-(2j+1)})$  then
11.        if  $((OccPos(\psi^{(i)}|_n, q) \wedge isExist(Qx)) \vee$ 
12.           $(\neg(OccPos(\psi^{(i)}|_n, q) \wedge \neg isExist(Qx))))$  then
13.           $p := extractMax(P_x^{\psi^{(i)}|_n})$ ;
14.           $\bar{k} := Father(p)$ ; // if one atom is of type  $yRx$ 
15.          if  $(x = \psi^{(i)}|_{n,j,2})$  then
16.            if  $(p = \bar{k} \cdot 1)$  then // and the other one of type  $xSz$ 
17.               $V_{\bar{l}} := V_{\bar{l}} \cup (V_{\bar{k}} \setminus \{v_{\bar{k}}\})$ ;
18.               $E_{\bar{l}} := ((E_{\bar{l}} \cup E_{\bar{k}})[v_{\bar{l}}/u_{\bar{k}}])[v_{\bar{k}}/v_{\bar{l}}]$ ;
19.               $Expr(\bar{l}) := Expr(\bar{l}); Lab(\psi^{(i)}|_{\bar{k}})$ ;
20.            else // if the other one is of type  $zSx$ 
21.               $V_{\bar{l}} := V_{\bar{l}} \cup (V_{\bar{k}} \setminus \{u_{\bar{k}}\})$ ;
22.               $E_{\bar{l}} := ((E_{\bar{l}} \cup E_{\bar{k}})[v_{\bar{l}}/v_{\bar{k}}])[u_{\bar{k}}/v_{\bar{l}}]$ ;
23.               $Expr(\bar{l}) := Expr(\bar{l}); Lab(\psi^{(i)}|_{\bar{k}})^{\smile}$ ;
24.            endif;
25.          else // if one atom is of type  $xRy$ 
26.            if  $(p = \bar{k} \cdot 1)$  then // and the other one of type  $xSz$ 
27.               $V_{\bar{l}} := V_{\bar{l}} \cup (V_{\bar{k}} \setminus \{v_{\bar{k}}\})$ ;
28.               $E_{\bar{l}} := ((E_{\bar{l}} \cup E_{\bar{k}})[u_{\bar{l}}/u_{\bar{k}}])[v_{\bar{l}}/u_{\bar{l}}][v_{\bar{k}}/v_{\bar{l}}]$ ;
29.               $Expr(\bar{l}) := Expr(\bar{l})^{\smile}; Lab(\psi^{(i)}|_{\bar{k}})$ ;
30.            else // if the other one is of type  $zSx$ 
31.               $V_{\bar{l}} := V_{\bar{l}} \cup (V_{\bar{k}} \setminus \{u_{\bar{k}}\})$ ;
32.               $E_{\bar{l}} := ((E_{\bar{l}} \cup E_{\bar{k}})[u_{\bar{l}}/v_{\bar{k}}])[v_{\bar{l}}/u_{\bar{l}}][u_{\bar{k}}/v_{\bar{l}}]$ ;
33.               $Expr(\bar{l}) := Expr(\bar{l})^{\smile}; Lab(\psi^{(i)}|_{\bar{k}})^{\smile}$ ;
34.            endif;
35.          endif;
36.           $G_n := G_n \setminus \{G_{\bar{k}}\}$ ;
37.           $\Phi_n := \Phi_n \setminus \{\bar{k}\}$ ;
38.           $G_{\bar{m}} := Rename(G_{\bar{l}}, \bar{m})$ ;
39.           $G_n := (G_n \setminus \{G_{\bar{l}}\}) \cup \{G_{\bar{m}}\}$ ;
40.           $\chi_{\bar{m}} := lNode(u_{\bar{m}})Expr(\bar{l})lNode(v_{\bar{m}})$ ;
41.           $\chi := \chi \wedge \chi_{\bar{m}}$ ;
42.        else
43.           $\chi := \chi \wedge \psi^{(i)}|_{\bar{l}}$ ;
44.        endif;
45.      endfor;
46.    endif;
47.     $m := m + 1$ ;
48.  endwhile;
end procedure

```

Fig. 7. The procedure of bypass.

multigraph with components G_{111} and G_{112} , representing x_1Ex_{-2} and $x_{-2}Ex_2$, respectively. Now, the α -formula $\psi^{(2)}|_{11} = x_1Ex_{-2} \wedge x_{-2}Ex_2$ is treated, and the *Bypass* procedure is employed. The universally quantified variable x_{-2} occurs twice in $\psi^{(2)}|_{11}$ and does not occur anywhere else in $\psi^{(2)}$. Moreover, $\psi^{(2)}|_{11}$ occurs negatively on the syntactic tree having as root the node labeled with $(\forall x_{-2})$.

```

procedure Bigamy( $n$ )
1.  $\Phi_n := \{j \in \mathbb{N} : j \in \text{Pos}(\psi^{(i)}|_n)\}$ ;
2.  $m := |\Phi_n|$ ;
3.  $\chi := \Lambda$ ;
4. while  $\Phi_n \neq \emptyset$  do
5.    $l := \text{extractMin}(\Phi_n)$ ;
6.   if  $\text{isAtom}(\psi^{(i)}|_{n.l})$  then
7.      $\text{BoundVarSingle}(\psi^{(i)}|_{n.j}) := \{x \in \text{Var}(\psi^{(i)}|_{n.j}) \cap \text{Var}^- : |\mathcal{P}_x^{\psi^{(i)}}| = |\mathcal{P}_x^{\psi^{(i)}|_n}| = 1\}$ ;
8.     for every  $x \in \text{BoundVarSingle}(\psi^{(i)}|_{n.j})$  do
9.        $q := \pi_{Qx}$ ; //  $q$  is the position of the quantifier binding  $x$ 
10.      if  $(\exists j \in \mathbb{N} : x = x_{-(2j+1)}) \wedge (\text{OccPos}(\psi^{(i)}|_n, q))$  then
11.         $p := \text{extractMax}(\mathcal{P}_x^{\psi^{(i)}|_n})$ ;
12.         $\bar{k} := \text{Father}(p)$ ;
13.        if  $(p = \bar{k}.1 \wedge \exists d \in \Phi_n : \text{isAtom}(d) \wedge (\psi^{(i)}|_{\bar{k}.2} = \psi^{(i)}|_{d.1} \vee \psi^{(i)}|_{\bar{k}.2} = \psi^{(i)}|_{d.2}))$  then
14.           $q := m + 1$ ;
15.           $V_q := \{u_q, v_q\}$ ;  $E_q := \{(u_q, v_q)\}$ ;
16.          if  $(\psi^{(i)}|_{\bar{k}.2} = \psi^{(i)}|_{d.1})$  then
17.             $\psi^{(i)}|_n := \psi^{(i)}|_n \wedge \psi^{(i)}|_{\bar{k}.1} \mathbf{1} \psi^{(i)}|_{d.2}$ ;
18.             $lNode(u_q) := \psi^{(i)}|_{\bar{k}.1}$ ;  $lNode(v_q) := \psi^{(i)}|_{d.2}$ ;
19.          elseif  $(\psi^{(i)}|_{\bar{k}.2} = \psi^{(i)}|_{d.2})$  then
20.             $\psi^{(i)}|_n := \psi^{(i)}|_n \wedge \psi^{(i)}|_{\bar{k}.1} \mathbf{1} \psi^{(i)}|_{d.1}$ ;
21.             $lNode(u_q) := \psi^{(i)}|_{\bar{k}.1}$ ;  $lNode(v_q) := \psi^{(i)}|_{d.1}$ ;
22.          endif;
23.           $lEdge((u_q, v_q)) := \mathbf{1}$ ;
24.           $G_n := G_n \cup \{G_q\}$ ;
25.           $m := m + 1$ ;
26.        endif;
27.      endif;
28.    endfor;
29.  endif;
30. endwhile;
end procedure

```

Fig. 8. The procedure *Bigamy*.

Thus, by applying the *Bypass* rule, we put $\psi^{(2)}|_{11} = x_1 \bar{E} \dagger \bar{E} x_2$, and $\psi^{(3)} = \psi^{(2)}$. $G_\psi^{(3)}$ is the multigraph with components G_{11} and G_{111} , where G_{11} has only one edge labeled with $x_1 \bar{E} \dagger \bar{E} x_2$, G_{111} represents $x_1 E; E x_2$, and $G_{11} \rightsquigarrow G_{111}$. Next, $\psi^{(3)}|_{12} = x_1 E x_2$ is processed. We put $\psi^{(4)} = \psi^{(3)}$ and $G_\psi^{(4)} = G_\psi^{(3)} \cup \{G_{12}\}$, where G_{12} is the multigraph representing $x_1 E x_2$. At this point, position 1 is considered: $\psi^{(4)}|_1 = x_1 \bar{E} \dagger \bar{E} x_2 \rightarrow x_1 E x_2$, can be rewritten as $\neg x_1 \bar{E} \dagger \bar{E} x_2 \vee x_1 E x_2$. Moreover, we put $\psi^{(4)}|_1 = \neg(\neg \neg x_1 \bar{E} \dagger \bar{E} x_2 \wedge \neg x_1 E x_2)$, $\mathcal{P} = \mathcal{P} \cup \{11, 111, 112, 1111\}$, and $G_\psi^{(5)} = (G_\psi^{(4)} \setminus \{G_{11}, G_{12}\}) \cup \{G_{112}, G_{1111}\}$, where G_{112} and G_{1111} are obtained from G_{12} and G_{11} , respectively, calling the *Rename* procedure. We proceed by applying three times the step 5 of the thinning algorithm (negation). This yields $\psi^{(8)}|_{11} = x_1 \bar{E} \dagger \bar{E} x_2 \wedge x_1 \bar{E} x_2$, whereas $G_\psi^{(8)}$ is the multigraph with components G_{111} , G_{112} , and G_{1111} where $G_{111} \rightsquigarrow G_{1111}$. Next, we apply the *Fusion* procedure to $\psi^{(8)}|_{11}$, put $\psi^{(8)}|_{11} = x_1 (\bar{E} \dagger \bar{E}) \cap \bar{E} x_2$, and then $\psi^{(9)} = \psi^{(8)}$. $G_\psi^{(9)}$ is the multigraph with components G_{11} and G_{111} such that G_{11} represents $\psi^{(9)}|_{11}$, and G_{111} $x_1 E; E x_2$. Now, formula $\psi^{(9)}|_1 = \neg(x_1 (\bar{E} \dagger \bar{E}) \cap \bar{E} x_2)$ is processed to obtain $\psi^{(10)}|_1 = x_1 (\bar{E} \dagger \bar{E}) \cap \bar{E} x_2$. The multigraph $G_\psi^{(10)}$ consists of the components G_1 ,

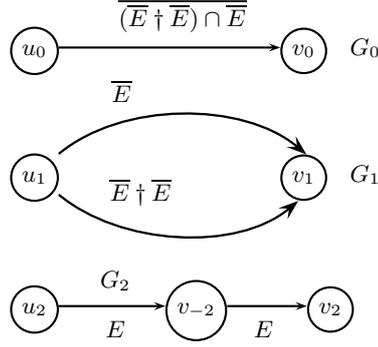


Fig. 9. The multigraph G_ψ associated with $\psi = \mathbf{x}_1(\overline{E \dagger E}) \cap \overline{E} \mathbf{x}_2$ (see Example 3).

G_{11} and G_{111} . In particular, G_1 has only one edge labeled with $\psi^{(10)}|_1$, G_{11} represents $\mathbf{x}_1(\overline{E \dagger E}) \cap \overline{E} \mathbf{x}_2$, G_{111} represents $\mathbf{x}_1 E; E \mathbf{x}_2$, and $G_1 \rightsquigarrow G_{11} \rightsquigarrow G_{111}$. Finally, we have $\psi^{(10)} = (\forall x_{-2}) \mathbf{x}_1(\overline{E \dagger E}) \cap \overline{E} \mathbf{x}_2$. Further steps involving cases 7, 5, and 4 of the thinning algorithm, allow us to eliminate the quantifier $(\forall x_{-2})$. Hence, we obtain the map translation of φ , $\psi = \mathbf{x}_1(\overline{E \dagger E}) \cap \overline{E} \mathbf{x}_2$. The corresponding multigraph is depicted in Fig. 9.

Conclusion and Future Work

We have enhanced preexisting algorithms for translating dyadic first-order logic into map calculus which rely on a specific graph representation of map expressions. As an outcome, we are able to deal with map expressions and with formulae containing the relational complement construct and the negation connective.

The first algorithm, called graph fattening algorithm, associates a graph with a given map expression. To do this, it decomposes the expression by a Smullyan-like unifying notation for map expressions. When applied to a map expression, it provides a representation which better conveys its meaning. The second algorithm, called graph thinning algorithm, translates formulae of dyadic first-order logic into map expressions. It works bottom-up (w.r.t. the structure of the formula), by building a graph which represents the output formula, and by translating the input formula “in place”.

We are currently working on the proofs of termination and confluence of the latter algorithm. Confluence is an important property of the graph thinning algorithm. In fact it states that if the algorithm succeeds (or, respectively, fails) in the translation of a given formula φ when a certain sequence of steps is chosen by the nondeterministic parts of the algorithm (e.g., the part treating α -formulae), then any alternative sequence of steps will likewise lead to success (resp., failure) the translation of φ .

Moreover, we plan to enhance the graph thinning algorithm by improving the bypass and bigamy rules (making them more liberal in case of nesting of

quantifiers) and integrating to the algorithm the rules exploiting the functionality information (like for instance that an expression P labeling an edge is single-valued) introduced in [2].

A first attempt in the implementation of a proof-assistant based on the graph representation of map expressions has been done in [7, 8]. In that case the algorithms described in [2, 5] have been implemented on top of the attributed graph transformation system AGG. In order to validate the approach we described in this paper, an implementation of the new algorithms is due (as a stand alone tool or in integration with standard theorem provers for first-order logic). Experimentation with such a tool has to be performed to validate the approach. These are interesting and challenging themes for further research.

Acknowledgements. We would like to thank the anonymous referees for their useful suggestions. This research was partially supported by PRIN project 2006/07 “Large-scale development of certified mathematical proofs” and by GNCS.

References

- [1] Cantone, D., Cavarra, A., and Omodeo, E. G. (1997). On existentially quantified conjunctions of atomic formulae of \mathcal{L}^+ . In M. P. Bonacina and U. Furbach, eds, *Proc. of FTP97*.
- [2] Cantone, D., Formisano, A., Omodeo, E. G., and Zarba, C. G. (2003) Compiling dyadic first-order specifications into map algebra. *TCS* 293(2):447-475.
- [3] Dershowitz, N. and Jouannaud, J.-P. (1990). Rewrite systems. In *Handbook of Theoretical Computer Science, Vol. B.: Formal Models and Semantics (B)*, pp.243-320.
- [4] Fitting, M. C. (1996). *First-order Logic and Automated Theorem Proving*, Graduate Texts in Computer Science. Springer-Verlag, New York, 2nd edition.
- [5] Formisano, A., Omodeo, E. G., and Simeoni, M. (2001). A graphical approach to relational reasoning. In W. Kahl, D. L. Parnas, and G. Schmidt, eds., *Proc. of RelMiS 2001 (ETAPS 2001)*. ENTCS, 44(3), Elsevier Science.
- [6] Formisano, A., Omodeo, E. G., and Temperini, M. (2000). Goals and benchmarks for automated map reasoning. *Journal of Symbolic Computation*, 29(2):259-297.
- [7] Formisano, A. and M. Simeoni, *Graphs and maps: rewriting techniques at work*, Tech. Rep. TU-Berlin 2001-01, Technische Universität Berlin, (2001).
- [8] Formisano, A., and Simeoni, M. (2001). An AGG application supporting visual reasoning. In L. Baresi and M. Pezzè, eds., *Proc. of GT-VMT'01 (ICALP 2001)*. ENTCS, 50(3), Elsevier Science.
- [9] Kwatinetz, M. K. (1981) *Problems of expressibility in finite languages*. PhD thesis, University of California, Berkeley.
- [10] Smullyan, R. M. (1995) *First-order Logic*, Dover Publications, New York.
- [11] Tarski, A. and Givant, S. (1987). *A formalization of Set Theory without variables*, vol. 41 of Colloquium Publications. American Mathematical Society.

Semantically Augmented DCG Analysis for Next-generation Search Engine

Stefania Costantini and Alessio Paolucci

Computer Science Dept., Univ. of L'Aquila
Via Vetoio Loc. Coppito, I-67010 L'Aquila (Italy)
E-mail: stefcost@di.univaq.it, alessiopaolucci@ieee.org

Abstract. In this paper, we describe an extension to the well-known DCG's (Definite Clause Grammars) to allow for semantic analysis, and generate semantically-based description of the sentence at hand. This approach has been the basis for the development of a new search engine called Mnemosine, which is able to interact with a user in natural language and to provide contextual answer at different levels of detail. Mnemosine has been fully implemented and has been applied to a practical case-study, i.e., to the Wikipedia Web pages. The Mnemosine system, though still a prototype, exhibits features which are more advanced than those of the (few) existing competitors and thus represents a successful proof-of-concept for the proposed approach.

1 Introduction

Natural language processing (NLP) is an area of interdisciplinary research that spans from artificial intelligence [8,10] to linguistics [5]. Its objective is to develop formal models and representations of natural language in order to allow automatic processing by a machine.

Up to now, despite several theoretical studies, methodologies and proposed technologies, implementing systems based on natural language processing is a critical activity where the quality of the results is far from granted. The increasing request for solutions based on natural language makes this research area particularly active. This work is part of this effort, and proposes a new approach to natural language processing. The current goal is to improve the quality of semantic search engines results, while future developments are aimed at introducing a new perspective in a wider class of applications.

Any approach to NLP involves, either explicitly or implicitly, a comparison with the dynamics of the functional biological machine that is the source of the language: the human brain. In fact, the proposed approach draws its inspiration from the dynamics that are supposed to be at the basis of communication processes that are conveyed through language rather than from work in engineering solutions for automatic processing of language (often based on tools inherited from the theory of formal languages). The ultimate goal is to achieve improved quality of NLP results by applying, in the context of Computer Science, an operational model elicited through a process of generalization and abstraction of the underlying biological structures. As a first result, we have defined an approach to syntactic-semantic fully informed analysis. The proposed model is called “syntactic-semantic” because, similarly to processes related to the human language elaboration, syntactic and semantic analysis interact in both serial and parallel way. The analysis is called “fully informed” because it is able to exploit pieces of information which are either extracted or inferred from an underlying Knowledge Base. In particular, we have defined an extension to classical DCG’s called SE-DCG’s for “Semantically Enhanced DCG’s”.

For the validation and evaluation of the proposed computational model we have developed Mnemosine, a “semantic search engine”. The application is a proof-of-concept of the developed theory. Users can submit questions, or more generally queries, in natural language. Mnemosine is able to return one or more answers formulated in natural language. It may also return references to web pages where the user may presumably find an answer to the query, and also references to pages that contain conceptually equivalent answers. A main feature of Mnemosine is that of identifying and indicating the possible contexts in which the query can find a different plausible interpretation: thus, the user can restrict results to the preferred context.

The Mnemosine system constitutes a step ahead, as similar features cannot be found in state-of-the-art competitor systems.

The plan of the paper is as follows: in Section 2 we illustrate the biological background our approach is based upon. In Section 3 we recall some well-known pitfalls of existing NLP approaches. In Section 4 we recall some concepts about linguistic analysis. In Section 5 we present SE_DCG’s. In Section 6 we present Mnemosine, and in Section 7 we compare it to related work. Finally we conclude.

2 Neuro-Biological Background

Though we admit that the ultimate goal of our research is hard to achieve, we illustrate in a nutshell the result of our investigation concerning work in the field of neuroscience, with particular attention to studies related to high-level cerebral functions related to the neocortex.

This investigation was aimed at rooting (at least in a tentative way) our approach and techniques into the best-consolidated theories that identify the functional areas of the brain in which the language is processed and try to elicit their role and interconnections (see the Web site of the Max Planck Institute for Human Cognitive and Brain Science (MPI CBS) - http://www.cbs.mpg.de/MPI_Base/NEU/home and [9]). In these studies, localization of functional areas has been obtained by means of functional magnetizing resonance imaging (fMRI).

The language processing temporal profile has been studied by means of a neuroimage technique with increased temporal resolution by the registration of event-related potential through the electroencephalography (EEG). The results show that the language functions are not the outcome of few well-defined areas (such as Broca's and Wernicke's areas): rather, language production and analysis result from the activity of a large part of the neocortex. Actually, language sentences activate the so-called Wernicke area for first, but the interested sub-areas can be the front, rear or median part. The front part of the Wernicke area appears to be especially related to syntactical and grammatical aspects, while the median part seems to deal with representations of the meanings. The rear area is most presumably involved in both aspects. This area acts as a catalyst to enable the synthesis of meaning and grammar: in fact, the interpretation of sentences implies a unified composition of both syntactic contents and meaning. A section of the temporal lobe and a section of the frontal lobe cooperate, respectively, to semantic and syntactic analysis, thus forming a mini-network.

The "data flow" aspect of these processes, analyzed by electroencephalography (EEG) has been related to changes in the voltage synchronized to the presence of stimuli. From the calculation of the average of EEG segments related to stimulation, wave forms (called components) are obtained, that can take either a positive or a negative value. The language processing temporal profile is extremely complex. The study of the components (waves) in summary shows that the brain proceeds first by analyzing the syntactical and grammatical structure, then proceeds with semantic analysis that includes numerous sub-processing activities related to syntactic aspects. Most presumably, in the "assembly line" of cortical

elaboration related to language processing the brain proceeds in part sequentially and in part by integrating in parallel the syntactical and semantic aspects.

Language is indeed not a stand-alone function: rather, it is related to other high-level cognitive functions. Thus, the various cortical areas in which certain kinds of language processing occur are interconnected to other neuronal groups which in turn constitute areas of operation for specific aspects of other high-level cognitive functions. In particular, memory has a very important role. Memory is the more complex and interesting functional system with which the neurobiological mechanisms underlying the development of language must cooperate and relate to. Even syntactic features require a semantic interpretation, that is based on extracting information from a “corpus” of knowledge that functionally belongs to long-term memory. Short-term memory also has a key role in the understanding of speech and text as a whole. Information extracted and deduced from the context and information extracted from previously-examined propositions are necessary for the understanding of the meaning of propositions currently under consideration. As discussed in the following, our approach introduces mechanisms in principle analogous to human language processing and memory.

In particular, the sequential and parallel components of syntactic and semantic analysis are emulated by the SE-DCGs, that we propose in this paper: this extension to classical DCG grammars introduces in fact the possibility of operating simultaneously on syntactical and semantic aspects, similarly to what happens in the human brain. For example, syntactic or morphological information needed for the understanding (semantic analysis) of a proposition are retrieved via SE-DCG embedded goals. SE-DCG embedded goals can be used e.g. for a proper semantic interpretation or for ambiguity resolution. Previous extracted knowledge can be usefully exploited. In the next sections we will show how these improvements may help overcome some problems (such as semantic ambiguity) that are not easily managed by using traditional NLP approaches based on sequential analysis only and with no memory use.

3 Pitfalls of classical NLP

Classical NLP solutions [2, 5, 6] suffer from functional deficiencies in both syntactic and semantic analysis. Consider for instance the sentence

“Le fontane in ghisa sono indistruttibili”

It means that fountains made in “ghisa”, which is a very resistant material (cast iron), are indestructible. A syntactical profile analysis would return something like the following, where “articolo” means “article”, “nome” means “noun”, “preposizione” means “preposition”, “verbo” means “verb” and “aggettivo” means “adjective”.

```
le -> Articolo
fontane -> Nome
in -> Preposizione
ghisa -> Nome
sono -> Verbo
indistruttibili -> Aggettivo
```

Concerning the semantic interpretation, while it is relatively easy to extrapolate that “fontane” (“fountains”) are objects which are the subject of a proposition, the semantic interpretation of “in ghisa” is conceptually much more complex. If for a person the solution is presumably trivial, for the machine things are different. Without further information, for the machine the interpretations can be at least two: “ghisa” could be seen as the place where the fountains are indestructible, or alternatively, “ghisa” can be given its proper semantic meaning and then “ghisa” is the material with which fountains are made. In some cases, inferential capabilities are essential. Given the proposition:

```
“Le fontane del grande Mastrofontani sono
costosissime”
```

(“The fountains by the great Mastrofontani are very expensive”, where in Italian “del” may mean either “made by” or “possessed by” or even “located in” and “grande” may mean “great”, “big” or “large”).

Assume that there is no knowledge about “Mastrofontani”. People can again perform with agility the proper semantic evaluation, inferring that this element is a lexical proper name. Deduction also makes use of common knowledge and memory to assess that “Mastrofontani” is presumably the proper name of the fountain maker. After having understood the sentence, a person will infer that the artist Mastrofontani is very well-known and well rated.

A classic NLP system simply does not consider the information that should be needed in order to properly cope with propositions of this kind. A branch of advanced NLP is related to probabilistic analysis of sentences [3, 4]: at present however, pure probabilistic systems for natural language processing have not demonstrated to have the potential for a good performance in general

contexts. Also, classical NLP systems, including probabilistic ones, have many difficulties to analyze sentences such as:

"Le fontane in ghisa sono indistruttibili"

and

"Le fontane in firenze sono indistruttibili"

The two sentences have identical syntactic structure, but have, for humans, a clear difference in semantic connotation. In classical automatic language processing methodologies, syntactic and semantic analysis do not have, in many cases, sufficient information neither to determine with certainty the syntactic aspects and details nor to give the correct semantic evaluation. Even probabilistic methods have difficulties. For example, in semantic analysis both "ghisa" and "firenze" (Florence) can be plausibly classified either as materials or as places. People instead, having knowledge about cast iron and Florence, can correctly attribute to "in ghisa" the logical meaning of complement of material and to "in firenze" the logical meaning of complement of place. Thus, for coping with these cases NLP systems must at least include ontological reasoning, and thus must become to some extent "intelligent". Consider now a short speech which is a composition of two propositions:

"The waiter recommended a Japanese dish. He did not like it very much."

Assuming that the system is able to successfully complete the analysis on the first proposition, in the next proposition "He did not like it very much." resolution of the pronoun, i.e. the semantic definition of "he" is a problem that cannot be solved without exploiting information extracted from the first proposition. Otherwise, failure on pronoun resolution prevents proper semantic evaluation of the second proposition. Solving the pronoun implies possessing deductive abilities and appropriate background knowledge. Once established the association "he=customer" it becomes possible to semantically analyze the proposition and the whole text. These operations are easily carried out by humans in a concurrent way. The associations are kept in the short term memory for a very limited time. An NLP system that does not have an equivalent mechanism cannot resolve properly the relations between propositions that compose a speech. Then, it cannot semantically analyze, in the proper way, a text as a whole.

4 Linguistics background

The study of language in its lexical, syntactic and semantic profiles has roots and is the result of the human interest in understanding the processes on which language is based. The grammatical and logical analysis have analytical purposes equivalent to those of syntactic and semantic analysis in computer science. Thus, studies in linguistics can be exploited for improving information technology methods and techniques. The purposes of grammar are to classify the different words by assigning each of them to their grammatical class and to indicate for each previously-classified word the morphological characteristics, like gender, number, type, and so on. In NLP, the purposes of grammar are similar to syntactical analysis and part-of-speech tagging. The purpose of logical analysis is the determination of the role that the components of the speech play in the formulation of thought. It distinguishes between analysis of a single sentence or simple proposition and analysis of a complex sentence or period. The analysis of the proposition aims at identifying the relationships between the various elements of the sentence, that is the subject, predicate, object (or direct) complements, indirect complements and attributes.

The analysis of a period consists of identifying the relationships between the various simple phrases that make up the period: it identifies the phrase regent (or “primary”) and the subordinate or coordinate ones for which it establishes the kind of either subordinating or coordinating (explicit or implicit) relationship and the logical role. For example, given the simple text consisting of the single proposition:

“L'aereo vola nel cielo blu”
(The airplane is flying in the blue sky).

the analysis must extract: i) subject: “the airplane”, ii) action: “flying” towards some destination, iii) place: the “blue sky”.

The logical and semantic analysis is based on syntactic and grammar information represented by the syntactic category to which each word (or token) belongs and morphological properties, i.e., the information returned by syntactic analysis.

5 Proposed approach: syntactic-semantic fully informed analysis

The syntactic-semantic fully informed analysis is an attempt towards the transposition to the computer science scenario, and in particular to natural language processing, of the operational dynamics studied in the context of

neuroscience. The proposed approach introduces advances not only in the syntactic and semantic analysis: rather, the approach has aspects concerning knowledge representation and the related automated reasoning methods. In fact, the approach relies on a background knowledge base in order to properly perform the analysis. The input of syntactic-semantic analysis is (as usual) a sequence of tokens obtained from lexical analysis. The results of syntactic-semantic analysis consist in (i) establishing the syntactic correctness of the sentence; (ii) creating a formal representation of extracted knowledge.; (iii) adding to the knowledge base this representation, as well as the consequences that can be drawn from it. I.e., the objective is to extrapolate the structure and the meaning of the natural language expression at hand, and to properly exploit it for enlarging or improving the available knowledge.

5.1 Background: Definite Clause Grammars

The enhanced analysis method that we propose is based on introducing suitable extensions to the DCGs (Definite Clause Grammars) formalism. The DCGs have been defined and introduced in prolog systems [6, 7]. DCGs have been demonstrated to be convenient ways of representing grammatical relationships for various parsing applications. They can be used for natural language, for creating formal command and programming languages. For example, DCG is an excellent tool for parsing and creating tagged input and output streams, such as HTML or XML. A DCG translates DCG rules into pure prolog. Syntax of a DCG rule is quite intuitive: a sample rule is shown below.

```
non_term1 --> non_term2, [term1].
```

where `non_term1` and `non_term2` are non terminal symbols and `[term1]` is a terminal symbol. On the left-hand side there is a non terminal symbol, while on the right-hand side there can be any sequence of terminal and non terminal symbols (for short “terminals”, indicated in square brackets). A simple DCG grammar, especially developed for propositions similar to the one of previous example, is given below and will be used in the following in order to illustrate proposed extensions.

Notation: Our examples refer to the Italian language. Below we list the correspondence between Italian and English terminology: “proposizione” stands for “proposition”; “soggetto” stands for “subject”; “verbo” stands for “verb”; “complemento” stands for “complement”; “compl_stato_in_luogo” stands for a complement of place; “nome” stands for “noun”; “aggettivo” stands for “adjective”; “articolo” stands for “article”; “proposizione” stands for “proposition” or equivalently “sentence”.

```

proposizione --> soggetto, verbo, complemento.
soggetto --> articolo, nome.
soggetto --> nome.
complemento --> compl_stato_in_luogo.
compl_stato_in_luogo --> preposizione, nome.
compl_stato_in_luogo --> preposizione, nome, aggettivo.
articolo --> [lo].
nome --> [aereo].
nome --> [cielo].
aggettivo --> [blu].
verbo --> [vola].
preposizione --> [nel].

```

Given the proposition seen before “Lo aereo vola nel cielo azzurro” (“L’aereo” is an abbreviation for “Lo aereo”) it is possible to verify, by means of a prolog interpreter, whether the proposition is valid w.r.t. the grammar. In fact, the answer to the query (in standard form)

```

?- proposizione([lo,aereo,vola,nel,cielo] , []).
Yes      (w.r.t. No for invalid sentences).

```

5.2 Extension: Semantically Enhanced Definite Clause Grammars

The “Semantic Enhanced DCGs” (for short SEDCGs) on the one hand can improve the syntactic analysis and on the other hand allow one to elicit semantic information from sentences and periods. In terms of performance, the SE-DCG are able to operate with high efficiency due to search space cutting that this formalism allows. Unlike DCG, the use of syntactical and semantical informations together with the previously deducted and stored knowledge, allows the SE-DCG to cut branches of search space as soon as they present syntactic, semantic or general logic inconsistencies (in relation to the knowledge already gained). This property allows to use the SE-DCG even in contexts where operational performance play a key role.

The extension of DCGs to SE-DCGs is very simple, and implies limited modifications to a DCG pre-processor. However, there is a significant gain in DCGs potential.

The proposed extension is the following. Non terminals may have one or more logical variables as argument, that will be instantiated during the parsing process to terminal symbols. The left-hand side of rules is enhanced by adding expressions in brackets that express constraints on these variables. Thus, a sample SE-DCG rule can look like:

```

non_term1(X1,X2) --> non_term2(X1),
                    non_term2(X2), [term1],
                    { constr(X1,X2,term1) }.

```

where there can be as many arguments as needed, and the constraint (here indicated as `constr(X1,X2,term1)`) is a formula involving variables and terminals.

5.3 Syntactic-semantic fully informed analysis: syntactic aspects

Below we discuss why the extension is useful for better coping with syntax, starting from a practical problem. Given the above DCG grammar, if we introduce some more terminals such as the articles “la” and “le” (female gender, singular and plural respectively) and “gli” (male gender, like “lo”, but plural) sentences with the wrong article will be assumed as correct as well. This problem might be coped by creating several copies of rules each dealing for instance which either singular or plural, either male or female gender, etc. However, this would result in a less elaboration-tolerant formulation of the grammar and a much less efficient parsing process. We mean to solve the problem taking however into account the following objectives: i) keep the grammar as simple as possible; ii) obtain an efficient parsing process.

The constraints that we have introduced can be exploited in order to check grammatical aspects, such as e.g. singular/plural male/female concordance while leaving the grammatical rules untouched. The following formulation of the example checks concordance (where “maschile” stands for “male gender”, “femminile” stands for “female gender”, `singolare` for “singular” and “plurale” for “plural”):

```
[...]
proposizione --> soggetto, verbo, complemento.
soggetto --> articolo(G,N), nome(G,N).
soggetto --> nome(G,N).
[...]
articolo(G,N) --> [lo], { G = maschile, N = singolare }.
articolo(G,N) --> [la], { G = femminile, N = singolare }.
articolo(G,N) --> [gli], { G = maschile, N = plurale }.
nome(G,N) --> [aereo], { G = maschile, N = singolare }.
[...]
```

Now we have:

```
?- proposizione([lo,aereo,vola,nel,cielo],[ ]).
Yes

?- proposizione([la,aereo,vola,nel,cielo],[ ]).
No
```

The improvement is in the simplicity of the grammar and in the improved efficiency of parsing, as much less backtracking is needed. All partial syntactic trees that violate concordance are simply not developed, thus

enormously reducing the search space. In fact, concordance is just one of the many constraints that any language includes. In a complementary way, DCG grammars can be used for generating sentences. With our solution, over-generation is greatly limited. Sentences such as: “la aereo vola nel cielo” and “gli aereo vola nel cielo” are simply not generated.

5.4 Syntactic-semantic fully informed analysis: basic semantic aspects

SE-DCGs have the potential of allowing one to extract semantic information from sentences and periods. Consider the following variant of the DCG grammar introduced before. The constraint in the first rule states that the “value” of a proposition is its subject. The constraint in the second rule states that the “value” of the subject is the name (without article). An expression like e.g. `nome(_)` states that in the present context we do not care about the value of the argument.

```

proposizione(S) --> soggetto(S), verbo, complemento.
soggetto(S) --> articolo, nome(N), { S = N }.
soggetto(S) --> nome(N), { S = N }.
complemento --> compl_stato_in_luogo.
compl_stato_in_luogo --> preposizione, nome(_).
compl_stato_in_luogo --> preposizione, nome(_), aggettivo.
                                articolo --> [lo].

nome(N) --> [aereo], { N = aereo }.
nome(N) --> [cielo], { N = cielo }.
aggettivo --> [blu].
verbo --> [vola].
preposizione --> [nel].

```

This simple extension allows a basic syntacticsemantic analysis to be performed via the prolog query:

```

?- proposizione(S, [lo, aereo, vola, nel, cielo], []).
S = aereo
Yes

```

The sentence has been, as before, recognized valid in relation to grammar, but there has been the possibility of extracting information such as, in this case, the subject of the proposition. This ability is important because extracting information will make it possible to perform deduction and reasoning. The extracted knowledge can be added to the knowledge base or exploited for performing useful computation. It is of course possible to introduce several arguments related to different kinds of information. For instance, if we extend the representation to `proposizione(S,V)` in order to extract subject and verb, in the example we are thus able to add to the knowledge base the new fact

```

vola(aereo).                                (=fly(airplane))

```

Note: the syntactic and semantic enhancements are by no means exclusive and can coexist in the same SE-DCG.

Thus, similarly to the above-discussed biological model, in SE-DCGs the analysis of a sentence proceeds in part sequentially and in part by integrating in parallel the syntactic and semantic aspects.

5.5 Syntactic-semantic fully informed analysis: advanced semantic aspects and the background knowledge base

The similarity with the biological model can be improved by better exploiting the potential of constraints. In fact, as mentioned they can be any formula involving the logical variables and the non-terminals occurring in a rule. Up to now, we have seen simple constraints involving equalities. Clearly, inequalities and other comparison operator may occur, as well as logical connectives. However, the real add-on is that a constraint may contain atomic formulas to be evaluated as true/false in the background knowledge base.

The knowledge base will certainly include a dictionary of known objects and ontological aspects to establish correspondences. In principle however, it may contain any form either commonsense knowledge or scientific/specialized knowledge so as to possibly integrate reasoning on language with other forms of reasoning. Therefore, syntactic-semantic fully informed analysis is characterized by an useful blend of enhanced syntactic analysis performed in parallel with basic semantic analysis, all of this integrated with reasoning and inference performed on a knowledge base. In this structure, the role of memory in understanding, assimilating and using language is explicit.

The knowledge base may include temporal aspects, thus allowing one to reason about the past, and introducing a distinction between short-term and long-term memory. The knowledge base will also in general include a “temporary” area where acquired new knowledge is kept in order to be evaluated: if it reveals to be plausible and useful it is transferred into “permanent” memory, otherwise it is either discarded or kept for further evaluation. Thus, our system is potentially able to learn and evolve, and its linguistic abilities will improve over time. Incomplete information about a sentence (notice that the presence of unknown words creates difficulties to people as well) can be filled by means of plausible reasoning such as default reasoning or abduction, and plausibility of results (that will initially be placed in the “temporary” area) will be checked according to subsequent interactions with the environment (about the authors’ experience in automated reasoning, evolving agents and learning see the research group publications: http://www.di.univaq.it/stefcost/pubbls_stefi.htm).

In the case studies that we have developed, common knowledge about lexical elements is defined as exemplified below, i.e., listing both semantic attributes (thing, inanimate, etc) and syntactic ones (female, singular).

```
is_nome('home', thing, concrete, common, inanimate,
        female, singular).
```

Verbal expressions are represented mainly by listing their grammar feature that also indicate how they are being used.

```
is_verbo('eat', transitive, active, prima, indicative, present,
        third, singular)
```

Constraints in SE-DCG rules allows the system to check e.g. whether the terminals occurring in the sentence are known, and, if so, if they are used in a plausible way. Coming back to the example of airplanes flying in the blue sky, we might modify the above discussed SE-DCG so as to extract the subject, verb, complement and kind of complement:

```
proposizione(S,V,C,Mode)--> soggetto(S), verbo(V),
                           complemento(C,Mode).
```

We might then add a constraint checking whether the subject can plausibly perform the action indicated by the verb, and if the complement is semantically acceptable:

```
proposizione(S,V,C,Mode) --> soggetto(S), verbo(V),
                           complemento(C,Mode), {plausibile_subj(S,V),
                           plausibile_compl(V,C,Mode)}
```

In the knowledge base, we might have rules such as:

```
plausibile_subj('vola',V):- flying_object(V).
flying_object('aereo').
```

```
plausibile_compl('vola',P,place) :- fly_where(P).
fly_where('cielo').
```

The constraint ensures that the sentence is plausible, while it would fail the check if for instance the sentence were mentioning an airplane flying “in the blue sea” as “sea” is not included in the fly_where list. The constraint can also include repair actions, related to what to do either in case of failure or in case of uncertainty.

Syntactic-semantic fully informed analysis is able to resolve ambiguities which are hardly treated by classical NLP. Consider the two sentences:

```
La porta è aperta.
(The door is open [= lies, remains open].)
```

```
La porta è aperta dal vento.
(The door has been opened by the wind.)
```

In the first case “aperta” is as adjective while in the second case “aperta” is participle used for the passive form of the verb. The following SE-DCG resolves the ambiguity:

```

predicato_verbale(Vrb) --> verbo(Vrb).
predicato_nominale --> copula, nome_del_predicato.
copula --> [Verbo], { is_verbo(Verbo, Infinito, _, _, _, _, _, _),
                    is_copulativo(Infinito),
                    controllaVeraCopula(...) }.

```

where `ControllaVeraCopula` succeeds if the verb actually is a “copula” (if indeed it preaches a property of the subject).

5.6 More about knowledge representation

As a side-effect of the analysis, the SE-DCG processor adds all valid sentences to the knowledge base in a quite rich format. In fact, natural language processing works in general on a text consisting of a series of periods, where each period is composed of one or more propositions. A period expresses a concept. The proposition is the atomic unity of thought and expresses a concrete thought.

The syntactic-semantic fully informed analysis phase returns as a result a list of “concepts” that represents the extracted knowledge. Each “concept” contains the elements of the concept expressed in a period. A concept is composed of one or more element (“concept elements”), which represents the formalization of propositions. For instance, the sentence:

```

L'informatica è una scienza che studia il trattamento
dell'informazione.
(Computer Science is a science concerning information
processing.)

```

once successfully analyzed is stored in the following form:

```

concept(
  croot( element(1, main, null,
    eroot( sog( art('la'), nome('informatica')),
      prdnomin( cop('è', 'essere'),
        prdnome(
          art('una'),
          nome('scienza')) )
    ),
    element(2, rel, 'che',
      eroot( sog(sogPREV_ELEMENT),
        vrb('studia', 'studiare'),
        c_ogg( art('il'), nome('trattamento') ),
        c_spe( art('della'), nome('informazione'))
      ) ) ).

```

Each “(concept) element” includes information about semantics of the period, such as an index (in chronological order) of the proposition in the period, the

kind of relationship that exists with the previous propositions and (if existing) the lexical connection.

In the last example, the input is a single period composed of two propositions. The first proposition is defined as “main”, the second one is a relative proposition. The representation is composed of a single concept (period) composed of two elements (propositions). The first element is the primary one (main), while the second is the relative (rel). This knowledge representation form has a key feature: it explicitly expresses the relationship between logical components of the text. As seen in neuroscience, it is possible to say that semantics relies both in information (inherited from lexical elements) and in the relationship between information. Thus, semantics results from the relationship between logical components of the text, as well as, of course, from the meaning of each lexical unit.

5.7 Inference

A system of natural language processing may constitute a component of one or more complex systems. Therefore, the final result of the syntactic and semantic analysis of text can be the basis for the other components functions. Consider the scenario of semantic search engines. Assume to have the following text:

```
“Un albero è composto da foglie”
(“A tree is composed of leaves”)
```

The extracted knowledge representation is as follows:

```
concept(
  croot(
    element(1, main, null,
      eroot(
        sog(art('un'), nome('albero')),
        vrb('è composto', 'comporre'),
        c_materia(preps('da'), nome('foglie'))
      ) ) ) )
```

It is possible to generalize this knowledge and add it to the knowledge base. By some simple reasoning steps based on general knowledge about being part of and its synonyms (see e.g. [1]), the system can answer questions such as the following:

```
“Di cosa fanno parte le foglie ?”
(“What are leaves part of?”)
```

The query can in fact be translated into the prolog-like counterpart:

```
?- is_part_of( foglie, O ).
O = albero
Yes
```

and the user is told, in natural language (by means of the opposite translation): “Le foglie fanno parte di un albero” (“Leaves are part of a tree”)

6 A next-generation search engine: Mnemosine

We have developed a prototype semantic search engine as a demonstration and a verification of the quality of the developed theory. This new system is called Mnemosine, after the Greek goddess of Memory.

We consider Mnemosine as a proof-of-concept of the presented approach, though the system design has been carefully performed in order to easily scale up (if this will be the case) to a real product. We have chosen to implement a search engine because of the high potential practical impact of real qualitative advances in this field.

We will try to demonstrate that Mnemosine constitutes a step ahead of traditional lexical engines. Another motivation lies in the possibility of performing a comparison with other innovative solutions available on Internet. This should allow us to evaluate the quality of adopted technology within a high profile scenario.

The Mnemosine user interface is quite essential (see Figure 1), but allows a user to formulate questions in natural language, such as “Che cosa è un albero?” (“What is a tree?”)



Fig. 1. The user interface

For the development and testing of Mnemosine, we have opted to using real data products from third parties. The choice of data is of primary importance: in the field of artificial intelligence in fact, many solutions operate properly and efficiently on the data on which they have been developed and tested and then their efficiency collapses dramatically as soon as switched to a real

operating environment. Also, many NLP systems fail when applied to real data.

For Mnemosine, the data set that we have considered is a subset of the pages of Italian Wikipedia. This choice has made it possible to combine the need for a set of heterogeneous data and the need to keep the size of the dataset compatible with computing resources and storage available for the prototype development and testing of the system. If asked for instance:

“Cosa è l'informatica ?” (“What is computer science?”)

Mnemosine returns up to three kinds of results, according to the submitted query and to the information either available or inferable from the knowledge base. If the knowledge base provides (directly or by means of reasoning) an answer, it is translated into natural language and returned to the user (see Figure 2).



Fig. 2. The answer to a query as a deduction from the knowledge base

“[Deduction] Computer science is a science that concerns information processing.”

The set of responses in natural language constitutes the first class of results. The answer is generated basing on the knowledge extracted from the analyzed Wikipedia pages and formalized as discussed in the previous section. As we have seen in fact, text extracted by the dataset (e.g. “Un albero è composto da foglie”) is analyzed and then stored as a concept. This concept can be reasoned about by means of the background knowledge base, and thus related questions such as “Di cosa fanno parte le foglie?” can be properly answered. This behaviour is innovative compared to current Query Answering (for short QA) technologies.

State-of-the-art QA systems can provide an answer to a query only if it is already lexically present in the indexed pages. Then, a QA system can provide an answer to the above question only if it has as a previously indexed text such as “Le foglie fanno parte di un albero”. Mnemosine instead is able to answer a question based on the meaning and on reasoning rather than on relatively simple lexical manipulation and matching.

Consider now the problem of ambiguity. Given e.g. the query "Che cosa è un albero?" ("What is a tree?") the word "tree" may refer either to trees as intended in botany or to trees as data structures in computer science or also to trees as understood in genetics. Mnemosine operates the distinction among different plausible contexts. For example, for the query "What is a tree?" several results are returned, each one associated to the proper context. Then, the user is allowed to contextualize the search by selecting the context of interest so to obtain, in a rapid manner, specific results without having to manually select the appropriate contents.

The context is used for the second and third class of results. Results of the second class consist in references to pages containing answers or anyway contents related to the query. The third class of results consists of references to pages containing, in relation to the query, answers or contents either similar or equivalent in terms of concepts or involving equivalent lexical items

For example, for the query "Cosa è l'informatica?" the results of the second class are shown in Figure 3.



Fig. 3. Second class of results

Each element of the second class of results includes: a reference (url) to the page containing the answer to the query, a short textual description and the link to click. The results of the third class are shown in Figure 4.

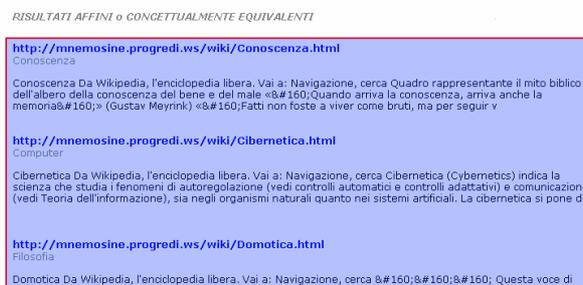


Fig. 4. Third class of results

The knowledge extracted from the pages to which they refer to are related, w.r.t. the given query, either to equivalent concepts or to equivalent lexical items. In summary, Mnemosine offers a wide range of search capabilities based on our new approach that exploits SE-DCGs and commonsense reasoning on a background knowledge base. Mnemosine has been fully implemented, and the authors of this papers will be glad to allow interest readers to experiment the system.

7 Related Work

Though Mnemosine is a product with just a few months of life, and despite the limited available resources for computation and storage, experiments show that the quality of the performed semantic search has reached an excellent level. This can be seen by comparison with other similar solutions far more mature and with substantial economic budgets behind. Namely, we have been unable to find products on the search market capable to formulate responses in natural language which are inferred from extracted information by means of automated reasoning techniques. Moreover, we have found evidence of the fact that most semantic search engines have been developed on the paradigm of Question Answering (QA).

Other solutions, instead, work mainly on syntax. We have examined in some depth AskWiki (http://askwiki.com/AskWiki/index.php/Main_Page), which has been defined as “the” search engine based on natural language for extracting information from Wikipedia. It has been developed by large companies which are leader in the semantic search engine scenario in a direct collaboration with Wikimedia Foundation (Wikipedia project owner). The user interface allows you to type in natural language queries such as: “What is a tree? “ The result is a fragment of text extracted from the (explicitly listed) page: <http://en.wikipedia.org/wiki/Tree>. This search engine has clearly been developed upon the Question Answering paradigm. The replies are portions of text extracted from the indexed pages (which are explicitly indicated). Also, AskWiki returns results related only to “tree” as defined in the context of botany. Differently from AskWiki, Mnemosine offers a set of semantic results covering all possible aspects related to the semantic, allowing then a user to choose the proper context and relevance of the results. Also, it provides conceptually equivalent results, a feature that is lacking in semantic search solutions currently on the market.

It is important to notice that Mnemosine is not a QA system: thus, the comparison with AskWiki or other QA systems in general is by no means exhaustive but is limited to certain Mnemosine's operational features only. As seen before, Mnemosine offers a wide range of results, from responses formulated in natural language to references to web pages containing a response or similar contents.

However, Mnemosine is an experimental research project while the other analyzed solutions are now coming to the market and are therefore more stable and mature. Among the future developments, we intend to assess its suitable metrics for evaluating the operational functions in order to make a more objective comparison among similar solutions feasible.

8 Concluding Remarks and Future Directions

The proposed theory of natural language processing based on syntactic-semantic fully informed analysis makes some steps ahead with respect to canonical NLP. We have introduced a computational model and a representation of extracted knowledge which is inspired and is meant to be related to biology and neuroscience. We have supported the claim of novelty and applicability of the proposed approach by means of the objective feedback provided by developing a project, Mnemosine, as a case study in a highly competitive industrial field: semantic search. This in order to prove the quality of our approach not just by means of suggestive arguments, but by means of that “quid” that ultimately counts and determines the difference: the results. The current role of Mnemosine is to be a proof-of-concept, a validation of the developed theory. In fact, Mnemosine can be improved under several aspects and can be extended in many directions. However, the underlying architecture has been designed so as to be ready for a timely transformation from a research prototype to an actual industrial product. A main current research goal is to improve the quality of the results provided by our semantic search engine. On the one hand, we mean to extend the role and usage of natural language processing. On the other hand, we mean to obtain further improvements related to introducing a “plausibility score” for extracted knowledge, to the handling of temporal logic aspects as well as of other typical aspects of human communication. In addition, as mentioned above, some metrics should be developed for the evaluation relating to the functionality offered by Mnemosine.

In our view, the future Mnemosine system will be able to learn and evolve, by updating its knowledge base in consequence of the interactions with the users. This is the focal point: validate the quality of the theory developed by the direct application in real operating context, and make the theory itself flexible and adaptable enough to be able to evolve.

References

- [1] J. Barklund, S. Costantini, P. Dell'Acqua and G. A. Lanzarone, "Meta-Reasoning Agents for Flexible Query Answering Systems", *Flexible Query Answering Systems*, Kluwer Academic Publishers, 1997.
- [2] P. Blackburn and J. Bos, *Representation and Inference for Natural Language*, Center for the Study of Language and Information, Stanford University - Lecture Notes, 2005.
- [3] R. Bod, J. Hay and S. Jannedy, *Probabilistic Linguistics*, MIT Press, 2003.
- [4] C. D. Manning and H. Schuetze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.
- [5] R. Mitkov, *The Oxford Handbook of Computational Linguistics*, Oxford University Press, 2005.
- [6] P. M. Nugues, *An Introduction to Language Processing with Perl and Prolog*, Springer, 2006.
- [7] F. N. Pereira and S. M. Shieber, *Prolog and Natural-Language Analysis*, Microtome Publishing, 2002 .
- [8] D. Poole, A. Mackworth and R. Goebel, *Computational Intelligence: A Logical Approach*, Oxford University Press, 1998.
- [9] D. Purves et al., *Neuroscience, Third Edition*, Sinauer Associates, 2004.
- [10] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (2nd Edition)*, Prentice Hall, 2002

Experimenting with Stochastic Prolog as a Simulation Language

Enrico Oliva, Luca Gardelli, Mirko Viroli, Andrea Omicini

ALMA MATER STUDIORUM–Università di Bologna

Via Venezia, 52 - 47023 Cesena, Italy

{ enrico.oliva, luca.gardelli, mirko.viroli, andrea.omicini } @unibo.it

Abstract. While simulation is an established tool for scientific analysis, it is recently gaining more interest also in other contexts, such as software engineering. Hence, more and more attention is devoted to the development of suitable simulation languages (and tools), as well as to their exploitation in application development and run-time. As already experienced in the context of general-purpose programming languages, we envision future developments towards expressiveness, with performance issues becoming less and less relevant.

Along this direction, we propose a preliminary stochastic simulation framework developed on top of a logic programming language, called *Stochastic Prolog*: this framework allows us to run simulations directly from Prolog-based specifications. Our objective, in this work, is to put the basis for future research on logic stochastic language used for simulation purpose. In our approach Prolog clauses can be labelled with *rates* modelling temporal/probabilistic aspects. The main advantage of using Prolog is that it is significantly more expressive than other languages typically used in simulation, allowing complex specifications to be more easily encoded. In order to evaluate our framework, we compare it with the stochastic language defined by the PRISM tool, by discussing as case study the *collective sorting* problem, a decentralised sorting strategy for multiagent systems (MAS) inspired by behaviours observed in social insects.

1 Introduction

There is a growing interest in simulation languages and tools, and an increasing use of them throughout the engineering process. This is related to the increasing complexity of today computational systems in both the scientific and the engineering community: complexity implies that it is essentially infeasible to fully predict the behaviour of a system from its design, since small changes in some of the surrounding condition can lead system behaviour to diverging dynamics. Simulation is hence used as a means to preview the behaviour of complex computational systems without resorting to complete implementations, and possibly using such observations to tune the design in the early phases of the engineering process.

Many current simulation languages are based on stochastic extensions of some very low-level language, such as Stochastic π -calculus [1], generic stochastic process algebras for quantitative analysis such as EMPA [2], and verification-oriented tools like PRISM [3]. As the systems to be modelled and simulated tend to grow in complexity, such tools are more and more becoming inadequate. What typically happens is that a designer is forced to define a system behaviour using low-level mechanisms and tricks, much in the same way a programmer in the 70s would have used Assembler for building a complex application. One of the main reasons why this situation is persisting is that performance is still usually considered the primary issue of simulation, rather than expressiveness of the languages and frameworks.

Much in the same way programming languages evolved from low- to high-level, including more and more expressive abstractions and requiring true virtual machines in spite of a significant performance overhead, we envision a similar situation for simulation languages and frameworks. Although we do not deny the importance of performance in simulation, we believe it can and should be handled at the level of the underlying execution engine: the designer is to be freed from this problem, and has to leverage expressive languages for specifying system behaviour. Towards this direction, we believe logic languages such as Prolog can be used as a basis for developing suitable simulation languages, since: (i) they are high-level, characterised by expressive and abstract constructs; (ii) they are still core languages with few constructs, paving the way towards formal analysis of probabilistic properties; and (iii) they allow to easily exploit the rule-based specification style—which is quite common in the context of distributed systems.

In this article we introduce a preliminary stochastic simulation framework on top of the Prolog language, called *Stochastic Prolog*: to this purpose we follow a probabilistic logic approach labelling clauses with rates. As a first step, we define a base stochastic extension of first order logic in order to follow the framework of Continuous Time Markov Chains CTMC [4]: rates define transition frequency according to an exponential distribution, and in the case of race conditions they implicitly define the probability for an action to be scheduled next.

As a second step, we evaluate our Stochastic extension of Prolog applying it to the case of collective sorting, a decentralised sorting algorithm inspired by *social insects behaviours* [5], and then compare it to the PRISM tool, taken here as a reference language for simulation. In this case study, a multiagent system (MAS) is composed of agents and tuple spaces: agents are in charge of moving tuples from one space to another according to their type, until reaching full ordering [6, 7]. We model a self-organising solution to the collective sorting problem using both our framework and the PRISM tool, then compare the expressiveness of the two solutions, finally showing how the framework proposed here can better scale with system complexity.

The remainder of the article is structured as follows: in Section 2 we describe the simulation framework; in Section 3 we provide a brief description of the case study and provide specifications both for the PRISM tool and our framework; in Section 4 we discuss related works and finally conclude.

2 A Stochastic Simulation Framework in Prolog

We describe a framework to run stochastic simulations based on Prolog logic language in order to exploit the expressive power of a declarative language. Prolog is very useful because of the uniform representation of code and data, both encoded as first-order logic (FOL) clauses, which makes writing (meta-)interpreters quite easy [8]. The framework allows the modelling of stochastic aspects, in particular according to the CTMC model, an important class of stochastic processes widely used in practice both to determine system performance and to predict system behavior.

As a CTMC is basically an automata where transitions from one state to another are labelled with rates, in our framework it is used to enhance a logic program with rates driving the goal resolution process—the label-based approach is commonly used to introduce stochastic aspects in formal languages, e.g. the Stochastic π Calculus [1] and also in logic programming [9].

The basic idea of our framework is to use stochastic operation towards FOL for simulation purpose. To this end it is necessary to add a couple of features to FOL: 1) clauses annotated by *rate*, 2) stochastic inference relation.

Definition 1. *A stochastic logic program is a set of clauses of the form $r : h \leftarrow b_1, \dots, b_n$ and $h \leftarrow b_1, \dots, b_n$ where h and b_i are **atomic formula**, r a frequency value.*

Syntactically, in a Stochastic Prolog program the set of labelled clauses are expressed using the following notation: `label:h:-b1,b2,..bn..`. Namely, each stochastic clause is a standard clause with a prefix `label`, which is a frequency value (or rate) $r(\mathbf{X})$, where \mathbf{X} should be a ground number.

The semantic of a stochastic clause is defined by a possible world semantic of a Herbrand interpretation of the classical underlying first order language. Thus a C clause defined $C = r : h \leftarrow b$ is true in a possible world W denoted as $W \models C$ if and only if C is true in the Herbrand interpretation belonging to W . The values indicated in C clauses are the frequency rates that are used to choose next ground instance $C\theta$ and proceed with the resolution derivation process.

The stochastic inference relation for labelled clauses is inspired by Gillespie’s algorithm [10] based on frequency values.

Definition 2. *The stochastic inference is expressed by the following algorithm:*

1. Find the set \mathcal{C} of labelled clauses whose head h unifies with the current goal b
2. Calculate $r_{tot} = \sum_{i=1}^n r_i$, where n is the cardinality of \mathcal{C} , where r_i is the rate of $C_i \in \mathcal{C}$
3. Generate a random number $n_1 \in [0, 1]$
4. Evaluate the relation

$$\sum_{i=1}^{k-1} r_i \leq n_1 \cdot r_{tot} \leq \sum_{i=1}^k r_i$$

in order to find k ,

5. Next head to consider in the resolution process is the body of $C_k \in \mathcal{C}$

More practically, the operational semantics of stochastic Prolog is expressed by the meta-interpreter in Figure 1. From a Prolog point of view we have introduced a *probabilistic cut*. The normal cut tells the the interpreter to remove a choice point from the stack, eliminating a branch of solution tree and limiting the backtracking. Instead, a *probabilistic cut* tells the meta-interpreter to make a probabilistic choice in the SLD resolution tree, discarding the other possibilities.

Each execution of a program always corresponds to the production of a single simulation trace, formed by a stream of simulation events each being a couple **event (State,Time)**—where **State** is the Prolog goal to be solved yet, and **Time** is the elapsed time. The goal resolution process depends on the kind of predicate involved, briefly: standard predicates are solved as in Prolog, i.e., the top-most clause is selected and others are reconsidered during backtracking; predicates with rates are solved by randomly selected a clause (the higher the rate, more likely they are selected) and causing an elapsed time according to the exponential distribution (see below). The time t of each event in the simulation is calculated only after the execution of a stochastic clause, and proportionally to the total rate with an exponential distribution $t = (1/rtot) * (\ln(1/n_2))$ where n_2 is a new random number $\in [0, 1]$.

As a basic example, let us consider a process in which each time unit a fair coin is tossed a hundred of times, but with a 1% probability of failing, in which case the process is to be stopped. Figure 1 lists the code for simulating this system.

Example 1. Simple Stochastic Prolog Program for simulating a coin toss with possible failures, and a possible trace. The predicate `coin/1` expresses the fact that the probability of tossing to head or tail is 49.5%, and probability of failure is 1% (causing a failure of the process): then since the sum of rates is 100, each tossing will take place at an average 0.01 of elapsed time. Predicate `toss/0` drives the process: after solving `coin/1`, if `manhole` is found the process is stopped, otherwise it is reiterated again.

```
r(49.5):coin(head).
r(49.5):coin(tail).
r(1):coin(manhole).

toss() :- coin(X), continuation(X).
continuation(manhole) :- !.
continuation(_) :- toss().

-----
?- solve_trace(toss(), 100, T).
yes.
T=[...,
event(28, coin(tail), 0.0102374),
event(29, coin(head), 0.0151458),
```

```
event(30, coin(head), 0.00339425),
event(31, coin(manhole), 0.00836014)]
```

Note that the result of the simulation is provided by a meta-interpreter `solve_trace`, accepting as input the initial state of the stochastic program (i.e. the goal to be solved), a maximum elapsed time, and returning as output a sequence of events. In order to ask the solution we hence try demonstrating goal `solve_trace(+Goal,+Time,-Trace)`. The output is a list of events, each with its own elapsed time. Such a trace can easily be passed to a drawing program for generating charts.

2.1 A Prolog Implementation

The first implementation of Stochastic Prolog is easily achieved by exploiting the well-known meta-programming capabilities of Prolog. Our meta-program structure in Figure 1 is similar to the standard Prolog interpreter `solve` [8]. Notice that our meta-interpreter, in addition to normal clauses resolution, has to make stochastic choices without backtracking, and calculate the simulation steps and the simulation time. Currently, all the functionality required by Stochastic Prolog are developed by changing the standard resolution process of Prolog through the meta-interpreter. The resolution is directed by label value and label type. To introduce the label syntax `label:clause`, we define a new operator “:” through definition `:-op(500, yfx, ':')`. Rate labels are expressed by using label predicates `r(X)`. Technically, backtracking of labelled clauses is disabled by the meta-interpreter using cuts operator. We can introduce also some specification facilities like `p(X)` label which means that all `p(X):C` unifiable clauses have the frequency value that sum to 1. They are solved by randomly selecting a clause (according to its probability) and causing no elapsed time.

Figure 1 shows the top level predicate of the interpreter, which is called to start the simulation. Last clause of `solve_trace/6` predicate realises most of the selection process: all clauses matching `Goal` and with some label `X` are retrieved combining built-in predicates `findall/3` and `clause/2`. The stochastic choice is made in `selection/4` whose details are not reported for the sake of brevity. The system has been tested with SWI-Prolog platform [11], but it could have been tested with any other standard Prolog interpreter because only ISO standard predicates are used.

3 Comparing Stochastic Prolog with PRISM

In order to evaluate our Stochastic Prolog framework, we consider the case of *collective sorting* as described in [6, 7]. We first describe the problem, then we model a self-organising solution according to the agent paradigm. We provide a specification for both the PRISM tool and our Stochastic Prolog framework.

In this paper, we considered PRISM as a reference language for the sake of comparison, for it is a paradigmatic case of how programming in such simulation/verification languages hardly scale with the system complexity. Since a

```

solve_trace(Goal,Time,Trace):-
    solve_trace(Goal,Time,Trace,0,NS,TBack).
solve_trace(_,Time,_,Time,_,T):-!,write('STOP ').
solve_trace(true,_,_,_,_):-!.
solve_trace((Goal1;Goal2),Time,Trace,Step,NS,TBack):-
    !,(solve_trace(Goal1,Time,Trace,Step,NS,TBack);
    solve_trace(Goal2,Time,Trace,Step,NS,TBack)).%Disjunction

solve_trace((Goal1,Goal2),Time,Trace,Step,NS,TBack):-
    !,
    solve_trace(Goal1,Time,Trace,Step,NS,TBack),
    NS < Time,
    solve_trace(Goal2,Time,Trace,NS,NS1,TBack).%Conjunction

solve_trace(Goal,Time,Trace,Step,NS,TBack):-
    builtin(Goal),!,
    call(Goal),NS = Step. %System predicate

solve_trace(Goal,Time,Trace,Step,NS,TBack):-
    clause(Goal,B),
    solve_trace(B,Time,Trace,Step,NS,TBack).%Normal predicate

solve_trace(Goal,Time,Trace,Step,NS,TBack):-
    findall(X:(Goal:-B),clause(X:Goal,B),L),
    not(empty_list(L)),!,
    sum_up(L,Tot),
    random(R),          % 0-1 random number
    Tot1 is Tot * R,
    selection(L,Tot1,0,Rate:(G)), %make stochastic selection
    arg(1,G,Goal),arg(2,G,Body),
    label_eval(Tot,Rate,Step,Goal,NS),%eval label for step & time
    solve_trace(Body,Time,Trace,NS,NS1,TBack1).%Label predicates

```

Fig. 1. The meta-interpreter code for predicates `solve_trace/3` and `solve_trace/6`

PRISM specification defines a transition system, we expect the specification to rapidly grow with the system complexity (number of guards increases): this is often the case when dealing with non-trivial algorithms, such as the collective sorting. What we expect using a more expressive language such as Prolog, is that it should be possible to provide a more compact and simpler specification.

3.1 Case Study: Collective Sorting

In this section, we describe a particular decentralised sorting strategy called *collective sorting*: the solution to this problem has been inspired by sorting behaviours displayed by social insects [5]. Collective sorting has potential applications to both distributed software and physical environments, although,

each time behaving like one of the n agents chosen probabilistically—each agent has the same probability of being chosen. Hence, we consider the following single agent protocol:

1. choose the source tuple space randomly
2. choose the destination tuple space randomly
3. uniformly read a tuple S from the source tuple space
4. uniformly read a tuple D from the destination tuple space
5. only if the tuple kinds are different, transfer a tuple of kind D from the source to the destination

This self-organising algorithm is shown to bring the system towards ordering independently of the initial configuration of tuples [7]—the ordering is yet complete only by slightly changing the algorithm as shown in [6], but this issue is of no interest here. The chart displayed in Figure 3 represents the dynamics of collective sorting starting from initial situation TS1(20,20) and TS2(20,20): the algorithm evolves the system towards the final state TS1(40,0) and TS2(0,40). It is worth noting that the dual solution may have occurred as well, and it is not possible to foretell where a specific cluster will appear.

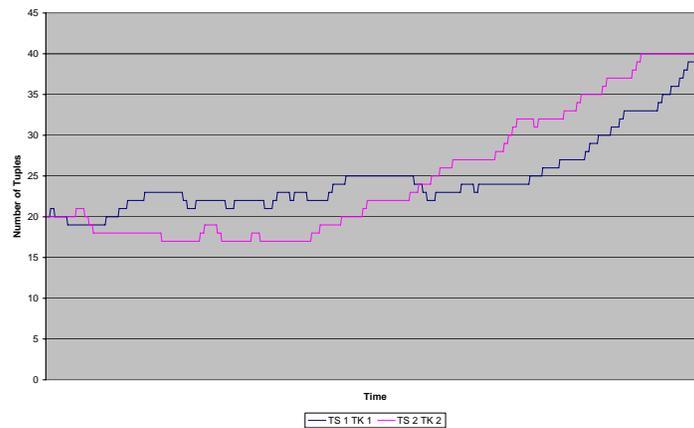


Fig. 3. The dynamics of the collective sorting, specifically tuple kind 1 in tuples space 1 and tuple kind 2 in tuples space 2

As a case to evaluate the simulation framework developed, we provide the specifications for the basic version of collective sorting using both the PRISM language and our framework.

3.2 Collective Sorting in PRISM

Among the many commercial and academic simulation frameworks, we have chosen the PRISM tool [3] as a comparison: other than allowing simulation, it offers a simple specification language and the possibility to perform formal analysis. In general, it is a paradigmatic case of low-level stochastic language.

In PRISM, models are specified using a state-based language based on Reactive Modules and is able to represent either probabilistic, non-deterministic and stochastic systems using, respectively, Discrete-Time Markov Chains (DTMC), Markov Decision Processes (MDP) and Continuous-Time Markov Chains (CTMC). Components of a system are specified using modules and the state is modelled as a set of finite-values variables: furthermore, modules composition and interaction is achieved in a process algebra style.

According to the agent agenda defined in the previous section, in Figure 4 we provide the PRISM commented specifications for collective sorting, considering the basic case of two tuple spaces and two tuple kinds. The first part of the specification consists in constants, variables and formulas: the behaviour of the agent is completely specified within the block `module agent .. endmodule`. Each step in the agent agenda is encoded using one or more transitions, specified using the notation `[] guard -> rate_1 : update_1 + ... + rate_n : update_n;`: the guard is a boolean expression that, when verified, leads to one of the next states specified in the updates according to the specified rates. The algorithm is articulated in four steps:

Step 0 choose the source and the destination tuple spaces

Step 1 randomly draw a tuple from source tuple space and observe its kind

Step 2 randomly draw a tuple from destination tuple space and observe its kind

Step 3 depending on the tuple kinds decide whether to move the tuple or not

It is worth noting that several transitions occur with rate value equals `decision`: this is an arbitrary large constant used to model a decisional process, i.e. a process having a duration small in comparison with the other durations of the system. It is easy to recognise that the PRISM language does not produce very compact specifications: indeed, specifications tend to grow combinatorially with the number of tuple spaces n (which directly influences the variables involved and the number of guards). For instance, even with $n > 5$ the specifications grows to several hundreds of rules: it is clear then that such a kind of language can be used only as a low-level one, whereas more expressive languages are instead required.

3.3 Collective Sorting in Stochastic Prolog

According to the agent agenda previously defined, we provide a Stochastic Prolog specification for collective sorting—see Figure 5. While the PRISM specification has been encoded for a specific instance, the prolog works for all instances of collective sorting problem, where the number N of tuple kinds and tuple spaces

can be any greater than 1. This is a clear expressiveness advantage over the PRISM language, where specifications grow very quickly.

A possible initial system with $n = 2$ is expressed by the following facts with a probability label:

```
p(80):cell(1,1,80).
p(50):cell(1,2,50).
p(30):cell(2,1,30).
p(20):cell(2,2,20).
p(1):ts(1).
p(1):ts(2).
```

Fact `cell(TS,TK,N)` stands for N tuples of kind TK residing in tuple space TS : we hence represent the system configuration with a matrix of weights. This choice ensures that while reading a tuple on a tuple space, it is more likely to get one of a bigger group. Moreover, we have two facts of the kind `ts(X)` with same probability, used to pick a tuple space probabilistically. These facts are considered as input state for the simulation, and could be changed throughout via assertion and retraction as usual in Prolog.

In the general sense the overall system could be considered like an instance of a CTMC. In the whole specification there is only one rule with a rate (with head `r(1):state(M)`), responsible for setting the agent working rate to 1, and four probabilistic selections: two for choosing source and destination tuple spaces `ts(S)` and `ts(D)`, and two for reading tuples in them `cell(S,KS,NKS)`, `cell(D,KD,NKD)`.

There are three main predicates that are involved in the problem specification: `transfer/4`, `state/1` and `start/0`. The predicate `transfer(+S,+D,+KS,+KD)` tests whether the transfer is possible or not and it accordingly executes the transfer—it is worth noting that this modifies the rates of facts. There, `S` and `D` are respectively tuple space source and destination identifiers, `KS` and `KD` are tuple kind of source and tuple kind of destination. Predicates `state/1`, `state0/1`, `state1/2`, `state2/4` represent states during protocol execution, and implement the CTMC: The `M` parameter contains the current state of system i.e. the matrix with all probability values, which is reified as argument for being tracked in the simulation trace. `start` predicate is called to start the simulation process.

In order to run a simulation, we invoke the meta-goal `solve_trace(start, 100, Trace)`, which produces as output the trace of the next 100 system states with corresponding elapsed time, e.g. as follows:

```
event(44, state([24:cell(2, 2), 50:cell(1, 2),
                26:cell(2, 1), 80:cell(1, 1)], 3), 0.767398)
event(45, state([50:cell(1, 2), 26:cell(2, 1),
                23:cell(2, 2), 81:cell(1, 1)], 3), 1.54405)
event(46, state([50:cell(1, 2), 26:cell(2, 1),
                80:cell(1, 1), 24:cell(2, 2)], 3), 0.455535)
event(47, state([80:cell(1, 1), 24:cell(2, 2),
```

```

                49:cell(1, 2), 27:cell(2, 1)], 3), 0.834995)
event(48, state([24:cell(2, 2), 49:cell(1, 2),
                26:cell(2, 1), 81:cell(1, 1)], 3), 0.119659)

```

In order to verify the generality of the Prolog specifications we evaluated the collective sorting in the instance with 3 tuple spaces and hence a 3×3 matrix, as in Example 2.

Example 2. collective sorting simulation with $N = 3$

```

p(80):cell(1,1,80). p(50):cell(1,2,50). p(10):cell(1,3,10).
p(30):cell(2,1,30). p(20):cell(2,2,20). p(20):cell(2,3,20).
p(70):cell(3,1,70). p(60):cell(3,2,60). p(10):cell(3,3,10).
p(1):ts(1). p(1):ts(2). p(1):ts(3).

?- solve_trace(start,100,T).
...
event(97, state([
p(11):cell(1,3,11), p(55):cell(3,2,55), p(8):cell(3,3,8),
p(32):cell(2,1,32), p(75):cell(1,1,75), p(22):cell(2,2,22),
p(27):cell(2,3,27), p(51):cell(1,2,51), p(69):cell(3,1,69)]),
1.85357)

event(98, state([
p(11):cell(1,3,11), p(55):cell(3,2,55), p(8):cell(3,3,8),
p(75):cell(1,1,75), p(22):cell(2,2,22), p(27):cell(2,3,27),
p(69):cell(3,1,69), p(50):cell(1,2,50), p(33):cell(2,1,33)]),
0.959323)

event(99, state([
p(11):cell(1,3,11), p(55):cell(3,2,55), p(8):cell(3,3,8),
p(22):cell(2,2,22), p(69):cell(3,1,69), p(50):cell(1,2,50),
p(33):cell(2,1,33), p(26):cell(2,3,26), p(76):cell(1,1,76)]),
0.366512)
STOP

```

4 Conclusion

In this article, we propose a stochastic framework based on Prolog that allows to perform stochastic simulation directly from Prolog specifications. The proposed extension, called Stochastic Prolog, follows the same approach used in other languages such as Stochastic π -calculus [1]. Clauses are labelled either with rates or probabilities, respectively modelling stochastic and probabilistic aspects: specifically rates define action duration according to an exponential distribution, while next event scheduling is based on the Gillespie's algorithm [10].

In the Logic Programming literature some concepts related to stochastic programming have already been introduced, but to the best of our knowledge they are not meant to target stochastic simulation. Muggleton [9] defines *stochastic logic programming*, where a stochastic logic program P is a set of labelled clause $p : C$ with probability $p \in [0, 1]$, and where for each symbol q in P the probability label for all clauses with q head sum to 1. The meaning of the label in Muggleton is probabilistic but it is not related to time or rate of execution. Therefore, our work is apparently the first one putting together timing and probabilistic aspects into Prolog, and then also the first to experiment it in the context of simulation of complex computational systems.

In order to evaluate our simulation framework we compare the specification of the collective sorting problem written in Stochastic Prolog with the same one expressed with the PRISM tool. Thus, we show that Stochastic Prolog specifications are more compact even with the simplest problem instances: while Stochastic Prolog specifications are not affected by the instance size, PRISM specifications tend to quickly grow up to hundreds of transition rules even with the simple case of $N = 4$ —4 tuple spaces and 4 tuple kinds. Although a more thorough study is now required, from this preliminary study it is clear that our approach generally provides for much more expressiveness, and could hence be used as an effective specification tool for system designers that need to simulate complex applications.

Further explorations will first involve a semantic study of the language, coding more case studies to better evaluate it, then extend the language in several ways, e.g. supporting concurrency operators as commonly found in process algebras.

References

1. Priami, C.: Stochastic π -calculus. *The Computer Journal* **38**(7) (1995) 578–589
2. Aldini, A., Bernardo, M., Gorrieri, R., Rocchetti, M.: Comparing the QoS of Internet audio mechanisms via formal methods. *ACM Trans. Model. Comput. Simul.* **11**(1) (2001) 1–42
3. University of Birmingham: The PRISM probabilistic model checker. Version 3.1.1 and documentation available online at <http://www.prismmodelchecker.org> (September 2007)
4. Kulkarni, V.G.: Modeling and analysis of stochastic systems. Chapman & Hall, Ltd., London, UK, UK (1995)
5. Deneubourg, J., Goss, S., Franks, N., C. Detrain, A.S.F., Chrétien, L.: The dynamics of collective sorting: Robot-like ants and ant-like robots. In Meyer, J.A., Wilson, S.W., eds.: *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. Classics. MIT Press, Cambridge, MA, USA (February 1991) 356–363
6. Viroli, M., Casadei, M., Gardelli, L.: A self-organising solution to the collective sort problem in distributed tuple spaces. In: *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007)*, Seoul, Korea, ACM (11–15March 2007) 354–359 Special Track on Coordination Models and Languages.

7. Casadei, M., Gardelli, L., Viroli, M.: Simulating emergent properties of coordination in Maude: the collective sort case. *Electronic Notes in Theoretical Computer Science* **175**(2) (June 2007) 59–80 5th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2006).
8. Sterling, L., Shapiro, E.: *The Art of Prolog: Advanced Programming Techniques*. 2nd edn. Logic Programming. MIT Press, Cambridge, MA, USA (1994)
9. Muggleton, S.: Stochastic logic programs. In De Raedt, L., ed.: *Proceedings of the 5th International Workshop on Inductive Logic Programming*, Department of Computer Science, Katholieke Universiteit Leuven (1995) 29
10. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry* **81**(25) (1977) 2340–2361
11. SWI-Prolog: Version 5.6.40. Documentation available online at <http://www.swi-prolog.org> (September 2007)

```

ctmc
const int MAX=40;
const int decision = 1000000;
global t1k1 : [0..MAX] init 20;
global t1k2 : [0..MAX] init 20;
formula chooset1k1 = t1k1/content1;
formula chooset1k2 = t1k2/content1;
formula content1 = t1k1+t1k2;
global t2k1 : [0..MAX] init 20;
global t2k2 : [0..MAX] init 20;
formula chooset2k1 = t2k1/content2;
formula chooset2k2 = t2k2/content2;
formula content2 = t2k1+t2k2;

module agent
//Variables encoding agent internal state
step : [0..3] init 0;
ts : [1..2];
td : [1..2];
s : [1..2];
d : [1..2];
//Randomly choose source and destination tuple spaces
[] step = 0 & content1 != 0 & content2 != 0 ->
  decision : (ts' = 1) & (td' = 2) & (step'=1) +
  decision : (ts' = 2) & (td'=1) & (step'=1);
// Choose source tuple kind
[] step = 1 & ts = 1 & content1 != 0 & content2 != 0 ->
  chooset1k1 * decision : (s' = 1) & (step' = 2) +
  chooset1k2 * decision : (s' = 2) & (step' = 2);
[] step = 1 & ts = 2 & content1 != 0 & content2 != 0 ->
  chooset2k1 * decision : (s' = 1) & (step' = 2) +
  chooset2k2 * decision : (s' = 2) & (step' = 2);
// Choose destination tuple kind
[] step = 2 & td = 1 & content1 != 0 & content2 != 0 ->
  chooset1k1 * decision : (d' = 1) & (step' = 3) +
  chooset1k2 * decision : (d' = 2) & (step' = 3);
[] step = 2 & td = 2 & content1 != 0 & content2 != 0 ->
  chooset2k1 * decision : (d' = 1) & (step' = 3) +
  chooset2k2 * decision : (d' = 2) & (step' = 3);
[] step = 3 & s = d ->
  decision : (step'=0);
//Tuple space source 1, Tuple space destination 2
[] step = 3 & ts = 1 & td= 2 & s = 1 & d = 2 &
  t1k2 > 0 & t2k2 < MAX ->
  1.0 : (step'=0) & (t1k2' = t1k2 - 1) & (t2k2'=t2k2 + 1);
[] step = 3 & ts = 1 & td= 2 & s = 1 & d = 2 &
  t1k2 = 0 ->
  1.0 : (step'=0);
[] step = 3 & ts = 1 & td= 2 & s = 2 & d = 1 &
  t1k1 > 0 & t2k1 < MAX ->
  1.0 : (step'=0) & (t1k1' = t1k1 - 1) & (t2k1'=t2k1 + 1);
[] step = 3 & ts = 1 & td= 2 & s = 2 & d = 1 &
  t1k1 = 0 ->
  1.0 : (step'=0);
//Tuple space source 2, Tuple space destination 1
[] step = 3 & ts = 2 & td= 1 & s = 1 & d = 2 &
  t2k2 > 0 & t1k2 < MAX ->
  1.0 : (step'=0) & (t2k2' = t2k2 - 1) & (t1k2'=t1k2 + 1);
[] step = 3 & ts = 2 & td= 1 & s = 1 & d = 2 &
  t2k2 = 0 -> 1.0 : (step'=0);
[] step = 3 & ts = 2 & td= 1 & s = 2 & d = 1 &
  t2k1 > 0 & t1k1 < MAX ->
  1.0 : (step'=0) & (t2k1' = t2k1 - 1) & (t1k1'=t1k1 + 1);
[] step = 3 & ts = 2 & td= 1 & s = 2 & d = 1 &
  t2k1 = 0 -> 1.0 : (step'=0);
endmodule

```

Fig. 4. Collective sorting specifications in the PRISM language

```

transfer(S,D,KS,KD) :-
    retract(p(N1):cell(S,KS,NKS)),
    retract(p(N2):cell(D,KD,NKD)),
    (N1 == 0,Nout1 is 0,Nout2 is N2;
     N1 =\= 0,exec(N1,N2,Nout1,Nout2)),
    assert(p(Nout1):cell(S,KS,Nout1)),
    assert(p(Nout2):cell(D,KD,Nout2)).
exec(TkS,TkD,TkSa,TkDa) :-
    TkDa is TkD + 1,
    (TkSa is TkS - 1).

r(1):state(M):- ts(S),state0(S).
state0(S) :- ts(D),state1(S,D).
state1(S,S) :- !,state0(S).
state1(S,D) :-
    cell(S,KS,_),
    cell(D,KD,_),
    state2(S,D,KS,KD).
state2(S,D,K,K) :- !,start.
state2(S,D,KS,KD) :-
    transfer(S,D,KS,KD),
    findall(R:(cell(X,Y,N)),R:(cell(X,Y,N)),M1),
    state(M1).
start:- ts(S),state0(S).

```

Fig. 5. Collective sorting $N \times N$ specification in Stochastic Prolog

A Nonmonotonic Soft Concurrent Constraint Language for SLA Negotiation

Stefano Bistarelli^{1,2} and Francesco Santini^{2,3}

¹ Dipartimento di Scienze, Università “G. D’Annunzio” di Chieti-Pescara, Italy
bista@sci.unich.it

² Istituto di Informatica e Telematica (CNR), Pisa, Italy
[stefano.bistarelli, francesco.santini]@iit.cnr.it

³ IMT - Institute for Advanced Studies, Lucca, Italy
f.santini@imtlucca.it

Abstract. We present an extension of the *Soft Concurrent Constraint* language that allows the nonmonotonic evolution of the constraint store. To accomplish this, we introduce some new operations: the *retract(c)* reduces the current store by *c*, the *update_X(c)* transactionally relaxes all the constraints of the store that deal with the variables in the set *X*, and then adds a constraint *c*; the *nask(c)* tests if *c* is not entailed by the store. We present this framework as a possible solution to the management of resources (e.g. web services and network resource allocation) that need a given *Quality of Service* (QoS). The QoS requirements of all the parties should converge, through a negotiation process, on a formal agreement defined as the *Service Level Agreement*, which specifies the contract that must be enforced. c-semirings are the algebraic structures that we use to model QoS metrics.

1 Motivations

Many real-life problems require computation mechanisms which are nonmonotonic in their nature. Consider for example an everyday scenario where clients need to reserve some resources, and service providers must allocate those resources providing also a desired *Quality of Service* (QoS). Negotiation [15] is the process by which a group of agents communicate among themselves and try to come to a mutually acceptable agreement on some matter. The means for achieving this goal consist in offering concessions and retracting proposals. When agents are autonomous and cooperation/coordination is attempted at run-time, automated negotiation represents a complex process [15]. Notice that this process is continuous because clients and providers can change their requirements during their execution.

To model and manage automated negotiation, in this paper we propose the *Nonmonotonic Soft Concurrent Constraint* (*nmsccp*) language, which extends *Soft Concurrent Constraint Programming* (*sccp*) [3, 7] in order to support the nonmonotonic evolution of the constraint store. In classical *sccp* the *tell* and *ask* agents can be equipped with a preference (or consistency) threshold which is used to determine their success, failure, or suspension: the action is enabled only if the

store is “consistent enough” w.r.t. the threshold. Since constraints can only be accumulated (via the *tell* operation), this consistency level can only monotonically decrease starting from the initial empty store: the function used to combine the constraints, i.e. the \times of the semiring, is intensive [6]. To go further, we propose some new actions that provide the user with explicit nonmonotonic operations which can be used to retract constraints from the store (i.e. *update* and *retract*), and a particular *ask* operation (i.e. *nask*), enabled only if the current store does not entail a given constraint.

The *nmsccp* language has two main difference w.r.t. classical *sccp*: *i*) the consistency level of the store can be increased by retracting constraints (i.e. it is not monotonic), and *ii*) some of the failures are transformed in suspension because of the nonmonotonicity of the store. According to *i*), we have extended the semantics of the actions to include also an upper bound on the store consistency (since it can be increased by a *retract*, for example), in order to prune also “too good” computations obtained at a given step. In this way, now we are able to model intervals of acceptability, while in *sccp* there is only a check on “not good enough” computations, i.e. decreasing too much the consistency w.r.t the lower threshold. This leads to *ii*): in *sccp* an agent fails if the resulting store is not consistent enough w.r.t. the threshold (i.e. a given semiring value or soft constraint); in *nmsccp* the same agent simply suspends waiting for a possible consistency increase of the current store, which enables the pending action.

We apply these extensions to model *Service Level Agreements (SLAs)* [2, 16] and their negotiation: soft constraints represent the needs of the agents on the traded resources and the consistency value of the store represents a feedback on the current agreement. In words, how much all the requirements are consistent among themselves, or how much the global satisfaction is being met. The thresholds on the actions are used to check this interval of preference values, and having a feedback value which is not a plain “yes or no” (i.e. true or false, as in crisp constraints) is clearly more informative. Using soft constraints (e.g. “at most *around* 10 Mbyte of bandwidth”) gives the service provider and clients more flexibility in expressing their requests w.r.t. crisp constraints (e.g. “*exactly* 10 Mbyte”), and therefore there are more chances to reach a shared agreement. Moreover, the cost model is very adaptable to the specific problem, since it is parametric with the chosen semiring, and its semantics is directly embedded in the requirement definition itself (i.e. the constraint) and in the language modeling the agent (e.g. the thresholds on the *tell* and *retract* actions).

The remainder of this paper is organized as follows. In Sec. 2 we summarize the background information. Section 3 features the nonmonotonic language, its operational semantics and how the consistency intervals are managed. In Sec. 4 we show how the language can be used to represent preference-driven negotiations. At last, Sec. 5 shows related works and Sec. 6 concludes by indicating future research directions.

Related Work on Nonmonotonic Extensions. The inspiration for this work comes from [10] and [12]: in [12] the authors present a nonmonotonic framework for *Concurrent Constraint Programming (ccp)* [22], together with its semantics. Our *nask* and *update* operations (see Sec. 3) are the soft versions of those described

in [12], while the *atell*, which adds a constraint only if it is consistent with the store, can be trivially modelled with the classical (valued) *tell* of *sccp*. A negative ask like our *nask* is described also in [21]. The idea for a fine-grained removal of constraints (the *retract* in Sec. 3) comes from [10], which describes a different nonmonotonic framework for *ccp*. Its main purpose was not to add any additional indeterminism (w.r.t. the choice operator) by keeping track of the dependencies among constraints in the same parallel computation, otherwise the nonmonotonic evolution could yield different results if executed with different scheduling policies. However, in our language we decided to allow this kind of indeterminism, since we believe it is more natural to experience this behaviour during the negotiation interactions in open systems. Other examples of nonmonotonic evolution of the constraint store in *ccp* are presented in [14], and their line of research is usually called *Linear Concurrent Constraint Programming*. Moreover, several Linda-like languages [9] contain similar primitives, as well as other languages for distributed programming such as KLAIM [17].

2 Background

Absorptive Semiring. An absorptive semiring [5] S can be represented as a $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ tuple such that: *i)* A is a set and $\mathbf{0}, \mathbf{1} \in A$; *ii)* $+$ is commutative, associative and $\mathbf{0}$ is its unit element; *iii)* \times is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. Moreover, $+$ is idempotent, $\mathbf{1}$ is its absorbing element and \times is commutative. Let us consider the relation \leq_S over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that (see [6]): *i)* \leq_S is a partial order; *ii)* $+$ and \times are monotonic on \leq_S ; *iii)* $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; *iv)* $\langle A, \leq_S \rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = \text{lub}(a, b)$ (where *lub* is the *least upper bound*). Informally, the relation \leq_S gives us a way to compare semiring values and constraints. In fact, when we have $a \leq_S b$ (or simply $a \leq b$ when the semiring will be clear from the context), we will say that *b is better than a*.

In [5] the authors extended the semiring structure by adding the notion of *division*, i.e. \div , as a weak inverse operation of \times . An absorptive semiring S is *invertible* if, for all the elements $a, b \in A$ such that $a \leq b$, there exists an element $c \in A$ such that $b \times c = a$ [5]. If S is absorptive and invertible, then, S is *invertible by residuation* if the set $\{x \in A \mid b \times x = a\}$ admits a maximum for all elements $a, b \in A$ such that $a \leq b$ [5]. Moreover, if S is absorptive, then it is *residuated* if the set $\{x \in A \mid b \times x \leq a\}$ admits a maximum for all elements $a, b \in A$, denoted $a \div b$. With an abuse of notation, the maximal element among solutions is denoted $a \div b$. This choice is not ambiguous: if an absorptive semiring is invertible and residuated, then it is also invertible by residuation, and the two definitions yield the same value.

To use these properties, in [5] it is stated that if we have an absorptive and complete semiring⁴, then it is residuated. For this reason, since all classical

⁴ If S is a absorptive semiring, then S is complete if it is closed with respect to infinite sums, and the distributivity law holds also for an infinite number of summands.

soft constraint instances (i.e. *Classical CSPs*, *Fuzzy CSPs*, *Probabilistic CSPs* and *Weighted CSPs*) are complete and consequently residuated, the notion of semi-ring division can be applied to all of them. Therefore, for all these semirings it is possible to use the \div operation as a “particular” inverse of \times ; its extension to soft constraints, defined as \ominus , can be used to (partially) remove soft constraints from the store (see next Paragraph).

Soft Constraint System. A soft constraint [6, 3] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a finite domain D , a soft constraint is a function which, given an assignment $\eta : V \rightarrow D$ of the variables, returns a value of the semiring. Using this notation $C = \eta \rightarrow A$ is the set of all possible constraints that can be built starting from S , D and V .

Any function in C involves all the variables in V , but we impose that it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ over variables x and y , is a function $c_{x,y} : V \rightarrow D \rightarrow A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$ (the *support* of the constraint, or *scope*). Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$. Note also that $c\eta$ is the application of a constraint function $c : V \rightarrow D \rightarrow A$ to a function $\eta : V \rightarrow D$; what we obtain, is a semiring value $c\eta = a$.

Given the set C , the combination function $\otimes : C \times C \rightarrow C$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ (see also [6, 3, 7]). Having defined the operation \div on semirings, the constraint division function $\ominus : C \times C \rightarrow C$ is instead defined as $(c_1 \ominus c_2)\eta = c_1\eta \div c_2\eta$ [5]. Informally, performing the \otimes or the \ominus between two constraints means building a new constraint whose support involves all the variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying or, respectively, dividing the elements associated by the original constraints to the appropriate sub-tuples. The partial order \leq_S over C can be easily extended among constraints by defining $c_1 \sqsubseteq c_2 \iff c_1\eta \leq c_2\eta$. Consider the set C and the partial order \sqsubseteq . Then an entailment relation $\vdash_{\sqsubseteq} \wp(C) \times C$ is defined s.t. for each $C \in \wp(C)$ and $c \in C$, we have $C \vdash c \iff \bigotimes C \sqsubseteq c$ (see also [3, 7]).

Given a constraint $c \in C$ and a variable $v \in V$, the *projection* [6, 3, 7] of c over $V \setminus \{v\}$, written $c \Downarrow_{(V \setminus \{v\})}$ is the constraint c' s.t. $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support. This is done by associating with each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. To treat the hiding operator of the language, a general notion of existential quantifier is introduced by using notions similar to those used in cylindric algebras. For each $x \in V$, the hiding function [3, 7] is defined as $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$.

To model parameter passing, for each $x, y \in V$ a diagonal constraint [3, 7] is defined as $d_{xy} \in C$ s.t., $d_{xy}\eta[x := a, y := b] = \mathbf{1}$ if $a = b$ and $d_{xy}\eta[x := a, y := b] = \mathbf{0}$ if $a \neq b$. Considering a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables

D , an ordered set of variables V and the corresponding structure C , then $S_C = \langle C, \otimes, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \exists_x, d_{xy} \rangle^5$ is a cylindric constraint system (“*a la Saraswat*”⁶ [7]).

3 The Language

The $retract(c)$ operation is at the basis of our nonmonotonic extension of the *sccp* language, since it permits to remove the constraint c from the current store σ . It is worth to notice that our $retract$ can be considered as a “relaxation” of the store, and not only as a strict removal of the token representing the constraint, because in soft constraints we do not have the concept of token. Thus if c (parameter of $retract$) satisfies $\sigma \sqsubseteq c$ then it can be removed, even if c is different from any other constraints previously added to σ .

To use a metaphor describing the sequence of actions, imagine to pour a liquid into and out a bowl with a spoon. The content of the bowl represents the store, and the liquid in the spoon represents the soft constraint we want to add and retract from the store; as the two liquids are mixed, we lose the identity of the added soft constraint, which can worsen the condition of the store by raising the level of the liquid in the bowl. When we want to relax the store, we remove some of the liquid with the spoon, and that corresponds to the removed constraint: the consistency is incremented because the level of the bowl is lowered. This “bowl example” is appropriate when \times is not idempotent, otherwise pouring the same constraint multiple times would not increase the liquid level.

The $update_X(c)$ primitive has been inspired by the work in [12]. It consists in a sort of “assignment” operation, since it transactionally relaxes all the constraints of the store that deal with variables in the set X , and then adds a constraint c (usually with $support = X$). This operation is variable-grained w.r.t. our $retract$, and for many applications (as ours, on SLA negotiation), it is very convenient to have a relaxation operation that is focused on one (or some) variable: the reason is that it could be required to completely renew the knowledge about a parameter (e.g. the bandwidth of the example in Sec. 4).

The $nask(c)$ operation (crisp examples are in [10, 18]) is enabled only if the current store does not entail c ; it is the negative version of ask , since it detects *absence* of information. Note that, in general, $ask(\neg c)$ is different from $nask(c)$, so it is necessary to introduce a completely new primitive. Consider for example the store $\{x \leq 10\}$: while the action $nask(x < 5)$ succeeds, $ask(x \geq 5)$ would block the computation. Consider also that the notion of $\neg c$ (i.e. the negation of a constraint) is not always meaningful with preferences based on semirings, except, for instance, for the *Boolean* semiring (i.e. $\langle \{0, 1\}, \vee, \wedge, 0, 1 \rangle$). It would be difficult to define $\neg c$ when using *Weighted* semirings [3, 6]. This operation improves the expressivity of the language, since it allows to check facts not yet derivable from the store (it can be valuable to add them), or no longer derivable

⁵ $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ respectively represent the constraints associating $\mathbf{0}$ and $\mathbf{1}$ to all assignments of domain values; in general, the \bar{a} function returns the semiring value a .

⁶ Notice that in *sccp*, algebraicity is not required, since the algebraic nature of C strictly depends on the properties of the semiring [7].

(to check if some constraints have been removed), or facts that we do not want to be implied by the store.

Given a soft constraint system as defined in Sec. 2 and any related constraint c , the syntax of agents in *nmsccp* is given in Fig. 1. P is the class of programs, F is the class of sequences of procedure declarations (or clauses), A is the class of agents, c ranges over constraints, X is a set of variables and Y is a tuple of variables.

$$\begin{aligned}
P &::= FA \\
F &::= p(Y) :: A \mid FF \\
A &::= \text{success} \mid \text{tell}(c) \succrightarrow A \mid \text{retract}(c) \succrightarrow A \mid \text{update}_X(c) \succrightarrow A \mid E \mid A \parallel A \mid \exists x.A \mid p(Y) \\
E &::= \text{ask}(c) \succrightarrow A \mid \text{nask}(c) \succrightarrow A \mid E + E
\end{aligned}$$

Fig. 1. Syntax of the *nmsccp* language.

In addition to the new operations, the other most important variation w.r.t. *sccp* is the action prefixing symbol \succrightarrow in the syntax notation, which can be considered as a general “checked” transition of the type $\rightarrow_{\phi_1}^{\phi_2}$, where $\phi_i = a_i$ (i.e. the threshold is a semiring element that summarize the consistency of the store into a plain value [7]) or $\phi_i = \phi_i$ (i.e. the threshold is a constraint, i.e. a pointwise comparison between the store and the ϕ_i constraint [7]) with $i = 1, 2$. In words, two conditions must be checked at the same time: a_1 or ϕ_1 (one of the two) will be used as a cut level to prune computations that at this point are not good enough (i.e. a lower bound), while a_2 or ϕ_2 to prune computations that are too good (i.e. an upper bound). The four possible instantiation of \succrightarrow are given in Fig. 2, i.e. $\rightarrow_{a_1}^{a_2}$, $\rightarrow_{a_1}^{\phi_2}$, $\rightarrow_{\phi_1}^{a_2}$ and $\rightarrow_{\phi_1}^{\phi_2}$ (the semantics of these checked transitions will be better explained in Sec. 3.1). Therefore, we can now model intervals of acceptability during the computation, while in classical *sccp* this is not possible: *sccp* being monotonic, since the consistency level of the store can only be decreased during the executions of the agents, it is only meaningful to prune those computations that decrease this level too much. On the other hand, in *nmsccp* there is the possibility to remove constraints from the store, and thus the level can be increased again (this leads to the absence of a fail agent). For this reason we claim the importance of checking also that the consistency level of the store will not exceed a given threshold. For instance, consider the preference as a cost for a given resource: the lower threshold of the interval will prevent us from paying that resource too much (i.e. a high cost means a low preference), while the upper threshold models a clause in the contract that forces us to pay at least a minimum price.

As in classical *sccp*, the semiring values a_1 and a_2 represent two *cut levels* that summarize the consistency of the store into a plain value. On the other hand, the constraints ϕ_1 and ϕ_2 represent a finer check of the store, since a pointwise comparison between the store and these constraints is performed.

3.1 The Operational Semantics

To give an operational semantics to our language we need to describe an appropriate transition system $\langle \Gamma, T, \rightarrow \rangle$, where Γ is a set of possible configurations, $T \subseteq \Gamma$ is the set of *terminal* configurations and $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation between configurations. The set of configurations is $\Gamma = \{\langle A, \sigma \rangle\}$, where $\sigma \in C$ while the set of terminal configurations is instead $T = \{\langle success, \sigma \rangle\}$. The transition rule for the *nmsccp* language are defined in Fig. 3.

The \rightarrow is a generic checked transition used by several actions of the language. Therefore, to simplify the rules in Fig. 3 we define a function $check_{\rightarrow} : \sigma \rightarrow \{true, false\}$ (where $\sigma \in C$), that, parametrized with one of the four possible instances of \rightarrow (**C1-C4** in Fig. 2), returns *true* if the conditions defined by the specific instance of \rightarrow are satisfied, or *false* otherwise. The conditions between parentheses in Fig. 2 claim that the lower threshold of the interval clearly cannot be “better” than the upper one, otherwise the condition is intrinsically wrong.

$$\begin{array}{ll}
 \mathbf{C1:} \frac{\rightarrow = \rightarrow_{a_1}^{a_2}}{} check(\sigma)_{\rightarrow} = true \text{ if } \begin{cases} \sigma \Downarrow_0 \not\prec_S a_2 \\ \sigma \Downarrow_0 \not\prec_S a_1 \end{cases} & \mathbf{C3:} \frac{\rightarrow = \rightarrow_{\phi_1}^{a_2}}{} check(\sigma)_{\rightarrow} = true \text{ if } \begin{cases} \sigma \Downarrow_0 \not\prec_S a_2 \\ \sigma \not\sqsupseteq \phi_1 \end{cases} \\
 \text{(with } a_1 \not\prec a_2 \text{)} & \text{(with } \phi_1 \Downarrow_0 \not\prec a_2 \text{)} \\
 \\
 \mathbf{C2:} \frac{\rightarrow = \rightarrow_{a_1}^{\phi_2}}{} check(\sigma)_{\rightarrow} = true \text{ if } \begin{cases} \sigma \not\sqsupseteq \phi_2 \\ \sigma \Downarrow_0 \not\prec_S a_1 \end{cases} & \mathbf{C4:} \frac{\rightarrow = \rightarrow_{\phi_1}^{\phi_2}}{} check(\sigma)_{\rightarrow} = true \text{ if } \begin{cases} \sigma \not\sqsupseteq \phi_2 \\ \sigma \not\sqsupseteq \phi_1 \end{cases} \\
 \text{(with } a_1 \not\prec \phi_2 \Downarrow_0 \text{)} & \text{(with } \phi_1 \not\sqsupseteq \phi_2 \text{)}
 \end{array}$$

Otherwise, within the same conditions in parentheses, $check(\sigma)_{\rightarrow} = false$

Fig. 2. Definition of the *check* function for each of the four checked transitions.

Notice that in Fig. 2 we use $\not\prec_S a_1$ instead of $\geq_S a_1$ because we can possibly deal with partial orders. Similar considerations can be done for $\not\sqsupseteq$ instead of \sqsupseteq .

Some of the intervals in Fig. 2 (**C1**, **C2** and **C3**) are checked by considering the least upper bound among the values yielded by the solutions of a *Soft Constraint Satisfaction Problem* (SCSP) [3] defined as $P = \langle C, con \rangle$ (C is the set of constraints and $con \subseteq V$, i.e. a subset the problem variables). This is called the *best level of consistency* and it is defined by $blevel(P) = Sol(P) \Downarrow_0$, where $Sol(P) = (\bigotimes C) \Downarrow_{con}$; notice that $supp(blevel(P)) = \emptyset$. We also say that: P is α -consistent if $blevel(P) = \alpha$; P is consistent iff there exists $\alpha >_S \mathbf{0}$ such that P is α -consistent; P is inconsistent if it is not consistent. In Fig. 2 **C1** checks if the α -consistency of the problem is between a_1 and a_2 .

In words, **C1** states that we need at least a solution as good as a_1 entailed by the current store, but no solution better than a_2 ; therefore, we are sure that some solutions satisfy our needs, and none of these solutions is “too good”. The semantics of these checks can easily be changed in order to model different requirements on the preference interval, e.g. to guarantee that all the solutions in the store (and not at least one) have a preference contained in the given interval.

$\mathbf{R1} \frac{check(\sigma \otimes c)_{\rightarrow}}{\langle tell(c) \rangle \rightarrow A, \sigma} \rightarrow \langle A, \sigma \otimes c \rangle$	Tell	$\mathbf{R6} \frac{\sigma \not\sqsubseteq c \quad check(\sigma)_{\rightarrow}}{\langle nask(c) \rangle \rightarrow A, \sigma} \rightarrow \langle A, \sigma \rangle$	Nask
$\mathbf{R2} \frac{\sigma \vdash c \quad check(\sigma)_{\rightarrow}}{\langle ask(c) \rangle \rightarrow A, \sigma} \rightarrow \langle A, \sigma \rangle$	Ask	$\mathbf{R7} \frac{\sigma \sqsubseteq c \quad \sigma' = \sigma \ominus c \quad check(\sigma')_{\rightarrow}}{\langle retract(c) \rangle \rightarrow A, \sigma} \rightarrow \langle A, \sigma' \rangle$	Retract
$\mathbf{R3} \frac{\langle A, \sigma \rangle \rightarrow \langle A', \sigma' \rangle}{\langle A \parallel B, \sigma \rangle \rightarrow \langle A' \parallel B, \sigma' \rangle}$ $\langle B \parallel A, \sigma \rangle \rightarrow \langle B \parallel A', \sigma' \rangle$	Parall1	$\mathbf{R8} \frac{\sigma' = (\sigma \Downarrow_{(V,X)}) \otimes c \quad check(\sigma')_{\rightarrow}}{\langle update_x(c) \rangle \rightarrow A, \sigma} \rightarrow \langle A, \sigma' \rangle$	Update
$\mathbf{R4} \frac{\langle A, \sigma \rangle \rightarrow \langle success, \sigma' \rangle}{\langle A \parallel B, \sigma \rangle \rightarrow \langle B, \sigma' \rangle}$ $\langle B \parallel A, \sigma \rangle \rightarrow \langle B, \sigma' \rangle$	Parall2	$\mathbf{R9} \frac{\langle A[x/y], \sigma \rangle \rightarrow \langle B, \sigma' \rangle}{\langle \exists x. A, \sigma \rangle \rightarrow \langle B, \sigma' \rangle}$ with y fresh Hide	
$\mathbf{R5} \frac{\langle E_j, \sigma \rangle \rightarrow \langle A_j, \sigma' \rangle \quad j \in [1, n]}{\langle \Sigma_{i=1}^n E_i, \sigma \rangle \rightarrow \langle A_j, \sigma' \rangle}$	Nondet	$\mathbf{R10} \frac{\langle A, \sigma \rangle \rightarrow \langle B, \sigma' \rangle}{\langle p(Y), \sigma \rangle \rightarrow \langle B, \sigma' \rangle} \quad p(Y) :: A \in F \text{ P-call}$	

Fig. 3. The transition system for *nmsccp*.

Here is a description of the transition rules in Fig. 3. In the **Tell** rule (**R1**), if the store $\sigma \otimes c$ satisfies the conditions of the specific \rightarrow transition of Fig. 2, then the agent evolves to the new agent A over the store $\sigma \otimes c$. Therefore the constraint c is added to the store σ . The conditions are checked on the (possible) next-step store: i.e. $check(\sigma')_{\rightarrow}$.

To apply the **Ask** rule (**R2**), we need to check if the current store σ entails the constraint c and also if the current store is consistent w.r.t. the lower and upper thresholds defined by the specific \rightarrow transition arrow: i.e. if $check(\sigma)_{\rightarrow}$ is true.

Parallelism and nondeterminism: the composition operators $+$ and \parallel are not modified w.r.t. [7]. A parallel agent (rules **R3** and **R4**) will succeed when both agents succeed. This operator is modelled in terms of *interleaving* (as in the classical *ccp*): each time, the agent $A \parallel B$ can execute only one between the initial enabled actions of A and B (**R3**); a parallel agent will succeed if all the composing agents succeed (**R4**). The nondeterministic rule **R5** chooses one of the agents whose guard succeeds, and clearly gives rise to global nondeterminism.

The **Nask** rule is needed to infer the absence of a statement whenever it cannot be derived from the current state: the semantics in **R6** shows that the rule is enabled when the consistency interval satisfies the current store (as for the *ask*), and c is not entailed by the store: i.e. $\sigma \not\sqsubseteq c$.

Retract: with **R7** we are able to “remove” the constraint c from the store σ , using the \ominus constraint division function defined in Sec. 2. According to **R7**, we require that the constraint c is entailed by the store, i.e. $\sigma \sqsubseteq c$. Notice that in [5] the division is instead always defined, but for the *nmsccp* language we decided to be able to remove a quantity c only if the store is “big” enough to permit the removal of c , i.e. we want that $a \div b$ is possible only if $a \leq_S b$. For example, consider the three weighted constraints in Fig. 4: the domain of the variable x is \mathbb{N} and the adopted semiring is instead the classical *Weighted* semiring $(\mathbb{R}^+, \min, +, +\infty, 0)$. It is possible to perform $c_2 \ominus c_1$ because $c_2 \sqsubseteq c_1$ (the c_1 function is completely dominated by c_2 for every $x \in \mathbb{N}$, and thus c_1 is better), but it is not possible to perform $c_3 \ominus c_1$ because, for $x = 1$ (for instance), $c_3(x) = 2$ is better than $c_1(x) = 4$: thus $2 \leq 4$ and the semiring division $2 \div 4$

cannot consequently be performed because of the **R7** definition. Clearly, it is also possible to completely remove a constraint as if using tokens:

Theorem 1 (Complete removal). *Given a soft constraint system C , where the semiring S is invertible by residuation and thus \ominus can be defined (see Sec. 2 and [5]), then the nmsccp agent $\langle \text{tell}(c_i) \succ \text{retract}(c_i) \succ A, \sigma_k \rangle$ is equivalent to $\langle A, \sigma_k \rangle$, for every constraint c_i , store σ_k and \succ (if enabled).*

As a sketch of the proof, the agents' equivalence comes from the properties explained in [5], i.e. $a \times b \div b = a$ always holds, given any two elements $a, b \in S$.

$$\begin{array}{ll}
c_1 : \{x\} \rightarrow \mathbb{N} \rightarrow \mathbb{R}^+ \text{ s.t. } c_1(x) = x + 3 & c_2 : \{x\} \rightarrow \mathbb{N} \rightarrow \mathbb{R}^+ \text{ s.t. } c_2(x) = 2x + 8 \\
c_3 : \{x\} \rightarrow \mathbb{N} \rightarrow \mathbb{R}^+ \text{ s.t. } c_3(x) = 2x & c_4 : \{x\} \rightarrow \mathbb{N} \rightarrow \mathbb{R}^+ \text{ s.t. } c_4(x) = x + 5 \\
c_5 : \{x\} \rightarrow \mathbb{N} \rightarrow \mathbb{R}^+ \text{ s.t. } c_5(x) = \bar{3} & c_6 : \{y\} \rightarrow \mathbb{N} \rightarrow \mathbb{R}^+ \text{ s.t. } c_6(y) = y + 1
\end{array}$$

Fig. 4. Six weighted soft constraints (notice that $c_4 = c_2 \ominus c_1$).

The semantics of **Update** rule (**R8**) [12] resembles the assignment operation in imperative programming languages: given an $\text{update}_X(c)$, for every $x \in X$ it removes the influence over x of each constraint in which x is involved, and finally a new constraint c is added to the store. To remove the information concerning all $x \in X$, we project (see Sec. 2) the current store on $V \setminus X$, where V is the set of all the variables of the problem and X is a parameter of the rule (projecting means eliminating some variables). If $X = V$, this operation finds the *blevel* of the problem defined by the store, before adding c . At last, the levels of consistency are checked on the obtained store, i.e. $\text{check}(\sigma')_{\rightarrow}$. Notice that all the removals and the constraint addition are transactional, since are executed in the same rule. Moreover, notice that the removal semantics of the *update* is quite different from that of the *retract*: the *update* operation can always be applied, while the *retract* can be applied only when $\sigma \sqsubseteq c$. In addition, performing an *update* is different from sequentially performing one (or some) *retract* and then a *tell*: the *retract* relaxes the store in a “clear” way, while the *update* “releases” one (or more) variable x by choosing the best semiring value for each constraint c supported by x (i.e. $\sigma \Downarrow_{(V \setminus \{x\})} = \sum_{d_i \in D} c\eta[x := d_i]$, where D is the domain of x). Therefore, if c is supported also by another variable y , c is somewhat still constraining y after the *update* operation. As an example of the different semantics between an *update* and a *retract-tell* sequence, the agent $\langle \text{tell}(c_5) \xrightarrow{0} \text{retract}(c_5) \xrightarrow{0} \text{tell}(c_2), \bar{0} \rangle$ (in the *Weighted* semiring $\mathbf{1} \equiv \bar{0}$) results in the store $c_5 \ominus c_5 \otimes c_2 = c_2$, while $\langle \text{tell}(c_5) \xrightarrow{0} \text{update}_{\{x\}}(c_2), \bar{0} \rangle$ results in the store $\bar{3} \otimes c_2$ (i.e. $c_5 \otimes c_2$), where $\bar{3} = c_5 \Downarrow_{(V \setminus \{x\})}$.

Hidden variables: the semantics of the existential quantifier in **R9** is similar to that described in [20] by using the notion of *freshness* of the new variable added to the store.

Procedure calls: the semantics of the procedure call (**R10**) is not modified w.r.t. the classical one: as usual, we use the notion of diagonal constraints (as defined in Sec. 2) to model parameter passing.

Given the transition system proposed in Fig. 3, we define for each agent A the set of final stores that collects the results of successful computations that A can perform (i.e. the *observables*): $\mathcal{S}_A = \{\sigma \downarrow_{\text{var}(A)} \mid \langle A, \bar{\mathbf{1}} \rangle \rightarrow^* \langle \text{success}, \sigma \rangle\}$.

No Failures. The *nmsccp* agents computation can only be successful or can suspend waiting for a change of the store in which it is possible to execute the action on which an agent is suspended on. This represents a further difference w.r.t. *sccp* where, when trying to execute a (valued or not) *ask/tell*, if the resulting level of the store consistency is lower than the threshold labeled on the transition arrow, then this is considered a failure (see [7]): in *sccp* the store consistency can only be monotonically decreased, and therefore a better level can never be reached during the successive steps. In *nmsccp*, another agent in parallel can instead perform a *retract* (or an *update*) and can consequently increase the consistency level of the store, then enabling the idle action.

Preference Representation and Operations The representational and computational issues are complex and would deserve a deep discussion [11]. However, some different considerations can be provided whether or not the language adopted to represent the constraints preference is finitary.

As a practical example of (a specific subset of) soft constraints that have a finitary representation, consider the *Weighted* semiring and consider a class of constraints whose soft preference (or cost) is represented by a polynomial expression over the variables involved in the constraints. In this case, adding a constraint to the store means to obtain a new polynomial form that is the sum of the new preference and the polynomial representing the current store; retracting a constraint means just to subtract the polynomial form from the store. Suppose we have three constraints $c_1(x, y) = x^2 - 3x + 4y$, $c_2(x) = 3x + 2$ and $c_3(y) = 3y - 2$: if the initial store contains $c_1(x, y)$, *tell*(c_2) gives $(c_1 \otimes c_2) = x^2 - 3x + 4y + 3x + 2 = x^2 + 4y + 2$, and then a *retract*(c_3) would result in the store preference $(c_1 \otimes c_2 \oplus c_3) = x^2 + 4y + 2 - (3y - 2) = x^2 + y$. To compute the result of an *update* _{y} (c_4) we need to project over $V \setminus \{y\}$ (see Sec. 2) before adding c_4 : therefore, if the store preference is $x^2 + y$, we must find the minimum of this polynomial by assigning $y = 0$ and finally obtaining $x^2 \otimes c_4 = x^2 + x + 5$ as result (see Fig. 4). Notice that in the *Weighted* semiring, to maximize the preference means to minimize the polynomial.

Otherwise, if soft constraints have not a finitary representation, we can model the store as an ordered list of constraints and actions. For examples, if the agents have chronologically performed the actions *tell*(c_1), *tell*(c_2) *retract*(c_3) and *update* _{x} (c_4), the store will be $c_1 \otimes c_2 \oplus c_3 \downarrow_{(V \setminus X)} \otimes c_4$ (whose composition is left-associative). Therefore, at each step it is possible to compute the actual store in order to verify the entailments among constraints and the consistency intervals. Thus, the actions ordering is important:

Theorem 2 (Actions ordering). *Given a soft constraint system C , where the semiring S is invertible by residuation (see Sec. 2 and [5]), the tell and retract actions of $nmsccp$ are neither commutative nor associative.*

In fact, if we suppose the \times of S as idempotent, we have the $nmsccp$ agent $\langle tell(c_i) \succ retract(c_i) \succ tell(c_i) \succ A, \sigma_k \rangle \equiv \langle A, c_i \otimes \sigma_k \rangle$, and by changing the ordering of actions it differs from $\langle tell(c_i) \succ tell(c_i) \succ retract(c_i) \succ A, \sigma_k \rangle \equiv \langle A, \sigma_k \rangle$, for every constraint c_i , store σ_k and \succ (if enabled). To prove it, we consider that for every semiring element $a \in S$, we have $a \times a \div a = \mathbf{1}$ (since $a \times a = a$, if \times is idempotent), but $a \div a \times a = a$. This is due to idempotency of \times and the properties of \div shown in [5]. Theorem 2 holds even if \times is not idempotent: for example (see the constraints in Fig. 4), $\langle tell(c_2) \succ retract(c_4) \succ success, c_1 \rangle$ successfully terminates with the store $c_1 \otimes c_2 \oplus c_4 \equiv 2x + 6$, while $\langle retract(c_4) \succ tell(c_2) \succ success, c_1 \rangle$ is suspended on the first *retract*, since the $\sigma \sqsubseteq c$ precondition of **R7** in Fig. 3 is false (here, $c_1 \sqsubseteq c_4$ is false).

This representation (i.e. keeping also the sequence of operations) differs from the classical one given by Saraswat [20] or in [8], since in these works a *retract* removes from the store only one instance of the token: $\langle tell(c_1) \rightarrow tell(c_1) \rightarrow retract(c_1) \rightarrow A, \mathbf{1} \rangle \equiv \langle A, c_1 \rangle$, even if \times is idempotent. Therefore, the ordering of the actions is useless and the store can be seen only as a set of tokens.

4 The Negotiation of Service Levels Agreement

One of the most meaningful application of the $nmsccp$ language is to model generic entities negotiating a formal agreement, i.e. a SLA [2, 16]. The main task consists in accomplishing the requests of all the agents by satisfying their QoS needs. Considering the fuzzy negotiation in Fig. 5 (Fuzzy semiring: $\langle [0, 1], max, min, 0, 1 \rangle$) both a provider and a client can add their request to the store σ (respectively $tell(c_p)$ and $tell(c_c)$): the thick line represents the consistency of σ after the composition (i.e. *min*), and the *blevel* of this SCSP (see Sec. 3.1) is the *max*, where both requests intersects (i.e. in 0.5).

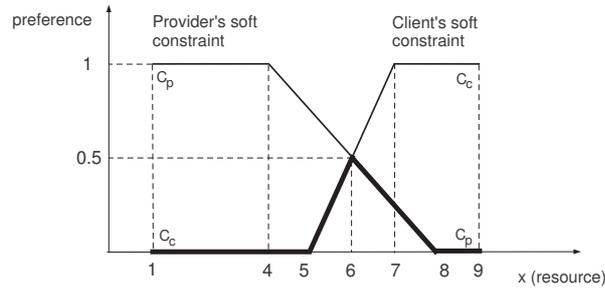


Fig. 5. The graphical interpretation of a fuzzy agreement

We present four short examples to suggest possible negotiation scenarios. We suppose to have two distinct companies (e.g. providers P_1 and P_2) that want to merge their services in a sort of pipeline, in order to offer to their clients a single structured service: e.g. P_1 completes the functionalities of P_2 . This example reminds the *cross-domain* management of services proposed in [2]. The variable x represents the global number of failures they can sustain during the service provision, while the preference models the number of hours (or a money cost in hundreds of euro) needed to manage them and recover from them. The preference interval on transition arrows models the fact that both P_1 and P_2 explicitly want to spend some time to manage the failures (the upper bound in Fig. 2), but no so much time (lower bound in Fig. 2). We will use the the *Weighted* semiring and the soft constraints given in Fig. 4. Even if the examples are based on a single criteria (i.e. the number of hours) for sake of simplicity, they can be extended to the multicriteria case, where the preference is expressed as a tuple of incomparable criteria.

Example 1 (Tell and negotiation). P_1 and P_2 both want to present their policy (respectively represented by c_4 and c_3) to the other party and to find a shared agreement on the service (i.e. a SLA). Their agent description is: $P_1 \equiv \langle tell(c_4) \rightarrow_{\infty}^0 tell(s_{p2}) \rightarrow_{\infty}^0 ask(s_{p1}) \rightarrow_{10}^2 success \rangle \| \langle tell(c_3) \rightarrow_{\infty}^0 tell(s_{p1}) \rightarrow_{\infty}^0 ask(s_{p2}) \rightarrow_4^1 success \rangle \equiv P_2$, executed in the store with empty support (i.e. $\bar{0}$). Variables s_{p1} and s_{p2} are used only for synchronization and thus will be ignored in the following considerations (e.g. replaced by the *SYNCHRO*_{*i*} agents in Ex. 2). The final store (the merge of the two policies) is $\sigma = (c_4 \otimes c_3) \equiv 2x + x + 5$, and since $\sigma \Downarrow_0 = 5$ is not included in the last preference interval of P_2 (between 1 and 4), P_2 does not succeed and a shared agreement cannot be found. The practical reason is that the failure management systems of P_1 need at least 5 hours (i.e. $c_4 = x + 5$) even if no failures happen (i.e. $x = 0$). Notice that the last interval of P_2 requires that at least 1 hour is spent to check failures.

Example 2 (Retract). After some time (still considering Ex. 1), suppose that P_1 wants to relax the store, because its policy is changed: this change can be performed from an interactive console or by embedding timing mechanisms in the language as explained in [4]. The removal is accomplished by retracting c_1 , which means that P_1 has improved its failure management systems. Notice that c_1 has not ever been added to the store before, so this retraction behaves as a relaxation; partial removal, which cannot be performed with tokens (see Sec 5), is clearly important in a negotiation process. $P_1 \equiv \langle tell(c_4) \rightarrow_{\infty}^0 SYNCHRO_{P1} \rightarrow_{10}^2 retract(c_1) \rightarrow_{10}^2 success \rangle \| \langle tell(c_3) \rightarrow_{\infty}^0 SYNCHRO_{P2} \rightarrow_4^1 success \rangle \equiv P_2$ is executed in $\bar{0}$. The final store is $\sigma = c_4 \otimes c_3 \oplus c_1 \equiv 2x + 2$, and since $\sigma \Downarrow_0 = 2$, both P_1 and P_2 now succeed (it is included in both intervals).

Example 3 (Nask). In a negotiation scenario, the *nask* operation can be used for several purposes. Since it checks the absence of information (see Sec. 3), for example it can be used to check if the own policy is still implied by the store or if it has been relaxed too much: e.g. $P_1 \equiv \langle retract(c_1) \rightarrow_{\infty}^0 SYNCHRO_{P1} \rightarrow_{\infty}^0 success \rangle \| \langle tell(c_4) \rightarrow_{\infty}^0 nask(c_4) \rightarrow_{\infty}^0 tell(c_4) \rightarrow_{\infty}^0 SYNCHRO_{P2} \rightarrow_{\infty}^0 success \rangle \equiv P_2$

(evaluated in $\bar{0}$). As soon as P_2 adds its policy (i.e. c_4), P_1 can relax it (by removing c_1); P_1 perceives this relaxation with the *nask* and adds again c_4 . The reason is that P_1 explicitly needs a global number of spent hours not better than that one defined by c_4 , which then must be entailed by the store: e.g. its recovery system works only with at least that time. Here the preference intervals of the two agents are not significative, since equal to the whole \mathbb{R}^+ .

Example 4 (Update). The *update* can be instead used for substantial changes of the policy: for example, suppose that $P_1 \equiv \langle \text{tell}(c_1) \rightarrow_{\infty}^0 \text{update}_{[x]}(c_6) \rightarrow_{\infty}^0 \text{success}, \bar{0} \rangle$. This agent succeeds in the store $\bar{0} \otimes c_1 \Downarrow_{(V \setminus \{x\})} \otimes c_6$, where $c_1 \Downarrow_{(V \setminus \{x\})} = \bar{3}$ and $\bar{3} \otimes c_6 \equiv y + 4$ (i.e. the polynomial describing the final store). Therefore, the first policy based on the number of failures (i.e. c_1) is updated such that x is “refreshed” and the new added policy (i.e. c_6) depends only on the y number of system reboots. The consistency level of the store (i.e. the number of hours) now depends only on the y variable of the SCSP. Notice that the $\bar{3}$ component of the final store derives from the “old” c_1 , meaning that some fixed management delays are included also in this new policy.

5 Related Work

Nonmonotonicity has been extensively studied for crisp constraints in the so-called *linear cc* programming [19] and in following works as [1, 10, 12, 18]. Regarding related SLA negotiation models, the process calculus introduced in [13] is focused on controlling and coordinating distributed process interactions while respecting QoS parameters expressed as c-semiring values; however, the model does not cover negotiation. In [2] and [16] the authors define SLAs at a lower level of abstraction and their description is separated from their negotiation (while soft constraint systems cover both cases).

The most direct comparison for *nmsccp*, since the two languages are both used for SLA negotiation, is with the work in [8], in which soft constraints are combined with a name-passing calculus (even if all the examples in the paper are then developed using crisp constraints). However, w.r.t our language there are some important differences: *i)* in *nmsccp* we do not have the concept of constraint token and it is possible to remove every c that is entailed by the store (i.e. $\sigma \sqsubseteq c$), even if c is syntactically different from all the c previously added (as the retraction of c_1 in Ex. 2). For example, even the removal of the $c_1 \otimes c_2$ composition from a store containing both c_1 and c_2 cannot be performed in [8], because it is a derived constraint. Therefore our *retract* is more like a “relaxation” operation, and not a “physical” removal of a token as in [8]; this feature is in the nature of negotiation, when a step back must be taken to reach a shared agreement.

Then, *ii)* with *nmsccp* we can reach a final agreement among the parties, knowing also “how consistently” (or “how expensively”) the claimed needs are being satisfied. This is accomplished by checking the preference level of the store and the consistency intervals conditioning the actions (Fig. 2). In this way, each of the agents can specify its desired preference for the final agreement. This is a

relevant improvement w.r.t. [8], where the final store collects all the consistent solutions without any distinction, i.e. each solution that satisfies $\sigma \Downarrow_0 = \alpha_i$, for every $\alpha_i >_S \mathbf{0}$.

At last, *iii*) we introduced the *update* operation (extending the semantics of the crisp *update* in [12]), which is a variable-grained relaxation, and the *nask* (whose crisp version is in [12]), that is very useful to have in a nonmonotonic framework to check absence of information. Notice that we do not need the *check* operation defined in [8] in order to verify if a given constraint is consistent with the store (without adding it). The reason is that we have the checked transitions of Fig. 2 to prevent the store from becoming not consistent “enough”.

6 Conclusions and Future Work

Monotonicity is one of the major drawbacks for practical use of concurrent constraint languages in reactive and open systems. In this paper we have proposed some new primitives (*nask*, *update* and *retract*) that allow the nonmonotonic evolution of the store. We have chosen to extend *sccp* because soft constraints [3, 6] enhance the classical constraints in order to represent consistency levels, and to provide a way to express preferences, fuzziness, and uncertainty. We think that having preference values directly embedded in the language represents a valuable solution to manage SLA negotiation, particularly when a given QoS is associated with the resources. Soft constraints can be used to model different problems by only parameterizing the semiring structure.

We are currently extending the language with timing mechanisms such as “timeout” and “interrupt” to further improve the expressiveness of the language [4]. These capabilities can be useful during complex interactions, e.g. to interrupt a long wait for pending conditions (or to interrupt a deadlock) or to trigger urgent actions.

Moreover, we would like to investigate the possibility of a distributed store instead of the centralized one we have assumed in this paper. In distributed CSP [23], variables and constraints are distributed among all the agents, thus the knowledge of the problem is not concentrated in a single agent only. This requirement is common in many practical application, and surely for (SLA) negotiating entities, where each agent has a private store collecting its resources (i.e. variables) and policies (i.e. constraints).

At last, we plan to provide the language with other formal tools, such as a denotational semantics, a study on agent equivalences in order to prove when two providers offer the same service. Moreover, we want to deepen the absence of failures in *nmsccp*.

References

1. E. Best, F. S. de Boer, and C. Palamidessi. Partial order and sos semantics for linear constraint programs. In *COORDINATION '97*, pages 256–273. Springer, 1997.
2. P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. *Comput. Networks*, 35(1):5–24, 2001.

3. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of LNCS. Springer, 2004.
4. S. Bistarelli, M. Gabrielli, M. C. Meo, and F. Santini. Timed concurrent constraint programs. In *To appear in COORDINATION '08*, LNCS. Springer, 2008.
5. S. Bistarelli and F. Gadducci. Enhancing constraints manipulation in semiring-based formalisms. In *European Conference on Artificial Intelligence (ECAI)*, pages 63–67, 2006.
6. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
7. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
8. M. G. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *ESOP'07*, volume 4421 of LNCS, pages 18–32. Springer, 2007.
9. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing three semantics for linda-like languages. *Theor. Comput. Sci.*, 240(1):49–90, 2000.
10. P. Codognet and F. Rossi. NMCC programming: Constraint enforcement and retracting in CC programming. In *ICLP'95*, pages 417–431, 1995.
11. D. Cohen, M. Cooper, P. Jeavons, and A. Krokhin. The complexity of soft constraint satisfaction. *Artif. Intell.*, 170(11):983–1016, 2006.
12. F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *ILPS*, pages 315–334, 1993.
13. R. De Nicola, G. L. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for QoS-aware applications. In *COORDINATION'95*, volume 3454 of LNCS, pages 33–48. Springer, 2005.
14. F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Inf. Comput.*, 165(1):14–41, 2001.
15. N. Jennings, P. Faratin, A. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: prospects, methods and challenges. *Int Journal of Group Decision and Negotiation*, 10(2):199–215, 2001.
16. A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11(1):57–81, 2003.
17. R. De Nicola, G. Luigi Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A formal basis for reasoning on programmable qos. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of LNCS, pages 436–479. Springer, 2003.
18. V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *POPL '95*, pages 272–285. ACM Press, 1995.
19. V. Saraswat and P. Lincoln. Higher-order Linear Concurrent Constraint Programming. Technical report, Xerox Parc, 1992.
20. V. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90*, pages 232–245. ACM Press, 1990.
21. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *POPL '95: Proceedings of Principles of Programming Languages*, pages 272–285, San Francisco, California, January 22–25, 1995. ACM Press.
22. V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM Press.
23. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.

Modeling and selecting countermeasures using CP-net and Answer Set Programming

Stefano Bistarelli^{1,2}, Fabio Fioravanti¹, Pamela Peretti¹, and Irina Trubitsyna³

¹ Dipartimento di Scienze, Università “G. d’Annunzio”, Pescara, Italy
`{bista,fioravanti,peretti}@sci.unich.it`

² Istituto di Informatica e Telematica, CNR, Pisa, Italy
`Stefano.Bistarelli@iit.cnr.it`

³ DEIS Università della Calabria, Rende, Italy
`irina@deis.unical.it`

Abstract. In this paper, we present CP-defense trees for modelling security scenarios and for expressing qualitative preferences over attacks and countermeasures, and we show how to select the set of preferred countermeasures able to protect a system by translating CP-defense trees to *Answer Set Optimization* programs which contains preferences among attacks and countermeasures. By computing the optimal answer set of the ASO program corresponding to the CP-defense tree we are able to automatically select the set of preferred countermeasure able to mitigate all the vulnerabilities in the modeled security scenario.

1 Introduction

Providing an adequate level of protection to an enterprise’s IT assets is becoming increasingly important. As a consequence, security spending constitutes a conspicuous part of an enterprise budget for IT. In order to focus the real and concrete threats that could affect an enterprise’s assets, a risk management process is needed in order to identify, describe and analyze the possible vulnerabilities that must be eliminated or reduced. The final goal of the process is to make security managers aware of the possible risks, and to guide them towards the adoption of a set of countermeasures which can bring the overall risk under an acceptable level, while minimizing the total cost of the security investment.

An instrument that can be used to determine the possible attacks that can harm a system, and the necessary countermeasures are *defense trees*. Defense trees, DT [1], an extension of attack trees [12], have been introduced as a methodology for the analysis of attack/defense security scenarios. A DT is an *and/or* tree, where leaf nodes represent the vulnerabilities and the set of countermeasures available for their mitigation, **and**-nodes represent attacks composed of a set of actions that have to be performed as a whole to damage the system, **or**-nodes represent attacks composed of a set of alternative actions that damage the system. Notice that to defeat **and** attacks it is enough to patch one of the vulnerabilities (by selecting a single countermeasure), whilst to stop **or** attacks, one countermeasure for each of the actions composing the attack has to be selected.

The overall goal is to use the defense tree representation of attacks and countermeasures for the selection of the *best* set of countermeasures (w.r.t. specific preference criteria such as cost, efficiency, etc.), that can stop all the attacks to the system.

Several works in literature [8, 11] address this problem by following a quantitative approach: the security manager must provide economic quantitative estimates to attacks and countermeasures which are then used to compute the cost-optimal set of countermeasures. However, the use of quantitative evaluations in the analysis of an IT system is often very difficult and expensive because many factors can influence the attacks and the selection of countermeasures. In most cases such a quantitative evaluation is not possible or reliable, and as a consequence security managers tend to take decisions according to their experience and intuition alone, without following any kind of structured process.

In these situations, it seems reasonable to provide qualitative estimates, which are usually easier to elicit, rather than quantitative ones. In [2] a method has been proposed to model preferences over attacks and countermeasures using CP-nets, but no implementation was provided, making the method unfeasible for big and complex scenarios.

The preference among countermeasures and the dependency between attacks and countermeasures are modeled in [6] by an *answer set optimization* (ASO) program. The **and** and **or** composition of a branch is then obtained by a syntactic composition of the ASO programs, whose semantics completely respects the intended meaning given in [2]. The semantics of the obtained ASO program provides a set of ordered answer sets representing the ordered sets of countermeasure to be adopted. In order to deal with ordered attacks (from the more to the less dangerous), the model is extended by introducing a corresponding rank (meta-preferences) among the preference rules of an ASO program. The introduction of this kind of meta-preferences allows us to prefer the adoption of countermeasures covering the more dangerous of the attacks.

The paper is structured as follows: we introduce CP-defense trees in Sec. 2. In Sec. 3 we propose two different methods to compose preferences: the **and**-composition and the **or**-composition. In Sec. 4 we present a method for translating - under suitable conditions - the CP-defense tree into an Answer Set Optimization (ASO) program.

2 CP-defense trees

The *CP-defense tree* structure allows a system administrator to determine, in a qualitative manner, the attack strategies that an attacker can follow to damage a system, the different actions that compose each attack and the security measures that can be introduced into the system. CP-defense trees are based on two different instruments: the *defence tree* and the *CP-network*.

Defence tree [1] are an instrument that can be used to determine and to graphically represent the possible attacks that can harm the assets of an IT system,

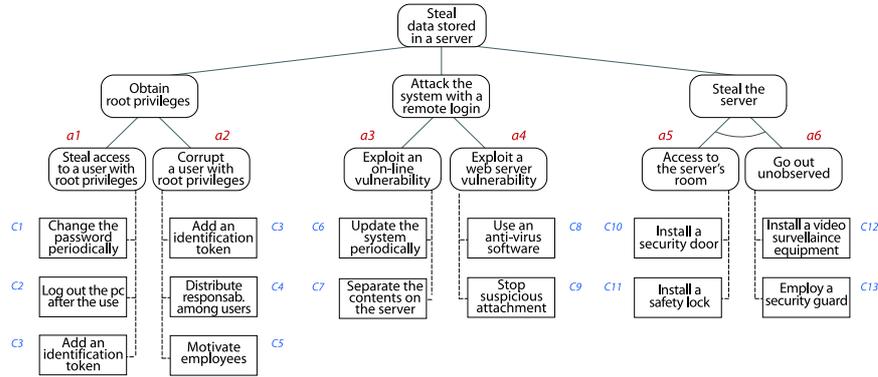


Fig. 1. An example of defense tree.

and the necessary countermeasures. They are an extension of attack trees [12], a formal way of describing how attacks against a system can be performed. An attack tree is built as follows:

- the *root* of the tree is associated with an asset of the IT system under consideration,
- *leaf nodes* represent simple subgoals which lead the attacker to (partially) damage the asset by exploiting a single vulnerability,
- *non-leaf nodes* (including the tree root) represent attack subgoals and can be of two different types: **or-nodes**: are used to represent subgoals that are completed as soon as any of its child nodes is achieved; **and-nodes**: are used to represent subgoals which require all of its child nodes to be completed. In the following we draw an horizontal line across the arcs connecting an **and**-node to its children to distinguish it from an **or**-node.

Each path from leaf nodes to the root ending in an achieved subgoal represents a different attack strategy in the considered scenario.

A defense tree is built by labeling each leaf node of an attack tree with a set of possible countermeasures able to stop that particular attack action. The attack tree is represented by using round nodes and solid edges, then it is enriched by labeling each leaf node with a set of possible countermeasures, represented by square nodes and connected by means of dotted lines.

Fig. 1 shows an example of a defense tree to model an attack/defense scenario for an enterprise's server used to store information about customers: rounded-box nodes denote the attack strategies and the different actions the attacker needs to perform, while square box nodes denote the different countermeasures the system administrator can adopt.

CP-networks [3–5] are a graphical formalism for specifying and representing qualitative conditional preference relations.

CP-nets capture *ceteris paribus* preference statements like, for example, “I prefer red wine to white wine if vegetable soup is served”. This preference statement

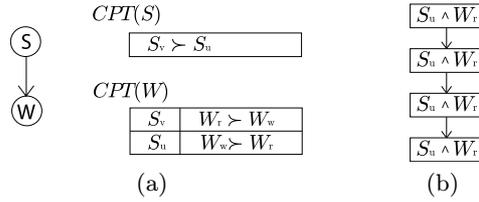


Fig. 2. An example of CP-net and the corresponding induced preference graph.

means that, given two meals both containing a vegetable soup, the meal with red wine is preferred to the meal with white wine, *all else being equal*.

The conditional *ceteris paribus* semantics, adopted in CP-nets, requires that for each variable x in the variable set V , a user specifies the parent variable $Pa(x)$ that can affect his preferences over the values of x . This information is used to create the CP-net graph in which each node x has $Pa(x)$ as its immediate predecessor. For instance, in the example above the variable set V consists of two variables S and W , representing the soup and the wine choice respectively. $Pa(W) = S$ as the assignment of S (the choice of soup) can impact the preference over the values of W (wine choice).

So, given a particular value assignment to $Pa(x)$, the user can determine a preference order over the domain of x , denoted as $\mathcal{D}(x)$, all *other things being equal*. This conditional preference over the values of X is collected in a *conditional preference table* (CPT). For each assignment to $Pa(X)$, $CPT(X)$ specifies a strict partial order over $\mathcal{D}(X)$. Notice that the right side of a conditional preference table is composed of a set of admissible values and by a partial order over these values. When the set is not specified, we assume that it is composed only of the values used in the partial order. Thus in the example above the conditional table associated with variable W contains only one line $S_v : W_r \succ W_w$, representing the preference of red wine (W_r) to white wine (W_w) in the case of the vegetable soup (S_v).

Example 1. Consider a preference specification described by the following sentences: “I prefer vegetable soup to another type of soup”, “I prefer red wine to white wine if vegetable soup is served” and “If another type of soup is served the white wine is preferred to the red wine”. This preference specification extends the above-discussed one and can be modeled by using a CP-net presented in Fig. 2(a). The variable set is $V = \{S, W\}$, $D(S) = \{S_v, S_u\}$, $D(W) = \{W_r, W_w\}$, where S_v and S_u represent the vegetable and the another type of soup, while W_r and W_w state for red and white wine. $Pa(W) = S$ because the type of soup influences the selection of wine. The $CPT(S)$ and $CPT(W)$ reports the preference over soup choice, described by the first sentence, and wine choice, described by the two latter sentences, respectively. \triangle

The CP-net can be used to build an *induced preference graph* [4], that is an acyclic directed graph where the nodes correspond to the complete assignments of the variables of the CP-net, and there is an edge from node o' to node o if and only if the assignments at o' and o differ only in the value of a single variable X and o is preferred to o' . For instance, Fig. 2(b) reports the induced

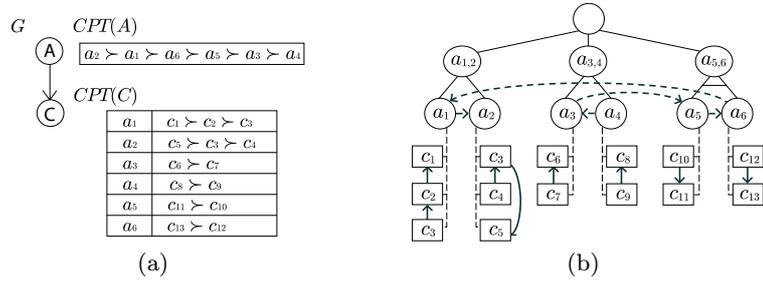


Fig. 3. A CP-defense tree.

preference graph corresponding to the CP-net presented in Ex. 1. It shows that the assignment $S_u \wedge W_r$, corresponding to the root of the induced preference graph, is the less preferred assignment, while the assignment $S_v \wedge W_r$ is the most preferred assignment.

CP-defense trees. The integration of defense trees and CP-nets has been recently proposed in [2] as a methodology to help the system administrator to analyze a security scenario and to give him a model to represent preferences among countermeasures. The resulting structure, called *CP-defense tree*, extends the defense tree representation with preference specification.

The idea is to use the defense tree representation for the selection of the best set of countermeasures (w.r.t. specific preference criteria such as cost, efficiency, etc.) that can stop all the attacks to the system. To guide this selection, a CP-net structure, able to model preferences over attacks and countermeasures can be used.

As an example consider the integration of the defense tree of Fig. 1 and the CP-net of Fig. 3(a). The variables A and C represent the (attack) action and the countermeasure selection respectively. The preference over countermeasures is conditioned by the type of action, so $Pa(C) = A$. $CPT(A)$ describes the preferences over actions $a_i \in D(A)$, ordered according to the impact produced on the system. For instance, the system administrator prefers to protect from the action a_2 rather than from the action a_1 because action a_2 is more dangerous than action a_1 . $CPT(C)$ collects the preference among countermeasures $c_k \in D(C)$ for each type of action. The reason for the preference specification can be different: efficiency, price, etc. In this example, for a given attack action a the application of the countermeasure c is preferable to c' when c is less expensive than c' .

The corresponding CP-defense tree is presented in Fig. 3(b).

Observe, that the preference order over actions described in $CPT(A)$ is represented with dotted arrows, while conditional preferences over countermeasures described in $CPT(C)$ are represented by using solid arrows. The arrows are directed from the less preferred to the more preferred outcome.

The final goal of a system administrator is to determine the best defense strategy. Thus, he has to derive the minimal sets of countermeasures able to stop all actions and the partial order among them. In particular, using the preference

tables $CPT(C)$ and $CPT(A)$, he has to combine the preference orders among countermeasures, described in $CPT(C)$, and obtain a new preference order over sets of countermeasures able to stop all actions.

3 Composition of preferences

The composition of preference orders can be performed by using two composition operations corresponding to the cases of **and**-attack and **or**-attack.

3.1 and-composition

An **and**-attack is an attack composed of a set of actions that an attacker has to successfully achieve to obtain his goal. To protect the system from this type of attack the system administrator can select a countermeasure for any of the actions composing the **and**-attack. In fact, it is enough to stop one of the actions to stop the attack.

In the following we say that a set of countermeasures S covers an **and**-attack A if there exists a countermeasure $c \in S$ covering at least an attack action $a \in A$.

Given an **and**-attack, denoted $u_1 \wedge \dots \wedge u_n$, composed of n actions u_1, \dots, u_n , with sets of countermeasures C_{u_1}, \dots, C_{u_n} respectively, the minimal sets of countermeasures able to cover all actions ($u_1 \wedge \dots \wedge u_n$) have just one countermeasure from $C_{u_1} \cup \dots \cup C_{u_n}$. The new partial order $\succ_{u_1 \wedge \dots \wedge u_n}$, describing the preference among countermeasures for $u_1 \wedge \dots \wedge u_n$, can be obtained by combining the orders $\succ_{u_1}, \dots, \succ_{u_n}$ describing the preference orders among countermeasures associated with actions u_1, \dots, u_n respectively.

The **and**-composition operator \wedge , modelling this situation, is defined as follows:

Definition 1. (and-composition) Let $\mathcal{N} = (V, E)$ be a CP-net, $V = \{x_1, \dots, x_n\}$ be a set of variables, and u_1, \dots, u_k be instantiations of the variable $x_j = Pa(x_i)$.

The **and**-composition $\wedge(\succ_{u_1}^i, \dots, \succ_{u_k}^i)$ of the partial orders $\succ_{u_1}^i, \dots, \succ_{u_k}^i$, described by $CPT(x_i)$, is a new partial order $\succ_{u_1 \wedge \dots \wedge u_k}^i$ among elements in $D(x_i)$, such that $\forall a, b \in D(x_i)$:

$$a \succ_{u_1 \wedge \dots \wedge u_k}^i b \iff \begin{cases} \exists j \in [1..k] \text{ s.t. } a \succ_{u_j}^i b \text{ or} \\ \exists j, l \in [1..k] \text{ s.t. } \forall x, y : a \not\succeq_{u_j}^i x \text{ and } y \not\succeq_{u_l}^i b \text{ and } u_j \succ u_l. \end{cases}$$

For the sake of simplicity, let consider now the **and**-composition of two partial orders. Suppose $\mathcal{N} = (V, E)$ be a CP-net, with $V = \{A, C\}$ and $Pa(C) = A$, describing the preferences of CP-defense tree. Suppose also that $u_1, u_2 \in D(A)$ are two attack actions. The **and**-composition operator can be applied to the partial orderings \succ_{u_1} and \succ_{u_2} described in $CPT(C)$ in order to obtain a new partial order $\succ_{u_1 \wedge u_2}$ among countermeasures. In particular, given two countermeasures $a, b \in D(C)$, the countermeasure a is preferred to the countermeasure b in $\succ_{u_1 \wedge u_2}$, if

- a is preferred to b in, at least, one of the partial orders \succ_{u_1}, \succ_{u_2} ;

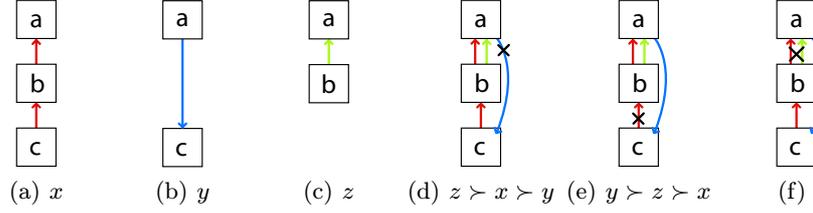


Fig. 4. An example of **and**-composition.

- a is a minimal (i.e. one of the worst) countermeasures in \succ_{u_1} , b is a maximal (i.e. one of the best) countermeasures in \succ_{u_2} and the attack u_1 is more dangerous than u_2 , i.e. $u_1 \succ u_2$.

Thus, the **and**-composition preserves the partial orderings among the countermeasures, corresponding to each attack action and introduces the *bridge preference relation*, connecting the corresponding orderings, by considering the preferences over the values of the parent variable $A = Pa(C)$: if $u_1 \succ u_2$ then the countermeasures able to mitigate the risk of u_1 are preferable to the countermeasures able to mitigate the risk of u_2 .

As an example consider the **and**-attack $x \wedge y$, in the case than $CPT(C) = \{x : a \succ b \succ c, y : b \succ c\}$ and $CPT(A) = \emptyset$. The preference order $\succ_{x \wedge y} = \wedge(\succ_x, \succ_y) = \{a \succ b \succ c\}$. The countermeasure a is the best choice for this situation.

Suppose now that $CPT(C) = \{x : a \succ b, y : c \succ d\}$, whereas $CPT(A) = \{y \succ x\}$. Observe that in this case the actions have disjoint sets of countermeasures. By applying the bridge preference relation $d \succ a$ we obtain the preference order $\succ_{x \wedge y} : c \succ d \succ a \succ b$, so c is the best countermeasure for $x \wedge y$.

It should be noted that, by applying the **and**-composition operator, we may introduce cycles in the induced preference graph. There are two possible solutions to this problem:

1. if there is some preference order between the parents $Pa(x_i)$ of the variable x_i , then we can use it to delete some edges from the induced preference graph. For example, if we know that an action is more dangerous than another one, we can say that the countermeasures for the first action are preferable;
2. if two actions are not comparable then there is no preference order between the assignment of $Pa(x_i)$. In this case we can use some algorithms like, for example the Floyd's algorithm [9] for remove cycles from the induced preference graph (by randomly removing an edge).

The following example shows how to use both these solutions.

Example 2. Consider a set of (attack) actions $A = \{x, y, z\}$, a set of countermeasures $C = \{a, b, c\}$ and $CPT(C) = \{x : a \succ b \succ c, y : c \succ a, z : a \succ b\}$, describing the partial orders \succ_x, \succ_y and \succ_z depicted in Fig. 4(a), 4(b) and 4(c) respectively.

The **and**-composition $\wedge(\succ_x, \succ_y, \succ_z)$ returns the preference order among the countermeasures able to stop the **and**-attack $x \wedge y \wedge z$: $\succ_{x \wedge y \wedge z} = \{a \succ b \succ c \succ a\}$, which introduces a cycle in the graph. If a preference order over actions exists, we can use

it to delete some edges from the network. For instance, if we know that z , is more dangerous than x and that x is more dangerous than y ($z \succ x \succ y$), we can remove the less preferred edge (a, c) , generating the cycle. Consequently, we obtain the preference order $\succ_1 = \{a \succ b \succ c\}$, presented in Fig. 4(d). Vice-versa, if we know that the action y is more dangerous than z and that z is more dangerous than x ($y \succ z \succ x$), we prefer the countermeasure associated with the actions y . Thus, we can delete the edge (c, b) , as shown in Fig. 4(e), and obtain a different order: $\succ_2 = \{c \succ a \succ b\}$. If there is no preference relation between actions, we have to randomly choose and delete an edge for removing the cycle. In this way we could obtain another order of preference like, for instance, $\succ_3 = \{b \succ c \succ a\}$, reported in Fig. 4(f). \triangle

3.2 or-composition

An **or-attack** is an attack that can be performed with different and alternative actions: the attacker can complete successfully any of its actions to obtain his goal. To protect the system from this type of attack, the system administrator has to select one countermeasure for each of the actions composing the **or-attack**.

In the following we say that a set of countermeasures S *covers an or-attack* A if for each action $a \in A$ there exists a countermeasure $c \in S$ covering such attack.

Example 3. Consider an **or-attacks** A composed of three actions x, y and z . Suppose that $CPT(C) = \{x : a \succ b \succ c, y : c \succ a$ and $z : a \succ b\}$. The set $\{a\}$ covers the **or-attack** A because it is able to mitigate the risk associated with actions x, y and z . The set $\{b\}$, on the contrary, doesn't cover A because it is not able to protect the system from the attack action y . \triangle

Given an **or-attack** $u_1 \vee \dots \vee u_k$, composed of k actions u_1, \dots, u_k , with sets of countermeasures C_1, \dots, C_k respectively, the sets of countermeasures able to cover all actions u_1, \dots, u_k can be derived by considering the combinations of countermeasures from C_1, \dots, C_k . The partial order among these sets can be obtained by taking into account (i) the minimality condition (redundant countermeasures should be avoided) and (ii) the partial orders $\succ_{u_1}, \dots, \succ_{u_k}$ describing the preference orders among countermeasures associated with actions u_1, \dots, u_k respectively.

Definition 2. (or-composition) Let $\mathcal{N} = (V, E)$ be a CP-net, where $V = \{x_1, \dots, x_n\}$ is a set of variables, u_1, \dots, u_k be instantiations of the variable $x_j = Pa(x_i)$ and $\succ_{u_1}^i, \dots, \succ_{u_k}^i$ be partial orders, described by $CPT(x_i)$, over the sets $C_1, \dots, C_k \subseteq D(x_i)$ respectively.

We say that the **or-composition** of the partial orders $\succ_{u_1}^i, \dots, \succ_{u_k}^i$ denotes a partial order \succ^t over tuples $\langle c_1, \dots, c_k \rangle$ of $C_1 \times \dots \times C_k$ defined as follows: $\forall \bar{c} = \langle c_1, \dots, c_k \rangle, \bar{c}' = \langle c'_1, \dots, c'_k \rangle$

$$\bar{c} \succ^t \bar{c}' \iff \exists j \in [1..k], c_j \succ_{u_j}^i c'_j \text{ and } \forall h \in [1..k] \text{ s.t. } h \neq j, (c'_h \not\succeq_{u_h}^i c_h \text{ or } u_j \succ u_h)$$

Let us denote by $Set(\bar{c})$ the set of countermeasures occurring in a tuple \bar{c} , and let us denote by $Cm(S)$ the set of tuples of countermeasures \bar{c} s.t. $Set(\bar{c}) = S$.

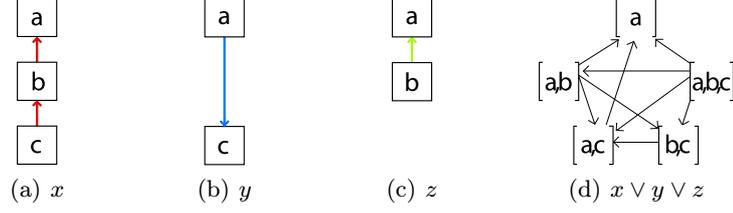


Fig. 5. An example of or-composition.

The or-composition $\vee(\succ_{u_1}^i, \dots, \succ_{u_k}^i)$ of the partial orders $\succ_{u_1}^i, \dots, \succ_{u_k}^i$ denotes a partial order $\succ_{u_1 \vee \dots \vee u_k}^i$ over sets of countermeasures defined as follows:

$$S \succ_{u_1 \vee \dots \vee u_k} S' \iff \begin{cases} S \subset S' \text{ or} \\ \exists \bar{c} \in Cm(S), \bar{c}' \in Cm(S') \quad \bar{c} \succ^t \bar{c}' \\ \text{and } \forall \bar{d} \in Cm(S), \bar{d}' \in Cm(S') \quad \bar{d}' \not\succeq^t \bar{d} \end{cases}$$

The following example shows how the or-composition operator works.

Example 4. Consider an or-attack $A = x \vee y \vee z$ and $CPT(C) = \{x : a \succ b \succ c, y : c \succ a, z : a \succ b\}$, describing the partial orders \succ_x, \succ_y and \succ_z depicted in Fig. 5(a), 5(b) and 5(c) respectively. We want to determine the preference order over sets of countermeasures covering the attack $x \vee y \vee z$. First of all we have to determine the possible tuples of countermeasures $\langle c_x, c_y, c_z \rangle$, those components c_x, c_y and c_z cover the actions x, y and z respectively. Then we derive the sets of countermeasures, by removing the duplicate elements from these tuples:

$$\begin{aligned} \{a\} &\leftarrow \langle a, a, a \rangle \\ \{a, b\} &\leftarrow \langle b, a, a \rangle, \langle a, a, b \rangle, \langle b, a, b \rangle \\ \{a, c\} &\leftarrow \langle a, c, a \rangle, \langle c, c, a \rangle, \langle c, a, a \rangle \\ \{b, c\} &\leftarrow \langle b, c, b \rangle, \langle c, c, b \rangle \\ \{a, b, c\} &\leftarrow \langle b, c, a \rangle, \langle a, c, b \rangle, \langle c, a, b \rangle \end{aligned}$$

Now we have to compare these sets: the set $\{a\}$ is preferable to the sets $\{a, b\}$ and $\{a, c\}$ because $\{a\} \subset \{a, b\}$, $\{a\} \subset \{a, c\}$; the sets $\{a\}$, $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$ are preferable to the set $\{a, b, c\}$ because they are proper subsets of it; the set $\{b, c\}$ is preferable to $\{a, b\}$ because $\langle b, c, a \rangle$ is preferable to $\langle b, a, a \rangle$ and no tuple in $Cm(\{a, b\})$ is preferable to a tuple in $Cm(\{b, c\})$; the set $\{a, c\}$ is preferable to $\{b, c\}$ because $\langle a, c, a \rangle$ is preferable to $\langle b, c, b \rangle$ and no tuple in $Cm(\{b, c\})$ is preferable to a tuple in $Cm(\{a, c\})$; the set $\{a, c\}$ is preferable to $\{a, b\}$ because $\langle a, c, a \rangle$ is preferable to $\langle b, a, a \rangle$ and no tuple in $Cm(\{a, b\})$ is preferable to a tuple in $Cm(\{a, c\})$. Fig. 5(d) shows the corresponding induced preference graph, where the set $\{a\}$ is the preferable solution for the or-attack $x \vee y \vee z$. \triangle

4 From CP-defense trees to ASO programs

The use of CP-defense trees and the methodology proposed in [2] aims to find best set of countermeasures able to stop all the attacks to the system by compos-

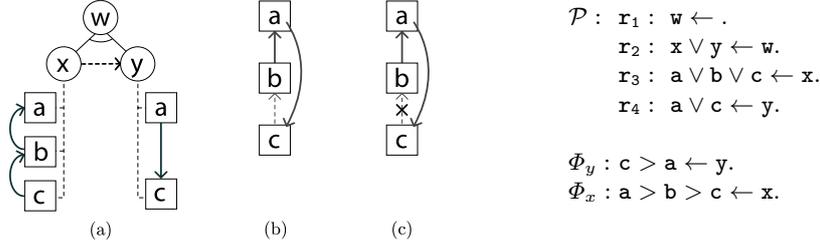


Fig. 6. An example of **and** attacks (with cycles) and the corresponding ASO program.

ing together the preference orderings associated to each branch of the tree. This approach is good and elegant for representing small scenarios, but the directly application of the composition operators becomes difficult for big and complex scenarios. This problem can be avoided by translating the CP-defense tree into ASO program, whose semantics completely respects the intended meaning given in [2], and by solving the obtained program with CHOPPER, an ASO solver described in [7].

4.1 Translation of *and-or* attacks into ASO programs

The ASO framework can profitably be used in order to represent a CP-defense tree structure and find the best set of countermeasures able to stop all the attacks. In this section the translation of **and/or** attacks into ASO programs whose semantics realizes **and/or**-composition of actions, presented in Sections 3.1 and 3.2, will be discussed.

Translation of and-attacks into ASO programs. Let us start with an example of **and**-attack. Consider the **and**-attack w , composed of two actions x and y , where $y \succ x$, presented in Fig. 6(a).

To protect the system from this kind of attack it is enough to stop just one action composing the attack. The order among countermeasures obtained by **and**-composition of orders corresponding to these actions is depicted in Fig. 6(b). We can notice that in this case a cycle is obtained. Since the countermeasure of the most dangerous attack has to be considered as preferred, the cycle can be broken by removing one of the arcs among the countermeasure of the less dangerous attack x . More precisely, the preference relations, described in (D_y, \succ_y) have to be considered as more important, and the relation $b \succ_x c$, generating (transitively) the conflict, has to be omitted. Graphically, this corresponds to removing the arc as shown in Fig. 6(c).

Let us now consider how to model this by using ASO programs. The attack action x and the preference order over the corresponding countermeasures a , b , and c generate the following ASO program $\langle \mathcal{P}_x, \Phi_x \rangle$:

$$\mathcal{P}_x \quad r_{x_1} : x \leftarrow . \quad r_{x_2} : a \vee b \vee c \leftarrow x. \quad \Phi_x \quad \rho_{x_1} : a > b > c \leftarrow x.$$

where the rules r_{x_1} and r_{x_2} introduce the action and the possible countermeasures respectively, while ϱ_{x_1} represents the preference order among them. The same result is obtained for the attack action y , the corresponding program $\langle \mathcal{P}_y, \Phi_y \rangle$ is the following:

$$\mathcal{P}_y \quad r_{y_1} : y \leftarrow . \quad r_{y_2} : a \vee c \leftarrow y. \quad \Phi_y \quad \varrho_{y_1} : c > a \leftarrow y.$$

In order to model the **and**-node presented in Fig. 6(a), a new program $\langle \mathcal{P}, \Phi_y, \Phi_x \rangle$ (see right side of Fig. 6) is generated combining the rules in $\langle \mathcal{P}_x, \Phi_x \rangle$ and $\langle \mathcal{P}_y, \Phi_y \rangle$. \mathcal{P} introduces two new rules: r_1 represents the root action w , while r_2 combines the action x and y in such way that only one of them must be stopped. The other rules are simply added without any change.

The answer sets of \mathcal{P} are $M_1 = \{w, x, a\}$, $M_2 = \{w, x, b\}$, $M_3 = \{w, x, c\}$, $M_4 = \{w, y, c\}$ and $M_5 = \{w, y, a\}$. In order to establish the optimal answer set, the ASO semantics firstly constructs the satisfaction vectors reporting the degree of satisfaction for ϱ_{y_1} and ϱ_{x_1} for each model: $V_{M_1} = [I, 1]$, $V_{M_2} = [I, 2]$, $V_{M_3} = [I, 3]$, $V_{M_4} = [1, I]$ and $V_{M_5} = [2, I]$.

Since ϱ_{y_1} is more important than ϱ_{x_1} , the comparison of these vectors under assumption that $I \equiv \infty$ establishes the following order among the answer sets: $M_4 > M_5 > M_1 > M_2 > M_3$. Consequently, M_4 is the optimal answer set and $\{c\}$ is the best set of countermeasures as $c \in M_4$.

The order among answer sets can be also used in order to derive all possible sets of countermeasures and the order among them. Firstly, observe that the possible sets of countermeasures correspond to the extraction of countermeasures from the answer sets and are $\{a\}$, $\{b\}$ and $\{c\}$. The preference order among these sets can be obtained by considering the preference order among the corresponding answer sets. In particular, when a set corresponds to multiple answer sets, the best answer set has to be considered.

In this example both M_1 and M_5 contain countermeasure a , $M_5 > M_1$, thus we have consider M_5 ; both M_3 and M_4 contain countermeasure c , $M_4 > M_3$, thus we have consider M_4 . Consequently, the order among the sets of countermeasures is $\{c\} > \{a\} > \{b\}$ as $M_4 > M_5 > M_2$.

Translation of or-attacks into ASO programs. Let now consider the case of an **or**-attack, i.e. an attack composed of a set of alternative actions one of which has to be successfully achieved to obtain the goal. The protection from this kind of attack consists in the protection from all the actions composing the **or**-attack. Intuitively, the most preferred strategy has to select the best countermeasure for each action.

Consider the **or**-attack w presented in Fig. 7. The corresponding ASO program is generated as follows: $\langle \mathcal{P}_x, \Phi_x \rangle$, $\langle \mathcal{P}_y, \Phi_y \rangle$ and $\langle \mathcal{P}_z, \Phi_z \rangle$ represent, respectively, the programs associated to the actions x , y and z , and the corresponding preferences over the countermeasures a , b and c .

$$\begin{array}{lll} \mathcal{P}_x \quad x \leftarrow . \quad a \vee b \vee c \leftarrow x. & \mathcal{P}_y \quad y \leftarrow . \quad a \vee c \leftarrow y. & \mathcal{P}_z \quad z \leftarrow . \quad a \vee b \leftarrow z. \\ \Phi_x \quad a > b > c \leftarrow x. & \Phi_y \quad c > a \leftarrow y. & \Phi_z \quad a > b \leftarrow z. \end{array}$$

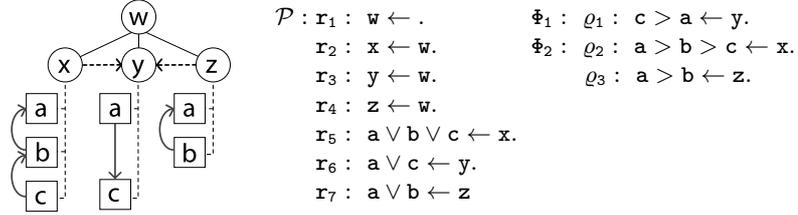


Fig. 7. An example of or attacks and the corresponding ASO program.

Then they are combined in the ranked ASO program $\langle \mathcal{P}, \Phi_1, \Phi_2 \rangle$, where $\Phi_1 = \Phi_y$, $\Phi_2 = \Phi_x \cup \Phi_z$ (see the right side of Fig. 7). A new rule r_1 , introduced in \mathcal{P} , represents the root of the or-attack w , while the rules r_2 , r_3 and r_4 model the or-attack, i.e. that all the three actions must be stopped to stop the w . The answer sets of \mathcal{P} are $M_1 = \{w, x, y, z, a\}$ and $M_2 = \{w, x, y, z, b, c\}$ and describes the application of two alternative sets of countermeasures $\{a\}$, $\{b, c\}$, protecting from the or-attack w . In order to establish the optimal answer set, the ASO semantics firstly construct the satisfaction vectors $V_{M_1} = [2, 1, 1]$ and $V_{M_2} = [1, 2, 2]$. Then it compares these vectors, by firstly considering $\rho_1 \in \Phi_1$, obtaining immediately that $V_{M_2} < V_{M_1}$. Consequently, M_2 is the optimal answer set and $\{b, c\}$ is the preferred set of countermeasures.

Observe that the application of ASO semantics models the or-composition operation in simple and elegant way. For instance, by collecting the minimal answer sets it avoids to select the redundant countermeasures. In fact, in the example above among the sets $M_1 = \{w, x, y, z, a\}$, $M_1' = \{w, x, y, z, a, b\}$, $M_1'' = \{w, x, y, z, a, c\}$, $M_1''' = \{w, x, y, z, a, b, c\}$ only M_1 is considered as the application of the countermeasure a is enough to protect the system.

Observe also that the most preferred strategy suggested by ASO semantics has to prefer the set, containing the best countermeasures for most dangerous actions, to the set containing the smaller number of countermeasures. In fact, in the above example, the set of countermeasures $\{b, c\}$ is preferred to $\{a\}$, having only one countermeasure. A possible extension of the ASO semantics, could be able to take into account the cardinality of the set of countermeasures.

4.2 Translation of CP-defense trees to ASO programs

Given an IT system $root$ and the corresponding CP-defense tree \mathcal{T} , the selection of the preferred defense strategy can be modeled by means of the corresponding logic program with preferences. In particular, if we assume that (i) attacks are totally ordered, and (ii) the set of countermeasures occurring in every conditional preference rule is totally ordered, a ranked ASO program $L(\mathcal{T}) = \langle \mathcal{P}, \Phi_1, \dots, \Phi_k \rangle$ can be constructed, where \mathcal{P} describes the possible defense strategies designed in \mathcal{T} , while Φ_1, \dots, Φ_k model preferences among the attacks, highlighted in \mathcal{T} , and establish the preference order among the attacks following the preference orderings among the countermeasures for each single attack. The application of the ASO semantics on $L(\mathcal{T})$ produces the best defense strategies w.r.t \mathcal{T} , thus the

optimal solutions of $L(\mathcal{T})$ can be used in order to find the best countermeasure selection.

Given a CP-defense tree with n leaf attack actions a ranked ASO program $L(\mathcal{T}) = \langle \mathcal{P}, \Phi_1, \dots, \Phi_k \rangle$, $k \leq n$, can be defined, where

\mathcal{P} is generating program consisting in the following rules r :

1. $\text{root} \leftarrow$, stating that the root of the tree must be protected;
2. $Y_1 \vee \dots \vee Y_n \leftarrow X$, for each **and**-node X having n child nodes Y_1, \dots, Y_n , meaning that at least one attack from Y_1, \dots, Y_n must be stopped to protect from X ;
3. $Y_i \leftarrow X$, $i = [1..n]$, for each **or**-node X having n child nodes Y_1, \dots, Y_n , stating that each attack represented by Y_1, \dots, Y_n must be stopped to protect from X ;
4. $C_1 \vee \dots \vee C_n \leftarrow X$, for each leaf node X decorated with n countermeasures C_1, \dots, C_n , stating that at least one countermeasure C_1, \dots, C_n is able to protect from X .

Each action in the defense tree induces a preference rule in the ASO program. Moreover, if the attack actions are ordered as $A_1 \succ \dots \succ A_k$, where each A_i is a set of actions $\{a_1, \dots, a_{h_i}\}$, a set of preference programs Φ_1, \dots, Φ_k , where $\Phi_i = \{\varrho_{a_1}, \dots, \varrho_{a_{h_i}}\}$, $i = [1, \dots, k]$, is constructed. For each attack action X the preference rule ϱ_X is defined as follows:

1. $C_1 > \dots > C_n \leftarrow X$, for each leaf node X decorated with n countermeasures C_1, \dots, C_n , where there is a solid arrow from C_i to C_{i-1} , $i = [2, \dots, n]$.
This preference rule states that to protect the attack X the countermeasure C_1 is preferred to C_2 , \dots , C_{n-1} is preferred to C_n .

The optimal answer set obtained by computing the semantics of the generated program will collect the best countermeasure. The following Example show the result of the above procedure when applied to the CP-defense tree of Fig. 3(b).

Example 5. In this example we present an application of the ASO semantics for analyzing the attack/defense scenario shown in the CP-defense tree of Fig. 3(b). It can be modeled by using the prioritizing program $\langle \mathcal{P}, \Phi \rangle$:

$$\begin{array}{lll}
 \mathcal{P} : & \text{root} \leftarrow & a_5 \vee a_6 \leftarrow a_{5,6} & \Phi : & \varrho_1 : c_1 > c_2 > c_3 \leftarrow a_1 \\
 & a_{1,2} \leftarrow \text{root} & c_1 \vee c_2 \vee c_3 \leftarrow a_1 & & \varrho_2 : c_5 > c_3 > c_4 \leftarrow a_2 \\
 & a_{3,4} \leftarrow \text{root} & c_3 \vee c_4 \vee c_5 \leftarrow a_2 & & \varrho_3 : c_6 > c_7 \leftarrow a_3 \\
 & a_{5,6} \leftarrow \text{root} & c_6 \vee c_7 \leftarrow a_3 & & \varrho_4 : c_8 > c_9 \leftarrow a_4 \\
 & a_1 \leftarrow a_{1,2} & c_8 \vee c_9 \leftarrow a_4 & & \varrho_5 : c_{10} > c_{11} \leftarrow a_5 \\
 & a_2 \leftarrow a_{1,2} & c_{10} \vee c_{11} \leftarrow a_5 & & \varrho_6 : c_{12} > c_{13} \leftarrow a_6 \\
 & a_3 \leftarrow a_{3,4} & c_{12} \vee c_{13} \leftarrow a_6 & & \\
 & a_4 \leftarrow a_{3,4} & & &
 \end{array}$$

The answer sets of \mathcal{P} are ninety, M_1 is an example of them: $M_1 = \{\text{root}, a_{1,2}, a_{3,4}, a_{5,6}, a_1, a_2, a_3, a_4, a_5, c_1, c_4, c_6, c_8, c_{10}\}$. Considering the preference order among attacks, as depicted in Fig. 3(b), we can specify the importance of preference rules: ϱ_2 is more important than ϱ_1 for expressing the preferences in the selection of countermeasures

for the attack a_2 that is preferred to a_1 , ϱ_1 is more important than ϱ_6 because a_2 is preferred to a_6 and so on. In this way we obtain the following ranked ASO program $\langle \mathcal{P}, \Phi_1, \Phi_2, \Phi_3, \Phi_4, \Phi_5, \Phi_6 \rangle$, where $\Phi_1 = \{\varrho_2\}$, $\Phi_2 = \{\varrho_1\}$, $\Phi_3 = \{\varrho_6\}$, $\Phi_4 = \{\varrho_5\}$, $\Phi_5 = \{\varrho_3\}$ and $\Phi_6 = \{\varrho_4\}$, whereas the generating program is the same. By applying the ASO semantics we obtain that $M = \{root, a_{1,2}, a_{3,4}, a_{5,6}, a_1, a_2, a_3, a_4, a_6, c_1, c_5, c_6, c_8, c_{13}\}$ is the optimal answer set, intuitively, M is the preferred answer set as it contains a_6 which is preferred w.r.t. a_5 and the best options for each preference rule $\varrho_1 \dots \varrho_4$ and ϱ_6 . Concluding, $\{c_1, c_5, c_6, c_8, c_{13}\}$ is the preferred set of countermeasures. \triangle

5 Conclusion and future works

In this paper we propose the use of two qualitative instruments for the selection of defense strategies to protect an IT system from the risk of attacks: we use defense trees to model attack/defense scenarios and CP-nets to model qualitative conditional preference over attacks and countermeasures.

Defense trees allow us to represent the possible attack actions that a person can perform to damage an asset of an IT system, and the possible countermeasures able to stop each action.

CP-nets, instead, allow us to model conditional preferences over attacks and countermeasures, and the qualitative preferences of a system administrator in the selection of countermeasures for each attack.

Our idea is to use CP-nets to model the selection of the countermeasures represented in a defense tree. We propose two methods for the composition of these preferences: an **and**-composition to model the preference order in the case of an **and**-attack, and an **or**-composition in the case of an **or**-attack.

The first operation we propose is an **and**-composition, we combine the preference order over the countermeasures represented in the CP-nets associated to attack actions composing the **and**-attack. In this case we also take into consideration the preference between attacks with the purpose of determining the preferred countermeasures. The second operation we propose is an **or**-composition of the CP-nets: we are interested in determining a preference order over sets of countermeasures which are able to cover all actions of an **or**-attack. In particular, we prefer countermeasures which are able to mitigate the risks associated with more than one action.

This paper also discusses the use of ASO programs to represent CP-defense trees, and to reason about them. The ASO approach uses preference rules in order to express the preference relations among the combinations of atoms and introduces the preference order among these rules. The ASO semantics gives simple and an intuitive way for modelling a CP-defense tree and is expressive enough to formulate preferences over countermeasures w.r.t. attacks.

The methodology presented in this paper provides a basis for future work along the following directions.

In a future version of this paper we will give formal proofs about the soundness and the completeness of the translation of CP-defense trees into ASO programs. When presenting the translation method in Section 4 we assumed that (i) attacks are totally ordered, and (ii) the set of countermeasures occurring in every

conditional preference rule is totally ordered. The application of the translation method to the more general case requires further investigation.

We faced the problem of determining defense strategies composed of sets of countermeasures (one for each attack) and only in the case of an **or**-attack we studied how a single countermeasure can cover more than one attack. A possible extension of this work is to investigate how to find sets of countermeasures able to mitigate sets of attacks, also in the case of **or**-attacks.

In this paper we use CP-nets considering only *strict* partial orders, a possible extension can be the use of *non strict partial orders* to model the preference over countermeasures and attacks, and the use of indifference between this variables (for instance $a_1 \succeq a_2$).

We also plan to consider attacks as uncertain variables. In [10] an approach is described to model a real-life problem as a set of variables with finite domains and a set of soft constraints among subsets of the variables. A variable is uncertain if we cannot decide its value. In this case, they associated a possibility degree to each value in its domain, which will tell how plausible it is that the variable will get that value.

References

1. S. Bistarelli, F. Fioravanti, and P. Peretti. Defense tree for economic evaluations of security investment. In *1st Int. Conf. on Availability, Reliability and Security (ARES'06)*, pages 416–423, 2006.
2. S. Bistarelli, F. Fioravanti, and P. Peretti. Using cp-nets as a guide for countermeasure selection. In *ACM Symp. on Applied Computing*, pages 300–304, 2007.
3. C. Boutilier, R. I. Brafman, C. Domshlak, H. Hoos, and D. Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *JAIR*, 21, 2004.
4. C. Boutilier, R. I. Brafman, C. Domshlak, H. Hoos, and D. Poole. Preference-based constrained optimization with cp-nets. *Computational Intelligence, Sp.I. on Preferences*, 2(200):137–157, 2004.
5. C. Boutilier, R. I. Brafman, H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *15th Conf. on Uncertainty in Artificial Intelligence*, pages 71–80, 1999.
6. G. Brewka, I. Niemela, and M. Truszczyński. Answer set optimization. In *18th Int. Joint Conf. on Artificial Intelligence*, pages 867–872. Morgan Kaufmann, 2003.
7. L. Caroprese, I. Trubitsyna, and E. Zumpano. Implementing prioritized reasoning in logic programming. In *ICEIS (2)*, pages 94–100, 2007.
8. J. Caulkins, E. Hough, N. Mead, and H. Osman. Optimizing investments in security countermeasures: A practical tool for fixed budgets. *IEEE Security and Privacy*, 5(5):57–60, 2007.
9. R. W. Floyd. Non-deterministic algorithms. *J. Assoc. Comp.*, pages 636–644, 1967.
10. M.S. Pini, F. Rossi, and K.B. Venable. Possibility theory for reasoning about uncertain soft constraints. In *ECSQARU*, pages 800–811, 2005.
11. Mehmet Sahinoglu. Security meter: A practical decision-tree model to quantify risk. *IEEE Security and Privacy*, 3(3):18–24, 2005.
12. B. Schneier. Attack trees: Modeling security threats. *Dr. Dobb's Journal*, 1999.

A Java Wrapper for Answer Set Programming Inferential Engines^{*}

Giovanni Pirrotta and Alessandro Provetti

Dept. of Physics, Informatics curriculum. The Univ. of Messina.
Sal. Sperone 31. S. Agata di Messina, I-98166 Italy
{gpirrotta,ale}@unime.it

Abstract. Answer set programming inferential engines often do not give much support to integration with Object-oriented software applications written in e.g., C++ and Java. Thus, they cannot be easily integrated in external applications. This work tries to overcome the above problem by the implementation in Java of a wrapper that supports several use cases. The new JASPWrapper for ASP inferential engines is presented and its usage and advantages are described in detail. The Wrapper consist of a Java library, to be included in larger projects, that offers a uniform interface to several inferential engines for Answer Set Programming.

1 Introduction

Answer set programming (ASP) is a form of declarative programming while the syntax of ASP largely coincides with that of Prolog, the semantics is alternative. It is based on the stable model (answer set) semantics of logic programming by Gelfond and Lifschitz [1] In ASP, search problems are rephrased as computing a minimal set of atoms that is *stable* w.r.t. a set of constraints defined in the program. In a more general sense, ASP can be useful in all applications of knowledge representation and reasoning. [1,2]. The areas of application where ASP has been successfully applied are¹

- **Software Configuration Management** - modern software products are large and may consist of thousands of components where each component provides some specific functionality. In configuration management, given a description of the components and a set of user requirements, the target is to find a valid configuration of components that satisfies the requirements [3].

^{*} This work was partially supported by M.U.R. under the PRIN 2006 project “Potenziamento e Applicazioni della Programmazione Logica.” A companion Web site to this article, with software and benchmarks is available at <http://mag.dsi.unimi.it/jasp-wrapper/>

¹ The following listing is broadly taken from the survey of the WASP working group: <http://www.kr.tuwien.ac.at/projects/WASP/showcase.html>.

- **Data and Information Integration** - Information integration is the task of providing a uniform interface to various pre-existing data sources, so as to enable users to focus on specifying what answers they want, rather than on how to obtain the answer. Information integration is one of the key problems in distributed databases, cooperative information systems, and data warehousing [4].
- **Security Engineering** - Incorporating Security Engineering into Software Requirements Engineering is an important research topic in particular since capturing trust and security requirements at an organizational level is a challenging problem [5].
- **Agent Systems** - Multi-agent systems have been recognized as a promising paradigm for distributed problem solving, and numerous multi-agent platforms and frameworks have been proposed, which allow to program agents in languages ranging from imperative over object-oriented to logic-based ones.
- **Semantic-Web Reasoning** - Reasoning support for the Semantic Web is currently mainly restricted to terminological reasoning in Description logics perhaps reflecting a consensus on what will constitute the logical layer of the Semantic Web. ASP can be used to support nonmonotonic formalisms and the representation of default knowledge, i.e., the mechanism of “jumping to conclusions” in the presence of incomplete or uncertain information [6].
- **Planning** - in contrast to classical planning, where complete knowledge about the scenario is assumed, knowledge-based planning deals with further, more real-world features, like incomplete knowledge about the domain and non-deterministic actions [7], where, through the work of Gelfond, Lifschitz, Baral, Son and others ASP finds a convincing application.

The existing ASP inferential engines [8], e.g., DLV (and its successor DLV* [9]), smodels, Cmodels and Clasp have been successfully applied to the reasoning problems defined above. However, most ASP inferential engines do not fully support (with the possible exceptions described in detail in the next Section) integration with software applications based on OO paradigm, such C++ and Java and cannot be easily interfaced in any other external program.

This work tries to overcome the above problem showing our novel implementation in Java a wrapper for ASP inferential engines named JASPWrapper. JASPWrapper invokes native compiled solver and grounder through a Java Virtual Machine and computes answer sets by intercepting and analyzing the output stream.

The main inspiration for this project came from the DLV Wrapper Project (JDLV) by Ricca [10]. JDLV provides a Java interface specific to the DLV system; to best of our knowledge it is intended only for the DLV system and remains closed-source. Our JASPWrapper project in contrast seeks to overcome the limits of the previous efforts by offering a widely adaptable wrapper for the most known ASP grounders and solvers.

2 State of the art

In this Section we survey existing implementation efforts that, to the best of our knowledge, focus, as JASPWrapper does, on making ASP inferential engines available as a component of software applications.

2.1 Jsmodels

The Jsmodels [11] project at New Mexico State University seeks to port the branch-and-bound algorithmic structure and the powerful heuristics of smodels to Java. Jsmodels has been successfully deployed in the area of automated planning. We can see Jsmodels as a re-coding of smodels in Java. Re-encoding in Java brings about several interesting possibilities which, in our opinion, are not yet fully exploited. Among the objectives of the Jsmodels project there is, rather than improving performance, the exploitation of threaded/parallel execution by Java virtual machine. Indeed, a preliminary benchmark on a single-processor architecture has shown a relative worsening of the performance (time before the first model is found). Clearly, including Jsmodels in a Java application is an easy task and well-supported by the Java platform.

2.2 The API for smodels

The smodels project comes with an API library that provides some basic communication features. According to Lparse manual [12], the following sets of functions are available:

api.h function for creating logic programs,
atomrules.h to define atoms and rules,
defines.h for general definitions,
stable.h functions for reading programs from files, and
smodels.h functions for computing stable models.

The headers file above mentioned contains all necessary functions needed to create and manipulate logic programs. For example, the rules are created using the following methods:

```
void begin_rule (RuleType type);
void add_head (Atom *a);
void add_body(Atom *a, bool pos);
void add_body(Atom *a, bool pos, Weight w);
void end_rule();
```

There are six functions that can be used to create and manipulate atoms.

```
virtual Atom *new_atom();
void set_name (Atom *a, const char *name);
Atom *get_atom (const char *name);
```

```
void set_compute(Atom *a, bool in_model);
void reset_compute(Atom *a, bool in_model);
done();
```

The `new_atom()` function creates a new atom and it has to be named with the `set_name()` function. The two last functions are used to define the compute statement. The function `done()` is called when all rules of the program have been constructed. This is used to signal the `Api` class that it can now create the internal data structures for the program.

2.3 JDLV

Ricca's JDLV [10] project provides a Java interface specific to the DLV system. It effectively allows the embedding of DLV in larger applications. JDLV is a library implemented in Java, that allows to embed disjunctive logic programs inside Object-Oriented programs. Basically, the DLV Wrapper executes, in an external native process, the DLV system, feeding input and getting output through a system pipe. Moreover, by using suitable hierarchy of classes, the DLV Wrapper allows to handle DLV Input and DLV output by using Java objects and to import and export data from/to commercial database systems implementing JDBC drivers. In this way, DLV Wrapper acts as a full interface between Java programs and the DLV system and allows user to manipulate ASP logic program thinking in OO paradigm.

3 The architecture of JASPWrapper

JASPWrapper is an object-oriented library that implements the following basic operations:

1. Load program (source files or grammar objects);
2. Setup engine (configuration of solver and grounder options);
3. Load program into chosen ASP engine;
4. Start engine, and
5. Parsing output stream by translating it into our object model abstraction.

The ASP engine is parametric, i.e., its name is passed by the invocation. The real computation takes place in an external process and JASPWrapper manages text output results coming from it. It must be noticed that before the JDK 5.0² release, the only way to fork off a process and execute it local to the user runtime was to use the `exec()` method of the `java.lang.Runtime` class. JDK 5.0 adds a new way of executing a command in a separate process, through a class called *ProcessBuilder*.

As for step 1, programs and/or instances are inserted by source files or grammar objects. Then the user must create a solver and -when needed- a grounder object

² <http://java.sun.com/j2se/1.5.0/>.

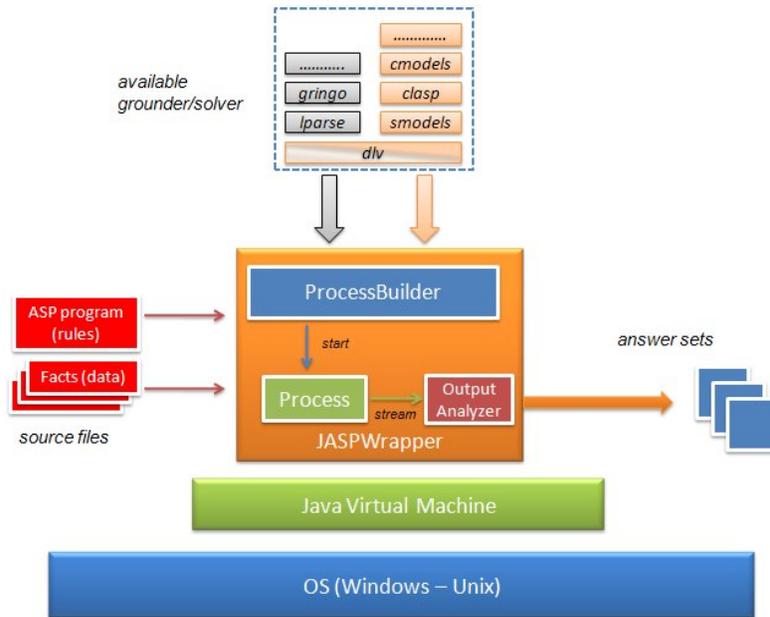


Fig. 1. The JASPWrapper Architecture

configuration. After the user passes these objects to JASPWrapper, external programs (dlv, smodels, lparse, etc.) are invoked by runtime and *Process* object is created. Finally, the output stream coming from *Process* is analyzed (essentially, filtered) by the *OutputAnalyzer*, to obtain the answer sets.

3.1 The Output Analyzer

Our goal to obtain a *universal* wrapper w.r.t. the available ASP grounders and solvers implies defining a uniform output format. The Output Analyzer module is in charge of creating a uniform output; its functioning is described as follow. At first, the module sits and listens to the output stream; the first (if any) answer set arriving is recognized and delimited by the applications of regular patterns. Next, for each subsequent answer set, a new search starts. For instance, when the output stream is from DLV, e.g.

```
{q(1,3), q(2,7), q(3,2), q(4,8), q(5,5), q(6,1), q(7,4), q(8,6)} (a)
q(1,3), q(2,7), q(3,2), q(4,8), q(5,5), q(6,1), q(7,4), q(8,6) (b)
q(1,3) (c)
.
.
q(8,6)
```

JASPWrapper

1. extracts contents delimited by brackets ($\{\}$); (*a*)
2. analyzes text delimited by commas; (*b*)
3. each atom recognized from output text (*c*) is reified in an *Atom* object, and
4. inserted in the *Model* object.

Afterwards, we continue parsing by going back to searching for a new model. In setting up the engine, one must specify which type of engine solver she wants to use. Hence, the wrapper will know which extraction strategy to apply to parsing stream and to obtaining model objects. JASPWrapper 0.9 has output analyzers for DLV, DLV* [13,9], Smodels and Clasp. The *OutputAnalyzer* abstract class can be extended to address new output formats, thus making JASPWrapper widely adaptable.

3.2 Synchronization strategy

JASPWrapper can be launched in two modalities: *all models at once*, sometimes referred to as synchronous-mode and *one model at a time*, a.k.a. model-synchronous mode. The former corresponds to the *all models* version of Answer Set Programming: the inferential engine's output is paused and all answer set objects are available at once. Obviously, when an ASP program yields exponentially many models this modality is not effective (with or without wrapper). In such modality, a *Vector* object is used to store models found and its contents will grow until last model was found. Afterward, models are ready to be consumed by the calling methods (or the final user, for that matter).

The model-synchronous mode instead proceeds to reify the answer sets piecemeal, i.e., as they become available. Each new model found is stored in the same *Vector* object. Immediately the thread passes in sleep mode and the new model search is paused until the user consumes the one and only model available. Then, the thread is woken-up and a new search can start. In model-synchronous mode, the memory stack does not overflow because the memory to store each model found is released immediately after consumption by the caller. Our approach differs from that of JDLV -in the same model-synchronous mode- in the memory management aspect: the memory used by JDLV to store models does not come released so, for large models, the wrapper may as well go into stack-overflow.

4 Using JASPWrapper in interactive-mode

In this modality, JASPWrapper can be launched in two ways:

- Stand-alone mode
- Console-mode

Interactive mode is used mainly for debug activities and for didactic-purpose. In this modes the answer sets are not *reified* into objects but simply printed out in the console window.

4.1 Stand-alone mode

The easiest way to use JASPWrapper is through invocation from a command-line shell. In this way, JASPWrapper can use only compiled solvers grounders; it passes to them the source ASP program (and options) and redirects the output stream to the console. In other words, JASPWrapper from command-line works exactly as DLV or smodels.

This is the syntax of the invocation:

```
java jaspwrapper.tools.JASPWrapper file(s) OPTIONS
```

where OPTIONS are defined as follows:

<code>-solver</code>	<code>path/to/solver</code>	
<code>-os</code>	<code>type</code>	Available: windows,unix
<code>[-solverOptions</code>	<code>options]</code>	Passed to the solver, put in double quotes i.e. "-nolookahead -internal"]
<code>[-grounder</code>	<code>path/to/grounder]</code>	
<code>[-grounderOptions</code>	<code>options]</code>	Passed to the grounder, put in double quotes i.e. "-c n=8 -D"]

4.2 JASPConsole mode

JASPConsole is a textual environment where the user can edit an ASP program and immediately compute models without a invoking a solver from the command line. At the start, the environment configuration (solver and grounder path, SO type, option) must be setup with the SET command; after, the user can enter the environment with the START command. Now, the RUN command will effectively launch the ASP computation.

Command:

```
java jaspwrapper.tools.JASPConsole
```

```
-----  
JASPWrapper 0.9 - Console (help for commands)  
-----
```

Commands:

<code>SET SOLVER</code>	<code>path</code>	<code>path/to/solver</code>
<code>SET OS</code>	<code>type</code>	operating system (unix/windows)
<code>[SET GROUNDER</code>	<code>path</code>	<code>path/to/grounder]</code>
<code>[SET SOLVER_OPTIONS</code>	<code>"options"</code>	options passed in double quote]
<code>[SET GROUNDER_OPTIONS</code>	<code>"options"</code>	options passed in double quote]
<code>START</code>		enter environment
<code> RUN</code>		compute models - inside environment
<code> NEW</code>		creates a new program
<code> UP</code>		exit environment
<code>QUIT</code>		exit console

5 Using JASPWrapper as library mode

Arguably, the best use of JASPWrapper is as an API library for Java applications: Inside JASPWrapper, a set of essential objects for building a complete object-oriented ASP program is available. The *Program* class is the most important of these and to use JASPWrapper in library mode, one must first of all configure an instance of this object. That may be done in several ways, which are described in detail next.

5.1 Grammar objects

With this approach, the input ASP program is paused, tokenized on each lexical entity is made to correspond to an instance of the relative grammar object. The following lexical objects are available:

- Constant
- Variable
- Atom
- Literal
- Rule
- Program
- Model

The following fragment shows the explicit instantiation of lexical objects representing a sample ASP program³.

```
Variable x = new Variable("X");
Constant tweety = new Constant("tweety");
Constant skippy = new Constant("skippy");
Atom bird = new Atom("bird");
bird.addTerm(x);
Atom abnormal = new Atom("abnormal");
abnormal.addTerm(x);
Atom penguin = new Atom("penguin");
penguin.addTerm(x);
Atom bird1 = new Atom ("bird");
bird1.addTerm(tweety);
Atom penguin1 = new Atom("penguin");
penguin1.addTerm(skippy);
Atom flies = new Atom("flies");
flies.addTerm(x);
Rule rule1 = new Rule();
rule1.addToHead(new Literal(flies));
rule1.addToPosBody(new Literal(bird));
```

³ Obviously, explicit instantiation from within the code is not viable as the program size grows.

```

rule1.addToNegBody(new Literal (abnormal));
Rule rule2 = new Rule();
rule2.addToHead(new Literal(abnormal));
rule2.addToPosBody(new Literal(penguin));
Rule rule3 = new Rule();
rule3.addToHead(new Literal(bird));
rule3.addToPosBody(new Literal(penguin));
Literal lit1 = new Literal(bird1);
Literal lit2 = new Literal(penguin1);

Program p1 = new Program();
p1.addFact(lit1);
p1.addFact(lit2);
p1.addRule(rule1);
p1.addRule(rule2);
p1.addRule(rule3);
p1.end();

```

The object ASP program may also be instantiated rule by rule:

```

Program p2 = new Program();
p2.addRule("a:-not b.");
p2.addRule("b:-not a.");
p2.addRule("c:-not d.");
p2.addRule("d:-not c.");
p2.addRule("e:-not f.");
p2.addRule("f:-not e.");
p2.end();

```

Finally, the most concise way to instantiate the object program is through file upload:

```

Program p3 = new Program();
p3.addFromFile("d:\\queens.sm");
p3.end();

```

Now one must proceed to configure Solver and Grounder objects, create the engine object, select the O.S., the extraction strategy , load the program and finally start the inferential engine.

```

Solver solver = new Solver ("c:\\asp\\smodels.exe");
Grounder grounder = new Grounder ("c:\\asp\\lparse.exe");
EngineRunner myEngine = new EngineRunner(solver,grounder);
myEngine.setOS("windows");
myEngine.setSolverType(OutputAnalyzerFactory.SMODELS);
myEngine.loadProgram(program);
myEngine.compute(EngineRunner.ONE_MODEL_AT_ONCE);

```

Let us now see an example run for program *p1*

```
myEngine.loadProgram(p1);
myEngine.compute();
while (myEngine.hasMore()) {
    Model model = myEngine.next();
    for(Iterator itAtom = model.iterator(); itAtom.hasNext();) {
        Atom atom = (Atom) itAtom.next();
        System.out.println(atom.getName());
        for (Iterator itArg = atom.iterator(); itArg.hasNext();) {
            Constant constant = (Constant) itArg.next();
            System.out.println(constant);
        }
    }
}
```

5.2 JASPWrapper output for library mode

Let us now see the output results for the programs described above. The output for program *p1* will be:

```
JASPWrapper 0.9 - Command executing...
cmd /C C:\lparse.exe C:\Temp\jaspwrapper12105.tmp | C:\asp\models.exe 0
-----
penguin(skippy) bird(tweety) bird(skippy) abnormal(skippy) flies(tweety)
```

For program *p2*:

```
JASPWrapper 0.9 - Command executing...
cmd /C C:\lparse.exe C:\Temp\jaspwrapper12106.tmp | C:\asp\models.exe 0
-----
e d b
e c b
e c a
e d a
f d a
f d b
f c b
f c a
```

For program *p3*:

```
JASPWrapper 0.9 - Command executing...
cmd /C C:\lparse.exe C:\Temp\jaspwrapper12107.tmp | C:\asp\models.exe 0
-----
has_q(1) has_q(2) has_q(3) has_q(4) q(4,1) has_q(5) has_q(6) q(1,2) q(5,3)
```

```

q(2,4) q(6,5) q(3,6) n_q(1,1) n_q(2,1) n_q(3,1) n_q(5,1) n_q(6,1) n_q(2,2)
n_q(3,2) n_q(4,2) n_q(5,2) n_q(6,2) n_q(1,3) n_q(2,3) n_q(3,3) n_q(4,3)
n_q(6,3) n_q(1,4) n_q(3,4) n_q(4,4) n_q(5,4) n_q(6,4) n_q(1,5) n_q(2,5)
n_q(3,5) n_q(4,5) n_q(5,5) n_q(1,6) n_q(2,6) n_q(4,6) n_q(5,6) n_q(6,6)
d(1) d(2) d(3) d(4) d(5) d(6)
.
.
.
has_q(1) has_q(2) q(2,1) has_q(3) has_q(4) has_q(5) has_q(6) q(4,2) q(6,3)
q(1,4) q(3,5) q(5,6) n_q(1,1) n_q(3,1) n_q(4,1) n_q(5,1) n_q(6,1) n_q(1,2)
n_q(2,2) n_q(3,2) n_q(5,2) n_q(6,2) n_q(1,3) n_q(2,3) n_q(3,3) n_q(4,3)
n_q(5,3) n_q(2,4) n_q(3,4) n_q(4,4) n_q(5,4) n_q(6,4) n_q(1,5) n_q(2,5)
n_q(4,5) n_q(5,5) n_q(6,5) n_q(1,6) n_q(2,6) n_q(3,6) n_q(4,6) n_q(6,6)
d(1) d(2) d(3) d(4) d(5) d(6)

```

Great care has been given to output visualization; the answer set objects resulting from computation can be returned also in following formats:

- Console
- File
- CSV
- XML
- LATEX
- HTML
- SQL

To obtain the desired output formatting the following instantiation shall be used, possibly in combination:

```

myEngine.loadProgram(p1);
myEngine.addOutputResult(new ConsoleOutputResult());
myEngine.addOutputResult(new FileOutputResult("file.txt"));
myEngine.addOutputResult(new CSVOutputResult("file.csv"));
myEngine.addOutputResult(new XMLOutputResult("results.xml"));
myEngine.addOutputResult(new LatexOutputResult("file.tex"));
myEngine.addOutputResult(new HTMLOutputResult("page.html"));
myEngine.addOutputResult(new SQLOutputResult("script.sql"));

```

In the final phase, up to six output files are created and models are redirected to the console output. Moreover, if a different format type is needed, one may easily implement a new *OutputResult* interface without changing existing sources.

6 Benchmarking JASPWrapper

In this Section we present our experimental assessment of JASPWrapper in terms of the computational overhead it would introduce vis-à-vis direct invocation of

the relative grounder/solver. JASPWrapper (version 0.9) has been tested on a server with Pentium Dual Core 3.40 GHz RAM 1 GB - Linux Fedora core 7 - kernel 2.6.21.

The following ASP engines (with lparsc as external grounder, when needed) have been considered:

- Clasp 1.0.4,
- dl_v (build BEN/Oct 11 2007),
- dl* (build BEN/May 12 2008),
- Smodels 2.32 and
- CModels 3.73.

For this benchmarking the stand-alone mode has been used: computation is wrapped by JASPWrapper, and output is instantly sent in console, thus skipping reification. The `time` unix command has been used to extract the times for the *Java Virtual Machine* and the chosen inferential engine. Table 6 reports the speed-up ratios wrapper (as a percentage), which are defined as follows:

- $JRT_k(p, n)$: JASPWrapper real time for program p in iteration n with engine (and grounder if required) k
- $GRT_g(p, n)$: Grounder real time for program p in iteration n with grounder g (equal to 0 if grounder is not required)
- $SRT_e(p, n)$: Solver real time for program p in iteration n with engine e

$$\bar{j} = \frac{\sum_{n=1}^{10} JRT_k(p, n)}{10} \quad \bar{s} = \frac{\sum_{n=1}^{10} GRT_g(p, n) + SRT_e(p, n)}{10}$$

$$ratio = \frac{\bar{j} * 100}{\bar{s}} - 100$$

From these benchmarking results we can observe how JASPWrapper introduces only a marginal time overhead w.r.t. the original command-line execution time. Obviously external process invoked by JASPWrapper is not faster than the same process invoked directly by command-line, since we expect the same times. In our results, however, we can note some negative ratio denoting faster execution for JASPWrapper. This anomalous behavior may have the following explanation. When JASPWrapper starts, the created subprocess does not have its own terminal or console. All its standard I/O (i.e. `stdin`, `stdout`, `stderr`) operations is redirected to the parent process through three streams (*input*, *output*, *error*). The parent process uses these streams to feed input to and get output from the subprocess. For some programs, perhaps, time stream communication among wrapper and subprocess added to time screen output is able to be slightly faster than time used by original engine to direct output results in console. In this manner, wrapper is able to gain something in time execution. It is also true that this anomaly can be seek to low tests effectively executed in our benchmarks. For further benchmarking details, please see JASPWrapper site described in the next section.

Category	Program	Clasp	DLV	DLV*	Smodels	CModels
Puzzle Game	15-puzzle.18	3,16%	-4,30%	0,19%	1,03%	3,58%
	15-puzzle.19	2,01%	-4,02%	0,00%	0,62%	4,04%
	15-puzzle.20	2,94%	13,96%	15,78%	2,77%	4,22%
Random Nontight	b5.lp	0,90%	0,52%	0,27%	0,10%	-0,10%
	b9.lp	0,78%	0,54%	0,30%	0,18%	-0,06%
	b10.lp	0,64%	0,43%	0,14%	0,17%	-0,46%
	b17.lp	0,37%	0,31%	0,07%	-0,07%	-0,41%
Random RLP-200	lp.200.01000.2	1,63%	-0,01%	0,10%	0,15%	-0,21%
	lp.200.00900.9	2,01%	-0,09%	-0,11%	0,07%	-0,64%
	lp.200.00900.19	1,62%	0,79%	0,58%	0,00%	-1,01%
	lp.200.01000.22	1,55%	-0,10%	0,06%	-0,16%	-0,16%
	lp.200.00900.23	2,05%	0,33%	-0,82%	0,14%	-0,30%
Eq. test	qeq.10	3,65%	1,42%	-0,61%	0,70%	1,08%
	qeq.11	0,48%	0,54%	-1,79%	0,02%	0,20%
	qeq.12	0,25%	-0,01%	-0,33%	0,07%	0,32%

Table 1. Benchmarking table - ratio values over real time

7 Licencing aspects: the GNU GPL License

JASPWrapper is a free software⁴ released with GPL v.3 license hence its realness can give an effective contribution to the adoption of ASP and to its further development.

Free software is defined by GNU GPL as follows⁵:

- the freedom to use the software for any purpose,
- the freedom to share the software with your friends and neighbors,
- the freedom to change the software to suit your needs, and
- the freedom to share the changes you make.

Finally, JASPWrapper can be downloaded from SourceForge⁶:
<http://jaspwrapper.sourceforge.net/>.

8 Conclusions and Future works

We have presented JASPWrapper, a Java library that allows to embed ASP grounder and inferential engines inside. Basically, the JASPWrapper executes

⁴ Free software is software that gives you the user the freedom to share, study and modify it. Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft

⁵ From <http://www.gnu.org/licenses/quick-guide-gplv3.html>

⁶ SourceForge.net is the world's largest Open Source software development web site. SourceForge.net provides free hosting to Open Source software development projects with a centralized resource for managing projects, issues, communications, and code.

ASP inferential engines as an external native process and captures their textual output using Java objects. We believe that the JASPWrapper will benefit the development of software applications, both in academia and in industry that employ ASP as their reasoning component. As for as current and future work Java-applications, We are testing the JASPWrapper library in real application contexts. We are optimizing both internal data structures and control structures in order to improve efficiency. Finally, we plan to enrich the wrapper implementation by adding new ASP inferential engines compatibility and by extending the output formats available i.e. implementing new output result converter.

Acknowledgments

Our work was in part motivated by the JDLV Wrapper Project started by F. Ricca and N. Leone. We thank them for the motivation and the support they gave us.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. *New Generation Computing* (1991) 365–387
2. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
3. Soinen, T., Niemelä, I., Tiihonen, J., Sulonen, R.: Representing configuration knowledge with weight constraint rules. In Proveti, A., Son, T.C., eds.: *Answer Set Programming*. (2001)
4. Faber, W., Greco, G., Leone, N.: Magic sets and their application to data integration. *J. Comput. Syst. Sci.* **73**(4) (2007) 584–609
5. Hietalahti, M., Massacci, F., Niemelä, I.: Des: a challenge problem for nonmonotonic reasoning systems. *CoRR cs.AI/0003039* (2000)
6. Eiter, T.: Answer set programming for the semantic web. In Dahl, V., Niemelä, I., eds.: *ICLP*. Volume 4670 of *Lecture Notes in Computer Science.*, Springer (2007) 23–26
7. Tu, P.H., Son, T.C., Pontelli, E.: Cpp: A constraint logic programming based planner with preferences. In Baral, C., Brewka, G., Schlipf, J.S., eds.: *LPNMR*. Volume 4483 of *Lecture Notes in Computer Science.*, Springer (2007) 290–296
8. Solvers: Web location of some of the most known asp solvers.
Aspps: <http://cs.engr.uky.edu/ai/aspps/>
CMODELS: <http://www.cs.utexas.edu/users/tag/>
Clasp: <http://www.cs.uni-potsdam.de/clasp/>
DLV: <http://www.dbai.tuwien.ac.at/proj/dlv/>
Smodels: <http://www.tcs.hut.fi/Software/smodels/> (2007)
9. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Functions, Lists and Set in DLV System. (2008) *CoRR Technical Report*.
10. Ricca, F.: The dlv java wrapper. In Buccafurri, F., ed.: *APPIA-GULP-PRODE*. (2003) 263–274
11. Le, H.V., Pontelli, E.: A java-based solver for answer set programming. Technical report, NMSU (2003)

12. Syrjänen, T.: Lparse 1.0 user's manual. Technical report, Helsinki University of Technology (2000)
13. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence* **50** (2007) 333–361

Towards introducing types in DLV*

(extended abstract)

Mario Ornaghi, Camillo Fiorentini, and Alberto Momigliano

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy*
{ornaghi,fiorenti,momiglia}@dsi.unimi.it

1 Introduction

The advantages of static type checking in programming languages are almost universally recognized and well-understood, although they made it into logic programming somewhat belatedly [13]. From the pioneering paper [11] advocating the introduction of types in Prolog, different approaches emerged, belonging roughly to two main camps: the descriptive and the prescriptive one. The former aims to capture some aspects of a program behavior, for example its success set, up to much more complex static and dynamic properties as with CiaoPP [7].

We favor the *prescriptive* approach, where types are an integral part of the program's meaning, thus allowing the user to discard ill-formed elements at compile time. This induces the developer to proceed in a more disciplined way, being able to receive early feedback about mistakes that may be real hard to find, especially in a purely declarative setting where, ideally, the only answers are yes/no.

The type theory of logic programming has significantly developed in the passing years – see the type system of Mercury [8] for a modern take to prescriptive typing. However, for the sake of this paper, we will adapt for disjunctive logic programs an elementary “à la SML” polymorphic discipline; however, we will enrich it semantically, following Lakshman & Reddy's approach [9], where typed logic programs are interpreted over first order many sorted structures.

Our objective here is to advocate the usefulness of prescriptive typing for Answer Set Programming in general. This extended abstract reports work in progress towards the introduction of static type-checking in DLV* [4], an extension of DLV [10] with function symbols. We motivate the issues by means of a detailed example before presenting a first basic type system (Fig. 2) and stating its basic properties.

2 Types for DLV*

We briefly recall some basic notions of DLV [10] and its extension DLV*. In the former one cannot use function symbols, hence terms t can only be variables or constant symbols. A classical atom A can either be of the form $p(t_1, \dots, t_r)$ or its classical negation $\neg A$. Literals, instead, can also use negation as failure, denoted $\text{not } A$. Finally, clauses

* Work partially supported by the MIUR Project “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva”.

have the form $A_1 \vee \dots \vee A_n \leftarrow L_1 \wedge \dots \wedge L_k$; we write $A_1 \vee \dots \vee A_n$ if $k = 0$, while if $n = 0$ we have what we call a *constraint* $\leftarrow L_1 \wedge \dots \wedge L_k$.

DLV* extends DLV by introducing function symbols. This is obtained through the concept of *value invention* [4], which is based on the possibility of using externally defined predicates in the body of clauses of DLV* programs. The syntax $\#_p(\dots)$ shows that p is external. External predicates must be (well) *moded*. For example, the moded predicate $\#_p(i, o)$ means that a call $p(c, X)$ with ground input c will “invent” an output value for X . A predicate may have many different modes, e.g., $\#_p(i, o)$ and $\#_p(o, i)$. A DLV* program with external predicates is required to be *safe* and *VI-restricted* (see [4] for those lengthy definitions, omitted here for lack of space). In this case, it is guaranteed that the grounding process will halt with a finite ground program whose answer sets are the expected ones. A function symbol $f(X)$ is introduced in DLV* by associating to it a constructor predicate $\#f(o, i)$ and a destructor predicate $\#f(i, o)$. Given a value c , $\#f(v, c)$ invents a value v representing the ground term $f(c)$, while $\#f(v, X)$ reconstructs the value c .

We Now introduce the type systems informally, by illustrating some of the relevant syntax:

```
type color --> red ; yellow ; blue.
type nat --> 0 ; s(nat).
type list(X) --> nil ; cons(X, list(X)).
```

The first declaration introduces an *enumeration* type `color`, generated by the constants `red`, `yellow` and `blue`. The second one defines an *inductive* type, generated by the constant `0:nat` and by the function `s : [nat] -> nat`. The third type is polymorphic, i.e., it parametrically depends on the type variable X . It represents the lists with elements of generic type X . For every type substitution t/X , `list(t)` is the set of all the lists generated by the constant `nil:list(t)` and by the function `cons: [t, list(t)] -> list(t)`.

A typed DLV* program starts with a type declaration section, introducing:

- all the enumerated types and all the types generated inductively. Essentially, the role of this section is to define the set of values that can be used in the grounding process;
- the declaration of all the predicates used in the programs, together with their arity, namely the number and types of their arguments. This is the basis for type checking¹.

In the rest of the section we will illustrate the motivations behind our proposal, while in Section 3 we will present the static semantics of Typed DLV*.

Example 1. One of the original motivations of our work was to analyze the possibility of using DLP in the validation of UML models, namely for automatic snapshot generation [5, 6]. Let’s consider the following problem domain: there are n interconnected restaurants. Waiters use PDAs to take orders and communicate them to the kitchen. These restaurants are small and need one or two waiters each. When the bill is asked,

¹ We leave the issue of type inference to the full paper.

orders are transmitted to the cash register; the latter writes down the receipt and sends the data on to a central server in charge of the accounting of the whole operation. Here, we only consider the interaction between restaurants and PDAs. The class diagram in Fig. 1 is the UML rendition of the problem domain.

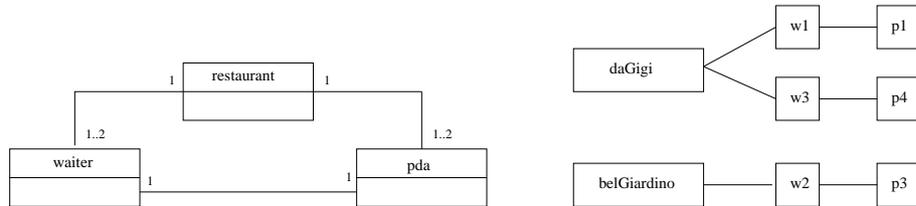


Fig. 1. UML diagram of the restaurant problem domain and a snapshot.

At first sight we do not seem to need types or function symbols to represent this in DLV – see the following code snippet:

```

restaurant(daGigi).
restaurant(belGiardino).

waiterId(w1).
waiterId(w2).
waiterId(w3).
waiterId(w4).

pdaId(p1).
pdaId(p2).
pdaId(p3).
pdaId(p4).

waiter(W) v -waiter(W) :- waiterId(W).
pda(P) v -pda(P) :- pdaId(P).
usesPda(W,P) v -usesPda(W,P) :- waiter(W), pda(P).
worksIn(W,R) v -worksIn(W,R) :- waiter(W), restaurant(R).

% at least 1 waiter works in R
:- restaurant(R), #count{W: worksIn(W,R)} < 1.
% at most 2 waiters work in R
:- restaurant(R), #count{W: worksIn(W,R)} > 2.
:- waiter(W), usesPda(W,P1), usesPda(W,P2), P1 != P2.

... plus other 5 constraints.
  
```

where #count is the DLV *aggregate* operator. We have used four constants for waiters and PDAs, because we want to incorporate, among all the possible models, those where there are two waiters for restaurant. Those constraints translate the UML multiplicity constraints. With this somewhat naive encoding, there are 864 solutions, where a large number of models represents equivalent situations. The snapshot in Fig. 1 corresponds to the answer set:

```
{worksIn(w1,daGigi), worksIn(w3,daGigi), worksIn(w2,belGiardino),
usesPda(w1,p1), ...}
```

With some ingenuity, we could decrease the size of ground instances by interpreting `worksIn(W,R)` as “W identifies a waiter in restaurant R”. For instance `worksIn(w1,daGigi), worksIn(w1,belGiardino)` means that there are two waiters, internally unidentified with `w1`, one each for restaurant. The same trick would work for PDAs. By adopting this representation, we do achieve a drastic reduction of the size of grounding and of the number of models. Unfortunately, we loose the tight connection between the DLV representation and what we wanted to model in the first place, in our case the correspondent UML object diagrams. Moreover, choosing different representations according to the situation yields the impossibility for a general method to map a UML model M and its snapshots in a DLV program P_M and its answer sets, respectively. On the other hand the use of well-typed functions offers a partial solution to this issue, while maintaining a clear and intuitive correspondence between the UML model and its representation. A typed DLV* version of the same code would go like that:

```
type restaurant --> daGigi ; belGiardino.
type code --> c1 ; c2.
type waiterId --> w(code, restaurant).
type pdaId --> p(code, restaurant).

pred waiter(waiterId, restaurant).
pred pda(pdaId, restaurant).
pred usesPda(waiterId, pdaId).

waiter(w(C,R),R) v -waiter(w(C,R),R).
pda(p(C,R),R) v -pda(p(C,R),R).

:- restaurant(R), #count{W: waiter(W,R)} < 1.
```

As you can see, the predicates usually needed for instance generation are included into the types and a single constraint is required. Note that the reduction to a DLV* program can be automated. More importantly, the number of models is now down to 9.

```
restaurant(belGiardino).
restaurant(daGigi).

code(c1).
code(c2).
```

```

waiter(w(C,R),R) v -waiter(w(C,R),R) :- code(C), restaurant(R).
pda(p(C,R),R) v -pda(p(C,R),R) :- code(C), restaurant(R).
usesPda(w(C,R),p(C,R)) :- code(C), restaurant(R).

:- restaurant(R), #count{W : waiter(W,R)} < 1.

```

Starting from the above example, we can now discuss some of the potential benefits of the introduction of types in DLV*. This impacts three directions:

- Using prescriptive types leads to clearer specifications and helps to discover specification errors at early stages. We believe that this is particularly important in a black-box environment such as Answer Set Programming; in fact, when, against expectation, no model is generated, it might be difficult to locate the mistakes, and even more frustrating to discover, in the end, that the problem was a trivial typing mistake. This is a well-known problem – compare it with the orthogonal work concerning *justifications* in ASP [14].
- Types may yield significant reduction of the size of grounding and impact its finiteness; it also reduces the generation of redundant variants of a solution.
- It makes possible to enforce some constraints “at compile time”, i.e. directly in the grounding phase.

Grounding with non recursive types Non recursive types are built on top of a set of predefined or user-defined *flat* types, namely types generated by constants. A new non-recursive type τ is introduced by a set of type constructors such as $c : \underline{\alpha} \rightarrow \tau$ that contain in their arity $\underline{\alpha}$ only types already defined. For example, the types used in the above example are non-recursive.

If all the types of a well-typed program are non-recursive, then its (well-typed) stable models are finite, even using infinite flat types, such as integers. This can be shown as follows. Let us consider a family of non recursive types generated by a set \mathcal{F} of function symbols and a set K of constant symbols. Let $C \subseteq K$ be a finite subset of constants, and let $Term(C)$ the set of the ground (well-typed) terms generated by the function symbols of \mathcal{F} and C . One easily sees that $Term(C)$ is finite. Now, let $grnd(P, Term(C))$ be the grounding of P by $Term(C)$ and $grnd(P, Term(K))$ be the grounding of P with respect to all the constants.

Theorem 1. *Let P be a well-typed and safe program and C the set of constant symbols occurring in P . Then, the stable models of $grnd(P, Term(C))$ and those of $grnd(P, Term(K))$ coincide.*

The proof is very similar to the proof of Theorem 3 of [4], the difference being that we are considering well-typed terms instead of constants.

Types and constraints There are cases where some constraints can be codified by typed functions. This essentially concerns different kinds of functional dependencies, such as those imposed in the UML by the multiplicity constraints that require a unique object on one of the association ends. For example, let’s assume that a class B is associated with a class A with multiplicity $1..*$, i.e., that every object of class A is to be

linked to a unique object of class B . We can enforce this constraint by generating the names of the objects of class A via the following type:

```
type name(A,B) --> nc(code(A), B).
```

The names of the objects of class A associated with an object b are $nc(a_1, b), \dots, nc(a_k, b)$. Clearly, we cannot have an object associated with two different b_1, b_2 because $nc(a_i, b_1)$ and $nc(a_i, b_2)$ are different names. We remark that A, B are type variables, i.e., that the type system is polymorphic, as we detail next. An instance could be $nc(\text{code}(\text{waiter}), \text{restaurant})$, similarly to what we did in Example 1.

3 Typed DLV*

The signature Σ of a typed DLV* specification $Spec$ has three parts:

- A set \mathcal{T} of type constructors, defining the set of *type expressions* of $Spec$. Let us indicate a type constructor with k arguments by c^k , a type expression by τ , and a type variable with v , where we assume denumerable and disjoint sets of individual and type variables. Hence

$$\tau := v \mid c^0 \mid c^k(\tau_1, \dots, \tau_k)$$

An *arity* will be a finite (possibly empty) list of type expressions $[\tau_1, \dots, \tau_m]$. We will denote an arity by $\underline{\alpha}$.

- A set \mathcal{F} of function declarations, of the form $f : \underline{\alpha}_f \rightarrow \tau_f$, introducing a function symbol f , its arity $\underline{\alpha}_f$ and its type τ_f .
- A set \mathcal{P} of predicate declarations, of the form $p : \underline{\alpha}_p$, introducing a predicate symbol p and its arity $\underline{\alpha}_p$.

Since a signature Σ will not change, we shall take it as fixed and leave it implicit in the judgments. A *context* is a finite set $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ of typed variables, where x_1, \dots, x_n are individual variables. We define well formed terms, atoms, literals together etc. by the following judgments:

- $\Gamma \vdash t : \tau$: t is a term of type τ in Γ ;
- $\Gamma \vdash \text{hlit } A$: A is of the form $p(\underline{t})$ or $\neg p(\underline{t})$; h denotes that A may occur in the head of a clause;
- $\Gamma \vdash \text{blit } A$: A is of the form $p(\underline{t})$, or $\neg p(\underline{t})$, or $\text{not } p(\underline{t})$ or $\text{not } \neg p(\underline{t})$; b denotes that A may occur in the body, but not in the head of a clause;
- $\Gamma \vdash \text{body } B$: B is a body in Γ ;
- $\Gamma \vdash \text{head } H$: H is a head in Γ ;
- $\vdash \text{clause } C$: C is a clause;
- $\vdash \text{prog } P$: P is a program.

$$\begin{array}{c}
\frac{}{\Gamma \cup \{X : \tau\} \vdash X : \tau} \text{ var} \\
\\
\frac{\Gamma \vdash t_1 : \tau_1 \delta \quad \dots \quad \Gamma \vdash t_n : \tau_n \delta}{\Gamma \vdash f(t_1, \dots, t_n) : \tau \delta} \text{ for } f : [\tau_1, \dots, \tau_n] \rightarrow \tau \in \mathcal{F} \ \delta \text{ a type subst.} \\
\\
\frac{\Gamma \vdash t_1 : \tau_1 \delta \quad \dots \quad \Gamma \vdash t_n : \tau_n \delta}{\Gamma \vdash \text{hlit } [-] p(t_1, \dots, t_n)} \text{ for } p : [\tau_1, \dots, \tau_n] \in \mathcal{P} \ \delta \text{ a type subst.} \\
\\
\frac{\Gamma \vdash t_1 : \tau_1 \delta \quad \dots \quad \Gamma \vdash t_n : \tau_n \delta}{\Gamma \vdash \text{blit } [\text{not}] [-] p(t_1, \dots, t_n)} \text{ for } p : [\tau_1, \dots, \tau_n] \in \mathcal{P} \ \delta \text{ a type subst.} \\
\\
\frac{\Gamma \vdash \text{blit } L_1 \quad \dots \quad \Gamma \vdash \text{blit } L_n}{\Gamma \vdash \text{body } L_1 \wedge \dots \wedge L_n} \quad \frac{\Gamma \vdash \text{hlit } A_1 \quad \dots \quad \Gamma \vdash \text{hlit } A_h}{\Gamma \vdash \text{head } A_1 \vee \dots \vee A_h} \\
\\
\frac{\Gamma \vdash \text{head } H \quad \Gamma \vdash \text{body } B}{\vdash \text{clause } \forall \Gamma (H \leftarrow B)} \quad \frac{\vdash \text{clause } C_1 \quad \dots \quad \vdash \text{clause } C_n}{\vdash \text{prog } C_1 \wedge \dots \wedge C_n}
\end{array}$$

Fig. 2. Typing rules

The rules are depicted in Fig. 2, where we put between square brackets optional connectives; δ is any substitution replacing type variables with types. Note that weakening (w) on any judgment J is admissible:

$$\frac{\Gamma \vdash J}{\Gamma \cup \Delta \vdash J} w$$

Let $C(\underline{v}) = \forall \{x_1 : \tau_1, \dots, x_n : \tau_n\} (H \leftarrow B)$ be a clause, where \underline{v} are the (possible) type variables occurring in τ_1, \dots, τ_n . We indicate with θ a substitution replacing \underline{v} by ground types, and by σ_θ a substitution replacing each individual variable x_j by a ground term t_j such that $\vdash t_j : \tau_j \theta$. We then say that $(H \leftarrow B)\sigma_\theta$ is a ground instance of $C(\underline{v})$. By $\text{ground}(P)$ we mean the set of all the ground instances of the clauses of P . Answer sets of $\text{ground}(P)$ are defined as usual². We can prove that the answer sets of a well-typed program are well-typed.

Theorem 2. *Let P be a program such that $\vdash \text{prog } P$, $\text{ground}(P)$ its grounding and I an answer set of P . Then, for every $L \in I$, $\vdash \text{hlit } L$.*

4 Conclusions

We have discussed the introduction of types in DLV* presenting a basic polymorphic type system. Although our research is at an early stage, we believe that it shows the potential interest of typed ASP, starting with DLV*, but also for other Answer Set systems

² Some care is needed to properly treat predefined types.

such as Smodels [12]. Beside the well-known advantages of prescriptive typing, in the ASP setting one may reap significant benefits w.r.t. the size of grounding; finally, there are constraints that can be directly codified in terms of typing, further reducing the size of the search space. Some of the next issues we plan to tackle are:

- Develop a systematic study of typed value invention;
- quantitatively evaluate the impact of well-typing on the efficiency of grounding;
- extend the type system with other constructs, such as aggregate types, subtyping, and possibly a limited form of dependent types. In particular, aggregate types can be defined in a clean way considering concrete data as in [2];
- investigate whether this approach can benefit finitely recursive programs [1, 3].

References

1. S. Baselice, P. A. Bonatti, and G. Criscuolo. On finitely recursive programs. In V. Dahl and I. Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2007.
2. A. Bertoni, G. Mauri, and P. Miglioli. A characterization of abstract data as model-theoretic invariants. In Hermann A. Maurer, editor, *ICALP*, volume 71 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 1979.
3. P. A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.
4. F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Ann. Math. Artif. Intell.*, 50(3-4):333–361, 2007.
5. M. Ferrari, C. Fiorentini, A. Momigliano, and M. Ornaghi. Snapshot generation in a constructive object-oriented modeling language. In A. King, editor, *Logic Based Program Synthesis and Transformation, LOPSTR'07. Revised Selected Papers*, volume 4915 of *LNCS*, pages 169–184. Springer-Verlag, 2008.
6. Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
7. Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Programming*, 58(1-2):115–140, 2005.
8. D. Jefferey. *Expressive Type Systems for Logic Programming Languages*. PhD thesis, The University of Melbourne, 2002.
9. T. L. Lakshman and Uday S. Reddy. Typed PROLOG: A semantic reconstruction of the Mycroft-O’Keefe type system. In *ISLP*, pages 202–217, 1991.
10. Nicola Leone and et al. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.
11. Alan Mycroft and Richard A. O’Keefe. A polymorphic type system for PROLOG. *Artif. Intell.*, 23(3):295–307, 1984.
12. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *LPNMR*, volume 1265 of *LNCS*, pages 421–430. Springer, 1997.
13. Frank Pfenning. *Types in logic programming*. MIT Press, Cambridge, MA, USA, 1992.
14. Enrico Pontelli and Tran Cao Son. Justifications for logic programs under answer set semantics. In Sandro Etalle and Mirosław Truszczyński, editors, *ICLP*, volume 4079 of *LNCS*, pages 196–210. Springer, 2006.

Modeling preferences on resource consumption and production in ASP

Stefania Costantini¹ and Andrea Formisano²

¹ Università di L'Aquila. stefcost@di.univaq.it

² Università di Perugia. formis@dipmat.unipg.it

Abstract. Recently we have proposed RASP, an extension of Answer Set Programming that permits declarative specification and reasoning on consumption and production of resources. Resources are modeled by introducing *amount-atoms*, involving *quantities* that represent the available amount of resources. Processes that use resources are easily described through program rules and solutions correspond to different possible allocations of available resources.

In this paper, we extend this framework to allow the declarative specification of preferences among alternative use of different resources.

We provide semantics for the resulting system and sketch a possible implementation based on standard ASP-solvers. The implementation consists of a standard translation of each rule into a set of plain ASP rules and of an inference engine that manages the firing of rules, the allocation of resources, the satisfaction of user constraints on resource usage. The preferences expressed on resource usage induce a preference order on answer sets. In this initial implementation, such an order is rendered through optimization features provided by standard ASP-solvers.

Key words: Answer set programming, quantitative reasoning, preferences, non-monotonic logic programming, language extensions.

Introduction

As it is well-known, Answer Set Programming (ASP) is a form of logic programming based on the answer set semantics [18], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [26, 28]. ASP is nowadays applied in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see, e.g., [8, 41, 2]).

However, the possibility was lacking of performing some kind of quantitative reasoning which is instead possible in non-classical logics such as e.g. Linear Logics [20] and Description Logics [6]. In recent work [13], we have extended ASP in order to support declarative reasoning on consumption and production of resources. The extension has been called RASP, standing for Resourced ASP.

In the present paper, we go further in this extension, by adding declarative *preferences* to the specification of production/consumption processes. In particular, in realizing the same process (modeled through the firing of rules), one may

prefer to produce and/or consume certain resources rather than another ones. This extension can be particularly useful in configuration applications where one can, for instance, prefer to save money while spending more time or vice versa or may prefer to employ a certain amount of cheap components rather than a little amount of expensive parts.

Let us briefly recall syntax and intended semantics of RASP programs through a simple example. We will then modify such example to informally introduce preferences.

A RASP program is composed of r-facts and r-rules, where numbers associated with the heads of r-facts and rules indicate which amount of a certain resource is respectively:

- available, in case of r-facts;
- produced, in case of r-rules, where production can take place if the body holds, which implies that required resources are either available or have been produced.

As resources that are either available or produced can be in turn consumed, then numbers (quantities) can be associated to atoms occurring in the bodies of r-rules as well, as shown below.

The example concerns the preparation of desserts. We may notice that different solutions stem in this case from the fact that, with the available ingredients, one may prepare either a cake or an ice-cream, but not both. Atoms of the form $q:a$ are called *amount atoms*, where the *amount symbol* a states which is the quantity of that resource which is either produced (if the amount atom is in the head of a rule) or consumed (if it is in the body) or available (if it is a fact) respectively.

```
cake:1 :- egg:3, flour:4, sugar:3.
ice_cream:1 :- egg:2, sugar:2, milk:2.
egg:3.                flour:8.
sugar:6.              milk:3.
```

In RASP one can specify that any given rule can be repeatedly fired. Bounds can be imposed on the number of times each rule is used. For instance, the rule below specifies that you can prepare a cake using eggs, flour, and sugar. The bound [2-4] states that this rule, if used, has to be used $2 \leq n \leq 4$ times (to make from two to four cakes). Enough resources have to be available to support all of the n firings. Notice that the possibility of not using the rule is still viable.

```
[2-4]: cake:1 :- egg:3, flour:4, sugar:3.
```

In general, usual ASP literals (possibly involving negation-as-failure) may occur in rules. Semantics of a RASP program is determined by interpreting usual literals as in ASP (i.e., by exploiting stable model semantics) and amount-atoms in an auxiliary algebraic structure (that supports operations and comparisons). For instance, we could modify the rule to make ice-cream by requiring that ice-cream can be made only when there is a fridge available and if there is someone who is a good cook:

```

ice_cream:1 :- egg:2, sugar:2, milk:2, fridge, a_cook_is_here.
a_cook_is_here :- is_here(remy), is_here(linguini).
is_here(remy).          is_here(linguini).

```

Intuitively, the first rule of this program is applicable only in correspondence of models that satisfy the literals `fridge` and `a_cook_is_here`.

RASP already offers some constructs that can be used to express basic forms of preferences on rule firing and resource consumption/production (cf. [13]). A number of budget policies are exploitable to control rule firings and, consequently, to influence what resources to produce and in which quantity:

Thrifty. An r-rule γ is fired only if this is forced.

Prodigal. Whenever an r-rule γ can be fired, it must be fired.

Optional. Different resource allocations can enable the firing of different rules, possibly in antithetic manners. Enabled rules might be not necessarily fired.

The various policies can be combined in a mixed strategy by choosing one of them for each single rule of the program. Another limited form of preferences can be realized in RASP by associating *costs* to rule's firings. Namely, if γ is an r-rule and C is an amount-symbol, then a writing of the form: $C : \gamma$ states that C represents the cost of each firing of γ . Depending on which rules are fired, we can associate a cost to each solution. Hence, it is possible to design specific cost-based criteria to impose preferences among answer sets. All of these features are fully accounted for in the RASP semantics and are dealt with in [13]. In what follows we do not treat these aspect of RASP, but we develop some more expressive form of preference on resource usage.

Recall the initial example and suppose that you might prepare a cake either with corn flour or with potato flour. The RASP rules:

```

cake:1 :- egg:3, flour:4, sugar:3.
cake:1 :- egg:3, potato_flour:3, sugar:3.

```

express the two possibilities, but do not say which one you would prefer, assuming both alternatives to be feasible.

We propose in this paper P-RASP (RASP with preferences), to allow one to explicitly state which resource (s)he would prefer to use, e.g., in the above example, the formulation

```

cake:1 :- potato_flour:3>flour:4, egg:3, sugar:3.

```

indicates that consuming potato flour is preferred onto consuming corn flour. Or also, if the recipe includes milk, one might prefer to use skim milk if available:

```

cake:1 :- potato_flour:3>flour:4, skim_milk:2>whole_milk:2,
egg:3, sugar:3.

```

In this reformulation, we have two *preference lists*, or for short p-lists. Actually, p-lists may involve any number of amount-atoms. The intuitive reading is that leftmost elements of a p-list have higher priority. P-lists may occur in the head

of r-rules, as shown in the example below, where one prefers to employ available ingredients to make an ice-cream instead of two cups of zabaglione:

```
ice-cream:1>zabaglione:2 :- skim_milk:2>whole_milk:2, egg:2, sugar:3.
```

The introduction of p-lists requires a concept of *preferred answer set*. In case several p-lists occur either in one rule or in different r-rules, it is necessary to establish which answer set better satisfies the preferences. Intuitively, if we choose to consider as “better” the answer sets which satisfy the higher number of leftmost elements, in the last example we would have:

- Producing an ice-cream with skim milk is the best solution.
- Producing:
 - (a) an ice-cream with whole milk or
 - (b) two zabagliones with skim milk
 would be equally good (but worse than the previous solution) as each of them employs the leftmost element of a p-list.
- Producing two zabagliones with whole milk is the less preferred solution.

Clearly, one has to choose the best *possible* solution, given the available resources. One might choose other strategies, e.g. one might give higher priorities to p-lists in rule heads, where consequently solution (a) above would become better than (b). One may also imagine to introduce a choice among different strategies to be employed in different contexts.

Finally, preferences may be conditional. One may for instance prefer skim milk when on a diet, i.e. the last rule may become:

```
ice-cream:1>zabaglione:2 :- (skim_milk:2>whole_milk:2 if diet),
                             egg:2, sugar:3.
```

If `diet` does not hold, then the preference list reduces to a disjunction, i.e. either skim milk or whole milk can indifferently be used. The above rule becomes equivalent to the couple of rules:

```
ice-cream:1>zabaglione:2 :- skim_milk:2, egg:2, sugar:3.
ice-cream:1>zabaglione:2 :- whole_milk:2, egg:2, sugar:3.
```

This generalizes to several p-lists, like in the example below:

```
(ice-cream:1>zabaglione:2 if summer) :-
  (skim_milk:2>whole_milk:2 if diet), egg:2, sugar:3.
```

In this paper, we propose a definition of P-RASP and its semantics, and a comparison with related work. We notice that P-RASP has been fully implemented. The implementation consists of a refinement of the treatment designed for RASP and fully described in [13]. For the sake of space, a full description of the implementation is out of the scope of this paper. The interested reader can refer to [13].

The plan of the paper is the following: in Section 1 we briefly summarize the RASP syntax and the needed extensions to deal with preferences. In Section 2 we

formally account for the semantics of P-RASP (which has RASP’s one as particular case). Section 3 introduces conditional preferences. In Section 4 we briefly sketch some features of the implementation. Finally, we outline a comparison with relevant related work (Section 5) and we draw some conclusions.

1 From RASP to P-RASP: Syntax

In order to formally introduce syntax and semantics of P-RASP, we need to briefly summarize the basic notions about RASP syntax as presented in [13].

To accommodate the new language expressions that involve resources and their quantities, the underlying language of RASP is partitioned into *Program* symbols and *Resource* symbols. Precisely, let $\langle \Pi, \mathcal{C}, \mathcal{V} \rangle$ be an alphabet where $\Pi = \Pi_P \cup \Pi_R$ is a set of predicate symbols such that $\Pi_P \cap \Pi_R = \emptyset$, $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_R$ is a set of symbols of constant such that $\mathcal{C}_P \cap \mathcal{C}_R = \emptyset$, and \mathcal{V} is a set of symbols of variable. The elements of \mathcal{C}_R are said *amount-symbols*, while the elements of Π_R are said *resource-predicates*. A *program-term* is either a variable or a constant symbol. An *amount-term* is either a variable or an amount-symbol.

Amount-atoms are introduced in addition to plain ASP atoms, here called program atoms. Let $\mathcal{A}(X, Y)$ denote the collection of all atoms of the form $p(t_1, \dots, t_n)$, with $p \in X$ and $\{t_1, \dots, t_n\} \subseteq Y$. Then, a *program atom* is an element of $\mathcal{A}(\Pi_P, \mathcal{C} \cup \mathcal{V})$. As usual, a *ground* amount-atom contains no variables. An *amount-atom* is a writing of the form $q:a$ where $q \in \Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C} \cup \mathcal{V})$ and a is an amount-term. Let $\tau_R = \Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C})$. We call elements of τ_R *resource-symbols*. E.g., in the two expressions $p:3$ and $q(2):b$, p and $q(2)$ are resource-symbols (with $p, q \in \Pi_R$ and $2 \in \mathcal{C}$) aimed at defining two resources which are available in quantity 3 and b , resp., (with $3, b \in \mathcal{C}_R$ amount-symbols). Expressions such as $p(X):V$ where V, X are variable symbols are also allowed, as quantities can be either directly defined as constants or derived. Notice that the set of variables is not partitioned, as the same variable may occur both as a program term and as an amount-term. *Ground* amount-atoms contain no variables. As usual, a *program-literal* L is a program-atom A or the negation *not* A of a program-atom (intended as negation-as-failure).¹ If $L = A$ (resp., $L = \text{not } A$) then \bar{L} denotes *not* A (resp., A).

Definition 1. A resource-literal (*r-literal*) is either a program-literal or an amount-atom.

Therefore, we do not allow negation of amount-atoms. (See [13] for a discussion about this point.) Finally, we distinguish between plain rules and rules that involve amount-atoms. In particular, a *program-rule* is defined as a regular ASP rule, including the case of ASP *constraints*, i.e., rules with empty head. Beside program-rules we introduce resource-rules which differ from program rules in that they may contain amount-atoms.

¹ We will only deal with negation-as-failure. Though, classical negation of program literals could be used in (P-)RASP programs and treated as usually done in ASP.

Definition 2. A resource-proper-rule has the form

$$Idx : H \leftarrow B_1, \dots, B_k \tag{1}$$

where B_1, \dots, B_k , $k > 0$ are r -literals and H is either a program-atom or a (non-empty) list of amount-atoms. Idx is of the form $[N_1-N_2]$, and each N_j is a variable or a positive integer number.

Intuitively, when N_1 and N_2 are integers, Idx denotes a (possibly void) interval in $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. It is intended to restrain the number of times the rule can be used: such number must be in Idx or the rule cannot be used at all. For the sake of generality, we admit that each N_j is a variable. Then, after grounding (see below), each N_j has to be instantiated to a positive integer.

A piece of notation: We will not write the list Idx when all N_j s are intended to be the constant 1 (meaning that at most one use of the rule is admitted).

Example 1. We can take advantage from having lists of amount-atoms as heads of rules, to model a chemical reaction. We consider a simplified description of the combustion of methane. In such a reaction the combination of a molecule of methane and two molecules of oxygen produces a molecule of carbon dioxide and two molecules of water. A chemical equation that schematize the reaction is: $CH_4 + 2O_2 \Rightarrow CO_2 + 2H_2O$. We can encode such a reaction with a single rule that produces two resources:

```
water:2, carbDioxide:1 :- methane:1, oxygen:2.
```

The following definition introduces the notion of resource-facts. Resource-facts that are not supposed to be iterable, since they are intended to model the fixed amount of resources that are available “from the beginning”.

Definition 3. A resource-fact (r -fact, for short) has the form

$$H \leftarrow$$

where H is an amount-atom $q:a$ and a is an amount-symbol.

According to the definition, the amount of an initially available resource has to be explicitly stated. Thus, in an r -fact the amount-term a cannot be a variable.

Definition 4. A resource-rule (r -rule, for short) can be either a resource-proper-rule or a resource-fact. A RASP-rule (rule, for short) γ is either a program-rule or a resource-rule. An r -program is a finite set of RASP-rules.

Let $\mathcal{B}_{\mathcal{H}}(X, Y)$ denote the collection of all ground atoms built up from predicate symbols in X and terms in Y . The grounding of a program P is the set of all ground instances of rules of P , obtained through ground substitutions over the Herbrand universe of P . As it is well-known, ASP solvers usually produce the grounding of the given program as a first step. Then, it is convenient to impose suitable constraints on r -programs so to ensure finiteness of the grounding process. To this aim, we impose usual restrictions (e.g., *range restriction*, cf. [8]) to

all variables occurring in a P-RASP program, including variables that restrain the number of iterations of a resource-proper-rule (cf. $N_{j,\ell}$ in Def. 2).

P-RASP programs are obtained from RASP programs by introducing alternatives in using resources expressed by preference lists:

Definition 5. A preference-list of amount-atoms (p-list, for short) is a writing of the form $q_1:a_1 > \dots > q_h:a_h$, where $h \geq 2$ and the q_i 's are distinct amount-atoms involving distinct resource-symbols. We say that the amount-atom $q_i:a_i$ has grade of preference i in the p-list.

We have now to extend the definition of an r-rule accordingly. This is done by including p-lists in r-literals:

Definition 6. A P-RASP resource-literal (*r-literal*) is either a program-literal or an amount-atom or a p-list.

In practice, P-RASP rules differ from RASP rules in that p-lists are admitted in place of amount-atoms. More precisely, the syntax of an r-rule in P-RASP is defined as in Def. 2 where in (1) some of the B_1, \dots, B_k, H may be p-lists.

Intuitively speaking, a p-list plays a role similar to an exclusive disjunction of amount-atoms. If a p-list occurs in the body (resp. head) of a rule, it encodes the requirement that one (and only one) resource among q_1, \dots, q_h has to be consumed (resp. produced), in the indicated amount, if the rule is fired. Moreover, q_i is preferred to q_j , for $i < j$.

2 Semantics of P-RASP

Semantics of a (ground) P-RASP program is determined by interpreting program-literals as in ASP and amount-atoms in an auxiliary algebraic structure that supports operations and comparisons. The rationale behind the proposed semantic definition is the following. On the one hand, we translate r-rules into a fragment of a plain ASP program, so that we do not have to modify the definition of stability which remains the same: this is of some importance in order to make the several theoretical and practical advances in ASP still available for RASP and P-RASP. However, an answer set of a P-RASP program will contain the head of a resource-rule only if: the rule is satisfied (in the usual way) as concerns its program-literals; and the requested amount is allocated for the resource-atoms. An interpretation involves the allocation of actual quantities to amount-atoms. In fact, this allocation is one of the components of an interpretation. A last component copes with the repeated firing of a rule: in case of several firings, the resource allocation must be iterated accordingly.

2.1 Modeling Amounts.

In order to define semantics of r-programs, we have to fix an interpretation for amount-symbols. This is done by choosing a collection Q of *quantities*, and the operations to combine and compare quantities. A natural choice is $Q = \mathbb{Z}$:

thus, we consider given a mapping $\kappa : \mathcal{C}_R \rightarrow \mathbb{Z}$ that associates integers to amount-symbols. Positive (resp. negative) integers will be used to model produced (resp. consumed) amounts of resources.

Remark 1. Alternative options for Q are possible. For instance, one could choose Q to be the set \mathbb{Q} of rational numbers. In general, amounts can be interpreted by choosing Q to be any set as carrier of a structure $\mathcal{Q} = \langle Q, \oplus_{\mathcal{Q}}, \preceq_{\mathcal{Q}}, \mathbf{0}_{\mathcal{Q}} \rangle$ that is a totally-ordered Abelian group (where $\mathbf{0}_{\mathcal{Q}}$ denotes the identity element in \mathcal{Q} w.r.t. the additive operation $\oplus_{\mathcal{Q}}$, and $\preceq_{\mathcal{Q}}$ is a translation invariant total order over Q .)

2.2 Notation.

Before going on, we introduce some useful notation. Given two sets X, Y , let $\mathcal{FM}(X)$ denote the collection of all finite multisets² of elements of X , and let Y^X denote the collection of all (total) functions having X and Y as domain and codomain, respectively. For any (multi)set Z of integers, $\sum(Z)$ denotes their sum. E.g., $\sum(\{\{2, 5, 3, 3, 5\}\}) = 18$.

Given a collection S of (non-empty) sets, a *choice function* $c(\cdot)$ for S is a function having S as domain and such that for each s in S , $c(s)$ is an element of s . In other words, $c(\cdot)$ chooses exactly one element from each set in S .

In order to deal with the disjunctive aspect of p-lists, we mark each resource amount with an integer index. Such indexes are intended to model the grade of preference of the amount-atoms occurring in p-lists. So, any amount-atom will be represented as a pair in $\mathbb{N} \times Q$ that we call an *amount couple*. Then, for each p-list, the composing amount-atoms will be associated from left to right with successive indexes starting from 1; for simple amount-atoms, the index will always be 0. For instance: for amount-atom `egg:2` the representation is $\{\{ \langle 0, 2 \rangle \}\}$ where the first element is the grade of preference (which in this case is 0 as no preference is involved) and the second element is the quantity; for p-list `skim_milk:2 > whole_milk:2`, the representation is $\{\{ \langle 1, 2 \rangle \}, \{\{ \langle 2, 2 \rangle \}\}$ where one can see the increasing grade of preference. The representation can be extended to entire rules and to the entire program.

Given an amount couple $r = \langle n, x \rangle \in \mathbb{N} \times Q$, let $grade(r) = n$ and $amount(r) = x$. Notice that the amount can in principle be negative (e.g., if $Q = \mathbb{Z}$). We extend such a notation to sets and multisets of amount couples, as one expects: namely, if X is a multiset then $grade(X)$ is defined as the multiset $\{\{ n \mid \langle n, x \rangle \text{ is in } X \}\}$. E.g., if $X = \{\{ \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 0, 1 \rangle, \langle 1, 2 \rangle \}\}$ then $grade(X)$ is $\{\{ 1, 2, 3, 0, 1 \}\}$ and $amount(X)$ is $\{\{ 2, 3, 1, 1, 2 \}\}$.

Let ℓ be either an amount-atom or a p-list in a resource-rule γ . Let

$$setify(\ell) = \begin{cases} \{\{ \langle 0, q, a \rangle \}\} & \text{if } \ell \text{ is } q:a \\ \{\{ \langle 1, q_1, a_1 \rangle, \dots, \langle h, q_h, a_h \rangle \}\} & \text{if } \ell \text{ is } q_1:a_1 > \dots > q_h:a_h \end{cases}$$

² A multiset (or bag) is a generalization of a set, where repeated elements are allowed. Then, a member of a multiset can have more than one membership.

We will use *setify* to represent the amount-atoms of rules as triples denoting: the position in each preference list where they occur; the resource-symbol they contain; the amount that is required for this resource-symbol in that preference list. We generalize the notion to any multiset X of amount-atoms and p-lists: $setify(X) = \{\{setify(\ell) \mid \ell \text{ in } X\}\}$.

Let $r\text{-head}(\gamma)$ and $r\text{-body}(\gamma)$ denote the multiset of amount-atoms or p-lists occurring in the head and in the body of γ , respectively. In order to distinguish, in the representation, between amount-atoms occurring in heads and in bodies, we define $setify_b(\gamma)$ and $setify_h(\gamma)$ as the multisets $\{\{setify(x) \mid x \in r\text{-body}(\gamma)\}\}$ and $\{\{setify(x) \mid x \in r\text{-head}(\gamma)\}\}$, respectively.

2.3 Interpretation of P-RASP Programs.

In what follows, we will apply a syntactical restriction on the form of the r-rules. Namely, we impose that each amount-atom cannot occur in more than one p-list within the same rule. (Clearly, a $q:a$ can occur in several p-lists of different rules.) Though this restriction is not strictly needed, for the sake of simplicity we focus on this simplified case.

We introduce now the notion of r-interpretation for r-programs. An interpretation of a (ground) r-program P must determine an allocation of amounts for all occurrences of such symbol in rules of P : in fact, below we introduce the first step of an interpretation, i.e., the actual assignment of amounts to the amount-atoms that occur in r-rules.

Since amounts and resource-symbols are used to model production and consumption of “real world” objects, we must take into account the obvious constraint that any resource cannot be consumed if it is not produced. Thus, we restrain to those allocations having (for each resource) a non-negative global balance. The collection \mathbb{S}_P of all potential allocations—for any single resource-symbol occurring in the program P (considered as a set of rules)—is the following collection of mappings:

$$\mathbb{S}_P = \left\{ F \in (\mathcal{FM}(\mathbb{N} \times Q))^P \mid 0 \leq \sum \left(\bigcup_{\gamma \in P} amount(F(\gamma)) \right) \right\} \quad (2)$$

The rationale behind the definition of \mathbb{S}_P is as follows: Let q be a given resource-symbol. Each element $F \in \mathbb{S}_P$ is a function that associates to every rule $\gamma \in P$ a (possibly empty) multiset $F(\gamma)$ of amount couples, assigning certain amounts to each occurrence of amount-atoms $q:a$ in γ . All such F s must satisfy the only requirement that, considering the entire P , the global sum of all the quantities F assigns must be non-negative. As we will see later, only some of these allocations will actually be acceptable as a basis for a model.

To interpret an r-program, we select a collection of elements of \mathbb{S}_P , one element for each resource-symbol in τ_R . More formally, an r-interpretation of a ground r-program P is defined by providing a mapping

$$\mu : \tau_R \rightarrow \mathbb{S}_P.$$

Such a function μ determines, for each resource-symbol $q \in \tau_R$, a mapping

$\mu(q) \in \mathbb{S}_P$. In turn, each mapping $\mu(q)$ assigns to each rule $\gamma \in P$ a multiset $\mu(q)(\gamma)$ of quantities. The use of multisets allows us to handle multiple copies of the same amount-atom. Each of these copies must be taken into account, since it corresponds to a different amount of resource.

Definition 7. An r-interpretation for a (ground) r-program P is a triple $\mathcal{I} = \langle I, \mu, \xi \rangle$, with $I \subseteq \mathcal{B}_{\mathcal{H}}(\Pi_P, \mathcal{C})$, $\mu : \tau_R \rightarrow \mathbb{S}_P$, and ξ a mapping $\xi : P \rightarrow \mathbb{N}^+$.

Intuitively: I plays the role of a usual answer set assigning truth values to program-literals; μ describes an allocation of resources; ξ associates to each rule an integer representing the number of times the (iterable) rule is used. By little abuse of notation, we consider ξ to be defined also for program-rules and r-facts. For these kind of r-rules we assume the interval $[N_1-N_2] = [1-1]$ as implicitly specified in the rule definition, as constraint on the number of firings.

Clearly, the firing of an r-rule (which may involve consumption/production of resources) can happen only if the truth values of the program-literals satisfy the rule. We reflect the fact that the satisfaction of an r-rule γ depends on the truth of its program-literals by introducing a suitable fragment of ASP program $\hat{\gamma}$. Let the r-rule γ , have L_1, \dots, L_k as program-literals and R_1, \dots, R_h as amount-atoms (or p-lists). The ASP-program $\hat{\gamma}$ is so defined:

$$\hat{\gamma} = \begin{cases} \{ \leftarrow \overline{L_1}, \dots, \leftarrow \overline{L_k} \} & \text{if the head of } \gamma \text{ consists of amount-atoms or p-lists} \\ \{ \leftarrow \overline{L_1}, \dots, \leftarrow \overline{L_k}, \\ \quad H \leftarrow L_1, \dots, L_k \} & \text{if } \gamma \text{ has the program-atom } H \text{ as head and } h > 0 \\ \{ \gamma \} & \text{otherwise (e.g., } \gamma \text{ is a program-rule).} \end{cases}$$

Def. 8, to be seen, states that in order to be a model, an r-interpretation that allocates non-void amounts to the resource-symbols of γ , has to model the ASP-rules in $\hat{\gamma}$. Some preliminary notion is in order.

We associate to each r-rule γ , the following set $\mathcal{R}(\gamma)$ of multisets. Each element of $\mathcal{R}(\gamma)$ represents a possible admissible selection of one amount-atom from each of the p-lists in γ and an actual allocation of an amount, taken in Q via the function κ , to the amount-symbol occurring in it. Notice that the quantities associated to amount-atoms occurring in the body of γ are negative, as these resources are *consumed*.³ Vice versa, the quantities associated to amount-atoms occurring in the head are positive, as these resources are *produced*.

$$\mathcal{R}(\gamma) = \left\{ \begin{aligned} & \{ \{ \langle i, q, \kappa(a) \rangle \mid \langle i, q, a \rangle = c_1(S_1) \text{ and } S_1 \text{ in } \textit{setify}_h(\gamma) \} \\ & \cup \{ \langle i, q, -\kappa(a) \rangle \mid \langle i, q, a \rangle = c_2(S_2) \text{ and } S_2 \text{ in } \textit{setify}_b(\gamma) \} \\ & \mid \text{for } c_1 \text{ and } c_2 \text{ choice functions for } \textit{setify}_h(\gamma) \text{ and } \textit{setify}_b(\gamma), \text{ resp.} \} \end{aligned} \right\}$$

where c_1 (resp. c_2) ranges on all possible choice functions for $\textit{setify}_h(\gamma)$ (resp. for $\textit{setify}_b(\gamma)$). In order to account for multiple firing of rules, we need to be able

³ To be precise, the assigned quantity corresponds to the negation of the amount occurring in an amount-atom of the body. One may also specify negative *byproducts* in the body, which are produced and not consumed: in such a case, the assigned quantity will be positive (cf., [13]).

to “iterate” the allocation of quantities for a number n of times: to this aim, for any $n \in \mathbb{N}^+$ and $q \in \tau_R$, let

$$\mathcal{R}^n(\gamma) = \left\{ \bigcup \llbracket X_1, \dots, X_n \rrbracket \mid \llbracket X_1, \dots, X_n \rrbracket \in \mathcal{FM}(\mathcal{R}(\gamma)) \right\}$$

and

$$\mathcal{R}^n(q, \gamma) = \left\{ \llbracket \langle i, v \rangle \mid \langle i, q, v \rangle \text{ is in } X \rrbracket \mid X \in \mathcal{R}^n(\gamma) \right\}$$

Definition 8. Let $\mathcal{I} = \langle I, \mu, \xi \rangle$ be an r -interpretation for a (ground) r -program P . \mathcal{I} is an answer set for P if the following conditions hold:

- for all rules $\gamma \in P$
 $(\forall q \in \tau_R (\mu(q)(\gamma) = \emptyset)) \vee (\forall q \in \tau_R (\mu(q)(\gamma) \in \mathcal{R}^{\xi(\gamma)}(q, \gamma)) \wedge (N_{1,1} \leq \xi(\gamma) \leq N_{1,2}))$
- I is a stable model for the ASP-program \widehat{P} , so defined

$$\widehat{P} = \bigcup \left\{ \widehat{\gamma} \mid \begin{array}{l} \gamma \text{ is a program-rule in } P, \text{ or} \\ \gamma \text{ is a resource-rule in } P \text{ and } \exists q \in \tau_R (\mu(q)(\gamma) \neq \emptyset) \end{array} \right\}$$

The two disjuncts in the formula in Def. 8 correspond to the two cases: a) the rule γ is not fired, so null amounts are allocated to all its amount-symbols; b) the rule γ is actually fired $\xi(\gamma)$ times and all needed amounts are allocated (by definition this happens if and only if $\exists q \in \tau_R (\mu(q)(\gamma) \neq \emptyset)$ holds). Notice that case b) imposes that the amount couples assigned by μ to a resource q in a rule γ reflect one of the possible choices in $\mathcal{R}^{\xi(\gamma)}(q, \gamma)$.

We now formally introduce the notions of *rule firing* and *resource balance*:

Definition 9. Let $\mathcal{I} = \langle I, \mu, \xi \rangle$ be an answer set for a (ground) r -program P . The resource balance for P , w.r.t. $\langle I, \mu, \xi \rangle$, is the mapping $\varphi : \tau_R \rightarrow \mathbb{Z}$ defined as:

$$\varphi(q) = \sum \left(\left\{ \sum (\text{amount}(\mu(q)(\gamma))) \mid \gamma \in P \right\} \right)$$

which summarizes consumptions and productions of all resources.

Finally, we say that an r -interpretation \mathcal{I} is an answer set of an r -program P if it is an answer set for the grounding of P . Note that, the above definition however does not in general fulfill the preferences expressed through p-lists.

Remark 2. As mentioned, the use of p-lists describes preferences among alternative uses of resources. Notice that such kind of preference have a *local scope*: each p-list is seen in the context of a particular rule (which models a specific process manipulating some amounts of resources). Clearly, such a local aspect is strictly correlated with the constraints on global resource balance and resource availability (as well as with the adopted policies and cost-based criteria). Consequently, preferences locally stated for different rules might/should be expected to interact “over distance” with those expressed in other rules.

In order to impose a preference order on the answer sets of an r -program, we need to provide a *preference criterion* to compare answer sets. Such a criterion should impose an order on the collection of answer sets by reflecting the (preference grades in the) p-lists. Any criterion \mathcal{PC} has to take into account that each rule

determines a (partial) preference ordering on answer sets. In a sense, \mathcal{PC} should aggregate/combine all “local” partial order to obtain a global one.

Fundamental techniques for combining preferences (seen as generic binary relations) can be found for instance in [1]. Regarding combination of preferences in Logic Programming, criteria are also given, for instance, in [7, 11, 10, 36].

Here we will just consider for P-RASP two of the simpler criteria among the variety of alternative possible choices. As a first example, we directly exploit the ordering of amount-atoms in the p-lists (i.e., their relative position). For any multiset m in $\mathcal{FM}(\mathbb{N} \times Q)$ and $i \in \mathbb{N}$, let be $\beta_i(m) = |\{\langle i, v \rangle \mid \langle i, v \rangle \text{ is in } m\}|$. A partial order on answer sets can be defined as follows. Given two answer sets $\mathcal{I}_1 = \langle I_1, \mu_1, \xi_1 \rangle$ and $\mathcal{I}_2 = \langle I_2, \mu_2, \xi_2 \rangle$ for an r-program P , with $\mu_1 \neq \mu_2$, let m_i be the multiset

$$m_i = \bigcup_{\gamma \in P, q \in \tau_R} \mu_i(q)(\gamma),$$

for $i \in \{1, 2\}$, and let j be the minimum natural number such that $\beta_j(m_1) \neq \beta_j(m_2)$. We put $\mathcal{I}_1 \prec_1 \mathcal{I}_2$ if and only if $\beta_j(m_1) > \beta_j(m_2)$.

Our first preference criterion \mathcal{PC}_1 states that \mathcal{I}_1 is preferred to \mathcal{I}_2 if it holds that $\mathcal{I}_1 \prec_1 \mathcal{I}_2$. The *preferred answer sets* with respect to \mathcal{PC}_1 are those answer sets that are \prec_1 -minimal. In a sense, the criterion \mathcal{PC}_1 has a “positional flavor”: the answer sets that selects the highest possible number of leftmost elements (in the p-lists) are preferred.

Our second criterion brings into play the magnitude of the preference grades. This can be done by considering the grades as weights and by optimizing with respect to the global weight expressed by the entire answer set. (Clearly, more complex assignments of weights are viable.) For any answer set $\mathcal{I} = \langle I, \mu, \xi \rangle$ let

$$\omega(\mathcal{I}) = \sum_{\gamma \in P, q \in \tau_R} \text{grade}(\mu_i(q)(\gamma)).$$

Given \mathcal{I}_1 and \mathcal{I}_2 as before, we put $\mathcal{I}_1 \prec_2 \mathcal{I}_2$ if and only if $\omega_j(m_1) < \omega_j(m_2)$. Consequently, our second preference criterion \mathcal{PC}_2 states that \mathcal{I}_1 is preferred to \mathcal{I}_2 if it holds that $\mathcal{I}_1 \prec_2 \mathcal{I}_2$. As before, the preferred answer sets, with respect to \mathcal{PC}_2 , are those that are \prec_2 -minimal.

3 Conditional preferences on resources.

Let us extend the syntax of r-rules by admitting p-lists (or amount-atoms) whose activation is subject to the truth of a conjunctive condition. A *conditional p-list* (cp-list, for short) is a writing of the form

$$(r \text{ \textbf{if}} L_1, \dots, L_m)$$

where r is a p-list $q_1:a_1 > \dots > q_h:a_h$, or simply an amount-atom, and L_1, \dots, L_m are program-literals. The intended meaning of a cp-list occurring in the body of a r-rule γ (the case of the head is analogous) is that whenever γ is fired the rule has to consume one of the resources occurring in r . If the firing occurs in correspondence of an answer set that satisfies the literals L_1, \dots, L_m , then the choice of which resource to consume is determined by the preference expressed by the p-list. Otherwise, if any of the L_i is not satisfied, a non-deterministic

choice is performed. (Hence the conjunction L_1, \dots, L_m need not to be satisfied in order to fire γ .) More precisely, the r-rule containing the cp-list becomes, if L_1, \dots, L_m does not hold, equivalent to h r-rules, each containing exactly one of the $q_j:a_j$'s, in place of the cp-list.

Such an extension of P-RASP can be treated by translating the rules involving cp-lists into regular r-rules. For instance, the rule $H \leftarrow B_1, \dots, B_k, (r \text{ if } L_1, \dots, L_m)$ is translated into this fragment of r-program:

$$\begin{array}{ll}
p \leftarrow \text{not } np. & np \leftarrow \text{not } p. \\
\leftarrow np, L_1, \dots, L_m. & \leftarrow p, \overline{L_i}. \quad \text{for } i \in \{1, \dots, m\} \\
H \leftarrow B_1, \dots, B_k, r, p. & \\
H \leftarrow B_1, \dots, B_k, q_j:a_j, np. & \text{for } j \in \{1, \dots, h\}
\end{array}$$

where p and np are fresh program atoms. Consequently, the semantics of cp-lists is given in terms of that of p-lists.

A similar approach can be adopted to handle multiple firing of rules, as well as to introduce cp-lists with different semantics. For example, one might imagine a cp-list that, differently from the previous case, when some L_i does not hold the firing does not require any consumption of resources in r .

4 Implementation of P-RASP

In this section, we briefly sketch our implementation of P-RASP. Once again, the translation consists of a refinement of the treatment designed for RASP and fully described in [13]. In particular, each (P-)RASP program is translated into an ASP program that constitutes an “intermediate form” which is then processed by an inference engine that performs reasoning on resources allocation. Such an engine consists in an ASP specification, defined for RASP and extended for P-RASP, which is independent from the specific program at hand. Since in the case of P-RASP the inference engine remains widely unchanged w.r.t. RASP, we hereafter describe only the extensions and the reader is referred to [13] for a detailed description of the parts not described here.

For the syntactic form of ASP programs, in what follows we adopt the language accepted by lparse [33, 37] (one of the grounders of commonly available ASP-solvers such as smodels [29]).

We proceed by giving, as an example, the translation of the ground rule:

$$[1-2]: \text{ cake:1} \leftarrow \text{ cook, potato_flour:2} > \text{ flour:3, egg:3, sugar:3.} \quad (3)$$

From such P-RASP rule a RASP rule is obtained:

$$[1-2]: \text{ cake:1} \leftarrow \text{ cook, pl}_1:1, \text{ egg:3, sugar:3} \quad (4)$$

where pl_1 is a new auxiliary resource symbol whose availability, in quantity equal to the maximum of firings admitted for rule (3), is stated by an r-fact:

$$\text{pl}_1:2.$$

Each r-fact such as this one, as well as the r-rule (4), is translated as explained in [13, Sect. 5].

The following fragment of ASP program establishes a dependence between the auxiliary resource `pl1`, each single firing of rule (4), and the amount-atoms in the p-list `potato.flour:2 > flour:3`:

```

auxres(n1,pl1).
1{use_pl(n1,flour,I,3), use_pl(n1,potato.flour,I,2)}1 :-
    use_n(n1,2,pl1,NumFirings), firings(n1,I),
    I<=NumFirings, num(NumFirings).

res_pl(n1,pl1,potato.flour,1).    res_pl(n1,pl1,flour,2).

```

where `n1` is a unique identifier assigned to rule (3). `NumFirings` denotes the number of firings of rule (3) and `I` is intended to range in $\{1, \dots, \text{NumFirings}\}$. Each program atom of the form `use_n(G,X,R,Q)` models consumption of an amount `Q` of resource `R` involved by firing the rule `G`. Predicate `use_pl` plays a similar role for auxiliary resources. Predicate `res_pl(G,Aux,R,Grade)` assigns the grade of preferences for the resources occurring in the p-list.

The inference engine developed for RASP in [13] is extended with the following ASP fragment that reflects consumption of auxiliary resources into consumption of real resources (a similar treatment is done for produced resources).

```

res(P1) :- auxres(G,P1), rule(G).
use_n(G,Idx,R,N) :- sum_use_pl(G,R,N), res_pl(G,P1,R,Grade),
    rule(G), val(N), cons(G,Idx,P1,1), N!=0.

```

Here, for each p-list, the literal `sum_use_pl` determines the contribution determined by repeated firing of the corresponding rule. This is evaluated by:

```

#weight use_pl(X,Y,W,Z) = Z.
sum_use_pl(G,R,N) :- N[use_pl(G,R,Iter,Q) : val(Q) : iter(Iter)]N,
    res_pl(G,P1,R,Grade), rule(G), val(N).
1{sum_use_pl(G,R,N) : val(N)}1 :- res_pl(G,P1,R,Grade), rule(G).

```

(Such a complicated use of weight literals to compute sums could be avoided in systems providing aggregate functions, e.g., `dlv`.) Finally, we are left with the fragment used to implement the preference criterion:

```

num_n_choice(Grade,N) :-
    N{use_pl(G,R,I,Q) : val(Q) : res_pl(G,P1,R,Grade) : val(I)}N,
    val(N), grade(Grade).
1{num_n_choice(Grade,N) : val(N)}1 :- grade(Grade).

#weight num_n_choice(X,Y) = Y.
maximize [num_n_choice(2,N) : val(N)].
maximize [num_n_choice(1,N) : val(N)].

```

For each **Grade**, the first two of the above rules evaluate the number times a choices of grade **Grade** is performed in the answer set. The **maximize** statements select the minimal answer sets w.r.t. the criterion \mathcal{PC}_1 seen in Section 2.

5 Related Work

Modeling resources in logic-based frameworks. Among the various logic-based approaches proposed to equip logic programming with some notion of resource and reason about them, [31] exploits (a variant of) linear logic [27] to define a *resource programming language (RPL)*. An operational semantics of RPL is given in terms of deduction rules: *Storage operators* and *resource transformation rules* model availability and transformation of resources, respectively. The deduction proceeds by applying these rules in a Prolog-like goal-directed fashion. A notion of step-by-step evolution of the state of the world is implicit in the rule application mechanism and RPL is shown expressive enough to model Petri nets.

To deal with resources, [24] proposes a concurrent Prolog inference engine for clauses enriched with pre/post-conditions on resource availability. Resources are represented by multisets of atoms and terms (non-unit amounts of a resource are rendered through multiple copies of the same atom/term).

Both in [24] and [31] the operational semantics of the proposed frameworks can be given in terms of (refinements of) the SLD-procedure and (default) negation is not handled. Moreover, both the programming languages of [31] and [24] offer little separation between the resource/amounts representation symbols and program symbols: resources and amounts are represented by program terms. The distinction is left to programmer's discipline. A valuable feature of RPL is the presence of a temporal dimension (which is implicit in the succession of rule applications). This makes RPL closer to planners than RASP. The introduction of time in RASP, would, in principle, move the system towards the action description languages. This could represent an interesting topic for future work and comparison.

A form of resource treatment is described in [35, 34] to model product configuration problems. The proposed framework is based on stable model semantics and encompasses default negation, disjunctive choices, and cardinality constraints. It differs from other Prolog-based approaches (e.g., [5]) because of the higher declarativeness offered by ASP.

Recently, [12] proposed the action description language *CARD*. Resources are rendered through multi-valued fluents and consumption/production of resources is implicitly modeled by the changes in fluents' values caused by actions' executions. The approach emphasizes the use of resources in planning problems and the semantics is given in terms of transition systems (in the spirit of [19]). *CARD* also supports some form of preferences on actions.

With respect to *CARD*, in RASP there is a neater distinction between what is a resource and what is not. Moreover, the arithmetic of amounts (as well as the constraints on balances) is implicitly handled by RASP's inference engine.

It seems that in *CARD* these aspects have to be encoded in the problem specification. On the other hand, since *CARD* is tailored to model action theories, time and state evolution are easily dealt with.

Preferences in logic-based frameworks. From the point of view of declarative knowledge representation and reasoning, many problems are more naturally represented by soft, i.e. flexible, rather than by hard descriptions. Hence, it is not surprising that much work has been done in introducing preferences in logic-based frameworks. Consequently, applications have been proposed in many context such as configuration [35, 34], information fusion [30, 25], planning [36], diagnosis [39], decision making [9].

Expressing and using preferences enables natural forms of commonsense reasoning, because the use of preferences plainly yields extremely expressive frameworks. See for instance [38], showing that a logic language equipped with a suitably defined hierarchy of preference relations can express complete problems for each level of the polynomial hierarchy.

Here we limit ourself to briefly mentioning some of the proposed approaches. A comprehensive treatment of preferences in non-monotonic reasoning can be found in [16].

As regards Prolog-based frameworks, we mention [23, 22, 21] and [14] as interesting attempts to introduce preferences in (constraint) logic programming.

Various forms of preferences have also been introduced in Answer Set Programming (see [16]). Most of the proposed approaches on preference reasoning in ASP are based on establishing priorities/preferences among rules. In [7], A-Prolog is enriched with ordered disjunction and preferences among rules are handled by means of a rule-naming mechanism. In the case of *ordered logic programs* [40], preferences are expressed through a partial order imposed on the set of rules. The order is used to implement defeating of less-preferred rules.

Other approaches express priorities among answer sets. Intuitively, this is done by declaring those atoms whose truth is “preferred” (typically, in these cases some forms of disjunction in the heads of rules is introduced). In *prioritized logic programs* [32], a set of *priorities* determines preferences on literals: From priorities, a preference relation on answer sets is drawn.

Another form of preferences on atoms is proposed in [11] by introducing *ordered disjunction* in the head of the rules. The atoms of the head are ordered according to some preference. Intuitively, the solver tries to satisfy a rule by satisfying the “best ranked” atom in its head. Considering a given answer set of a program, for each rule a *degree of satisfaction* is determined depending on which atom of the head is satisfied. Satisfaction degrees of all rules are then combined, according to some criterion, to rank the answer sets. Through similar ideas, a *Preference Description Language* is defined in [10] to formalize penalty-based preference handling in Answer Set Optimization. A comparison of these approaches can be found in [40].

Notice that in almost all the above mentioned cases, preferences are expressed globally, e.g., by providing an order relation that applies on all the rules (or

atoms) of the program. In P-RASP, as shown, preferences are imposed, by using p-lists, on some of the atoms of a rule. In this sense preferences in P-RASP have a local character, cf., Remark 2.

6 Conclusions

In this paper, we have presented a refinement of the RASP approach (that allows for production/consumption of resources in ASP) to include preferences on which resources to exploit/produce. Preferences are expressed by means of p-lists of amount-atoms, where leftmost ones are assumed to have higher priority in consumption/production. P-lists, that can be conditional, can in fact occur both in the body and in the head of r-rules. We have extended both syntax and semantics of RASP to account for this kind of preferences and we have introduced a concept of preferred answer set as a partial ordering among possible solution according to a certain strategy.

In future work, we intend to further generalize P-RASP by introducing preferences among sets of amount-atoms (i.e., one might e.g. prefer to use resources a and b instead of resources x, y and z), as well as (explicit) preferences on rules. We intend to apply P-RASP to practical problems, e.g., of configuration, so as to have the ground for defining and experimenting different strategies for choosing preferred answer sets.

References

- [1] H. Andréka, M. Ryan, and P.-Y. Schobbens. Operators and laws for combining preference relations. *J. Log. Comput.*, 12(1):13–53, 2002.
- [2] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See <http://asparagus.cs.uni-potsdam.de>.
- [3] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *J. of Logic Programming*, 19/20:9–72, 1994.
- [4] Web references for some ASP solvers. ASSAT: <http://assat.cs.ust.hk>; Ccalc: <http://www.cs.utexas.edu/users/tag/ccalc>; Clasp: <http://www.cs.uni-potsdam.de/clasp>; Cmodels: <http://www.cs.utexas.edu/users/tag/cmodels>; DLV: <http://www.dbai.tuwien.ac.at/proj/dlv>; Smodels: <http://www.tcs.hut.fi/Software/smodels>.
- [5] T. Axling and S. Haridi. A tool for developing interactive configuration applications. *J. Log. Program.*, 26(2):147–168, 1996.
- [6] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
- [7] M. Balduccini and V. S. Mellarkod. CR-Prolog₂ with ordered disjunction. In *Proc. of ASP'03*, 2003.
- [8] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
- [9] G. Brewka. Answer sets and qualitative decision making. *Synthese*, 146(1-2):171–187, 2003.
- [10] G. Brewka. Complex preferences for answer set optimization. In *Proc. of KR'04*, pages 213–223, 2004.

- [11] G. Brewka, I. Niemelä, and T. Syrjänen. Logic programs with ordered disjunction. *Comput. Intell.*, 20(2):335–357, 2004.
- [12] S. Chintabathina, M. Gelfond, and R. Watson. Defeasible laws, parallel actions, and reasoning about resources. In *Proc. of CommonSense'07*, 2007.
- [13] S. Costantini and A. Formisano. Modeling resource production and consumption in answer set programming. In *Proc. of ASP07*, 2007. Extended version in www.dipmat.unipg.it/~formis/papers/report2008_04.ps.gz.
- [14] B. Cui and T. Swift. Preference logic grammars: Fixed point semantics and application to data standardization. *Artif. Intell.*, 138(1-2):117–147, 2002.
- [15] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
- [16] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Comput. Intell.*, 20(12):308–334, 2004.
- [17] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In M. Gabbrielli and G. Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005, Proceedings*, volume 3668 of *LNCS*, pages 67–82. Springer, 2005.
- [18] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. of the 5th Intl. Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
- [19] M. Gelfond and V. Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
- [20] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [21] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference logic programming. In *Proc. of ICLP'95*, pages 731–745, 1995.
- [22] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference queries in deductive databases. *New Generation Comput.*, 19(1):57–86, 2000.
- [23] H.-F. Guo and B. Jayaraman. Mode-directed preferences for logic programs. In *Proc. of ACM-SAC'05*, pages 1414–1418, 2005.
- [24] J.-M. Jacquet and L. Monteiro. Towards resource handling in logic programming: The PPL framework and its semantics. *Comput. Lang.*, 22(2/3):51–77, 1996.
- [25] S. Konieczny and E. Grégoire. Logic-based approaches to information fusion. *Information Fusion*, 7(1):2–3, 2006.
- [26] V. Lifschitz. Answer set planning. In *Proc. of the 16th Intl. Conference on Logic Programming*, pages 23–37, 1999.
- [27] Linear logic programming references, 1995. Web site: <ftp://ftp.cis.upenn.edu/pub/papers/miller/ComputNet95/11survey.html>.
- [28] V. W. Marek and M. Truszczyński. *Stable logic programming - an alternative logic programming paradigm*, pages 375–398. Springer, 1999.
- [29] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proc. of the 8th Workshop on Non-Monotonic Reasoning*, 2000.
- [30] S. Pradhan and J. Minker. Using priorities to combine knowledge bases. *Int. J. Cooperative Inf. Syst.*, 5(2-3):333–364, 1996.
- [31] Y. U. Ryu. A logic-based modeling of resource consumption and production. *Decision Support Systems*, 22(3):243–257, 1998.
- [32] C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artif. Intell.*, 123(1-2):185–222, 2000.
- [33] P. Simons, I. Niemelä, and T. Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

- [34] T. Soinen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proc. of PADL'99*, pages 305–319, 1999.
- [35] T. Soinen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proc. of ASP'01*, 2001.
- [36] T. C. Son and E. Pontelli. Planning with preferences using logic programming. *TPLP*, 6(5):559–607, 2006.
- [37] T. Syrjänen. Lparse 1.0 user's manual, 2000. <http://www.tcs.hut.fi/Software/smodels>.
- [38] D. Van Nieuwenborgh, S. Heymans, and D. Vermeir. On programs with linearly ordered multiple preferences. In *Proc. of ICLP'04*, pages 180–194, 2004.
- [39] D. Van Nieuwenborgh and D. Vermeir. Ordered diagnosis. In *Proc. of LPAR'03*, pages 244–258, 2003.
- [40] D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. *TPLP*, 6(1-2):107–167, 2006.
- [41] WASP-WP5 Report: Model applications and proofs-of-concept, 2005. Working Group on Answer Set Programming (WASP): Web site: <http://www.kr.tuwien.ac.at/projects/WASP/report.html>.

A ASP in nutshell

Let us briefly recall the basics about Answer Set Programming [26, 28]. In this logical framework, a problem can be encoded —by using a function-free logic language— as a set of properties and constraints which describe the (candidate) solutions. More specifically, an *ASP-program* is a collection of *rules* of the form

$$H \leftarrow L_1, \dots, L_m, \text{not } L_1, \dots, \text{not } L_n$$

where H is an atom $m \geq 0$, $n \geq 0$ and each L is an atom. The symbol *not* stands for negation-as-failure. Various extensions to the basic paradigm exist, that we do not consider as they are not essential in the present context. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. A rule with empty head is a *constraint*. (The literals in the body of a constraint cannot be all true, otherwise they would imply falsity.)

Semantics of ASP is expressed in terms of *answer sets* (or equivalently *stable models*, [18]). Consider first the case of an ASP-program P which does not involve negation-as-failure (i.e., $n = 0$). In this case, a set of atoms X is said to be an answer set for P if it is the (unique) least model of P . Such a definition is extended to any program P containing negation-as-failure by considering the *reduct* P^X (of P) w.r.t. a set of atoms X . P^X is defined as the set of rules of the form $H \leftarrow L_1, \dots, L_m$ for all rules of P such that X does not contain any of the literals L_1, \dots, L_n . Clearly, P^X does not involve negation-as-failure. The set X is an answer set for P if it is an answer set for P^X .

Once a problem is described as an ASP-program P , its solutions (if any) are represented by the answer sets of P . Unlike other semantics, a logic program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set: $\{a \leftarrow \text{not } b. b \leftarrow \text{not } c. c \leftarrow \text{not } a.\}$. The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the

existence of answer sets. For a survey of this and other semantics of logic programs with negation, the reader may refer to [3].

Let us consider the program P consisting of the three rules

$$r \leftarrow p. \quad p \leftarrow \text{not } q. \quad q \leftarrow \text{not } p.$$

Such a program has two answer sets: $\{p, r\}$ and $\{q\}$. If we add the rule (actually, a constraint) $\leftarrow q$ to P , then we rule-out the second of these answer sets, because it violates the new constraint.

This simple example reveals the core of the usual approach followed in formalizing/solving a problem with ASP. Intuitively speaking, the programmer adopts a “generate-and-test” strategy: first (s)he provides a set of rules describing the collection of (all) potential solutions. Then, the addition of a group of constraints rules-out all those answer sets that are not desired real solutions.

Expressive power of ASP, as well as, its computational complexity have been deeply investigated. The interested reader can refer to [15], among others.

To find the solutions of an ASP-program, an ASP-solver is used. Several solvers have become available [4], each of them being characterized by its own prominent valuable features.

Let us give a simple example of ASP-program (see [8, 17], among others, for a presentation of ASP as a tool for declarative problem-solving). In doing this, we will recall the syntax and the main features of `lparse/smodels` (one of the available ASP-solvers which we exploited in this article, see [37] for more details). The problem we want to formalize in ASP is the well-known *n-queens* problem: “Given a $n \times n$ chess board, place n queens in such a way that no two of them attack each other”. The two clauses below state that a candidate solution is any disposition of the queens, provided that each column of the board contains one and only one queen. The fact that a queen is placed on the n^{th} column and on the m^{th} row is encoded by the atom `queen(n, m)`. (In the syntax of `lparse` ‘:-’ denotes implication \leftarrow . The value of the constant `n` occurring in the first clause is a parameter of the program supplied to `lparse` at run-time.)

`pos(1..n). 1 {queen(Col,Row) : pos(Col)} 1 :- pos(Row).`

The second rule is a particular form of constraint available in `lparse`’s language. The general form of such a kind of clauses is

$$k\{\langle \text{property_def} \rangle : \langle \text{range_def} \rangle\}m :- \langle \text{search_space} \rangle$$

where: the conditions $\langle \text{search_space} \rangle$ in the body define the set of objects of the domain to be checked; the atom $\langle \text{property_def} \rangle$ in the head defines the property to be checked; the conjunction $\langle \text{range_def} \rangle$ defines the possible values that the property may take on the objects defined in the body, namely by providing a conjunction of unary predicates each of them defining a range for one of the variables that occur in $\langle \text{property_def} \rangle$ but not in $\langle \text{search_space} \rangle$; k and m are the minimum and maximum number of distinct values that the specified property may take on the specified objects. (Notice that this form of constraint, available in `smodels`, actually is syntactic sugar, since it can be translated into “proper” ASP-clauses thanks to negation, cf. [33, 37].)

We now introduce two constraints, in order to rule out those placements where two queens control either the same row or the same diagonal of the board (here `pos(C;R1;R2)` is a shorthand notation for the three facts `pos(C)`, `pos(R1)`, `pos(R2)`):

```
:- queen(C,R1), queen(C,R2), pos(C;R1;R2), R1<R2.
:- queen(C1,R1), queen(C2,R2), pos(C1;C2;R1;R2),
   R1<R2, abs(C1-C2)==abs(R1-R2).
```

Here are two of the answer sets produced by smodels (in this case we put $n=8$):

```
Answer 1: queen(4,1) queen(6,2) queen(1,3) queen(5,4)
           queen(2,5) queen(8,6) queen(3,7) queen(7,8) ...
Answer 2: queen(4,1) queen(2,2) queen(8,3) queen(5,4)
           queen(7,5) queen(1,6) queen(3,7) queen(6,8) ...
```

Eliciting Multi-Dimensional Relational Patterns

Nicola Di Mauro, Teresa M.A. Basile, Grazia Bombini, Stefano Ferilli, and
Floriana Esposito

Università degli Studi di Bari, Dipartimento di Informatica, 70125 Bari, Italy
{ndm, basile, gbombini, ferilli, esposito}@di.uniba.it

Abstract. Here the issue of discovery of frequent multi-dimensional patterns from relational sequences is addressed. The great variety of applications of sequential pattern mining makes this problem one of the central topics in data mining. Nevertheless, sequential information may concern data on multiple dimensions and, hence, the mining of sequential patterns from multi-dimensional information results very important. This work takes into account the possibility to mine complex patterns, expressed in a first-order language, in which events may occur along different dimensions. Specifically, multi-dimensional patterns are defined as a set of atomic first-order formulae in which events are explicitly represented by a variable and the relations between events are represented by a set of dimensional predicates. A complete framework and an *Relational Learning* algorithm to tackle this problem are presented along with some experiments on artificial and real multi-dimensional sequences.

1 Introduction

The rapid growth of the amount of data stored in large databases has led to an increasing interest in the data mining research area and, in particular, towards methods to discover hidden structured patterns in large databases. The sequences are the simplest form of structured patterns and different methodologies have been proposed to face the problem of sequential pattern mining, firstly introduced by R. Agrawal and R. Srikant in [2], with the aim of capturing the existent maximal frequent sequences in a given database. The issue of discovering all frequent sequences of itemsets in a dataset is crucial when the data to be mined have some sequential nature like events in the case of temporal information. Furthermore, some real world domains such as user profiling, medicine, local weather forecast and bioinformatics show an inherent propension to be modelled by means of sequences of events/objects related to each other. This great variety of applications of sequential pattern mining makes this problem one of the central topics in data mining as showed by the research efforts produced in recent years [1, 23, 7, 18, 19].

However, some environments involve very complex components and features. Thus, the classical existing data mining approaches, that look for patterns in a single data table, have been extended to the multi-relational data mining approaches that look for patterns involving multiple tables (relations) from a relational database. This has led to the exploitation of a more powerful knowledge

representation formalism as first-order logic. Some works facing the problem of knowledge discovery from spatial and temporal data in multi-relational data mining research area are present in literature [16, 20, 5, 21, 14]. Nevertheless, there exists no contributions presenting a framework to manage the general case of multi relational data in which, for example, spatial and temporal information may co-exist.

On the other hand, it is worth to note that sequential information might concern data on multiple dimensions and, hence, the mining of sequential patterns from multi-dimensional information turns out to be very important. An attempt to propose a (two-dimensional) knowledge representation formalism to represent spatio-temporal information based on multi-dimensional modal logics is proposed in [3], while the first work presenting algorithms to mine multi-dimensional patterns has been presented in 2001 by Pinto et al. [19]. However, all the works in multi-dimensional data mining have been restricted to the propositional case, not involving a first-order representation formalism.

In this paper we provide an Inductive Logic Programming (ILP) [17] algorithm for discovering *first-order* (DATALOG) *maximal frequent patterns* in *multi-dimensional* relational sequences. Multi-dimensional patterns are defined as a set of atomic first-order formulae in which events are explicitly represented by a variable and the relations between events are represented by a set of dimensional predicates.

2 Mining Multi-Dimensional Patterns

We used Datalog [22] as representation language for the domain knowledge and patterns, that here is briefly reviewed. A *first-order alphabet* consists of variables, predicate symbols and function symbols (including constants). A *term* is defined as a constant symbol (a function symbol of arity 0, i.e. followed by a 0-tuple of terms) or a variable. An atom $p(t_1, \dots, t_n)$ (or atomic formula) is a predicate symbol p of arity n applied to n terms t_i . Both l and its negation \bar{l} are said to be *literals* (resp. positive and negative literal) whenever l is an atomic formula.

A *clause* is a formula of the form $\forall X_1 \forall X_2 \dots \forall X_n (L_1 \vee L_2 \vee \dots \vee \bar{L}_i \vee \bar{L}_{i+1} \vee \dots \vee \bar{L}_m)$ where each L_i is a literal and X_1, X_2, \dots, X_n are all the variables occurring in $L_1 \vee L_2 \vee \dots \bar{L}_i \vee \dots \bar{L}_m$. Most commonly the same clause is written as $L_1, L_2, \dots \leftarrow L_i, L_{i+1}, \dots, L_m$. Clauses, literals and terms are said to be *ground* whenever they do not contain variables. A *Horn clause* is a clause which contains at most one positive literal. A *Datalog clause* is a clause with no function symbols of non-zero arity; only variables and constants can be used as predicate arguments.

A *substitution* θ is defined as a set of bindings $\{X_1 \leftarrow a_1, \dots, X_n \leftarrow a_n\}$ where $X_i, 1 \leq i \leq n$ is a variable and $a_i, 1 \leq i \leq n$ is a term. A substitution θ is applicable to an expression e , obtaining the expression $e\theta$, by replacing all variables X_i with their corresponding terms a_i .

Definition 1 (1-dimensional relational sequence). A 1-dimensional relational sequence may be defined as an ordered list of Datalog atoms separated by the operator $<$, $l_1 < l_2 < \dots < l_n$.

However, in order to make the proposed framework more general, the concept of *fluents* introduced by J. McCarthy in [15] should be considered: “After having defined a *situation*, s_t , as the complete state of the universe at an instant of time t , then a fluent is defined as a function whose domain is the space of situations. In particular, a *propositional fluent* ranges in (true,false). For example, $\text{raining}(x, s_t)$ is true if and only if it is raining at the place x in the situation s_t .”

If we consider a sequence as an ordered succession of events for each dimension, a fluent may be used to indicate that an atom is true for a given event. In particular, in our description language we can distinguish two kinds of Datalog atoms: *dimensional* and *non-dimensional* atoms. Specifically:

- non-dimensional atoms, that may be divided into
 - *fluent atoms*: explicitly referring to a given event (i.e., in which one of its argument denotes an event);
 - *non-fluent atoms*: denoting relations between objects (with arity greater than 1), or characterizing an object (with arity 1) involved in the sequence;
- dimensional atoms: referring to dimensional relations between events involved in the sequence.

The choice to add the event as an argument of the predicates is necessary for the general case of n -dimensional sequences with $n > 1$. In this case, indeed, the operator $<$ is not sufficient to express multi-dimensional relations and we must use its general version $<_i, 1 \leq i \leq n$. Specifically, $(e_1 <_i e_2)$ denotes that the event e_1 gives rise to the event e_2 in the dimension i . Hence, in our framework a multi-dimensional data is supposed to be a set of events, and to each dimension corresponds a sequence of events.

Definition 2 (Multi-dimensional relational sequence). A multi-dimensional relational sequence is a set of Datalog atoms, involving k events and concerning n dimensions, in which there are non-dimensional atoms (*fluents* and *non-fluents*) and each event may be related to another event by means of the $<_i$ operators, $1 \leq i \leq n$.

In order to represent multi-dimensional relational patterns, some dimensional operators must be introduced. The following symbols for describing general event relationships along many dimensions have been adopted. In particular, given a set \mathcal{D} of dimensions, in the following are reported the multi-dimensional operators:

- $<_i$: *next step on dimension i* , $\forall i \in \mathcal{D}$. This operator indicates the direct successor on the dimension i . For instance, $(x <_{time} y)$ denotes that the event y is the direct successor of the event x on the dimension *time*. `next_i/2` is the corresponding Datalog predicate used to denote the successor operator;

- \triangleleft_i : *after some steps on dimension $i, \forall i \in \mathcal{D}$.* This operator encodes the transitive closure of $<_i$. For example, $(y \triangleleft_{\text{spatial}x} z)$ states that the event z occurs somewhere after the event y on the dimension *spatialx*. `follow_i/2` is the corresponding Datalog representation;
- \bigcirc_i^n : *exactly after n steps on dimension $i, \forall i \in \mathcal{D}$.* In particular it calculates the n -th direct successor. For instance, $(x \bigcirc_{\text{spatial}z}^n w)$ states that the event w is the n -th direct successor of the event x on the dimension *spatialz*. The `followat_1/3` Datalog predicate is used to represent such a situation.

Note that, the dimensional characteristics in the sequences will be described by using the $<_i$ operator, while the two dimensional operators \triangleleft_i and \bigcirc_i^n , will be used, in combination with $<_i$ operator, to represent the frequent discovered patterns.

Definition 3 (Subsequence [9]). *Given a sequence $\sigma = (e_1 e_2 \dots e_m)$ of m elements, a sequence $\sigma' = (e'_1 e'_2 \dots e'_k)$ of length k is a subsequence (or pattern) of the sequence σ if*

1. $1 \leq k \leq m$
2. $\forall i, 1 \leq i \leq k, \exists j, 1 \leq j \leq m : e'_i = e_j$
3. $\forall i, j, 1 \leq i < j \leq k, \exists h, l, 1 \leq h < l \leq m : e'_i = e_h$ and $e'_j = e_l$.

The frequency of a subsequence in a sequence is the number of different mappings from elements of σ' into the elements of σ such that the previous conditions hold.

Note that this is a general definition of subsequence, in our case the *gaps* represented by the third condition are modelled by the \triangleleft_i and \bigcirc_i^n operators as reported in the following definition.

Definition 4 (Multi-dimensional relational pattern). *A multi-dimensional relational pattern is a set of Datalog atoms, involving k events and regarding n dimensions, in which there are non-dimensional atoms and each event may be related to another event by means of the operators $<_i$, \triangleleft_i and \bigcirc_i^n operators, $1 \leq i \leq n$.*

We are interested in finding maximal frequent patterns with a high frequency in long sequences. A pattern σ' of a sequence σ is *maximal* if there is no pattern σ'' of σ more frequent than σ' and such that σ' is a subsequence of σ'' . In order to calculate the frequency of a pattern over a sequence it is important to define the concept of sequence subsumption.

If we indicate the operator $<_i$ with the Datalog predicate `next_i(X,Y)`, the Datalog definition of the operators \bigcirc_i^k and \triangleleft_i can be formulated as follows:

```
followat_i(1,X,Y) ← next_i(X,Y), !.
followat_i(K,X,Y) ← next_i(X,Z), K1 is K - 1, followat_i(K1,Z,Y).
```

```
follow_i(X,Y) ← next_i(X,Y).
```

`follow_i(X,Y) ← next_i(X,Z), follow_i(Z,Y).`

These definitions are added to the background knowledge \mathcal{B} and used to prove the dimensional operators appearing in the patterns using the following definition of subsumption. Given S a multi-dimensional relational sequence, in the following we will indicate by Σ the set of Datalog clauses $\mathcal{B} \cup U$, where U is the set of ground atoms in S .

Definition 5 (Subsumption). *Given P a multi-dimensional relational pattern and S a multi-dimensional relational sequence, let $\Sigma = \mathcal{B} \cup U$. The pattern P subsumes the sequence S , written as $P \subseteq S$, iff there exists an SLD_{OI} -deduction of P from Σ .*

An SLD_{OI} -deduction is an SLD-deduction under Object Identity. In the Object Identity framework, within a clause, terms that are denoted with different symbols must be distinct, i.e. they must represent different objects of the domain.

3 The algorithm

After having defined the formalism for representing sequences and patterns, here we describe the algorithm for frequent multi-dimensional relational pattern mining based on the same idea of the generic level-wise search method, known in data mining from the APRIORI algorithm [1]. The level-wise algorithm makes a breadth-first search in the lattice of patterns ordered by a specialization relation \preceq . The search starts from the most general patterns, and at each level of the lattice the algorithm generates candidates by using the lattice structure and then evaluates the frequencies of the candidates. In the generation phase, some patterns are taken out using the monotonicity of pattern frequency (if a pattern is not frequent then none of its specializations is frequent).

The mining method is outlined in Algorithm 1. The generation of the frequent patterns is based on a top-down approach. The algorithm starts with the most general patterns. These initial patterns are all of length 1 and are generated by adding to the empty pattern a non-dimensional atom. Successively, at each step it tries to specialize all the potential frequent patterns, discarding the non-frequent patterns and storing the ones whose length is equal to the user specified input parameter *maxsize*. Furthermore, for each new refined pattern, semantically equivalent patterns are detected, by using the θ_{OI} -subsumption relation, and discarded. Note that the length of a pattern is defined as the number of non-dimensional atoms. In the specialization phase, the specialization operator under θ_{OI} -subsumption is used. Basically, the operator adds atoms to the pattern.

The algorithm uses a background knowledge \mathcal{B} (a set of Datalog clauses) containing the sequence and a set of constraints that must be satisfied by the generated patterns. In particular \mathcal{B} contains:

- *maxsize(M)*: maximal pattern length (i.e., the maximum number of non-dimensional predicates that may appear in the pattern);

Algorithm 1 MDLS

Require: $\Sigma = \mathcal{B} \cup U$, where \mathcal{B} is the background knowledge and U is the set of ground atoms in the sequence S .

Ensure: P_{max} : the set of maximal frequent patterns

```
1:  $P \leftarrow \{ \text{initial patterns} \}$ 
2:  $P_{max} \leftarrow \emptyset$ 
3: while  $P \neq \emptyset$  do
4:    $P_s \leftarrow \emptyset$ 
5:   for all  $p \in P$  do
6:     /* generation step */
7:      $P_s \leftarrow P_s \cup \{ \text{all the specializations of } p \text{ that satisfy all the constraints} \}$ 
8:      $P \leftarrow \emptyset$ 
9:     for all  $p \in P_s$  do
10:      /* evaluation step */
11:      if  $\text{freq}(p) \geq \text{minfreq}$  then
12:        if  $\text{length}(p) = \text{maxsize}$  then
13:           $P_{max} \leftarrow P_{max} \cup \{p\}$ 
14:        else
15:           $P \leftarrow P \cup \{p\}$ 
```

- $\text{minfreq}(m)$: this constraint indicates that the frequency of the patterns must be larger than m ;
- $\text{dimension}(\text{next}_i)$: this kind of atom indicates that the sequence contains events on the dimension i . One can have more than one of such atoms, each of which denoting a different dimension. In particular, the number of these atoms represents the number of the dimensions.
- $\text{type}(p)$: denotes the type of the predicate's arguments p ;
- $\text{mode}(p)$: denotes the input output mode of the predicate's arguments p ;
- $\text{negconstraint}([p_1, p_2, \dots, p_n])$: specifies a constraint that the patterns must not fulfill, i.e. if the clause (p_1, p_2, \dots, p_n) subsumes the pattern then it must be discarded. For instance, $\text{negconstraint}([p(X,Y), q(Y)])$ discards all the patterns subsumed by the clause $(p(X,Y), q(Y))$;
- $\text{posconstraint}([p_1, p_2, \dots, p_n])$: specifies a constraint that the patterns must fulfill. It discards all the patterns that are not subsumed by the clause (p_1, p_2, \dots, p_n) ;
- $\text{atmostone}([p_1, p_2, \dots, p_n])$: this constraint discards all the patterns that make true more than one predicate among p_1, p_2, \dots, p_n . For instance, $\text{atmostone}([\text{red}(X), \text{blue}(X), \text{green}(X)])$ indicates that each constant in the pattern can assume at most one of red, blue or green value;
- $\text{key}([p_1, p_2, \dots, p_n])$: it is optional and specifies that each pattern must have one of the non-dimensional predicates p_1, p_2, \dots, p_n as a starting literal.

The use of the `dimension(next_i)` literals, that specify the number of dimensions the sequence is based on, allows to the corresponding definitions of the predicates `followat_i/3` and `follow_i/2` to be automatically generated and added to the background knowledge \mathcal{B} .

Classical mode and type declarations are used to specify a language bias indicating which predicates can be used in the patterns and to formulate constraints on the binding of variables. The solution space is further pruned by using some positive and negative constraints specified by the `negconstraint` and `posconstraint` literals. The last pruning choice is defined by the `atmostone` literals. This last constraint is able to describe that some predicates are of the same type.

Since each pattern a) must start with a non-dimensional predicate, or with a predefined key, and b) its frequency must be less than the sequence length, the frequency of a pattern can be defined as follows.

Definition 6 (Frequency). *Given a multi-dimensional relational pattern $P = (p_1, p_2, \dots, p_n)$ and S a multi-dimensional relational sequence, the frequency of pattern P is equal to the number of different ground literals used in all the possible SLD_{OI} -deductions of P from $\Sigma = \mathcal{B} \cup U$ that make true the literal p_1 .*

The refinement of pattern is obtained by using a refinement operator ρ that maps each pattern to a set of specializations of the pattern, i.e. $\rho(p) \subset \{p' | p \preceq p'\}$ where $p \preceq p'$ means that p is more general of p' or that p subsumes p' . In particular, given the set \mathcal{D} of dimensions, the set \mathcal{F} of fluent atoms, the set \mathcal{P} of non-fluent atoms, the refinement operator for specializing the patterns is defined as follows:

adding a non-dimensional atom

- the pattern S is specialized by adding a non-dimensional atom $F \in \mathcal{F}$ (a fluent) referring to an event already introduced in S ;
- the pattern S is specialized by adding a non-dimensional atom $P \in \mathcal{P}$;

adding a dimensional atom

- the pattern S is specialized by adding the dimensional atom $(x <_i y)$ $i \in \mathcal{D}$, relating the events x and y , iff \exists a fluent $F \in \mathcal{F}$ in S which event argument is x and there not exist the atoms $(x \triangleleft_i y)$ and $(x \bigcirc_i^n y)$ in S ;
- the pattern S is specialized by adding the dimensional atom $(x \triangleleft_i y)$ $i \in \mathcal{D}$, relating the events x and y , iff \exists a fluent $F \in \mathcal{F}$ in S which event argument is x and there not exist the atoms $(x <_i y)$ and $(x \bigcirc_i^n y)$ in S ;
- the pattern S is specialized by adding the dimensional atom $(x \bigcirc_i^n y)$ $i \in \mathcal{D}$, relating the events x and y , iff \exists a fluent $F \in \mathcal{F}$ in S which event argument is x and there not exist the atoms $(x <_i y)$ and $(x \triangleleft_i y)$ in S .

The dimensional atoms are added iff there exists a fluent atom referring to its starting event. This is to avoid useless chains of dimensional predicates like this $\mathbf{p}(e_1, \mathbf{a}) (e_1 <_i e_2) (e_2 <_i e_3) (e_3 <_i e_4)$, that is naturally subsumed by $\mathbf{p}(e_1, \mathbf{a}) (e_1 \bigcirc_i^3 e_4)$. We recall that the length of a pattern P is equal to the number of non-dimensional atoms in P .

4 Experiments

MDSL has been implemented in Yap Prolog and evaluated by making some experiments on an artificial dataset and on trace files collected from different

users of Unix csh [6, 9]. The analysis of the use of Unix command shell represents one of the classic applications in the domain of adaptive user interfaces and user modeling. Greenberg [6] collected logs from 168 users of the unix csh, divided into 4 target groups: 55 novice programmers, 36 experienced programmers, 52 computer scientists and 25 non-programmers. Table 1 reports statistics of finding frequent patterns for 3 users logs from the Greenberg dataset.

Each Greenberg's log file corresponding to a user is divided into login sessions denoted by a starting and an ending time record. Each command entered in each session has been annotated with the current working directory, alias substitution, history use and error status. Furthermore, each command name may be followed by some options and some parameters. For instance the command `ls -a *.c` has name `ls`, option `-a` and parameter `*.c`.

As pointed out in [9], this problem is a relational problem, since commands are interrelated by their execution order (or time), and each command can be eventually related to one or more parameters. A shell log may be viewed as a 2-dimensional sequence, since each command is followed by another command (the first dimension) and each command line is composed by an ordered sequence of tokens (i.e., command name, options and parameters). Each shell log has been represented as a set of logical ground atoms as follows.

`command(e)` is the predicate used to indicate that `e` is a command. The command name has been used as a predicate symbol applied to `e`;
`parameter(e,p)` has been used to indicate that `e` has the parameter `p`. The parameter name has been used as a predicate symbol applied to `p`;
`current_directory(c,d)` indicates that `d` is the current directory of the command `c`;
`next_c(c1,c2)` ($\langle c \rangle$) indicates that the command `c2` is the direct command successor of `c1`;
`next_p(p1,p2)` ($\langle p \rangle$) indicates that the parameter `p2` is the direct parameter successor of `p1`.

For instance the following shell log

```
cp paper.tex newspaper.tex
latex newspaper
xdvi newspaper
```

should be translated as

```
command(c1), '$cp'(c1),
  next_p(c1,c1p1), parameter(c1p1,'paper.tex'),
  next_p(c1p1,c1p2), parameter(c1p2,'newpaper.tex'),
next_c(c1,c2), '$latex'(c2),
  next_p(c2,c2p1), parameter(c2p1,'newpaper'),
next_c(c2,c3), '$xdvi'(c3),
  next_p(c3,c3p1), parameter(c3p1,'newpaper')
```

In this way it is possible to describe patterns such as

```

command(L), '$latex'(L),
  next_p(L,LP), parameter(LP,P),
next_c(L,X), '$xdvi'(X),
  next_p(X,XP), parameter(XP,P)

```

Table 1 reports statistics on MDLS performance on the Greenberg dataset. The first four columns denote, respectively, the user name, the number of literals of the sequence, the number of sessions for each user log file and the total number of commands in the log file. For each user some experiments has been made. The kind of experiment carried out is denoted in the fifth column that reports the operators used in the experiment. For instance \langle_C and \langle_P indicate that only these two operators have been used in the experiment. We see that, as the number of commands and dimensional operators grows, the execution time increases. Note that each session represents a separate sequence and a log file is a collection of sequences. There is no correlation between two session in a log file.

Table 1. MDLS performances (time in secs.). |S|: n. of literals in the sequence; |Ses|: n. of sessions for user log file; |C|: total number of commands in the log file; Op: dimensional operators used; L: max length of the patterns; F: min freq of the patterns; |MP|: n. of found maximal patterns; Sp: required specializations.

User	S	Ses	C	Op	L	F	Time	MP	Sp
n9	2654	73	357	\langle_C	5	5	1.17	45	3597
				$\langle_C \langle_P$	5	5	2.68	45	9513
				$\langle_C \bigcirc_C^n$	5	5	2.58	91	8495
				$\langle_C \triangleleft_C \bigcirc_C^n$	5	5	28.94	149	29081
				$\langle_{C,P} \triangleleft_{C,P} \bigcirc_{C,P}^n$	5	5	99.06	218	76299
n17	5366	61	848	\langle_C	5	10	2.36	38	4512
				$\langle_C \langle_P$	5	10	3.58	18	6434
				$\langle_C \bigcirc_C^n$	5	10	7.95	47	10808
				$\langle_C \triangleleft_C \bigcirc_C^n$	5	10	37.06	39	19198
				$\langle_{C,P} \triangleleft_{C,P} \bigcirc_{C,P}^n$	5	10	144.38	25	25142
n7	12355	80	1231	\langle_C	5	15	6.10	64	7716
				$\langle_C \langle_P$	5	15	19.33	77	24046
				$\langle_C \bigcirc_C^n$	5	15	16.30	139	20744
				$\langle_C \triangleleft_C \bigcirc_C^n$	5	15	138.06	162	51363
				\langle_C	5	80	1.78	9	953
				$\langle_C \langle_P$	5	80	4.06	11	2493
				$\langle_C \bigcirc_C^n$	5	80	4.14	16	2479
				$\langle_C \triangleleft_C \bigcirc_C^n$	5	80	42.55	29	6745
				$\langle_{C,P} \triangleleft_{C,P} \bigcirc_{C,P}^n$	5	80	164.99	49	17505

5 Related Work

As already pointed out, the problem of sequential pattern mining is a central one in a lot of data mining applications and many efforts have been done in order to propose purposely designed methods to face it. Most of the works have been restricted to propositional patterns, that is, patterns not involving first order predicates. One of the early domains that highlights the need to describe with structural information the sequences was the bioinformatic. Thus, the need to represent many real world domains with structured data sequences became more unceasing, and consequently many efforts have been done to extend existing or propose new methods to manage sequential patterns in which first order predicates are involved. On the other hand, for a fair description of some application domains, the sequences must involve not only relational objects but also the evolution of each object in more than one dimension. Unfortunately, to our knowledge, there are no methods able to manage sequences whose description involves both relations and more than one dimensions. In the following a brief survey of the techniques proposed to deal with relational or multi-dimensional sequences is presented along with the modelled application domain.

In [9] is presented a work, in the domain of user modelling, that helps shell users by creating scripts (a sequence of commands) from shell logs, that automate frequent performed tasks. The authors see this task as a relational learning problem, indeed commands may be interrelated by their execution order, and each command is possibly related to one or more parameters, giving out a representation of a shell log as a set of logical ground atoms. After having transformed shell logs in a relational representation, they applied the Warmr [5] system, an upgrade of the propositional Apriori algorithm that can detect first order logic association rules, for generating scripts. They used a specific predicate to specify that two commands are considered next to each other in a sequence.

Warmr [5] is based on the level-wise search of conventional association rule learning systems of the Apriori-family [1]. It extends these systems by looking for frequent patterns that may be expressed as conjunction of first-order literals. In Warmr a pattern is defined as a conjunction of first order literals. It performs a top-down level-wise search, starting with the key and refining patterns by adding literals to them. Infrequent patterns (i.e. patterns whose frequency is below a predefined threshold) are pruned as are their refinements. With Warmr it is possible to generate patterns that are syntactically different but semantically equivalent. This is due to the redundant conditions that may be added to a pattern or to the fact that the same pattern may be expressed in different ways.

As already described in [4], this problem may be avoided by using the Warmr's configurable language bias or by its constraint specification language. However, this solution does not solve the problem at all. Indeed, the constraints that may be defined in Warmr, by using its constraint specification language, are only syntax based, and they are not sufficient to handle semantic dependencies. For this and other limitations already described in [9, 8], in some cases the Warmr system is not able to calculate frequent subsequences and it is difficult to correctly represent the specific sequence mining task.

In [13] are presented a logic language, SeqLog, for mining sequences of logical atoms, and the inductive mining system MineSeqLog, that combines principles of the level-wise search algorithm with the version space in order to find all patterns that satisfy a constraint by using an optimal refinement operator for SeqLog. SeqLog is a logic representational framework that adopts two operators to represent the sequences: one to indicate that an atom is the direct successor of another and the other to say that an atom occurs somewhere after another. Furthermore, based on this language, the notion of subsumption, entailment and a fix point semantic are given. However, with SeqLog one can represent unidimensional sequences only.

In [12] it is proposed an extension of classical Fisher kernels, working on sequences over flat alphabets, in order to make them able to model logical sequences, i.e., sequences over an alphabet of logical atoms. Fisher kernels were developed to combine generative models with kernel methods, and have shown promising results for the combinations of support vector machines with (logical) hidden Markov models and Bayesian networks. Successively, in [10] the same authors proposed an algorithm for selecting logical hidden Markov models from data. Hidden Markov models are one of the most popular methods for analyzing sequential data, but they can be exploited to handle sequence of flat/unstructured symbols. The proposed logical extension [11] overcomes such weakness by handling sequences of structured symbols by means of a probabilistic ILP framework.

The work above reported extend/propose techniques to mine sequences involving relational objects. However, these methods, both logical and propositional, do not mention the possibility to manage patterns in which more than one dimension is taken into account. On the other hand, it is not wrong to affirm that for most of the applications of real world domain generally the pattern sequences deal with different events each of which should be associated with different dimensions.

6 Conclusions

The issue of discovering sequential patterns from sequence data have drawn a lot of research efforts both in single data table and in multiple data table, known as multi-relational data. Although much work has been done in the area, no previous research revealed ways to find sequential patterns from multidimensional sequence data and in particular sequential patterns from multi-dimensional sequence expressed in first-order logic. Indeed, some works face the problem of knowledge discovery from spatial and temporal data in the multi-relational data mining research area but there exists no contributions to manage the general case of multi-dimensional data in which, for example, spatial and temporal information may co-exist. Other works on multi-dimensional data mining, in some cases thinking to the concept of multi-dimension of a sequence as the presence of multiple attributes in data descriptions, have been restricted to the propositional case, not involving a first-order representation formalism. Finally, other works

propose a (two-dimensional) knowledge representation formalism to represent spatio-temporal information based on multi-dimensional modal logics.

In this paper we proposed a logical framework for mining multi-dimensional patterns in which many dimensions can be specified. What we can obtain are maximal frequent multi-dimensional patterns described in a first order language. One of the most important characteristic of using logical framework for sequences is that we can incorporate additional information by using a background knowledge, and that any relation between atoms can be expressed or learned. The result is a dedicated system in which are incorporated specific language bias for multi-dimensional data in order to rise a faster execution and a smaller search space.

Acknowledgements

The authors would like to thank Jan Ramon for giving useful suggestions on setting the system Warmr and Saul Greenberg for providing the test Unix log data.

This work is partially funded on the Apulian Regional Project DDTA “Distretto Digitale a Supporto della Filiera Produttiva del Tessile-Abbigliamento”.

References

1. R. Agrawal, H. Manilla, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.
2. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Int. Conf. on Data Engineering (ICDE95)*, pages 3–14, 1995.
3. Brandon Bennett, Anthony G. Cohn, Frank Wolter, and Michael Zakharyashev. Multi-dimensional modal logic as a framework for spatio-temporal reasoning. *Applied Intelligence*, 17(3):239–251, 2002. Submitted 2000.
4. H. Blockeel, J. Fürnkranz, A. Prskawetz, and F. Billari. Detectin temporal change in event sequences: an application to demographic data. In *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 29–41. Springer, 2001.
5. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
6. S. Greenberg. Using unix: collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Alberta, 1988.
7. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, 2000.
8. N. Jacobs. *Relational Sequence Learning and User Modelling*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, October 2004.
9. N. Jacobs and H. Blockeel. From shell logs to shell scripts. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157, pages 80–90. Springer, 2001.

10. K. Kersting, L. De Raedt, , and T. Raiko. Logical hidden markov models. *Journal of Artificial Intelligence Research - JAIR*, 25:425–456, April 2006. Submitted 2005.
11. K. Kersting and T. Raiko. ‘Say EM’ for selecting probabilistic models for logical sequences. In F. Bacchus and T. Jaakkola, editors, *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI05)*, pages 300–307. AUAI Press, 2005.
12. Kristian Kersting and Thomas Gärtner. Fisher kernels for logical sequences. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Machine Learning: ECML 2004, Proceedings of the 15th European Conference on Machine Learning*, volume 3201 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 2004.
13. S.D. Lee and L. De Raedt. Constraint based mining of first order sequences in SeqLog. In R. Meo, P.L. Lanzi, and M. Klemettinen, editors, *Database Support for Data Mining Applications*, volume 2682 of *LNCS*, pages 155–176. Springer, 2004.
14. D. Malerba and F.A. Lisi. Discovering associations between spatial objects: An ilp application. In *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *LNCS*, pages 156–166. Springer, 2001.
15. J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.
16. S. Moyle and S. Muggleton. Learning programs in the event calculus. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 205–212. Springer, 1997.
17. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
18. P.J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. In *Proceedings of the 11th ACM International Conference on Information and Knowledge Management*, pages 18–25, 2002.
19. H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal. Multi-dimensional sequential pattern mining. In *CIKM ’01: Proceedings of the tenth international conference on Information and knowledge management*, pages 81–88, New York, NY, USA, 2001. ACM Press.
20. L. Popelínsky. Knowledge discovery in spatial data by means of ILP. In *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 185–193. Springer, 1998.
21. J. Rodríguez, C. Alonso, and H. Böstrom. Learning first order logic time series classifiers. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Workshop on Inductive Logic Programming*, pages 260–275. Springer, 2000.
22. J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.
23. M.J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal: Special issue on Unsupervised Learning*, 42(1/2):31–60, 2001.

Compiling Minimum and Maximum Aggregates into Standard ASP^{*}

Mario Alviano, Wolfgang Faber, and Nicola Leone

Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
{alviano, faber, leone}@mat.unical.it

Abstract. The introduction of aggregate functions in answer set programming (ASP) allows for representing many problems in a succinct way. The possibility to encode aggregate functions with the constructs of standard ASP results from previous studies about the complexity of answer set programming with aggregates (ASP^A). In this paper, possibilities for encoding minimum and maximum aggregates are analyzed. The straightforward encoding, which has also been proposed in the literature, is shown to be inappropriate in the presence of recursive aggregates. As a remedy, a new encoding is proposed, the adequacy of which is formally proved.

1 Introduction

Answer set programming (ASP, disjunctive logic programming under the answer set semantics [1, 2]) is a powerful language for knowledge representation and common-sense reasoning. While the standard formalism does not include aggregations, many extensions in this direction have been proposed [3–12]. In this work we will focus on the approach of [10]. Previous work [13] about the complexity of answer set programming with aggregates (ASP^A) showed that the complexity of model checking and cautious reasoning does not increase with respect to standard ASP. As a consequence, it must be possible to find an efficient transformation from programs with aggregates to programs without aggregates, which will be referred to as compilation in the sequel.

In this paper, transformations of this kind for minimum and maximum aggregates are examined. In Datalog folklore, there is a standard way of encoding the determination of a minimum or maximum of a relation, which is briefly recalled and discussed. A compilation method for minimum and maximum aggregates has been proposed, which is based on this folklore encoding [14, 15]. However, it can be shown that this compilation does not work for recursive aggregates, that is, when the minimum or maximum to be determined by the aggregate depends on a relation, which in turn depends (at least partially) on the aggregate in question. In particular, the compiled program in general admits fewer answer sets than the original program.

As a remedy, a new compilation method is described, which avoids the complications of the folklore encoding. For this compilation strategy, a formal proof of equivalence of the compiled program to the original ASP^A program is provided.

* Supported by M.I.U.R. within projects “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva” and “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione.”

Summarizing, this work provides the following main contributions.

- The inadequacy of the compilation method of [14, 15] for minimum and maximum aggregates is shown in presence of recursive aggregates.
- A new transformation, which avoids the problems of the folklore encoding, is provided.
- The equivalence of the compiled program with the original one is formally proved.

The remainder of this paper is structured as follows. In Section 2, syntax and semantics of answer set programming with minimum and maximum aggregates is introduced. In Section 3 the folklore encoding of minima and maxima, and the compilation method of [14, 15] is recalled and analyzed. Then, in Section 4, an alternative encoding is proposed and formally proved to be correct. Finally, in Section 5 the relation of these results with existing work is discussed, and in Section 6 the conclusions are drawn.

2 Logic Programs with Min and Max Aggregates

Syntax and semantics of logic programs with min and max aggregates are concisely described in this section.

2.1 Syntax

The reader is assumed to be familiar with standard Logic Programs (as in ASP) [16, 2], whose constructs are referred to as *standard atoms* (composed of a predicate, constants, and variables), *standard literals*, *standard rules*, and *standard programs*. Unless otherwise specified, words starting by a lower-case letter and numbers denote predicate and constant symbols, while words starting by an upper-case letter denote variable symbols. In addition, constants and terms overlined denote comma-separated lists (possibly empty) of constants and terms, respectively.

Set Terms. An ASP^A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{C : p(\overline{X}, \overline{Y}, C)\}$, where \overline{X} are bound terms, \overline{Y} are free terms, and C is a free variable called *aggregation term*. Intuitively, a symbolic set represents the set of C -values such that $p(\overline{X}, \overline{Y}, C)$ is true, i.e. $\{C \mid \exists \overline{Y} s.t. p(\overline{X}, \overline{Y}, C) \text{ is true}\}$, where the variables in \overline{X} are assumed to be fixed. A *ground set* is a set of pairs of the form $\langle c : a \rangle$, where c is a constant and a is a ground (variable free) standard atom.

Aggregate Functions. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping sets (or multisets) of constants to a constant. In this work we analyze the property of minimum and maximum aggregate functions, so f is $\#_{\min}$ for the minimum term, or $\#_{\max}$ for the maximum term. Both of these functions are undefined on the empty set.

Aggregate Literals. An *aggregate atom* is of the form $f(S) \prec k$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a predefined comparison operator, and k is a constant referred to as guard.

Example 1. The following are two aggregate atoms, where the second contains a ground set and could be a ground instance of the first.

$$\#\max\{C : p(Y, C)\} > 1 \quad \#\max\{\langle 1 : p(a, 1) \rangle, \langle 1 : p(b, 1) \rangle, \langle 2 : p(a, 2) \rangle\} > 1$$

An *atom* is either a standard atom or an aggregate atom. A *literal* ℓ is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, then ℓ is an *aggregate literal*.

ASP^A Programs. An ASP^A rule r is a construct

$$a_1 \vee \dots \vee a_n :- b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m.$$

where a_1, \dots, a_n are standard atoms, b_1, \dots, b_m are atoms, and $n \geq 1, m \geq k \geq 0$. The disjunction $a_1 \vee \dots \vee a_n$ is referred to as the *head* of r and is denoted by $H(r)$, while the conjunction $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$ is the *body* of r and is denoted by $B(r)$. We will usually refer to $H(r)$ and $B(r)$ as the set of literals which they contain. Although this syntax does not explicitly allow integrity constraints, to be more precise rules with an empty head, they can be simulated in the usual way by using a new symbol and negation.

Example 2. The truth value of a fresh atom co can be forced to be false by adding the rule “ $p :- co, \text{not } p.$ ”, where p is also a fresh atom. Then, a constraint “ $:- a, b, c, \dots$ ” can be written as “ $co :- a, b, c, \dots$ ”.

An ASP^A program is a set of ASP^A rules. In the sequel, when it is clear from the context, we will omit the qualifier ASP^A.

Given a rule r , variables appearing in a standard atom of r , which is not inside an aggregate atom, are *global* variables, while all other variables (i.e. variables occurring only inside aggregate atoms) are *local* variables of r .

Safety. A rule r is *safe* if (i) each global variable of r appears in a positive standard literal of $B(r)$, and (ii) each local variable of r appearing in a symbolic set $\{C : p(\overline{X}, \overline{Y}, C)\}$ is a variable of $p(\overline{X}, \overline{Y}, C)$. A program \mathcal{P} is safe if all its rules are safe. In the following, ASP^A programs are assumed to be safe.

2.2 Semantics

Universe and Base. Given an ASP^A program \mathcal{P} , let $U_{\mathcal{P}}$ denote the set of constants appearing in \mathcal{P} , $pred(\mathcal{P})$ denote the set of predicates in \mathcal{P} , and $B_{\mathcal{P}}$ be the set of standard atoms constructible from predicates in $pred(\mathcal{P})$ with the constants in $U_{\mathcal{P}}$.

Instantiation. A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r is a *global substitution for r* , while a substitution from the set of local variables of a symbolic set S is a *local substitution for S* . If σ is a substitution and Obj an ASP^A object (rule, set, etc.), then $\sigma(Obj)$ denotes the object obtained from Obj by replacing each occurrence of variable X by $\sigma(X)$. Given a symbolic set without global variables $S = \{C : p(\bar{x}, \bar{Y}, C)\}$, the *instantiation of S* is the following ground set of pairs:

$$inst(S) = \{\langle \gamma(C) : \gamma(p(\bar{x}, \bar{Y}, C)) \rangle \mid \gamma \text{ is a local substitution for } S\}$$

Finally, a *ground instance* of a rule r is obtained in two steps. First of all, a global substitution σ for r is applied over r . Therefore, every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program \mathcal{P} is the set of all possible instances of the rules of \mathcal{P} .

Example 3. A program \mathcal{P}_1 and its ground version $Ground(\mathcal{P}_1)$ are shown below.

$$\begin{aligned} \mathcal{P}_1 &= \{ a(0) \vee a(1), \\ &\quad b(X) :- a(X), \#min\{C : a(C)\} < 1. \} \\ \\ Ground(\mathcal{P}_1) &= \{ a(0) \vee a(1), \\ &\quad b(0) :- a(0), \#min\{\langle 0 : a(0) \rangle, \langle 1 : a(1) \rangle\} < 1., \\ &\quad b(1) :- a(1), \#min\{\langle 0 : a(0) \rangle, \langle 1 : a(1) \rangle\} < 1. \} \end{aligned}$$

Interpretations. An *interpretation* for an ASP^A program \mathcal{P} is a consistent set of standard ground literals, that is $I \subseteq (B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}})$ such that $I \cap \neg.I = \emptyset$. A standard ground literal ℓ is true (resp. false) w.r.t I if $\ell \in I$ (resp. $\ell \in \neg.I$). If a standard ground literal is neither true nor false w.r.t I , then it is undefined w.r.t I . I^+ (resp. I^-) denotes the set of all atoms occurring in standard positive (resp. negative) literals in I , while \bar{I} denotes the set of undefined atoms w.r.t. I , i.e. $B_{\mathcal{P}} \setminus I^+ \cup I^-$. An interpretation I is *total* if \bar{I} is empty (i.e., $I^+ \cup \neg.I^- = B_{\mathcal{P}}$), otherwise I is *partial*. A *totalization* of a (partial) interpretation I is a total interpretation J containing I ; in other words, J is a total interpretation such that $I \subseteq J$. Moreover, total interpretations can be written in a more compact form omitting negative literals.

An interpretation also provides a meaning for aggregate literals. Their truth value is first defined for total interpretations, and then generalized to partial ones. Let I be a total interpretation. A standard ground conjunction is true (resp. false) w.r.t I if all (resp. some) of its literals are true (resp. false). The meaning of a set, an aggregate function, and an aggregate atom under an interpretation is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the constants of the elements in S whose conjunction is true w.r.t. I . More precisely, $I(S)$ denotes the multiset $[c \mid \langle c : p(\bar{x}, \bar{y}, c) \rangle \in S \wedge p(\bar{x}, \bar{y}, c) \text{ is true w.r.t. } I]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f , then $I(f(S)) = \perp$ (where \perp is a fixed symbol not occurring in \mathcal{P}). A ground aggregate atom A of the form $f(S) \prec k$ is *true w.r.t. I* if (i) $I(f(S)) \neq \perp$, and (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal $\text{not } A = \text{not } f(S) \prec k$ is *true w.r.t. I* if (i) $I(f(S)) \neq$

\perp , and (ii) $I(f(S)) \prec k$ does not hold; otherwise, not A is false. If I is a *partial* interpretation, an aggregate literal A is true (resp. false) w.r.t. I if it is true (resp. false) w.r.t. *each* totalization J of I ; otherwise it is undefined.

Example 4. Consider the atom $\#\min\{\langle 1:p(1)\rangle, \langle 2:p(2)\rangle\} \geq 1$, and let S be its ground set. For the interpretation $I_1 = \{p(2)\}$, each total interpretation extending it contains either $p(1)$ or not $p(1)$. Therefore, either $I_1(S) = [2]$ or $I_1(S) = [1, 2]$ and the application of $\#\min$ yields either 2 or 1. Hence, since they are both greater or equal than 1, the aggregate atom is true w.r.t. I_1 .

Our definitions of interpretation and truth values preserve “knowledge monotonicity”. If an interpretation J extends I , that is $I \subseteq J$, then each literal which is true w.r.t. I is true w.r.t. J , and each literal which is false w.r.t. I is false w.r.t. J as well.

Minimal Models. Given an interpretation I and a ground rule r , the head of r is *true w.r.t. I* if some literal in $H(r)$ is true w.r.t. I , while the body of r is *true w.r.t. I* if all literals in $B(r)$ are true w.r.t. I . Then, the rule r is *satisfied w.r.t. I* if its head is true w.r.t. I whenever its body is true w.r.t. I . A total interpretation M is a *model* of an ASP^A program \mathcal{P} if each $r \in \text{Ground}(\mathcal{P})$ is satisfied w.r.t. M . Furthermore, a model M for \mathcal{P} is (subset) *minimal* if no model N for \mathcal{P} exists such that $N^+ \subset M^+$.

Answer Sets. Given a ground program \mathcal{P} and a total interpretation I , the *reduct* of \mathcal{P} w.r.t. I is the program \mathcal{P}^I obtained from \mathcal{P} by deleting the rules with body false w.r.t. I [10]. Then, I is an *answer set* of \mathcal{P} iff I is a minimal model for $\text{Ground}(\mathcal{P})^M$.

Example 5. Let $I_2 = \{p(0), p(1)\}$ and $I_3 = \{\text{not } p(0), p(1)\}$ be two interpretations for two programs $\mathcal{P}_2 = \{p(1), p(0) :- \#\min\{C : p(C)\} < 1.\}$ and $\mathcal{P}_3 = \{p(1), p(0) :- \#\min\{C : p(C)\} \geq 1.\}$, whose ground versions are shown below.

$$\text{Ground}(\mathcal{P}_2) = \{ p(1), \\ p(0) :- \#\min\{\langle 0 : p(0)\rangle, \langle 1 : p(1)\rangle\} < 1. \}$$

$$\text{Ground}(\mathcal{P}_3) = \{ p(1), \\ p(0) :- \#\min\{\langle 0 : p(0)\rangle, \langle 1 : p(1)\rangle\} \geq 1. \}$$

Then, $\text{Ground}(\mathcal{P}_2)^{I_2} = \text{Ground}(\mathcal{P}_2)$ and $\text{Ground}(\mathcal{P}_3)^{I_3} = \text{Ground}(\mathcal{P}_3)$, while $\text{Ground}(\mathcal{P}_2)^{I_3} = \text{Ground}(\mathcal{P}_3)^{I_2} = \{p(1).\}$. Indeed, I_3 is the only answer set of \mathcal{P}_2 , whereas \mathcal{P}_3 has no answer set at all.

By the way they are defined, each answer set M of \mathcal{P} is also a model of \mathcal{P} because $\text{Ground}(\mathcal{P})^M \subseteq \text{Ground}(\mathcal{P})$, and the rules in $\text{Ground}(\mathcal{P}) \setminus \text{Ground}(\mathcal{P})^M$ are satisfied w.r.t. M .

Monotonicity. Given two interpretations I and J , $I \leq J$ if $I^+ \subseteq J^+$ and $J^- \subseteq I^-$. For a ground literal ℓ and for all interpretations I and J with $I \leq J$, if (i) ℓ true w.r.t. I implies ℓ true w.r.t. J and (ii) ℓ false w.r.t. J implies ℓ false w.r.t. I , then ℓ is *monotone*. On the other hand, if the opposite happens, that is (i) ℓ false w.r.t. I implies ℓ false w.r.t.

J and (ii) ℓ true w.r.t. J implies ℓ true w.r.t. I , then ℓ is *antimonotone*. Finally, if ℓ is neither monotone nor antimonotone, then it is *nonmonotone*.

Positive standard literals are monotone and negative standard literals are antimonotone, while aggregate literals can be monotone, antimonotone or nonmonotone, regardless whether they are positive or negative. More specifically, concerning extrema predicates, positive aggregates with $\#min$ and $<$ or \leq are monotone, while positive aggregates with $\#min$ and $>$ or \geq are antimonotone or nonmonotone. In fact, the satisfaction of the aggregate depends on two conditions, that is (i) the aggregate function is defined, and (ii) the minimum meets the comparison. Then, with $<$ or \leq , positive $\#min$ aggregates have two monotone “components”, whereas with $>$ or \geq one component is monotone and the other is antimonotone. Aggregates with $\#max$ behave in a dual way, and also negation inverts monotonicity. Some examples are shown below and the complete picture is summarized in Figure 1.

Example 6. Ground instances of $\#max\{Z : r(Z)\} > 1$ and $\text{not } \#max\{Z : r(Z)\} < 1$ are monotone, while for $\#max\{Z : r(Z)\} < 1$ and $\text{not } \#max\{Z : r(Z)\} > 1$ they are antimonotone or nonmonotone.

Function	Operator	Polarity	Character
$\#min$	$<, \leq$		monotone
		not	<i>nonmonotone</i>
	$>, \geq$	not	<i>nonmonotone</i> monotone
$\#max$	$<, \leq$		<i>nonmonotone</i>
		not	monotone monotone
	$>, \geq$	not	<i>nonmonotone</i>

Table 1. Character of aggregate literals

Another interesting property of aggregate literals is that they can always be written in a positive form. In fact, in our setting a negative aggregate literal is equal to a positive one with the opposite comparison operator. For this reason, in the following only positive aggregate literals are considered for simplicity.

3 Existing Compilation of Minimum and Maximum Aggregates

In this section first the standard encoding for isolating the minimum or maximum of a relation, which is “folklore” in the deductive databases community, is recalled and its properties are briefly discussed. Subsequently, the method proposed by [14, 15], which is based on the folklore encoding, is reviewed.

3.1 Determining Minima and Maxima of a Relation

A standard problem in deductive databases is providing an encoding for determining the minimum value of a unary relation p , the constants of which are comparable via a relation $<$, usually a built-in. The standard encoding is as follows:

$$\begin{aligned} \text{nonminp}(X) &:- p(X), p(Y), Y < X. \\ \text{minp}(X) &:- p(X), \text{not nonminp}(X). \end{aligned}$$

Example 7. A simple instance of this schema is

$$\begin{aligned} p(1). \\ p(2). \\ \text{nonminp}(X) &:- p(X), p(Y), Y < X. \\ \text{minp}(X) &:- p(X), \text{not nonminp}(X). \end{aligned}$$

The grounding is

$$\begin{aligned} p(1). \\ p(2). \\ \text{nonminp}(1) &:- p(1), p(1), 1 < 1. \\ \text{nonminp}(2) &:- p(2), p(2), 2 < 2. \\ \text{nonminp}(1) &:- p(1), p(2), 2 < 1. \\ \text{nonminp}(2) &:- p(2), p(1), 1 < 2. \\ \text{minp}(1) &:- p(1), \text{not nonminp}(1). \\ \text{minp}(2) &:- p(2), \text{not nonminp}(2). \end{aligned}$$

and it is easy to see that the only answer set is $\{p(1), p(2), \text{nonminp}(2), \text{minp}(1)\}$.

We can observe, however, that the encoding relies on default negation. As long as it occurs in a stratified way, as in Example 7, its behavior is very intuitive. However, as soon as it occurs in a recursive way, that is if the relation p depends on minp , its behavior is arguably less intuitive. In particular, in ASP an odd negative cycle may inhibit answer sets. Let us look at a small elaboration of Example 7.

Example 8. Consider now a scenario in which it is known that $p(2)$ holds, but in which $p(1)$ should hold only if the minimum value is less than or equal to 2. It is fairly obvious that the condition for $p(1)$ is already guaranteed, and so we would expect an answer set containing $p(1)$ and $p(2)$.

$$\begin{aligned} p(2). \\ p(1) &:- \text{minp}(X), X \leq 2. \\ \text{nonminp}(X) &:- p(X), p(Y), Y < X. \\ \text{minp}(X) &:- p(X), \text{not nonminp}(X). \end{aligned}$$

The grounding is

$$\begin{aligned} p(2). \\ p(1) &:- \text{minp}(1), 1 \leq 2. \\ p(1) &:- \text{minp}(2), 2 \leq 2. \\ \text{nonminp}(1) &:- p(1), p(1), 1 < 1. \\ \text{nonminp}(2) &:- p(2), p(2), 2 < 2. \\ \text{nonminp}(1) &:- p(1), p(2), 2 < 1. \\ \text{nonminp}(2) &:- p(2), p(1), 1 < 2. \\ \text{minp}(1) &:- p(1), \text{not nonminp}(1). \\ \text{minp}(2) &:- p(2), \text{not nonminp}(2). \end{aligned}$$

which, however, does not admit any answer set. The reduct with respect to the expected answer set $A = \{p(1), p(2), \text{nonminp}(2), \text{minp}(1)\}$ is

$$\begin{aligned} & p(2). \\ & p(1) :- \text{minp}(1), 1 \leq 2. \\ & \text{nonminp}(2) :- p(2), p(1), 1 < 2. \\ & \text{minp}(1) :- p(1), \text{not } \text{nonminp}(1). \end{aligned}$$

the minimal model of which is $\{p(2)\}$. A is therefore not an answer set.

The case of maxima is symmetric to minima, and thus the same considerations apply for that case.

3.2 Compilation of Aggregates: The Existing Approach

The ideas of Section 3.1 had been generalized in the literature for compiling minimum and maximum aggregates [14, 15]. We review and discuss this “classical” encoding in the sequel. In particular, we argue that it is inadequate for recursive aggregates. To simplify the presentation, since maximum is symmetric to minimum, in the following only minimum aggregates are considered.

The key idea of the classical encoding is that a value c is the minimum over a set term $\{C : p(\bar{X}, \bar{Y}, C)\}$ w.r.t. a ground assignment \bar{x} to the bounded variables iff $p(\bar{x}, \bar{y}, c)$ is true for some \bar{y} , and no $p(\bar{x}, \bar{y}', c')$, with $c' < c$ and any \bar{y}' , is true. Also, \bar{x} determines the instantiation of the aggregate and, consequently, $\{C : p(\bar{x}, \bar{Y}, C)\}$ identifies a unique ground set. This is very similar to the encodings in Section 3.1, the difference being that now there may be “externally bound” variables and also “existential” variables.

Definition 1. Let \mathcal{P} be a program with minimum aggregation. The classical encoding of \mathcal{P} in standard ASP consists of \mathcal{P} with each aggregate $\#\min\{C : p(\bar{X}, \bar{Y}, C)\} \prec k$ replaced by the conjunction

$$p(\bar{X}, \bar{Y}, C), \text{not } a\text{Lesser}_p(\bar{X}, \bar{Y}, C), C \prec k$$

and the new rule

$$a\text{Lesser}_p(\bar{X}, \bar{Y}, C) :- p(\bar{X}, \bar{Y}', C'), C' < C.$$

The resulting program is denoted by $se(\mathcal{P})$.

Intuitively, $a\text{Lesser}_p(\bar{x}, \bar{y}, c)$ true means that c is not the minimum value for the particular \bar{x} , where \bar{x} is a vector of witness constants for the existential variables. These atoms may be viewed as the generalization of *nonminp* of Example 7 and 8. So, the conjunction is intended to be true when the comparison operation holds for the minimum value.

We note that the rules added to $se(\mathcal{P})$ are unsafe, since \bar{Y} are global variables that do not appear in any positive literal of the body, and since C occurs in a non-restricting builtin predicate. This can, however, be fixed by adding a suitable domain predicate to the body, obtaining

$$a\text{Lesser}_p(\bar{X}, \bar{Y}, C) :- p(\bar{X}, \bar{Y}', C'), C' < C, p(\bar{X}, \bar{Y}, C).$$

There is another minor problem with the encoding as originally defined: It would use the same predicate $aLesser_p$ for each occurrence of p in an aggregate, which is of course not intended. However, this can be easily repaired by using an additional qualifier in the predicate name for each occurrence of p .

Let us look at an example for understanding how the transformation works.

Example 9. The following is the classical encoding of \mathcal{P}_3 in Example 5 and its instantiation.

$$se(\mathcal{P}_3) = \{ p(1),, \\ p(0) :- p(C), \text{not } aLesser_p(C), C \geq 1., \\ aLesser_p(C) :- p(C'), C' < C, p(C). \}$$

$$Ground(se(\mathcal{P}_3)) = \{ p(1),, \\ p(0) :- p(1), \text{not } aLesser_p(1), 1 \geq 1., \\ p(0) :- p(0), \text{not } aLesser_p(0), 0 \geq 1., \\ aLesser_p(0) :- p(0), 0 < 0, p(0),, \\ aLesser_p(0) :- p(1), 1 < 0, p(0),, \\ aLesser_p(1) :- p(1), 1 < 1, p(1),, \\ aLesser_p(1) :- p(0), 0 < 1, p(1). \}$$

Just as \mathcal{P}_3 , $se(\mathcal{P}_3)$ has no answer set.

It is not by chance that $se(\mathcal{P}_3)$ has no answer set, in general programs compiled using $se()$ cannot have new answer sets.

Theorem 1. *Each answer set of $se(\mathcal{P})$, restricted to the symbols in \mathcal{P} , is an answer set of \mathcal{P} .*

Proof. Let M' be an answer set of $se(\mathcal{P})$. Let M be an interpretation for \mathcal{P} obtained from M' by removing each $aLesser_p(\bar{x}, \bar{y}, c)$ literal. $Ground(\mathcal{P})^M$ has a rule r with $\#min\{C : p(\bar{x}, \bar{Y}, C)\} < k$ in the body iff $\exists p(\bar{x}, \bar{y}, c) \in M$ with $c < k$ and $\nexists p(\bar{x}, \bar{y}', c') \in M$ with $c' < c$. This implies that $aLesser_p(\bar{x}, \bar{y}, c) \notin M$ and, consequently, $\exists r' \in Ground(se(\mathcal{P}))^{M'}$ with $p(\bar{x}, \bar{y}, c), \text{not } aLesser_p(\bar{x}, \bar{y}, c), c < k$ in the body. Then M is a model for $Ground(\mathcal{P})^M$. Suppose $N \subset M$ is a model for $Ground(\mathcal{P})^M$. Let N' be an interpretation for $Ground(se(\mathcal{P}))^{M'}$ obtained from N by adding $aLesser_p(\bar{x}, \bar{y}, c)$ where $\exists p(\bar{x}, \bar{y}', c') \in N$ and $c' < c$. Then N' satisfies each rule of $Ground(se(\mathcal{P}))^{M'}$ with an $aLesser$ atom in head. For each rule $r \in Ground(\mathcal{P})^M$ with an aggregate literal $\#min\{C : p(\bar{x}, \bar{Y}, C)\} < k$ true in the body, all rules $r' \in Ground(se(\mathcal{P}))^{M'}$ related to r are satisfied by N' . In fact, the aggregate is true w.r.t. N iff $\exists p(\bar{x}, \bar{y}, c) \in N$ with $c < k$ and $\nexists p(\bar{x}, \bar{y}', c') \in N$ with $c' < c$. It follows that $aLesser_p(\bar{x}, \bar{y}, c) \notin N'$ and then N' is a model for $Ground(se(\mathcal{P}))^{M'}$, contradicting the hypothesis. Therefore, M is a minimal model of $Ground(\mathcal{P})^M$ and so an answer set of \mathcal{P} .

However, there is also a major problem with the encoding, which has the same root as the unintuitive behavior described in Example 8. If the truth assignment of the aggregated atoms depends from the evaluation of the aggregate, that is if the aggregate is recursive, it is possible that some answer sets are lost. In fact, the following example demonstrates that not all the answer sets of a program \mathcal{P} are answer sets of $se(\mathcal{P})$.

Example 10. Let \mathcal{P}_4 and $se(\mathcal{P}_4)$ be the program below and its standard encoding, respectively.

$$\begin{aligned}
\mathcal{P}_4 &= \{ p(1),, \\
&\quad p(0) :- \#min\{C : p(C)\} \leq 1. \quad \} \\
se(\mathcal{P}_4) &= \{ p(1),, \\
&\quad p(0) :- p(C), \text{not } aLesser_p(C), C \leq 1., \\
&\quad aLesser_p(C) :- p(C'), p(C), C' < C. \quad \} \\
Ground(se(\mathcal{P}_4)) &= \{ p(1),, \\
&\quad p(0) :- p(0), \text{not } aLesser_p(0), 0 \leq 1., \\
&\quad p(0) :- p(1), \text{not } aLesser_p(1), 1 \leq 1., \\
&\quad aLesser_p(0) :- p(0), p(0), 0 < 0., \\
&\quad aLesser_p(0) :- p(1), p(0), 1 < 0., \\
&\quad aLesser_p(1) :- p(1), p(1), 1 < 1., \\
&\quad aLesser_p(1) :- p(0), p(1), 0 < 1. \quad \}
\end{aligned}$$

While \mathcal{P}_4 has one answer set, $A = \{p(0), p(1)\}$, no interpretation is an answer set of $se(\mathcal{P}_4)$. In particular, we would expect $B = \{p(0), p(1), aLesser_p(1)\}$, but the reduct is

$$\begin{aligned}
Ground(se(\mathcal{P}_4))^B &= \{ p(1),, \\
&\quad p(0) :- p(0), \text{not } aLesser_p(0), 0 \leq 1., \\
&\quad aLesser_p(1) :- p(0), p(1), 0 < 1. \quad \}
\end{aligned}$$

the minimal model of which is $\{p(1)\}$.

In this example, the problem is that $p(0)$ is inferred by the satisfaction of the aggregate. But the concept of aggregate satisfaction is not directly present in this encoding, which instead determines the exact result of the aggregation and, as a second step, compares this value with the guard. In this case, the minimum p is exactly $p(0)$ and, as a consequence, the only rule that supports $p(0)$ depends on the truth of $p(0)$, which causes the instability.

4 Correct Compilation of Minimum and Maximum Aggregates

In this section we propose an alternative strategy for compiling aggregates. This strategy has to take care of comparison operations and the monotone, antimonotone, or nonmonotone character of aggregates. In fact, to infer that a minimum aggregation is less than or equal to a value, it is sufficient to have one element less than or equal to that value, rather than determining the absolute minimum. On the other hand, a minimum aggregation is greater than or equal to a value iff its aggregate set is not empty, that is the aggregate is defined, and every element in the aggregate set is greater than or equal to that value. Similar if the comparison operator is strictly less or strictly greater. Again, we treat only minimum aggregates for space reasons, maximum aggregates being exactly symmetric.

Definition 2. Let \mathcal{P} be a program with minimum aggregates. Then, $mae(\mathcal{P})$ is the program obtained from \mathcal{P} by substituting each aggregate $\#\min\{C : p(\overline{X}, \overline{Y}, C)\} \prec k$ with a new atom $\#\min_{p \prec k}(\overline{X})$ and by adding only the required new rules

- (a) $\#\min_p(\overline{X}) \quad :- \quad p(\overline{X}, \overline{Y}, C).$
- (b) $\#\min_{p \leq k}(\overline{X}) \quad :- \quad p(\overline{X}, \overline{Y}, C), C \leq k.$
- (c) $\#\min_{p < k}(\overline{X}) \quad :- \quad p(\overline{X}, \overline{Y}, C), C < k.$
- (d) $\#\min_{p \geq k}(\overline{X}) \quad :- \quad \#\min_p(\overline{X}), \text{not } \#\min_{p < k}(\overline{X}).$
- (e) $\#\min_{p > k}(\overline{X}) \quad :- \quad \#\min_p(\overline{X}), \text{not } \#\min_{p \leq k}(\overline{X}).$

where $\#\min_p(\overline{X})$ stands for “the minimum of p for the global variables \overline{X} is defined” and \prec determines which rules are required: (b) for \leq ; (c) for $<$; (a), (c) and (d) for \geq ; (a), (b) and (e) for $>$.

These rules take care of the monotone, antimonotone, or nonmonotone character of aggregates. In fact, rules (a), (b) and (c) have only monotone literals in the body, while (d) and (e) have both monotone and antimonotone literals. Let us examine how this transformation works on program \mathcal{P}_4 , which was problematic for transformation $se()$.

Example 11. Consider the transformation $mae()$ applied on \mathcal{P}_4 of Example 10:

$$mae(\mathcal{P}_4) \quad = \{ p(1), \\ p(0) :- \#\min_{p \leq 1}, \\ \#\min_{p \leq 1} :- p(C), C \leq 1. \}$$

$$Ground(mae(\mathcal{P}_4)) = \{ p(1), \\ p(0) :- \#\min_{p \leq 1}, \\ \#\min_{p \leq 1} :- p(0), 0 \leq 1, \\ \#\min_{p \leq 1} :- p(1), 1 \leq 1. \}$$

Obviously, $\{p(0), p(1), \#\min_{p \leq 1}\}$ is the only answer set of $mae(\mathcal{P}_4)$.

In order to formulate a correspondence result, we need some notion for correlating the answer sets of the original and the compiled programs.

Definition 3. An interpretation I for \mathcal{P} is extended to an interpretation $ext(I)$ for $mae(\mathcal{P})$ by adding

- $\#\min_p(\overline{x})$ if $\#\min_p \in \text{pred}(mae(\mathcal{P}))$ and there is $p(\overline{x}, \overline{y}, c) \in I$,
- $\#\min_{p \leq k}(\overline{x})$ if $\#\min_{p \leq k} \in \text{pred}(mae(\mathcal{P}))$ and there is $p(\overline{x}, \overline{y}, c) \in I$ with $c \leq k$,
- $\#\min_{p < k}(\overline{x})$ if $\#\min_{p < k} \in \text{pred}(mae(\mathcal{P}))$ and there is $p(\overline{x}, \overline{y}, c) \in I$ with $c < k$,
- $\#\min_{p \geq k}(\overline{x})$ if $\#\min_{p \geq k} \in \text{pred}(mae(\mathcal{P}))$, $\#\min_p(\overline{x}) \in ext(I)$ and $\#\min_{p < k}(\overline{x}) \notin ext(I)$,
- $\#\min_{p > k}(\overline{x})$ if $\#\min_{p > k} \in \text{pred}(mae(\mathcal{P}))$, $\#\min_p(\overline{x}) \in ext(I)$ and $\#\min_{p \leq k}(\overline{x}) \notin ext(I)$.

Also, an interpretation I' for $mae(\mathcal{P})$ is restricted to an interpretation $res(I') = I' \cap U_{\mathcal{P}}$, that is by eliminating all atoms with predicates that are not in \mathcal{P} .

Example 12. By extending answer set $I_2 = \{p(0), p(1)\}$ of \mathcal{P}_4 we obtain

$$ext(I_2) = \{p(0), p(1), \#\min_{p \leq 1}\}.$$

which is the only answer set of $mae(\mathcal{P}_4)$.

By construction, each $ext(I)$ satisfies the rules added to $mae(\mathcal{P})$. In addition, $\#min_{p \prec k}(\bar{x}) \in ext(I)$ iff $\#min\{C : p(\bar{x}, \bar{Y}, C)\} \prec k$ is an atom of \mathcal{P} true w.r.t. I . As a consequence, if I is a model for \mathcal{P} , then $ext(I)$ is a model of $mae(\mathcal{P})$ and each rule r of $Ground(\mathcal{P})^I$ has a unique counterpart r' in $Ground(mae(\mathcal{P}))^I$. We observe thus that ext and res determine a one-to-one correspondence between the answer sets of \mathcal{P} and $mae(\mathcal{P})$.

Theorem 2. *There is a bijection between the answer sets of \mathcal{P} and $mae(\mathcal{P})$.*

Proof. Let M be an answer set of \mathcal{P} and $M' = ext(M)$ be an interpretation for $mae(\mathcal{P})$. By construction, M' is a model of $Ground(mae(\mathcal{P}))^{M'}$. Suppose, by contradiction, that $N' \subset M'$ is a model of $Ground(mae(\mathcal{P}))^{M'}$. If $N = res(N')$ is not a model of $Ground(\mathcal{P})^M$, then $\exists r \in Ground(\mathcal{P})^M$ such that $H(r)$ is false and $B(r)$ is true w.r.t. N . $N \neq M$ must hold, as otherwise one of rules (a) to (e) of Definition 2 would not be satisfied. $B(r)$ has to contain a $\#min\{C : p(\bar{x}, \bar{Y}, C)\} \prec k$. If $\prec \in \{<, \leq\}$, then $\exists p(\bar{x}, \bar{y}, c) \in N$ such that $c \prec k$. It follows that $\#min_{p \prec k}(\bar{x})$ belongs to N' . If $\prec \in \{>, \geq\}$, then the minimum is defined and $\nexists p(\bar{x}, \bar{y}, c) \in N$ with $c \not\prec k$. Therefore, $\#min_{p \prec k}(\bar{x}) \in N'$ and so the counterpart of r in $Ground(mae(\mathcal{P}))^{M'}$ is not satisfied by N' , which is a contradiction. Thus, M' is a minimal model of $Ground(mae(\mathcal{P}))$ and so an answer set of $mae(\mathcal{P})$.

Let M' be an answer set of $mae(\mathcal{P})$ and $M = res(M')$ be an interpretation for \mathcal{P} . Since M' is minimal, $\#min_{p \prec k}(\bar{x}) \in M'$ with $\prec \in \{<, \leq\}$ implies that there is $p(\bar{x}, \bar{y}, c) \in M$ with $c \prec k$ and, then, $\#min\{C : p(\bar{x}, \bar{Y}, C)\} \prec k$ is true w.r.t. M . Moreover, if $\prec \in \{>, \geq\}$, then the minimum is defined and $p(\bar{x}, \bar{y}, c) \in M$ implies $c \prec k$. So, $\#min\{C : p(\bar{x}, \bar{Y}, C)\} \prec k$ is true w.r.t. M . It follows that M is a model of $Ground(\mathcal{P})^M$. Suppose, by contradiction, that $N \subset M$ is a model of $Ground(\mathcal{P})^M$. Then, a model for $Ground(mae(\mathcal{P}))^{M'}$ can be obtained by removing from M' the atoms removed from M and the $\#min$ atoms that no longer have a reason to be true, viz. $N' = M' \cap ext(N)$. In fact, if a rule $r' \in Ground(mae(\mathcal{P}))^{M'}$ is not satisfied by N' , then $B(r')$ has to contain a $\#min$ literal. Therefore, the counterpart of r' in $Ground(\mathcal{P})^M$ is unsatisfied by N , leading to a contradiction. Thus, M is a minimal model of $Ground(\mathcal{P})$ and so an answer set of \mathcal{P} .

5 Related Work

Extrema aggregates like $\#min$ and $\#max$ for logic programming languages have been studied together with other aggregates since the 1980ies, when their utility has been perceived in deductive databases, such as LDL [17]. After that, many studies on this topic have been undertaken [18, 19]. In fact, different semantics have been defined for logic programs with aggregates and especially recursive aggregates [20, 5, 21–23, 9, 12, 10]. Almost all of the proposals are different on some language fragment, with pros and cons of each approach [23, 24, 12]. For our purposes, we adopted the semantics that seems to have found a consensus [10], as attested by its alternative characterizations proposed in other works [25–27].

About minimum and maximum aggregates in particular, relatively few works are present in literature. Some of them examine the problem for positive Datalog and adopt

the classical encoding described in Section 3 [14, 15]. Nevertheless, the equivalence of that encoding with the original program was not formally examined in these papers. In fact, the program \mathcal{P}_4 in Example 10 is positive and also monotone, and thus can serve as a rebuttal for that way of defining the semantics of minimum and maximum aggregates.

6 Conclusion

The properties of answer set programming with minimum and maximum aggregates have been analyzed in this work. In particular, we have examined possibilities for compiling these aggregates into standard ASP, which must be possible as a consequence of results of previous complexity studies. First, an encoding given in the literature has been examined, but shown to be inadequate in Section 3.

An alternative compilation method has then been proposed, which takes the monotone, antimonotone, or nonmonotone components of aggregates into account. The resulting *monotone/antimonotone encoding*, defined in Section 4, has been shown to admit answer sets which are in a one-to-one correspondence to the answer sets of the original program.

These results can be utilized to implement a frontend for minimum and maximum aggregates which uses a standard ASP^A solver as a computational engine. It should be noted that the transformation produces nondisjunctive rules only, so this method may be used for disjunctive logic programming, using solvers such as DLV [28], GnT [29], or Cmodels3 [30] as backends, but also for nondisjunctive programs, using for example Smodels [31] or clasp [32] (among many others) as backends.

References

1. Przymusiński, T.C.: Stable Semantics for Disjunctive Programs. *New Generation Computing* **9** (1991) 401–424
2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
3. Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In Saraswat, V.A., Ueda, K., eds.: *Proceedings of the International Symposium on Logic Programming (ISLP'91)*, MIT Press (1991) 387–401
4. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In Codognet, P., ed.: *Proceedings of the 17th International Conference on Logic Programming*, Springer Verlag (2001) 212–226
5. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A.C., Sadri, F., eds.: *Computational Logic. Logic Programming and Beyond*. Volume 2408 of LNCS. Springer (2002) 413–451
6. Simons, P., Niemelä, I., Sooinen, T.: Extending and Implementing the Stable Model Semantics. *Artificial Intelligence* **138** (2002) 181–234
7. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, Acapulco, Mexico, Morgan Kaufmann Publishers (2003) 847–852

8. Pelov, N., Truszczyński, M.: Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In: Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada. (2004) 327–334
9. Pelov, N., Denecker, M., Bruynooghe, M.: Partial stable models for logic programs with aggregates. In: Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). Volume 2923 of Lecture Notes in AI (LNAI), Springer (2004) 207–219
10. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In Alferes, J.J., Leite, J., eds.: Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004). Volume 3229 of Lecture Notes in AI (LNAI), Springer Verlag (2004) 200–212
11. Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. *Theory and Practice of Logic Programming* **7** (2007) 355–375
12. Son, T.C., Pontelli, E., Elkabani, I.: On Logic Programming with Aggregates. Technical Report NMSU-CS-2005-006, New Mexico State University (2005)
13. Faber, W., Leone, N.: On the Complexity of Answer Set Programming with Aggregates. In Baral, C., Brewka, G., Schlipf, J.S., eds.: *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR 2007*. Volume 4483 of Lecture Notes in AI (LNAI), Tempe, AZ, USA, Springer Verlag (2007) 97–109
14. Furfaro, F., Greco, S., Ganguly, S., Zaniolo, C.: Pushing extrema aggregates to optimize logic queries. *Information Systems* **27**(5) (2002) 321–343
15. Ganguly, S., Greco, S., Zaniolo, C.: Extrema predicates in deductive databases. *Journal of Computer and System Sciences* **51**(2) (1995) 244–259
16. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
17. Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S.A., Tsur, S., Zaniolo, C.: The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering* **2**(1) (1990)
18. Ross, K.A., Sagiv, Y.: Monotonic Aggregation in Deductive Databases. *Journal of Computer and System Sciences* **54**(1) (1997) 79–97
19. Kemp, D.B., Ramamohanarao, K.: Efficient Recursive Aggregation and Negation in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering* **10** (1998) 727–745
20. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In Dix, J., Furbach, U., Nerode, A., eds.: *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97)*. Volume 1265 of LNCS., Springer (1997) 290–309
21. Dell’Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in DLV. In de Vos, M., Proveti, A., eds.: *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, Messina, Italy (2003) 274–288 Online at <http://CEUR-WS.org/Vol-78/>.
22. Niemelä, I., Simons, P., Sooinen, T.: Stable Model Semantics of Weight Constraint Rules. In Gelfond, M., Leone, N., Pfeifer, G., eds.: *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’99)*. Volume 1730 of Lecture Notes in AI (LNAI), El Paso, Texas, USA, Springer Verlag (1999) 107–116
23. Pelov, N.: *Semantics of Logic Programs with Aggregates*. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium (2004)
24. Pelov, N., Denecker, M., Bruynooghe, M.: *Well-founded and Stable Semantics of Logic Programs with Aggregates*. *Theory and Practice of Logic Programming* (2007) Accepted for publication, available in CoRR as cs.LO/0509024.

25. Ferraris, P.: Answer Sets for Propositional Theories. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05*, Diamante, Italy, September 2005, Proceedings. Volume 3662 of *Lecture Notes in Computer Science.*, Springer Verlag (2005) 119–131
26. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*. (2005) 406–411
27. Faber, W.: Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05*, Diamante, Italy, September 2005, Proceedings. Volume 3662 of *Lecture Notes in Computer Science.*, Springer Verlag (2005) 40–52
28. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* 7(3) (2006) 499–562
29. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*. Volume 2923 of *LNAI.*, Springer (2004) 331–335
30. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05*, Diamante, Italy, September 2005, Proceedings. Volume 3662 of *Lecture Notes in Computer Science.*, Springer Verlag (2005) 447–451
31. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A System for Answer Set Programming. In Baral, C., Truszczyński, M., eds.: *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA (2000) Online at <http://xxx.lanl.gov/abs/cs/0003033v1>.
32. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Morgan Kaufmann Publishers (2007) 386–392

Normal Form Nested Programs^{*}

Annamaria Bria, Wolfgang Faber, and Nicola Leone

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{a.bria, faber, leone}@mat.unical.it

Abstract. Disjunctive logic programming under the answer set semantics (*DLP*, *ASP*) has been acknowledged as a versatile formalism for knowledge representation and reasoning during the last decade. Lifschitz, Tang, and Turner have introduced an extended language of *DLP*, called Nested Logic Programming (*NLP*), in 1999 [1]. It often allows for more concise representations by permitting a richer syntax in rule heads and bodies. However, that language is propositional and thus does not allow for variables, one of the strengths of *DLP*.

In this paper, we introduce a language similar to *NLP*, called Normal Form Nested (*NFN*) programs, which does allow for variables, and present the syntax and semantics. The presence of variables, however, gives rise to potential problems, such as the lack of domain independence. We study these issues in depth and define the class of safe *NFN* programs, which are guaranteed to be domain independent. Moreover, we show that for *NFN* programs which are also *NLP*s, our semantics coincides with the one of Lifschitz, Tang, and Turner. Finally, we provide an algorithm which translates *NFN* programs into *DLP* programs, and does so in an efficient way.

1 Introduction

In disjunctive logic programming (*DLP*) the heads (resp. the bodies) of the rules are disjunctions (resp. conjunctions) of simple constructs, viz. atoms and literals. *DLP*, under the answer sets semantics [2, 3], are widely recognized as important tools for knowledge representation and reasoning [4]. This is evidenced by the availability of software systems for computing answer sets for *DLP*, such as DLV [5], GnT [6], and Cmodels3 [7].

Lifschitz, Tang and Turner [1] extended the answer set semantics (in the propositional or ground case) to a class of logic programs where the heads and the bodies of the rules are nested expressions. Nested expressions are formed from negation-as-failure literals, conjunction and disjunction, nested arbitrarily. This class of programs, called *nested logic programs*, generalize the class of disjunctive logic programs. However, as shown in [1, 8], nested logic programs can be transformed into disjunctive logic programs. These results allow for evaluating ground nested logic programs using *DLP* systems, such as DLV, GnT, or Cmodels3.

^{*} Supported by M.I.U.R. within projects “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva” and “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione.”

In this paper, we extend nonground *DLP* to a class of programs, in which rule heads are boolean formulas in disjunctive normal form made of atoms, and in which the rule bodies are boolean formulas in conjunctive normal form made of literals. These programs referred to as Normal Form Nested (*NFN*) programs, and different to nested logic programs of [1], they may contain variables. We study some important properties of this class of programs, and provide a polynomial translation from *NFN* programs to *DLP*.

Summarizing, the main contributions of this paper are the following:

- We extend Disjunctive Logic Programming with variables introducing conjunctions in the head of the rules and disjunctions in the body of the rules, obtaining a new language, Normal Form Nested Programs (*NFN*). We formally define the syntax and semantics of this language.
- We study the properties of the *NFN* programs showing in the following results:
 - The answer sets for *NFN* coincide with the the answer sets of [1] for Nested Logic Programs, on the common language fragment.
 - We provide a definition of safe *NFN* programs. We show that every safe program is domain independent, that is, it has the same answer sets on each universe extending the constants of the program.
- We present an efficient algorithm that transforms normal form nested programs to disjunctive logic programs such that there is a one-to-one correspondence between their answer sets.

2 *NFN* Language

In this section, we provide a formal definition of the syntax and semantics of the *NFN* language.

2.1 Syntax

A variable or a constant is a term. An atom is $a(t_1, \dots, t_n)$ where a is a predicate of arity n and t_1, \dots, t_n are terms. A literal is either a positive literal p or a negative literal **not** p , where p is an atom. A *basic conjunction* is l_1, \dots, l_n where each l_1, \dots, l_n is a literal; if each l_1, \dots, l_n is an atom, the basic conjunction is *positive*. A *basic disjunction* is $k_1 \vee \dots \vee k_n$ where each k_1, \dots, k_n is a literal; if each k_1, \dots, k_n is an atom, the basic disjunction is *positive*. A (*normal form nested*) *rule* r is a syntactic of the following form:

$$C_1 \vee \dots \vee C_n \text{ :- } D_1, \dots, D_m. \quad n, m \geq 0$$

where C_1, \dots, C_n are positive basic conjunctions and D_1, \dots, D_m are basic disjunctions. The disjunction $C_1 \vee \dots \vee C_n$ is the *head* of r ($H(r)$) while the conjunction D_1, \dots, D_m is the *body* of r ($B(r)$). The set of all positive basic disjunctions of $B(r)$ is denoted by $B^+(r)$ and the set of basic disjunctions that contain a negative literal is denoted by $B^-(r)$. A rule is *positive* if $B(r) = B^+(r)$. If all C_i and all D_j are atoms and literals respectively, the rule is called *standard*.

Example 1 (Normal Form Nested Rule). The following is a normal form nested rule:

$$a(X) \vee (b(X), c(X)) :- d(X), (e(X) \vee \mathbf{not} f(Y)), (d(X) \vee s(Z) \vee f(X)).$$

where $d(X)$ and $(d(X) \vee s(Z) \vee f(X))$ are positive basic disjunctions, while $(e(X) \vee \mathbf{not} f(Y))$ is not since it contains the negative literal $\mathbf{not} f(Y)$.

An *NFN* program P is a finite set of rules. P is a *positive program* if all rules of P are positive. P is a *standard program* if all its rules are standard. Next we use Σ to denote a general construct (literal, disjunct, conjunct, etc.). We denote by $const(\Sigma)$ the set of constants that appear in a construct Σ and $vars(\Sigma)$ the set of variables that appear in Σ . A construct Σ is *ground* iff $vars(\Sigma) = \emptyset$.

2.2 Semantics

Program Instantiation Given an *NFN* program P , let U_P be the set of constants appearing in P and B_P be the set of atoms constructible from the predicates of P with constants in U_P . Let $U \supseteq U_P$ be a set of constants and r an *NFN* rule. A *substitution* is a function $\sigma : vars(r) \mapsto U$ that maps the variables of r to some constants in U . Given a substitution σ , a *ground instance* of a construct Σ w.r.t. U is denoted by $\Sigma\sigma$. The *instantiation* of r , denoted by $Ground(r, U)$, is the set of all ground instances of r w.r.t. U .

Example 2. Let $U = \{1, 2\}$ be a set of constants and r the following rule:

$$(a(X), b(Y)) \vee c(X) :- (p(X, Y) \vee q(Y)), q(X).$$

The instantiation $Ground(r, U)$ is the following set of rules:

$$\begin{aligned} (a(1), b(1)) \vee c(1) &:- (p(1, 1) \vee q(1)), q(1). \\ (a(1), b(2)) \vee c(1) &:- (p(1, 2) \vee q(2)), q(1). \\ (a(2), b(2)) \vee c(2) &:- (p(2, 2) \vee q(2)), q(2). \\ (a(2), b(1)) \vee c(2) &:- (p(2, 1) \vee q(1)), q(2). \end{aligned}$$

Given a program P , the *instantiation* of P w.r.t. U , denoted by $Ground(P, U)$ is defined as follows:

$$Ground(P, U) = \bigcup_{r \in P} Ground(r, U).$$

The instantiation of P w.r.t. U_P , is denoted by $Ground(P)$ (i.e. $Ground(P) = Ground(P, U_P)$).

Interpretation and Models An *interpretation* I , for an *NFN* program P , is a set of ground atoms $I \subseteq B_P$. A ground atom a is *true* (resp. *false*) w.r.t. I if $a \in I$ (resp. $a \notin I$). A ground negative literal $\mathbf{not} a$ is *true* (resp. *false*) w.r.t. I if $a \notin I$ (resp. $a \in I$).

A ground basic disjunction L is *true* w.r.t. I if at least one literal of L is true w.r.t. I ; otherwise L is *false* w.r.t. I . A ground basic conjunction A is *true* w.r.t. I if all atoms

of A are true w.r.t. I ; otherwise A is *false* w.r.t. I . Similarly, the head of a ground *NFN* rule r is true w.r.t. I if at least one basic conjunction of $H(r)$ is true w.r.t. I and the body of r is true w.r.t. I if all basic disjunctions of $B(r)$ are true w.r.t. I .

A ground *NFN* rule r is *satisfied* w.r.t. I if the head is true w.r.t. I or the body is false w.r.t. I . Next we denote truth and falseness of a construct Σ with $I \models \Sigma$ and $I \not\models \Sigma$ respectively.

Given a rule or a program Δ , we also denote the satisfiability of Δ w.r.t. I by $I \models \Delta$, and unsatisfiability of Δ w.r.t. I by $I \not\models \Delta$. A *model* for P is an interpretation M for P such that for each rule $r \in \text{Ground}(P)$, $M \models r$. A model M for P is *minimal* if no model N for P exists such that $N \subsetneq M$.

Answer sets Next we define a reduct for ground *NFN* programs w.r.t. an interpretation that is an adaptation of the reduct defined by Lifschitz, Tang, and Turner for nested logic programs (see Theorem 1).

Definition 1 (Reduct). *Let r be a ground *NFN* rule and I an interpretation. The reduct of r w.r.t. I , denoted by r^I , is derived from r by deleting all false literals w.r.t. I from each disjunction of the body. If any disjunction of r becomes empty, the reduct of r is void.*

*The reduct of a ground *NFN* program P w.r.t. I , denoted by P^I , is the union of the reducts of the rules in P .*

Observation 1 r^I exists (is not void) iff $I \models B(r)$.

Example 3. Consider the following *NFN* program P :

$$a. \quad b. \quad f \vee (d, e) :- (a \vee \mathbf{not} c). \quad p :- (\mathbf{not} a \vee \mathbf{not} b). \quad g :- (b \vee \mathbf{not} a).$$

and interpretation $I = \{a, b, f, g\}$, then P^I is the following program:

$$a. \quad b. \quad f \vee (d, e) :- (a \vee \mathbf{not} c). \quad g :- b.$$

Definition 2 (Answer set for *NFN* program). *Given an *NFN* program P , an interpretation I is an answer set for P if I is a minimal model for $\text{Ground}(P)^I$.*

The set of answer sets for P is denoted by $AS(P)$.

Example 4. In Example 3, I is an answer set for the program P .

3 Language Properties

In this section we study important properties of *NFN* programs.

3.1 Equivalence to the Semantics of Lifschitz, Tang, and Turner

Definition 3 (Reduct of Lifschitz, Tang, and Turner [1]). Let I a set of atoms. The reduct w.r.t. I of a construct (atom, literal, conjunction, etc.) is defined recursively as follows:

- $a^I = a$, if a is an atom;
- $(\mathbf{not} a)^I = \perp$ if $I \not\models (\mathbf{not} a)$; \top otherwise;
- $(l_1, \dots, l_n)^I = (l_1^I, \dots, l_n^I)$;
- $(k_1 \vee \dots \vee k_m)^I = (k_1^I \vee \dots \vee k_m^I)$;
- $r^I = (H :- B.)^I = H^I :- B^I$, for a rule r ;
- $P^I = \{(H :- B.)^I : H :- B. \in P\}$, for a program P .

Next we denote by Δ^{I_L} the reduct of Δ according to Definition 3.

Example 5. Consider P and I of Example 3, P^{I_L} is the following program:

$$a. \quad b. \quad f \vee (d, e) :- (a \vee \top). \quad p :- (\perp \vee \perp). \quad g :- (b \vee \perp).$$

Definition 4 (Answer Set of Lifschitz, Tang, and Turner [1]). A set of atoms I is an NLP answer set for an NFN program P if it is a minimal model of P^{I_L} .

Example 6. In Example 5, I is an NLP answer set of the program P^{I_L} .

Looking at Definition 1, similar to the reduct for *DLP* programs defined in [9], all rules with false body are deleted. Furthermore, from rule bodies of the remaining rules all false body literals are deleted. The latter deletion is necessary to have a program that has same minimal models of the program obtained after the reduct in Definition 3, as shown in the following example.

Example 7. Let us consider program $P = \{a :- (b \vee \mathbf{not} c), \quad d :- \mathbf{not} f.\}$ and interpretation $I = \{b, c\}$. If we delete all rules with false body w.r.t. I , we obtain a reduct program $P_1 = P$. The program P_2 , obtained using the reduct of Lifschitz, Tang, and Turner, is $\{a :- (b \vee \perp), \quad d :- \top.\}$.

$\{d\}$ is a model for P_2 but $\{d\}$ is not a model for P_1 , because the body of the first rule of P is true w.r.t. $\{d\}$ but the head is false w.r.t. $\{d\}$. The reduct P^I of P , according to Definition 1, is $\{a :- b, \quad d :- \mathbf{not} f.\}$, and $\{d\}$ is indeed a (minimal) model of P^I .

Lemma 1. Let r be a ground NFN rule and I a set of atoms, then:

$$I \not\models B(r) \Leftrightarrow I \not\models B(r^{I_L}) \tag{1}$$

and

$$\text{if } r^I \text{ exists, } \forall J \subseteq I : \quad J \models r^I \Leftrightarrow J \models r^{I_L}. \tag{2}$$

Proof. (1) $I \not\models B(r)$ iff a basic disjunction $D \in B(r)$ exists s.t. $I \not\models D$ iff for all $l \in D$, $I \not\models l$. The corresponding basic disjunction $D^{I_L} \in B(r^{I_L})$ of D contains the same positive literals of D and, if $l = \mathbf{not} a$, D^{I_L} contains \perp . Then $I \not\models B(r^{I_L})$. Vice versa, $I \not\models B(r^{I_L})$ iff $D^{I_L} \in B(r^{I_L})$ exists s.t. $I \not\models D^{I_L}$ iff for each $l \in D^{I_L}$, $I \not\models l$ where l is an atom or $l = \perp$. The corresponding basic disjunction $D \in B(r)$ of D^{I_L}

contains the same atoms of D^{I_L} and, for each $\perp \in D^{I_L}$, D contains a negative literal l_n , s.t. $I \not\models l_n$. Consequently, $I \not\models D$ and, hence, $I \not\models B(r)$.

(2) $J \models B(r^I)$ iff for each basic disjunction $D^I \in B(r^I)$ a literal $l \in D^I$ exists s.t. $J \models l$. If $l = \mathbf{not} a$, $I \models l$ follows from Definition 1 and each corresponding basic disjunction $D^{I_L} \in B(r^{I_L})$ of D^I contains \top . Otherwise, if l is a positive literal, the corresponding D^{I_L} of D^I also contains l . As a result $J \models B(r^{I_L})$. Since both reducts do not modify the head of the rules, if $J \models r^I$ and $J \models B(r^I)$ then $J \models H(r^I) = H(r^{I_L})$ and $J \models B(r^{I_L})$ hence $J \models r^{I_L}$; if $J \models r^{I_L}$ and $J \models B(r^{I_L})$ then $J \models H(r^{I_L}) = H(r^I)$ and $J \models B(r^I)$, hence $J \models r^I$.

Theorem 1. *Given an NFN program P , an interpretation I is an answer set of P according to Definition 2 iff I is an NLP answer set of P according to Definition 4.*

Proof. (\Rightarrow) If I is a minimal model for $Ground(P)^I$, for each $r \in Ground(P)$, s.t. r^I exists, from Observation 1 and from (2) of Lemma 1, $I \models r^{I_L}$. For rules $r \in Ground(P)$, for which no r^I but a r^{I_L} exists, $I \not\models r$ and from (1) of Lemma 1, $I \not\models B(r^{I_L})$. Consequently I is a model for $Ground(P)^{I_L}$. Moreover, no $J \subset I$ is a model for $Ground(P)^{I_L}$, as it would also be a model for $Ground(P)^I$ because of (2) of Lemma 1.

(\Leftarrow) Let I be a minimal model for $Ground(P)^{I_L}$. For each $r^I \in Ground(P)^I$, since $I \models r^{I_L}$ by (2) of Lemma 1, $I \models r^I$. As a result, I is a model for $Ground(P)^I$. Furthermore, no $J \subset I$ is a model for $Ground(P)^I$ because J would be also a model for $Ground(P)^{I_L}$. In fact, for each $r^I \in Ground(P)^I$ from (2) of Lemma 1, $J \models r^{I_L}$. For each $r \in Ground(P)$ s.t. no r^I exists, from Observation 1 $I \not\models B(r)$, therefore a disjunction $D \in B(r)$ exists s.t. $I \not\models D$ iff $I \not\models l$ for all $l \in D$. The corresponding disjunction $D^{I_L} \in B(r^{I_L})$ contains the same atoms of D and D^{I_L} contains \perp for each negative literal of D . Consequently $J \not\models D^{I_L}$ for all $J \subseteq I$ therefore $J \not\models B(r^{I_L})$ and $J \not\models r^{I_L}$.

3.2 Safety and Domain Independence

Let us review the definition of domain independence, stating in our case that the semantics should be independent of the universe, as long as it is large enough.

Definition 5. *Let P be an NFN program and U_P be the set of constants appearing in P . P is domain independent if for each $U \supseteq U_P$:*

$$AS(Ground(P, U)) = AS(Ground(P, U_P)).$$

Let us examine some examples related to domain independence.

Example 8. Consider the following program P_{us}

$$c(1). \quad d(2). \quad a(X) \vee b(Y) :- (c(X) \vee d(Y)).$$

where $U_{P_{us}} = \{1, 2\}$. Then $Ground(P_{us}, U_{P_{us}})$ is the following program:

$$\begin{array}{ll} c(1). & d(2). \\ a(1) \vee b(1) :- (c(1) \vee d(1)). & a(1) \vee b(2) :- (c(1) \vee d(2)). \\ a(2) \vee b(1) :- (c(2) \vee d(1)). & a(2) \vee b(2) :- (c(2) \vee d(2)). \end{array}$$

The answer sets are: $AS(\text{Ground}(P_{us}, U_{P_{us}})) = \{\{a(1), a(2), c(1), d(2)\}, \{b(1), b(2), c(1), d(2)\}, \{a(1), b(2), c(1), d(2)\}\}$. Now we consider $U = U_{P_{us}} \cup \{3\}$. Then we obtain $\text{Ground}(P_{us}, U)$ as the previous program union the following rules:

$$a(1) \vee b(3) :- (c(1) \vee d(3)). \quad a(2) \vee b(3) :- (c(2) \vee d(3)). \\ a(3) \vee b(3) :- (c(3) \vee d(3)). \quad a(3) \vee b(1) :- (c(3) \vee d(1)). \quad a(3) \vee b(2) :- (c(3) \vee d(2)).$$

In this case we have the following answer sets: $AS(\text{Ground}(P_{us}, U)) = \{\{a(1), a(2), a(3), c(1), d(2)\}, \{b(1), b(2), b(3), c(1), d(2)\}, \{a(1), b(2), c(1), d(2)\}\}$. We obtain different answer sets for the program P_{us} , depending on the considered universe. The problem is that in the main of P_{us} the body can be satisfied without “binding” one of the two variables that occur in the head.

Example 9. Consider the program P_{us2} :

$$c(1). \quad a :- (b(X) \vee \mathbf{not} c(X)).$$

where $U_{P_{us2}} = \{1\}$. Then $\text{Ground}(P_{us2}, U_{P_{us2}})$ is the following program:

$$c(1). \quad a :- (b(1) \vee \mathbf{not} c(1)).$$

so the program has the only one answer set:

$$AS(\text{Ground}(P_{us2}, U_{P_{us2}})) = \{c(1)\}.$$

Now if $U_2 = U_{P_{us2}} \cup \{2\}$, $\text{Ground}(P_{us2}, U_2)$ is the previous program union the following rule:

$$a :- (b(2) \vee \mathbf{not} c(2)).$$

So we have the following answer set:

$$AS(\text{Ground}(P_{us2}, U_2)) = \{a, c(1)\}.$$

In this case the variable in the negative literal is not necessarily “bound” when the body is true.

Theorem 2. *Given an NFN program P , P is in general not domain independent.*

Proof. Shown by Examples 8 and 9.

Safety

Definition 6 (Safe variable). *Let r be an NFN rule. A variable $X \in \text{vars}(r)$ is safe if there exists a positive basic disjunction $D \in B(r)$, such that $\forall a \in D, X \in \text{vars}(a)$.*

A variable X is saved or made safe by a basic disjunction D , if D is positive and $\forall a \in D, X \in \text{vars}(a)$. $\text{safeVars}(D)$ denotes the set of variables that are made safe by a basic disjunction D .

Example 10. Consider the following rule:

$$a :- (b(X) \vee c(X) \vee d(X)), e(Y), (s(Z) \vee t(X)).$$

The safe variables of the rule are X and Y . Indeed, the variable X is safe because it appears in the positive basic disjunction $(b(X) \vee c(X) \vee d(X))$ while the variable Y occurs in the positive basic disjunction $e(Y)$.

Definition 7 (Safe rule). An NFN rule r is safe if each variable that appears in the head of r and each variable that appears in negative body literals of r is safe.

Example 11. Consider the following NFN rules:

$$ok :- (a(X) \vee b(X)), \mathbf{not} c(X).$$

$$(ok(Z), ok1(X)) :- (a(X, Z) \vee b(X)), c(Z).$$

$$(ok1(X), ok2(X)) :- (a(X) \vee b(Z)), (c(X) \vee \mathbf{not} s(Z)).$$

The first rule is safe. In fact, the variable X , which appears in the negative literal $\mathbf{not} c(X)$, occurs in all atoms of the positive basic disjunction $(a(X) \vee b(X))$.

The second rule is safe since the variables of the head, X and Z , appear in all atoms of the first and second positive basic disjunction of the rule body, respectively. The last rule is unsafe, in fact the variable X occurs in the head but there is no positive basic disjunction in the body, in which X occurs in each atom. Moreover, the variable Z , occurring in the negative body literal $\mathbf{not} s(Z)$ is also unsafe.

An NFN program P is safe if each rule is safe.

Lemma 2. Let r be a safe NFN rule, I a set of ground atom, $U \supseteq \text{const}(I)$ and $U' \supset U$, then

$$I \models \text{Ground}(r, U) \Rightarrow I \models \text{Ground}(r, U') \quad (3)$$

Proof. Assume $I \not\models \text{Ground}(r, U')$ then a substitution $\sigma : \text{vars}(r) \rightarrow U'$ exists s.t. $I \not\models r\sigma$, then

$$I \models B(r\sigma) \quad \text{and} \quad I \not\models H(r\sigma). \quad (4)$$

Since $I \models B(r\sigma)$, then for each positive basic disjunction $D_p \in B^+(r)$ an atom $a \in D_p$ exists s.t. $a\sigma \in I$ so $\text{const}(a\sigma) \subseteq U$. Moreover, for each negative basic disjunction $D_n \in B^-(r)$ a literal $l \in D_n$ exists s.t. $I \models l\sigma$. Therefore, if l is positive, $l \in I$ and $\text{const}(l\sigma) \subseteq U$; if $l = \mathbf{not} a$, $a\sigma \notin I$ and since r is safe, for all $X \in \text{vars}(a)$ a positive disjunction $D_p \in B^+(r)$ exists s.t. for all $\bar{a} \in D_p$, $X \in \text{vars}(\bar{a})$ and thus $\text{const}(l\sigma) \subseteq U$. Then, a substitution $\sigma' : \text{vars}(r) \mapsto U$ as follows, exists

- $\forall X/c \in \sigma$ s.t. $c \in (U' \setminus U) \exists s \in U$ s.t. $X/s \in \sigma'$;
- $\forall X/c \in \sigma$ s.t. $c \in U$, $X/c \in \sigma'$.

From previous considerations it holds that $I \models B(r\sigma')$ and $I \models H(r\sigma')$ because $r\sigma' \in \text{Ground}(P, U)$. Furthermore, r is safe so for all $X \in \text{vars}(H(r))$ a positive disjunction $D_p \in B^+(r)$ exists s.t. for all $\bar{a} \in D_p$, $X \in \text{vars}(\bar{a})$ and thus $\text{const}(H(r\sigma)) \subseteq U$, hence $H(r\sigma') = H(r\sigma)$. Consequently we obtain a contradiction with (4).

Lemma 3. Let r be a safe NFN rule, I is a set of ground atoms and $U \supseteq \text{const}(I)$

$$\text{Ground}(r, U)^I = \text{Ground}(r, U')^I \quad \forall U' \supset U. \quad (5)$$

Proof. Since $U' \supset U$ from the definition of grounding it holds that $\text{Ground}(r, U) \subseteq \text{Ground}(r, U')$ and therefore, $\text{Ground}(r, U)^I \subseteq \text{Ground}(r, U')^I$. Let $\bar{r} \in \text{Ground}(r, U')^I$ and assume $\bar{r} \notin \text{Ground}(r, U)^I$ then there exists a literal $l \in B(\bar{r})$ s.t. $I \models l$ (from Definition 1) and $\text{const}(l) \cap (U' \setminus U) \neq \emptyset$. If l is positive, $l \in I$ and $\text{const}(l) \subseteq U$. If $l = \text{not } a$ then $a \notin I$ but since r is safe, for all $X \in \text{vars}(a)$ a positive disjunction $D_p \in B^+(r)$ exists s.t. for all $\bar{a} \in D_p$, $X \in \text{vars}(\bar{a})$ and thus $\text{const}(l) \subseteq U$. So we have a contradiction with the hypothesis $\text{const}(l) \cap (U' \setminus U) \neq \emptyset$.

Theorem 3. If P is safe then P is Domain Independent.

Proof. We split the proof in two parts.

1. If P is safe, $\forall U \supseteq U_P, \Rightarrow$

$$AS(\text{Ground}(P, U_P)) \subseteq AS(\text{Ground}(P, U)). \quad (6)$$

Let P be a safe program and assume that $I \in AS(\text{Ground}(P, U_P))$ and $\exists U \supset U_P$ s.t. $I \notin AS(\text{Ground}(P, U))$.

- If I is not a model of $\text{Ground}(P, U) \Rightarrow \exists r \in P$ and a substitution $\sigma : \text{vars}(r) \mapsto U$ s.t. $I \not\models r\sigma, r\sigma \in \text{Ground}(P, U)$. Since $I \models \text{Ground}(P, U_P) = \bigcup_{r' \in P} \text{Ground}(r', U_P)$, for each $r' \in P$ $I \models \text{Ground}(r', U_P)$ and from Lemma 2, $I \models \text{Ground}(r', U)$, $\forall r' \in P$. Then $I \models r\sigma$ for each $r\sigma \in \text{Ground}(P, U)$ and we obtain a contradiction.
 - If I is a model for $\text{Ground}(P, U)$ but I is not a minimal model for $\text{Ground}(P, U)^I$, then $\exists J \subset I$ s.t. J is a model for $\text{Ground}(P, U)^I$. Since $\text{Ground}(P, U_P) \subset \text{Ground}(P, U)$, then $(\text{Ground}(P, U_P)^I) \subseteq (\text{Ground}(P, U)^I)$ and J is a model for $\text{Ground}(P, U_P)^I$ contradicting the hypothesis $I \in AS(\text{Ground}(P, U_P))$.
2. If P is safe, $\forall U \supseteq U_P, \Rightarrow$

$$AS(\text{Ground}(P, U)) \subseteq AS(\text{Ground}(P, U_P)). \quad (7)$$

Let P be a safe program and we assume that $I \in \text{Ground}(P, U)$ and $\exists U \supset U_P$ s.t. $I \notin AS(\text{Ground}(P, U_P))$.

- Since $\text{Ground}(P, U_P) \subset \text{Ground}(P, U)$ then I is a model for $\text{Ground}(P, U_P)$.
- If I is a model for $\text{Ground}(P, U_P)$ but I is not a minimal model for $\text{Ground}(P, U_P)^I$ then there exists $J \subset I$ s. t. $J \in AS(\text{Ground}(P, U_P)^I)$. By Lemma 3 it holds that $\text{Ground}(P, U_P)^I = \text{Ground}(P, U)^I$ then $J \models \text{Ground}(P, U)^I$ and this is a contradiction to the hypothesis that $I \in AS(\text{Ground}(P, U))$.

4 An Efficient Translation from NFN to DLP

Results in [1] show that ground nested logic programs can be transformed into ground standard disjunctive logic programs. This allows for evaluating nested logic programs

using a disjunctive logic programming system, such as DLV [5], GnT [6], or Cmodels3 [7], as a back-end. However, the naïve transformation will in general produce very large programs, which may be exponentially larger than the input program. Inspired by structure-preserving normal form translations [10], a much more efficient transformation has been described in [8], which is guaranteed to run in polynomial time and will also provide transformed programs of polynomial size with respect to the input program. However, also the latter transformation works for ground programs only.

Unfortunately, a generalization of these techniques to programs with variables is not straightforward. A major obstacle is the requirement of *DLP* systems that *DLP* rules must be safe, that is each variable in a rule must occur in a positive body literal. When one would just add variables to the method of [8], one easily obtains unsafe rules. Indeed, if one just introduced a new predicate that replaces basic disjunctions in rule bodies and adds all variables of the basic disjunction that also occur elsewhere in the rule, the defining rules for the new predicate would in general not be safe. For instance, if we rewrite the rule $ok :- (b(X, Z) \vee c(X, Y)), (d(Z) \vee e(Y))$, the first disjunction $(b(X, Z) \vee c(X, Y))$ of the rule could be replaced by an auxiliary predicate l and we would need two defining rules for l , $l(X, Y, Z) :- b(X, Z)$. and $l(X, Y, Z) :- c(X, Y)$. Both of these rules are unsafe. However, the unsafe variables Y and Z must in some way be included in the atom for l , as in the rewritten rule other occurrences of Y and Z would otherwise be decoupled from the occurrences in the rewritten disjunction.

In this section we present an algorithm which efficiently translates normal form nested programs, which may contain variables, to disjunctive logic programs, elaborating on some ideas of [8].

Algorithm *rewriteNFNprogram*, shown in Fig. 1, takes as input a safe *NFN* program P and it returns a safe standard *DLP* program, P_{DLP} . The overall structure of the algorithm can be described as follows. For each safe rule $r \in P$ a corresponding standard rule, r_{DLP} , is built by means of functions *rewriteHead* and *rewriteBody* (described in the sequel).

Function *rewriteHead* builds $H(r_{DLP})$ by replacing each basic conjunction of $H(r)$ with a new atom. Moreover, it adds new standard rules to P_{DLP} , which are needed to correlate each basic conjunction with the corresponding new atom. Similarly, function *rewriteBody* introduces a new atom for each basic disjunction. Then, it builds $B(r_{DLP})$ as the conjunction of all new atoms. Finally, it adds a number of standard rules to P_{DLP} both for correlating each basic disjunction with the corresponding new atom and to preserve the join of the variables with the disjunctions. Functions *rewriteHead* and *rewriteBody* create new atoms for each *NFN* rule. These atoms have distinct predicate names for distinct rules.

Example 12. Let P the *NFN* program:

$$r1 : \quad (a(X), b(Y)) \vee c(Y) :- f(Y), (g(X) \vee h(X, Z)), (s(Z) \vee \mathbf{not} t(X)).$$

The corresponding program P_{DLP} , produced by algorithm *rewriteNFNprogram*, is the following:

$$\begin{aligned}
&auxh_1^{r_1}(X, Y) \vee c(Y) :- f(Y), aux_1^{r_1}(X, Z_1), aux_2^{r_1}(Z_2), match_Z^{r_1}(Z_1, Z_2, Z_{12}). \\
&auxh_1^{r_1}(X, Y) :- a(X), b(Y). \quad a(X) :- auxh_1^{r_1}(X, Y). \quad b(Y) :- auxh_1^{r_1}(X, Y). \\
&aux_1^{r_1}(X, \#unRestr) :- g(X). \quad aux_1^{r_1}(X, Z) :- h(X, Z). \\
&aux_2^{r_1}(Z) :- s(Z). \quad aux_2^{r_1}(\#unRestr) :- \mathbf{not} t(X), pred_X^{r_1}(X). \\
&match_Z^{r_1}(Z, Z, Z) :- univ_Z^{r_1}(Z). \quad match_Z^{r_1}(\#unRestr, Z, Z) :- univ_Z^{r_1}(Z). \\
&match_Z^{r_1}(Z, \#unRestr, Z) :- univ_Z^{r_1}(Z). \\
&match_Z^{r_1}(\#unRestr, \#unRestr, \#unRestr). \\
&univ_Z^{r_1}(Z) :- h(X, Z). \quad univ_Z^{r_1}(Z) :- s(Z). \\
&pred_X^{r_1}(X) :- aux_1^{r_1}(X, Z).
\end{aligned}$$

The algorithm possesses the following properties.

Proposition 1. *Let P an NFN program and P_{DLP} the DLP program obtained from P by algorithm *rewriteNFNprogram* and let \mathcal{A}_N and \mathcal{A}_D be the sets of predicate symbols that appear in P and in P_{DLP} , respectively ($\mathcal{A}_N \subseteq \mathcal{A}_D$). Then $AS(P) = \{I \cap \mathcal{A}_N \mid I \in AS(P_{DLP})\}$.*

Proof (Sketch). For each answer set I of P we can build a set of atoms $I_D \subseteq \mathcal{A}_D$ adding to I corresponding *aux*, *pred*, *univ* and *match* atoms for each basic disjunction and basic conjunction appearing in $Ground(P)$ true w.r.t. I . The new set is an answer set of P_{DLP} .

For each NFN rule $r \in P$, the standard program P_{DLP} contains a standard rule corresponding to r and several auxiliary rules. We can show that these auxiliary rules guarantee that each answer set of P_{DLP} satisfies $Ground(P)$. Consequently, after deleting all atoms $\mathcal{A}_D \setminus \mathcal{A}_N$ contained in an answer set of P_{DLP} , the remaining set is a model of P and we can furthermore prove that the set is actually an answer set of P .

Proposition 2. *For all NFN programs P_1, P_2 , $rewriteNFNprogram(P_1 \cup P_2) = rewriteNFNprogram(P_1) \cup rewriteNFNprogram(P_2)$.*

Proposition 3. *Let P be an NFN program. Computing $rewriteNFNprogram(P)$ takes polynomial time in the size of P .*

Function *rewriteNFNprogram*

Input: NFN program P

Output: DLP program P_{DLP} .

var *bodyRule* conjunction of literals; *headRule* disjunction of atoms.

1. **for each** $r \in P$ **do**
2. $headRule := rewriteHead(r)$; $bodyRule := rewriteBody(r)$;
3. $P_{DLP} += \{headRule :- bodyRule.\}$;
4. **return** P_{DLP} ;

Fig. 1. Algorithm: *rewriteNFNprogram*

Function *rewriteHead*

Input: *NFN* rule r

Output: disjunction of atoms *headRule*;

1. **for each** $C \in H(r)$ **do**
2. **if** C has one atom a
3. $headRule = headRule \vee a$;
4. **else**
5. $vars_C := \{\}$; $bodyAux_C^r := \{\}$;
6. **for each** $a \in C$ **do**
7. $bodyAux_C^r = bodyAux_C^r, a$; $vars_C += vars(a)$;
8. **for each** $a \in C$ **do**
9. $P_{DLP} += \{a :- aux_C^r(vars_C).\}$;
10. $P_{DLP} += \{aux_C^r(vars_C) :- bodyAux_C^r.\}$;
11. $headRule := headRule \vee aux_C^r(vars_C)$;
12. **return** *headRule*;

Fig. 2. Function: *rewriteHead*

4.1 The *rewriteHead* function

Function *rewriteHead*, shown in Fig. 2, receives a safe *NFN* rule r as input. For each basic conjunction $C \in H(r)$, if C contains strictly more than one atom, *rewriteHead* creates an auxiliary atom, with predicate name aux_C^r that substitutes C in $H(r_{DLP})$. $vars(aux_C^r)$ is the set of variables that appear in all atoms of C .¹ Furthermore, this function adds new rules to P_{DLP} which relate the atoms of C to aux_C^r . If C contains only one atom, aux_C^r is not built, and the atom is directly included in the resulting rule head.

4.2 The *rewriteBody* function

Function *rewriteBody*, shown in Fig. 3, takes as input an *NFN* rule r , and is more complex than *rewriteHead*. For each basic disjunction $D \in B(r)$, containing strictly more than one literal, the function builds a new auxiliary atom with predicate name aux_D^r and returns the conjunction of the new atoms, $B(r_{DLP})$. To compute the variables of each atom aux_D^r , the function uses the following definitions.

Definition 8 (Unrestricted Shared Variables). Let D a basic disjunction of a safe *NFN* rule r . A variable $X \in vars(D)$ is *unrestricted shared* if X is not a safe variable of r and there exists a disjunction $D' \neq D$ s.t. $X \in vars(D')$.

Definition 9 (Shared Variables). Let D be a basic disjunction of a safe *NFN* rule r . A variable $X \in vars(D)$ is *shared* if X is a safe variable of r but X is not made safe by D (i.e. $X \notin safeVars(D)$).

Suppose that for each $D \in B(r)$ the sets $safeVars_D$, $unrestrictedSharedVars_D$, and $sharedVars_D$ have been computed. If $safeVars_D \neq \emptyset$ then the auxiliary atom is $aux_D^r(safeVars_D, sharedVars_D, unrestrictedSharedVars_D)$, otherwise it is just $aux_D^r(unrestrictedSharedVars_D)$.

¹ Note that this set is then used as a sequence when building the atom; we use the set notation only for simplicity.

Function *rewriteBody*

Input: NFN rule r

Output: conjunction of literals *bodyRule*

var $PRED$, AUX_{safe} , AUX_u , $MATCH$ and $UNIV$ **vectors** of atoms; *bodyRule* and *bodyAux* conjunction of literals;

1. $bodyRule := \{ \}$;
2. **for each** $D \in B(r)$ **do**
3. **if** D has one literal l **then**
4. **if** l is positive **then** $AUX_{safe} += l$;
5. $bodyRule = bodyRule, l$;
6. **else for each** $l \in D$ **do**
7. $bodyAux := \{ \}$; $varAux := \{ \}$;
8. **if** $safeVars_D \neq \emptyset$ **then**
9. **for each** $X \in sharedVars_D$ **do**
10. **if** $X \in l$ **then** $varAux += X$; **else** $varAux += \#unRestr$;
11. **else if** $safeVars_D = \emptyset$ **then**
12. **for each** variable $X \in l$ **do**
13. **if** $X \in sharedVars_D$ **then**
14. $PRED += pred_X^r(X)$; $bodyAux = bodyAux, pred_X^r(X)$;
15. **for each** $X \in unrestrictedSharedVars_D$ **do**
16. **if** $X \in l$ **then**
17. $P_{DLP} += \{ univ_X^r(X) :- l. \}$; $UNIV += univ_X^r(X)$;
18. $varAux += X$;
19. **else** $varAux += \#unRestr$;
20. $P_{DLP} += \{ aux_D^r(safeVars_D \cup varAux) :- l, bodyAux. \}$;
21. **if** $varSafe_D \neq \emptyset$ **then**
22. $V = safeVars_D \cup sharedVars_D \cup unrestrictedSharedVars_D$;
23. **if** $sharedVars_D \neq \emptyset$ **then**
24. $V' = renameShared(D, PRED, bodyRule)$;
25. $V = safeVars_D \cup V' \cup unrestrictedSharedVars_D$;
26. **if** $unrestrictedSharedVars_D \neq \emptyset$ **then** $AUX_u += aux_D^r(V)$;
27. **else** $bodyRule = bodyRule, aux_D^r(V)$; $AUX_{safe} += aux_D^r(V)$;
28. **else if** $varSafe = \emptyset$ **then**
29. **if** $unrestrictedSharedVars_D \neq \emptyset$ **then**
30. $AUX_u += aux_D^r(unrestrictedSharedVars_D)$;
31. **else** $bodyRule = bodyRule, aux_D^r$;
32. $insertPredRules(PRED, AUX_{safe})$;
33. $MATCH = renameUnrestrictedShared(UNIV, AUX_u)$;
34. **for each** $aux \in AUX_u$ **do** $bodyRule = bodyRule, aux$;
35. **for each** $match \in MATCH$ **do** $bodyRule += match$;
36. **return** $bodyRule$;

Fig. 3. Function: *rewriteBody*

Function *rewriteBody* also creates rules that define the auxiliary predicates. For each literal l of a basic disjunction, the function builds a *DLP* rule, r_l . The head of r_l is formed by the respective auxiliary atom, placing the special constant $\#unRestr$ in place of variables that do not occur in l , indicating that the corresponding variable is not bound. The body of r_l consists of l and an atom $pred_X^r(X)$ for each variable

in a negative l for safety. For each $pred_X^r$ atoms a defining rule is eventually added by means of Function $insertPredRules$ depicted in Fig. 6, the body of which is the conjunction of all auxiliary atoms for basic disjunctions of r that save X .

If the original rule body contained shared variables, these variables may not be bound by some basic disjunctions, but for those that do bind them, it must be guaranteed that they bind them to equal values. We have introduced the special constant $\#unRestr$ in the definition of the auxiliary predicates to this end. But obviously this constant cannot be equal to any value that might be bound to the variable from another basic disjunction. To this end, we have to make the matching explicit, such that it can tolerate the presence of $\#unRestr$, and this is the purpose of the *match* predicates.

Example 13. Let P be the *NFN* program consisting only of the rule

$$: \quad ok :- (b(X, Y) \vee a(X)), (c(Y, X) \vee d(Y)).$$

Neglecting the *match* predicates, the program P_{DLP} would be:

$$\begin{aligned} aux_1^r(X, Y) &:- b(X, Y). & aux_1^r(X, \#unRestr) &:- a(X). \\ aux_2^r(Y, X) &:- c(Y, X). & aux_2^r(Y, \#unRestr) &:- d(Y). \end{aligned}$$

$$r_{DLP} : \quad ok :- aux_1^r(X, Y), aux_2^r(Y, X).$$

If we add $\{a(1), c(2, 1)\}$ to P and P_{DLP} then $I = \{ok, a(1), c(2, 1)\}$ is the only answer set for P and $I_D = \{a(1), c(2, 1), aux_1^r(1, \#unRestr), aux_2^r(2, 1)\}$ is the answer set of P_{DLP} , missing *ok*.

The reason is that the constants $\#unRestr$ and 2 do not match. To overcome this, we modify rule r_{DLP} by adding a new atom in $B(r_{DLP})$, obtaining

$$r_{DLP} : \quad ok :- aux_1^r(X, Y1), aux_2^r(Y2, X), match_Y^r(Y1, Y2, Y12).$$

Furthermore, we add to P_{DLP} some additional rules:

$$\begin{aligned} match_Y^r(\#unRestr, Y, Y) &:- univ_Y^r(Y). & match_Y^r(Y, \#unRestr, Y) &:- univ_Y^r(Y). \\ match_Y^r(Y, Y, Y) &:- univ_Y^r(Y). & match_Y^r(\#unRestr, \#unRestr, \#unRestr) &:- univ_Y^r(Y). \\ univ_Y^r(Y) &:- b(X, Y). & univ_Y^r(Y) &:- c(Y, X). \end{aligned}$$

The answer set of the modified P_{DLP} is the set $\{a(1), c(2, 1), aux_1^r(1, \#unRestr), aux_2^r(2, 1), univ_Y^r(2), match_Y^r(\#unRestr, 2, 2), match_Y^r(2, \#unRestr, 2), match_Y^r(2, 2, 2), match_Y^r(\#unRestr, \#unRestr, \#unRestr), ok\}$.

The code in Figures 4, 5, 6 generalizes and optimizes the idea shown in Example 13.

5 Conclusion

We have introduced *NFN* programs, an extension of nonground disjunctive logic programs, where conjunctions of atoms and disjunctions of literals are permitted in the heads and in the bodies of the rules, respectively. We have defined syntax and semantics of the new language and, since ground *NFN* programs are *NLP* programs, we

Figure *renameShared*

Input: basic disjunction D , **vector** of atoms $PRED$, basic conjunction $bodyRule$

Output: set of variables V ;

1. **for each** $X \in varShared_D$ **do**
2. $V+ = X_D$; $PRED+ = pred_X^r(X)$;
3. $P_{DLP}+ = \{match_X^r(X, \#unRestr) :- pred_X^r(X).\}$;
4. $P_{DLP}+ = \{match_X^r(X, X) :- pred_X^r(X).\}$;
5. $bodyRule = bodyRule, match_X^r(X, X_D)$;
6. **return** V ;

Fig. 4. Function *renameShared*

Function *insertPredRules*

Input: **vectors** of atoms $PRED$, AUX_{safe}

1. **for each** $pred(X) \in PRED$ **do**
2. $body = \{\}$;
3. **for each** $aux \in AUX_{safe}$ **do**
4. **if** $X \in vars(aux)$ **then**
5. $body+ = aux$;
6. $P_{DLP}+ = \{pred(X) :- body.\}$;
7. **return**;

Fig. 5. Function *insertPredRules*

Procedure *renameUnrestrictedShared*

Input: **vectors** of atoms $UNIV$, AUX_u

1. **for each** $aux_i \in AUX_u$ $i = 1, \dots, length(AUX_u) - 1$ **do**
2. **for each** $X \in aux_i$ **do**
3. **if** X is not a safe variable of r **then**
4. $X_{corr} = X$;
5. **for each** $aux_j \in AUX_u$ $j = i, \dots, length(AUX_u)$ **do**
6. **if** $X \in aux_j$ **then**
7. $MATCH+ = match_X^r(X_{corr}, X_j, X_{ij})$;
8. $X_{corr} = X_{ij}$;
9. substitute X by X_j in $vars(aux_j)$;
10. **for each** $univ \in UNIV$ **do**
11. **if** $X \in univ$ **then**
12. $P_{DLP}+ = \{match_X^r(X, \#unRestr, X) :- univ(X).\}$;
13. $P_{DLP}+ = \{match_X^r(\#unRestr, X, X) :- univ(X).\}$;
14. $P_{DLP}+ = \{match_X^r(X, X, X) :- univ(X).\}$;
15. $P_{DLP}+ = \{match_X^r(\#unRestr, \#unRestr, \#unRestr).\}$;
16. **return**;

Fig. 6. Procedure *renameUnrestrictedShared*

showed that the semantics coincide on this fragment. Furthermore, we have defined the class of safe *NFN* programs and showed that each program of this class is domain independent, that is, has the same answer sets on all universe containing the constants of the program. Finally, we have developed an efficient algorithm that rewrites safe *NFN* programs into safe *DLP* programs. Ongoing work concerns the implementation of the presented algorithm, allowing for the computation of answer sets for *NFN* programs by exploiting disjunctive ASP systems as back-ends.

References

1. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. *AMAI* **25**(3–4) (1999) 369–389
2. Przymusiński, T.C.: Stable Semantics for Disjunctive Programs. *NGC* **9** (1991) 401–424
3. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *NGC* **9** (1991) 365–385
4. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
5. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* **7**(3) (2006) 499–562
6. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: *LPNMR-7*. LNCS 2923, (2004) 331–335
7. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: *LPNMR'05*. LNCS 3662, (2005) 447–451
8. Pearce, D., Sarsakov, V., Schaub, T., Tompits, H., Woltran, S.: A Polynomial Translation of Logic Programs with Nested Expressions into Disjunctive Logic Programs: Preliminary Report. In: *ICLP 2002*. LNCS 2401, (2002) 405–420
9. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: *JELIA 2004*. LNCS 3229, (2004) 200–212
10. Plaisted, D.A., Greenbaum, S.: A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation* **2**(3) (1986) 293–304

GASP: Answer Set Programming with Lazy Grounding

A. Dal Palù¹, A. Dovier², E. Pontelli³, and G. Rossi¹

¹ Dip. Matematica, Univ. Parma,

{alessandro.dalpalu|gianfranco.rossi}@unipr.it

² Dip. Matematica e Informatica, Univ. Udine, dovier@dimi.uniud.it

³ Dept. Computer Science, New Mexico State Univ., epontell@cs.nmsu.edu

Abstract. In this paper we present a novel methodology to compute stable models in Answer Set Programming. The process is performed with a bottom-up approach that does not require the preprocessing of the typical grounding phase. The implementation is completely in Prolog and Constraint Logic Programming over finite domains. The code is very simple and can be used for didactic purposes.

1 Introduction

In recent years, we have witnessed a significant increase of interest towards the *Answer Set Programming* (briefly, ASP) paradigm [13, 14]. The growth of the field has been sparked by two essential activities:

- The development of effective implementations (e.g., SMODELS [15], DLV [6], CMODELS [1], CLASP [7])
- The continuous development of *knowledge building blocks* (e.g., [2]) enabling the application of ASP to various problem domains.

The majority of ASP systems rely on a two-stage computation model. The actual computation of the answer set is performed only on propositional programs—either directly (as in SMODELS and DLV and CLASP) or appealing to the use of a SAT solver (as in ASSAT and CMODELS). On the other hand, the convenience of ASP programming vitally builds on the use of first-order constructs. This introduces the need of a *grounding* phase, typically performed by a separate grounding program (e.g., LPARSE or GRINGO, or the grounding module of DLV).

The development of sophisticated applications of ASP in real-world domains (e.g., planning [10], phylogenetic inference [3]) has highlighted the strengths and weaknesses of this paradigm. The high expressive power enables the compact and elegant encoding of complex forms of knowledge (e.g., common-sense knowledge, defaults). At the same time, the technology underlying the execution of ASP is still lagging behind and it is often unable to keep up with the demand of complex applications. This has been, for example, highlighted in a recent study of use of ASP to address complex planning problems (drawn from the recent international planning competitions) [17]. A problem like Pipeline (from International

Planning Competition n. 5, IPC-5), whose first 9 instances can be effectively solved by state-of-the-art planners like FF [9], can be solved only for instance 1 using ASP. Using SMODELS, instances 2 through 4 do not terminate within several hours of execution, while instance 5 generates a ground image that is beyond the input capabilities of SMODELS.

In this manuscript, we propose a novel implementation of ASP—hereafter named *GASP* (*Grounding-lazy ASP*). The spirit of our effort can be summarized as follows:

- The execution model relies on a novel bottom-up scheme;
- The bottom-up execution model does not require preliminary grounding of the program;
- The internal representation of the program and the computation make use of constraint logic programming over finite domains [4].

This combination of ideas provides a novel system with significant potentials. In particular:

- It enables the simple integration of new features in the solver, such as constraints and aggregates. If preliminary grounding was required, these features would have to be encoded as ground programs, thus reducing the capability to devise general strategies to optimize the search, and often further growing the size of the ground program.
- The adoption of a non-ground search allows the system to control the search process effectively at a higher level, enabling the adoption of Prolog-level implementations of search strategies and the use of static analysis techniques. While in theorem proving-based ASP solvers the search is driven by literals (i.e., the branching in the search tree is generated by alternatively trying to prove p and **not** p), here the search is “*rule-driven*” in the sense that an applicable rule (possibly not ground) is selected and applied.
- It reduces the impact of grounding the whole program before execution, as observed in some domains (e.g., [17]). Grounding is lazily applied to the rules being considered during construction of an answer set, and the ground rules are not kept beyond their needed use.

Given a ASP program P , the key ingredients of the proposed system are:

1. an efficient implementation of the immediate consequence operator T_P (for definite and normal programs);
2. an implementation of an alternating fixpoint procedure for the computation of well-founded models of a (non-ground) program;
3. a mechanism for nondeterministic selection and application of a ground rule to a partial model.

GASP has been developed into a prototype, completely implemented in Prolog and available from <http://www.dimi.uniud.it/dovier/CLPASP>. For efficiency, the representation of predicates is mapped to *finite domain sets* (*FDSETS*), and

techniques are developed to implement a permutation-free search, which limits the risk of repeatedly reconstructing the same answer sets.

The prototype is aimed at demonstrating the feasibility of the proposed approach; at the current stage the system is slower than state-of-the-art ASP solvers. Performance improvements could be gained by using low level data structures that allow constant time w.r.t. linear time access to rules and models. Moreover, as shown in Section 6, our implementation can benefit from *Finite Domain (FD)* and *Finite Domain Set (FDSET)* constraint primitives to significantly speed-up the answer sets search. However, the code is rather short (less than 700 lines, including comments and auxiliary I/O predicates) and it can be used for pedagogical purposes at three levels: for the T_P computation of definite programs, for the computation of well-founded models, and, finally, for the computation of stable models. The first two levels have already performances comparable to other systems, in spite of the simplicity and compactness of the Prolog code.

2 Preliminaries

Let us consider a logic language composed of a collection of atoms \mathcal{A} . An ASP rule has the form:

$$p \leftarrow p_0, \dots, p_n, \mathbf{not} p_{n+1}, \dots, \mathbf{not} p_m$$

where $\{p, p_0, \dots, p_n, p_{n+1}, \dots, p_m\} \subseteq \mathcal{A}$. A program is a collection of ASP rules. Given a rule $a \leftarrow body$, let us denote with $body^+$ the collection of positive literals in $body$ and with $body^-$ the atoms that appear in negative literals in $body$. We also refer to a as $head(a \leftarrow body)$.

We view an interpretation I as a subset of the set of atoms $I \subseteq \mathcal{A}$. I satisfies an atom p if $p \in I$ (denoted by $I \models p$). The interpretation I satisfies the literal $\mathbf{not} p$ if $p \notin I$ (denoted by $I \models \mathbf{not} p$). The notion of entailment can be generalized to conjunctions of literals in the obvious way. An interpretation I is a *model* of a program P if for every rule $p \leftarrow p_0, \dots, p_n, \mathbf{not} p_{n+1}, \dots, \mathbf{not} p_m$ of P , if $I \models p_0 \wedge \dots \wedge p_n \wedge \mathbf{not} p_{n+1}, \dots, \mathbf{not} p_m$ then $I \models p$. We refer to these classical interpretations as to *2-interpretations*.

The traditional immediate consequence operator T_P is generalized to the case of ASP programs as follows:

$$T_P(I) = \{a \in \mathcal{A} \mid (a \leftarrow body) \in P, I \models body\} \quad (1)$$

Any program admits the trivial model \mathcal{A} (w.l.o.g., let us assume that $\mathcal{A} = B_P$, the Herbrand base of the program P). However, this model does not reflect the meaning of the program. It is widely accepted that the notion of *stable model* [8] is a necessary and sufficient condition for being an intended model of a program. A model I of P is stable if I is the least fixpoint of the operator T_{P^I} of the definite clause program P^I obtained adding to the definite clauses in P the

rules $p \leftarrow p_0, \dots, p_n$ such that $p \leftarrow p_0, \dots, p_n$, **not** p_{n+1}, \dots , **not** p_m is in P and $p_{n+1} \notin I, \dots, p_m \notin I$. Stable models are also known as *answer sets*.

Unfortunately, stating the existence of a stable model is NP-complete. When looking for stable models it appears clear (for some atoms) that they must be present in all stable models and (for some other atoms) that they cannot be present in any stable model. This suggested a 3-valued representation of interpretations.

A *3-interpretation* I is a pair $\langle I^+, I^- \rangle$ such that $I^+ \cup I^- \subseteq \mathcal{A}$ and $I^+ \cap I^- = \emptyset$. Intuitively, I^+ denotes the atoms that are known to be true while I^- denotes those atoms that are known to be false. Let us observe that it is not required that $I^+ \cup I^- = \mathcal{A}$. If $I^+ \cup I^- = \mathcal{A}$, then the interpretation I is said to be *complete*.

Given two 3-interpretations I, J , we introduce the relation $I \subseteq J$ to denote the fact that $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. The notion of entailment for 3-interpretations can be defined as follows. If $p \in \mathcal{A}$, then $I \models p$ iff $p \in I^+$; $I \models \mathbf{not} p$ iff $p \in I^-$.

The *well-founded model* [18] of a general program P is a 3-interpretation denoted as $\text{wf}(P)$. Intuitively, the well-founded model of P contains only (possibly not all) the literals that are necessarily true and the ones that are necessarily false in all stable models of P . The remaining literals are undefined, because they depend on loops of dependent literals in P and thus there is no unique interpretation for them. It is well-known that a general program P has a unique well-founded model $\text{wf}(P)$ [18]. If $\text{wf}(P)$ is complete then it is also a stable model (and it is the unique stable model of P).

A well-founded model can be computed deterministically using the idea of alternating fixpoint [19]. This technique uses pairs of 2-interpretations (denoted by I and J) for building the 3-interpretation $\text{wf}(P)$. The immediate consequence operator is extended, with the introduction of another interpretation J :

$$T_{P,J}(I) = \{a \in \mathcal{A} \mid (a \leftarrow bd^+, \mathbf{not} bd^-) \in P, I \models bd^+, (\forall p \in bd^-)(J \not\models p)\} \quad (2)$$

With this extension, the computation of the well-founded model of P is obtained as follows:

$$\begin{cases} K_0 = \text{lfp}(T_{P^+, \emptyset}) & U_0 = \text{lfp}(T_{P, K_0}) \\ K_i = \text{lfp}(T_{P, U_{i-1}}) & U_i = \text{lfp}(T_{P, K_i}) \quad i > 0 \end{cases}$$

where P^+ , used for computing K_0 , is the subset of P composed by definite clauses (facts and rules without negation in body).

When $(K_i, U_i) = (K_{i+1}, U_{i+1})$, the fixpoint is reached and the well-founded (possibly partial) model is the 3-interpretation:

$$\text{wf}(P) = \langle K_i, B_P \setminus U_i \rangle$$

where B_P is the Herbrand base of the program P .

3 Computation-based characterization of stable models

The computation model adopted in GASP has been derived from recent investigations about alternative models to characterize answer set semantics for various extensions of ASP—e.g., programs with *abstract constraint atoms* [16].

3.1 Computations and Answer Sets

The work described in [11] provides a *computation-based* characterization of answer sets for programs with abstract constraints. One of the side-effects of that research is the development of a computation-based view of answer sets for general logic programs. The original definition of answer sets [8] requires guessing an interpretation and successively validating it—through the notion of reduct (P^I) and the ability to compute minimal models of a definite program (e.g., via repeated iterations of the immediate consequence operator [12]).

The characterization of answer sets derived from [11] does not require the initial guessing of a complete interpretation; instead it combines the guessing process with the construction of the answer set.

Let us present this alternative characterization in the case of propositional programs.

Definition 1 (Computation). *A computation of a program P is a sequence of interpretations I_0, I_1, I_2, \dots that satisfies the following conditions:*

- $I_0 = \emptyset$
- $I_i \subseteq I_{i+1}$ for all $i \geq 0$ (Persistence of Beliefs)
- $\bigcup_{i=0}^{\infty} I_i$ is a model of P (Convergence)
- $I_{i+1} \subseteq T_P(I_i)$ for all $i \geq 0$ (Revision)
- if $a \in I_{i+1} \setminus I_i$ then there is a rule $a \leftarrow \text{body}$ in P such that $I_j \models \text{body}$ for each $j \geq i$ (Persistence of Reason).

The results presented in [11] imply the following theorem.

Theorem 1. *Given a program P and a 2-interpretation I , I is an answer set of P if and only if there exists a computation that converges to I .*

The notion of computation characterizes answer sets through an incremental construction process, where the choices are performed at the level of what rules are actually applied to extend the partial answer set.

3.2 A Refined View of Computation

This original notion of computation can be refined in various ways:

- The Persistence of Beliefs rule, together with the convergence rule, indicate that all elements that have a uniquely determined truth value at any stage of the computation can be safely added.

- The notion of computation can be made more specific by enabling the application of only one rule at each step (instead of an arbitrary subset of the applicable rules).

These two observations allow us to rephrase the notion of computation as follows, in the context of 3-interpretations. Given a rule $a \leftarrow \text{body}$ and an interpretation I , we say that the rule is *applicable* w.r.t. I if

$$\text{body}^+ \subseteq I^+ \text{ and } \text{body}^- \cap I^+ = \emptyset.$$

We extend the definition of applicable to a non ground rule R w.r.t. I iff there exists a grounding r of R that is applicable w.r.t. I .

For the sake of simplicity we also overload the \cup operation between interpretations. Given two interpretations I, J , we denote $I \cup J = \langle I^+ \cup J^+, I^- \cup J^- \rangle$. Additionally, given a program P and an interpretation I , we denote with $P \cup I$ the program

$$P \cup I = (P \setminus \{r \in P \mid \text{head}(r) \in I^-\}) \cup I^+.$$

Intuitively, $P \cup I$ is the program P modified in such a way to guarantee that all elements in I^+ are true and all elements of I^- are false.

Definition 2 (GASP Computation). *A GASP computation of a program P is a sequence of 3-interpretations I_0, I_1, I_2, \dots that satisfies the following properties:*

- $I_0 = \text{wf}(P)$
- $I_i \subseteq I_{i+1}$ (Persistence of Beliefs)
- if $I = \bigcup_{i=0}^{\infty} I_i$, then $\langle I^+, \mathcal{A} \setminus I^+ \rangle$ is a model of P (Convergence)
- for each $i \geq 0$ there exists a rule $a \leftarrow \text{body}$ in P that is applicable w.r.t. I_i and $I_{i+1} = \text{wf}(P \cup I_i \cup \langle \text{body}^+, \text{body}^- \rangle)$ (Revision)
- if $a \in I_{i+1} \setminus I_i$ then there is a rule $a \leftarrow \text{body}$ in P which is applicable w.r.t. I_j , for each $j \geq i$ (Persistence of Reason).

The following result holds:

Theorem 2. *Given a program P , a 3-interpretation I is an answer set of P if and only if there exists a GASP computation that converges to I .*

Proof Sketch. One direction is quite simple, by observing that the computations described in Def. 2 are special cases of the computations of Def. 1 (thus they produce answer sets). The other direction builds on the observation that each computation as in Def. 1 can be rewritten as a computation as in Def. 2, thanks to the Revision and Persistence of Reason properties. \square

4 Computing models using FDSETs

In this section we show how to encode and handle interpretations and answer sets in Prolog using FDSETs. In particular, we show how to represent an interpretation and how to compute the operator T_P and the well-founded model.

FDSETs are a data structure available in the `clpfd` library of SICStus Prolog that allows to efficiently store and compute on sets of integer numbers. Basically, a set $\{a_1, a_2, \dots, a_n\}$ is recognized as the union of a set of intervals $[a_{b_1}..a_{e_1}], \dots, [a_{b_k}..a_{e_k}]$ and stored consequently as $[[a_{b_1}|a_{e_1}], \dots, [a_{b_k}|a_{e_k}]]$. A library of built-in predicates for dealing with this data structure is made available.

4.1 Representation of Interpretations

Most existing front-ends to ASP systems allow the programmer to express programs using a first-order notation. Program atoms are expressed in the form $p(t_1, \dots, t_n)$, where each t_i is either a constant or a variable. Each rule represents a syntactic sugar for the collection of its ground instances. Languages like those supported by the SMODELS system impose syntactic restrictions to facilitate the grounding process and ensure finiteness of the collection of ground clauses. In particular, SMODELS requires each rule in the program to be *range restricted*, i.e., all variables in the rule should occur in *body*⁺. Furthermore, SMODELS requires all variables to appear in at least one atom built using a *domain predicate*—i.e., a predicate that is not recursively defined. Domain predicates play for variables the same role as types in traditional programming languages.⁴

In the scheme proposed here, the instances of a predicate that are true and false within an interpretation are encoded as sets of tuples, and handled using FD techniques.

We identify with p^n a predicate p with arity n . In the program, a predicate p^n appears as $p(X_1, \dots, X_n)$ where, in place of some variables, a constant can occur (e.g., $p(a, X, Y, d)$). The interpretation of the predicate p^n can be modeled as a set of tuples (a_1, a_2, \dots, a_n) , where $a_i \in \text{Consts}(P)$ —where $\text{Consts}(P)$ denotes the set of constants in the language used by the program P . The explicit representation of the set has the maximal cardinality $|\text{Consts}(P)|^n$. The idea is to use a more compact representation based on FDSETs, after a mapping of tuples to integers. Without loss of generality, we assume that $\text{Consts}(P) \subseteq \mathbb{N}$. Each tuple $\mathbf{a} = (a_0, \dots, a_{n-1})$ is mapped to the *unique* number $\text{map}(\mathbf{a}) = \sum_{i \in [0..n-1]} a_i \mathbb{M}^i$, where \mathbb{M} is a “big number”, $\mathbb{M} \geq |\text{Consts}(P)|$. We also extend the `map` function to the case of FD variables. In this case, if $X_i \in \text{Vars}(P)$ then $\text{map}(X)$ is a new FD variable constrained to be equal to the sum defined above, using FD operators. In case of predicates without arguments (predicates of arity 0), for the empty tuple $()$ we set $\text{map}() = 0$.

A 3-interpretation $\langle I^+, I^- \rangle$ can be represented by a set of 4-tuples

$$(p, n, \text{Pos}_{p,n}, \text{Neg}_{p,n}),$$

one for each predicate symbol, where p is the predicate name, n its arity, and

$$\begin{aligned} \text{Pos}_{p,n} &= \{\text{map}(\mathbf{x}) : I^+ \models p(\mathbf{x})\} \\ \text{Neg}_{p,n} &= \{\text{map}(\mathbf{x}) : I^- \models \text{not } p(\mathbf{x})\} \end{aligned}$$

⁴ Some of these restrictions have been relaxed in other systems, e.g., DLV.

The sets Pos and Neg are represented and handled efficiently, by using FD-SETS (see e.g., SICStus Prolog manual). For instance, if

$$Pos_{p,3} = \{\text{map}(0, 0, 1), \text{map}(0, 0, 2), \text{map}(0, 0, 3), \text{map}(0, 0, 8), \\ \text{map}(0, 0, 9), \text{map}(0, 1, 0), \text{map}(0, 1, 1), \text{map}(0, 1, 2)\}$$

and $M = 10$, then its representation as FDSET is simply: $[[1|3], [8|12]]$, in other words, the disjunction of two intervals.

4.2 Minimum model computation

```
(1) apply_def_rule(rule([H],PBody,NBody),I,[atom(F,AR,NEWPDOM,NEG)|PARTI]):-
(2)   copy_term([H,PBody,NBody],[H1,PBody1,NBody1]),
(3)   term_variables([H1,PBody1],VARS),
(4)   bigM(M),M1 is M-1,domain(VARS,O,M1),
(5)   build_constraint(PBody1,I,C3,pos),
(6)   build_constraint(NBody1,I,C4,negknown),
(7)   H1 =.. [F|ARGS],
(8)   build_arity(ARGS,VAR,AR),
(9)   select(atom(F,AR,OLD,NEG),I,PARTI),
(10)  nin_set(AR,VAR,OLD,C1),
(11)  nin_set(AR,VAR,NEG,C2),
(12)  findall(X,(C1+C2+C3+C4 #>= 4, X #= VAR,labeling([ff],VARS)),LIST),
(13)  list_to_fdset(LIST,SET),
(14)  fdset_union(OLD,SET,NEWPDOM).
(15) build_constraint([R|Rs],I,C,Sign):-
(16)   R =.. [F|ARGS],
(17)   builtin(F),!,
(18)   expression(F,ARGS,C2),
(19)   C #<=> C2 #/\ C1,
(20)   build_constraint(Rs,I,C1,Sign).
(21) build_constraint([R|Rs],I,C,Sign):-
(22)   R =.. [F|ARGS],!,
(23)   build_arity(ARGS,VAR,ARITY),
(24)   member(atom(F,ARITY,DOMF,NDOMF),I),
(25)   (Sign=neg,!, nin_set(ARITY,VAR,DOMF,C2);
(26)   Sign=pos,!, C2 #<=> VAR in_set DOMF;
(27)   Sign=negknown, C2 #<=> VAR in_set NDOMF ),
(28)   build_constraint(Rs,I,C1,Sign),
(29)   C #<=> C1 #/\ C2.
(30) build_constraint([],-,1,-).
(31) build_arity(ARGS,VAL,ARITY):-
(32)   (ARGS = [],!, VAL=0, ARITY=0;
(33)   ARGS = [VAL],!, ARITY=1;
(34)   ARGS = [Arg1,Arg2],!, bigM(M), ARITY=2, VAL #= M*Arg1+Arg2;
(35)   ARGS = [A1,A2,A3], bigM(M), ARITY=3, VAL #= M*A1+M*A2+A3).
```

Fig. 1. T_P computation

We start showing how the computation of the T_P (see equation (1)) can be implemented using finite domains and FDSET. Actually, the T_P implemented is slightly more complex than the one needed for definite programs, but it has the advantage to be used also during the answer set computation of general programs, when negative information is also known.

We define a simple fixpoint procedure that calls recursively `apply_def_rule` until the model is no longer modifiable. The definition of this predicate is reported in Figure 1. Observe in line 1 that `rule([H],PBody,NBody)` is the internal representation of the rule $H \leftarrow PBody, \text{not} NBody$, while `atom(F,AR,PDOM,NDOM)` is the internal representation of the tuple for the current interpretation of F as described in Section 4.1. In line 2, the variables of the (possibly non-ground) rule $H \leftarrow PBody, \text{not} NBody$ are renamed with fresh variables. These variables are assigned to the finite set domain $0..M1$ where $M1 = M - 1$ (and the “big number” M introduced in the previous subsection is defined by a fact `bigM(M)`). Constraints on the atoms of the body are set in lines 5 and 6. The predicate `build_constraint` sets membership (`in_set`) and non membership (`nin_set`) constraints on the variables of the atoms of the positive part (resp. negative part) of the body. The idea is that $C3 = 1$ iff the positive body is satisfied by the model I and $C4 = 1$ iff the negative body is satisfied by I . The predicate `build_arity` converts a tuple into a unique FD value according to the function `map` defined in the previous section. In lines (10) and (11), we add the constraints expressing the fact that the head is not yet in the positive part of the model (useless rule application) and it is not in the negative part of the model (inconsistent rule application). The variables C_1 and C_2 take care of these constraints.

In line (12), we collect all the ground instantiations of the rule that lead to new positive introductions of instances of the head. The list of new values for the atom F (see line (7)) is converted to the positive domain and added to the previous values. Observe that `build_constraint` takes care also of built-in predicates (e.g., equalities) calling the auxiliary predicate `expression` that parses the terms in the built-in atom. The additional parameter will be used later to support the computation of the well-founded model.

Observe that (local) grounding is performed in line (12), making sure that useless or redundant grounding are avoided a priori.

4.3 Well-founded model computation

Computing a well-founded model is a deterministic step during GASP computation. As done for T_P in the previous section, the implementation is based on FD constraint programming and FDSETs representation of interpretations. The idea of alternating fixpoint [19] is realized in Prolog.

The implementation boils down to controlling the alternating fixpoint computation and to realizing the $T_{P,J}$ operator (2). We present here the core procedure that analyzes a clause and computes the head predicates to be included in the resulting interpretation.

Let us consider a clause of P :

$$p^n(\mathbf{Y}_0) \leftarrow p_1^{n_1}(\mathbf{Y}_1), \dots, p_k^{n_k}(\mathbf{Y}_k), \text{not } p_{k+1}^{n_{k+1}}(\mathbf{Y}_{k+1}), \dots, \text{not } p_j^{n_j}(\mathbf{Y}_j)$$

with $|\mathbf{Y}_i| = n_i$ and $\mathbf{Y}_i \in (Vars(P) \cup Consts(P))^{n_i}$. Note that \mathbf{Y}_i may contain repeated variables and it can share variables with other literals in the clause.

Given two interpretations I and J , the application of $T_{P,J}$ to I considers each clause such that $I \models body^+$, $J \not\models body^-$. For these clauses, a set of new head

predicates is produced and added to the resulting interpretation. Instead of a generate and test approach, in which the new tuples in the head are generated using unification, we adopt a constraint-based approach. For each clause and interpretation I , we build a CSP that characterizes the atoms $p^n(\mathbf{X})$ derivable from the body.

For each predicate $p_i^{n_i}$ we introduce a FD variable V_i and we create the constraint $V_i = \text{map}(\mathbf{Y}_i)$. Recall that this constraint allows us to connect the individual variables in \mathbf{Y}_i to the variable V_i representing the possible tuples associated to $p_i^{n_i}$. The domain of V_i is Pos , where $(p_i^{n_i}, n_i, Pos, Neg)$ is part of the current interpretation I . For each literal **not** $p_j^{n_j}$, we use the same scheme, except for the fact that the domain of V_j is disjoint from the set Pos' , with $(p_j^{n_j}, n_j, Pos', Neg') \in J$. For the head of the rule, we define a FD variable V_0 similarly. These FD constraints allow us to link the tuples assignments, according to the variables occurrences.

An additional constraint is posted to ensure that the tuples in V_0 (the ones that are to be added to the interpretation) are not already present in the model and are not causing contradictions with the known negative tuples. According to the semantics of $T_{P,J}$, we use a call to the predicate `build_constraint(NBody, J, C4, neg)` in Figure 1. The difference between this constraint and the corresponding one used in the computation of T_P computation is that, by definition, the negative part of the rule may not appear in J^+ .

Finally, a labeling phase for the variables occurring in \mathbf{X} allows us to produce the set of ground instances of \mathbf{X} that satisfy the CSP. The atoms $p^n(\mathbf{X})$ are added to the interpretation, exploiting FDSETs operations.

Example 1. Let us consider an example of the application of $T_{P,\emptyset}(I)$ using a clause and the FD representation of domains. In particular, let us consider the clause

$$p(X) \leftarrow q(X, Y), \text{not } r(Y)$$

where $I = \langle \{q(1, 1), q(1, 2), q(2, 2), p(1)\}, \emptyset \rangle$. Let $\mathbb{M} = 3$, then I is represented as

$$(p, 1, [[1|1]], []), (q, 2, [[4|4], [7|8]], []), (r, 1, [], []).$$

The CSP induced is: $V_0 = X, V_1 = X + 3Y, V_2 = Y$, where the initial domains for V_0 and V_2 is the set $\{0, \dots, \mathbb{M} - 1\} = \{0, 1, 2\}$, while $V_1 \in \{4, 7, 8\}$. Using constraint propagation, the other domains can be restricted to $V_0 \in \{1, 2\}$ and $V_2 \in \{1, 2\}$. Moreover, the predicate p is constrained to be different from the values already known: $V_0 \notin \{1\}$; the predicate p is also constrained not to be in contradiction to its negative facts: in this case no constraint is added. The solution to this CSP is $X \in \{2\}$ and thus the fact $p(2)$ can be added to the interpretation.

4.4 Computing answer sets

The complete answer set enumeration is based on the well-founded model computation, alternated to a non deterministic choice phase. Basically, we collect

the well-founded model I of P . If the model is complete, we return the result. Otherwise, if there are some unknown literals, we start the stable model computation with I as initial interpretation. The call to $\text{wf}(P)$ save the first can detect inconsistent interpretations (failed I). In Figure 2, we summarize in pseudocode the algorithm.

```

(1)  rec_search(I)
(2)    R = applicable_rules(I)
(3)    if (R =  $\emptyset$  and I is a model) output: I is a stable model
(4)    else select  $a \leftarrow \text{body}^+, \text{body}^- \in R$ 
(5)      I = wf( $P \cup \{\text{body}^-\} \cup I$ )
(6)      if (I not failed) rec_search(I)

```

Fig. 2. The answer set computation

Each applicable rule represents a non deterministic choice in the computation of a stable model. The stable model computation explores each of these choices (line 4), and computes I_{i+1} using the wf operator starting for $P \cup \{\text{body}^-\} \cup I_i$ (line 5), as defined in the GASP computation. Note that a is immediately inserted into the model. This step requires the local grounding of each applicable rule in P , according to the interpretation I_i . The local grounding phase is repeated several times during computation, however it should be noted that each ground rule is produced only once along each branch, due to the constraints introduced. However, every time the local grounding is invoked, a CSP is built. We believe that the enhancement of this step (e.g., building CSPs less often) could reduce the search time significantly.

The process may encounter a contradiction while adding new facts to the interpretation, and consequently the computation may encounter failures. Whenever there are no more applicable rules, a leaf in the search tree is reached (line 3) and the corresponding stable model is obtained (convergence property).

The applicable rules w.r.t. an interpretation I_i are determined (line 2) as defined in GASP computation, i.e., solving the CSP using FD and FDSETs with similar techniques to the ones described in the previous sections. Here, the constraints for the negative part are slightly different from the well-founded rule applicability, since the negative part of a clause may not appear in the current I^+ , while in wf we used the additional interpretation J^+ .

In order to separate failure nodes from leaf nodes (stable models with no applicable rules), we organized the expansion as follows. We first collect every applicable rule at a node of the computation (line 2), then we apply the rule and add the new heads to the next interpretation. If the head produces a literal in contradiction to the interpretation, a failure is raised by a consistency check and the search backtracks (line 6). Note that I can be failed, since the well-founded computation is based on a new program that could introduce some contradictions (line 5).

From the implementation point of view, we verified that computing well-founded models at every non deterministic application of a rule is time consuming. In particular, the extraction of the extension of P with new facts from the positive interpretation is inefficient.

To gain efficiency, we substituted the call to the well-founded computation with a variant of the T_P operator. The extension of T_P to ASP considers rules where $body^+ \in I^+$ and $body^- \in I^-$ (actually, the same implemented in the code in Figure 1). The T_P operator adds new positive atoms as stated by the head of the rule. Other consequences that a call to the well-founded solutions could detect in one call will be detected in the successive part of computation (we are not losing correctness or completeness).

5 A permutation-free labeling

When enumerating stable models with a bottom up tree-based search, special care is needed in order to avoid producing repeated models. In fact, the concept of applicable rules and their non deterministic applications, allow the exploration of equivalent branches, where the order of rule applications is swapped, while the interpretation converges to the same set.

Let us abstract the process of search over a tree as follows. Every node u in the search tree is related to a set of possible choices $C(u)$ to be tried, where $C(u) \subseteq R$ and R is a set of applicable rules. The applicable rules at u depend on the interpretation $I(u)$ present at that node. Each non deterministic choice $r \in C(u)$ at u opens a branch, labeled by r , and it defines a child node of u in the search tree.

We now show a simple example of the combinatorial explosion of the number of branches due to all permutations. Let us assume that a simple search tree has a root u with $C(u) = \{a, b, c\}$. The expansion of the tree produces 6 different leaves, according to the possible permutations in the application of available rules. The unique interpretation $I = \{a, b, c\}$ is explored under every possible order of application of the facts in the program.

Let us introduce a transitive relation $\prec: R \times R \cup \{\perp\}$. Given $A = a_1 \prec a_2 \prec \dots \prec a_n$ and $B = b_1 \prec b_2 \prec \dots \prec b_m$, the *extension* of A w.r.t. B is the new order $a_1 \prec a_2 \prec \dots \prec a_n \prec c_1 \prec \dots \prec c_k$, where c_1, \dots, c_k are the elements in $B \setminus A$ retrieved in the same order as in B .

For example, let $a, b, c \in R$ and $a \prec b \prec c$. The order can be used to guide the nodes expansion, in particular to force the backtracking whenever the order is not respected along a branch. In the previous example, assuming to expand the rule b at the root, the applicable rules at the next node are $\{a, c\}$. The application of a constructs a branch in which b is applied before a and thus that branch fails. The application of c is allowed, and produces a new node in which only a is applicable. However the application of a generates a branch $b \prec c \not\prec a$ which results in a failure. The complete exploration of the search tree leads to a single success leaf, which corresponds to the application of rules in the order $a \prec b \prec c$.

Given a node u , for each applicable rule $r \in C(u)$ at u , a child of u' is expanded. We denote with $r = rule(u')$ the rule that caused the generation of u' . Note that all applicable rules must be collected at a node. When expanding the corresponding child, a test on the order is performed and a backtrack is invoked in case of a failure. This strategy allows us to distinguish between a success node (no applicable rules) and a failure node (every applicable rule violates the order). In fact, filtering out rules that are applicable but violate the ordering would produce an ambiguity on the type of node at hand.

When computing stable models, there is no a-priori order which is suitable to avoid permutations. In fact, the order is built dynamically, while exploring the nodes. Basically, a partial ordering is defined at each node, starting with the empty order at the root. Every time a node u is considered, the order $Ord(u)$ is extended with the applicable rules $C(u)$ of the node. Any expanded child v inherits the new order. To detect the order violation, it is sufficient to test if $r(u) \prec r(v)$ using the order $Ord(v)$. If the test fails, then backtracking is forced, since there is a left branch that contains the expansion $u \prec v$.

6 Experiments

The prototype implementing the ideas described above, as well as the tests described in this section, is available at www.dimi.uniud.it/dovier/CLPASP. The prototype has been developed using SICStus Prolog 4 (<http://www.sics.se/is1/sicstuswww/site/>), chosen for its rich library of FDSET primitives.

We performed some preliminary experiments, using different classes of ASP programs, and we report execution times in Table 1. All the experiments have been performed on an AMD Opteron 2GHz Windows XP machine.

In the set of experiments (Definite), we define a recursive binary predicate p , with the semantics $p(1, 2), p(2, 3), \dots, p(N - 1, N)$, and a predicate h which represents the transitive closure of p . The growth of the running time is quadratic in N , and the ratio between Smodels execution and GASP execution is a (low) constant. Let us observe that Smodels was unable to execute the instance $N = 256$, due to **Error in input** caused by the large size of the ground program. $N = 228$ is the largest instance handled by lparse+Smodels.⁵ We indicate with ∞ the computations that required more than a day to terminate. We also report the grounding times required by Lparse.

We then add to the definite program the following clause defining the predicate r :

$$r(X, Y) :- h(X, Y), \text{not } p(X, Y).$$

The whole program admits a well-founded and stable model. The same complexity observations made for definite programs continue to hold.

Finally, we run GASP and Smodels on a naive ASP modeling of the Schur number problem (see, e.g., [5] for a description).

`number(1..n).`

⁵ smodels 2.26 under Windows XP/cygwin.

```

part(1..p).
1 { inpart(X,P):part(P) } 1 :- number(X).
:- number(X), number(Y), number(Z), part(P),
   inpart(X,P), inpart(Y,P), inpart(Z,P),
   X < Y+1, Z = X+Y.

```

In this case the behavior of GASP (looking for one solution) is worse than in the above simpler cases. This is probably due to the amount of backtracking to be handled at a meta-interpreter level.

The current GASP implementation admits a natural and effective extension if cardinality constraints are used in the ASP program. We focus here on a particular case of a *function* constraint, namely an ASP rule of the form:

```

1 { function(X,Y): range(Y) } 1 :- domain(X).

```

typically used in encoding constraint satisfaction problems [5]. Let us assume that the predicate `range` is defined by the facts `range(a1) . . . , range(an)`. Then, the above rule can be encoded as:

```

function(X,a1) :- domain(X),
                not function(X,a2), ..., not function(X,an).
                ...
function(X,an) :- domain(X),
                not function(X,a1), ..., not function(X,an-1).

```

and GASP will be able to find the correct stable models.

We can anticipate the calling of the `fixpoint` procedure by a non-deterministic, constraint-based, generation of the values of the functions that satisfy ASP *constraints* in the ASP code we are reasoning on. In the implementation available on internet, we have implemented this idea. Basically, we look for pairs (X, Y) to be assigned to the predicate `inpart` that represents *functions* and fulfilling the following constraints generated by unfolding the ASP constraint modeling Schur:

- (1) forall X: number(X) forall Y:number(Y) forall Z:number(Z)
- (2) forall P1: part(P1) forall P2:part(P2) forall P3:part(P3)
- (3) if inpart(X,P1) \wedge inpart(Y,P2) \wedge inpart(Z,P3) \wedge X+Y = Z
- (4) add the constraint $P1 = P2 \Rightarrow P1 \neq P3$

As reported in Table 1 (column GASP-fun) this leads to a dramatic speed-up of the running time and this is encouraging for extensions of the naive implementation.

7 Conclusions

In this paper, we provided the foundation for a bottom-up construction of stable models of a program P without preliminary program grounding. The notion of

	N (n,p)	Lparse	Smodels	GASP	GASP/Smodels	GASP-fun
Definite	32	0.06	0.03	0.14	4.6	0.14
	64	0.09	0.12	0.45	3.7	0.45
	128	0.26	0.79	1.89	2.4	1.89
	228	0.75	1.95	6.78	3.5	6.78
	256	0.65	Error	8.81	-	8.81
Well Founded	32	0.06	0.08	0.98	12.2	0.98
	64	0.11	0.23	3.58	27.8	3.58
	128	0.32	1.07	15.38	14.4	15.38
	228	0.90	3.39	58.70	17.3	58.70
	256	0.68	Error	78.34	-	78.34
Schur	(6,3)	0.04	0.09	1.64	18.2	0.04
	(7,3)	0.05	0.18	6.59	36.6	0.06
	(8,3)	0.05	0.25	27.43	109.7	0.06
	(9,3)	0.06	0.48	113.59	236.65	0.06
	(10,3)	0.06	0.19	480.85	2530.8	0.09
	(30,4)	0.09	0.03	∞	-	0.36
	(35,4)	0.10	0.09	∞	-	0.44
	(40,4)	0.11	77.31	∞	-	0.50
	(45,4)	0.15	∞	∞	-	8315

Table 1. Timings (expressed in seconds). ∞ means ≥ 1 day.

GASP computation has been introduced; this model does not rely on the explicit grounding of the program. Instead, the grounding is local and performed on-demand during the computation of the answer sets. We believe this approach can provide an effective avenue to achieve greater efficiency in space and time w.r.t. a complete program grounding.

We illustrated a preliminary implementation of GASP using CSP on FD variables and FDSETs. We showed how to design T_P , well-founded and stable models computation based on CSPs. This allowed us to encode the entire process in Prolog. Interestingly, the running times for T_P and the well-founded computation are comparable to Smodels. Some ASP programs run slower, due to Prolog overheads and the limited efficiency of some (naive) data structures used.

We plan to investigate how to extend the model to enable the integration of other language features commonly encountered in ASP languages, and how to effectively use such features as constraints to guide the construction of the FDSET search space. We will also explore how global properties of the program and of the partial model can be used by the GASP implementation to improve efficiency.

Acknowledgments

The work is partially supported by MUR FIRB RBNE03B8KK and PRIN projects, and NSF grants HRD0420407 and CNS0220590. We really thank Andrea Formisano for the several useful discussions.

References

1. Y. Babovich and M. Maratea. Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. <http://www.cs.utexas.edu/users/yuliya>, 2003.
2. C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, 2003.
3. D. Brooks, E. Erdem, S. Erdogan, J. Minett, and D. Ringe. Inferring Phylogenetic Trees Using Answer Set Programming. *Journal of Automated Reasoning*, 39(4):471–511, 2007.
4. P. Codognot and D. Diaz. A Minimal Extension of the WAM for `clp(fd)`. In *International Conference on Logic Programming*. MIT Press, 1993.
5. A. Dovier, A. Formisano, and E. Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In *International Conference on Logic Programming*, pages 67–82. Springer Verlag, 2005.
6. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.
7. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. `clasp`: a Conflict-driven Answer Set Solver. In *Logic Programming and Non-Monotonic Reasoning*, pages 260–265. Springer Verlag, 2007.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *International Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.
9. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
10. V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.
11. L. Liu, E. Pontelli, S. Tran, and M. Truszczynski. Logic Programs with Abstract Constraint Atoms: the Role of Computations. In *International Conference on Logic Programming*, pages 286–301. Springer Verlag, 2007.
12. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Heidelberg, 1987.
13. V.W. Marek and M. Truszczynski. Stable Models and an Alternative Logic Programming Paradigm. In K.R. Apt, V.W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm*. Springer Verlag, 1999.
14. I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, (to appear).
15. I. Niemela and P. Simons. `Smodels` - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Logic Programming and Non-monotonic Reasoning*, pages 421–430. Springer Verlag, 1997.
16. T. Son and E. Pontelli. Set Constraints in Logic Programming. In *Logic Programming and Non-Monotonic Reasoning*, pages 167–179. Springer Verlag, 2004.
17. T. Son and E. Pontelli. Planning for Biochemical Pathways: a Case Study of Answer Set Planning in Large Planning Problem Instances. In *First International Workshop on Software Engineering for Answer Set Programming*, pages 116–130, 2007.
18. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
19. U. Zukowski, B. Freitag, and S. Brass. Improving the Alternating Fixpoint: The Transformation Approach. Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 4–59, 1997.

Compiling and Executing Declarative Modeling Languages in Gecode

Raffele Cipriano, Agostino Dovier, and Jacopo Mauro

Univ. di Udine, Dip. di Matematica e Informatica.
(cipriano|dovier)@dimi.uniud.it

Abstract. We developed a compiler from SICStus Prolog CLP(FD) to Gecode as well as a compiler from MiniZinc to Gecode. We compared the running times of the executions of (standard) codes directly written in the three languages and of the compiled codes for a series of classical problems. Performances of the compiled codes in Gecode improve those in the original languages and are comparable with running time of native Gecode code. This is a first step towards the definition of a unified declarative modeling language for combinatorial problems that will allow the user to define problem instances and solving meta-algorithms using high-level declarative language, and then to automatically translate the specification into a low-level encoding, that allows the solving algorithms to run efficiently.

1 Introduction

Combinatorial problems like planning, scheduling, timetabling and, in general, resource management problems, are daily handled by industries, societies and research companies. Although, in principle, problems in this family can be mathematically modeled and then solved by computers, the intrinsic NP-hardness of most of them prevents us from expecting from a tool an optimum solution in acceptable time. Therefore, we must focus on developing techniques for finding “good” solutions (as much as possible close to the optimal one) in acceptable time. In the past decades a lot of techniques and solvers have been developed to cope with this kind problems, like branch and bound/branch and cut/branch and price algorithms, constraint programming techniques, local search, heuristics, and so on. Moreover, several modeling languages have been proposed to easily model problems and instances, and to easily interact with the solvers. In fact, it would be desirable to work with a user-friendly modeling language that allows the user to define the problem in an easy and flexible way interfaced with an efficient solver able to explore in a clever way the space of the solutions.

In this work we have focused our attention on the Gecode solver, a recent C++ constraint solving platform with excellent performances [11] and on declarative modeling languages, like constraint logic programming [13] and MiniZinc [15]. Encoding constraint satisfaction/optimization problems using the C++ framework Gecode is not very user friendly as using high/medium-level declarative modeling languages. We present a compiler from SICStus Prolog CLP(FD) [1] to Gecode as well as a compiler from MiniZinc to Gecode. We have chosen a set of (well-known) benchmarks, and we

have compared their running times in the original paradigms (SICStus, MiniZinc, and Gecode) and the running time in Gecode of their translation. We also compared our results with the translation of MiniZinc into Gecode offered by MiniZinc developers.

The results are rather encouraging. Native code executions are typically faster in Gecode than in SICStus and MiniZinc (in the case of SICStus a consistent comparison is reported in [11]). However, in all cases, compilation (and then execution) in Gecode improves the performance of the native execution, and, moreover, these times are comparable with running time of native Gecode code.

This would allow the user to model problems at high level keeping all the well-known advantages of this programming style, without losing efficiency w.r.t. C++ encoding. Moreover, our encoding of MiniZinc in Gecode outperforms the similar translation (via FlatZinc) presented in [10].

The present work is part of a general project of developing a programming tool for combinatorial problems. This tool will be made of three main parts: the modeling one, the translating one, and the solving one. In the modeling part the user will define in a high-level style the problem and the instance he wants to solve and the algorithm to use (constraint programming search, eventually interleaved with local search, integer linear programming, heuristics or meta-heuristics phases). In the translating phase the model and the meta-algorithm defined by the user will be automatically compiled into the solver languages, like Gecode or other ones. During the third phase, the overall compiled program will be run and the various solvers will interact in the way specified by the user in the first phase, to find the solution for the instance of the problem modeled. We are planning to test the tool on different families of problems including those we have already cope with: a hospital rostering (timetable) problem [2], the protein structure prediction problem [5, 3], and the planning problem [9].

The paper is organized as follows. In Section 2 we describe the languages used to model and solve our problems, starting from the high level ones (SICStus Prolog and MiniZinc, respectively in subsections 2.1 and 2.2) and ending with the low-level encoding in Gecode (subsection 2.3). In section 3 we explain how we perform the translation from the high-level languages to the low-level one: this process passes through an intermediate language (named CNT) that we describe in subsection 3.1; the translation from SICStus Prolog and MiniZinc code into CNT are described in subsections 3.2 and 3.3; the translation from CNT to the C++ final encoding in Gecode is described in 3.4. In section 4 we present the problems and the instances we used to test the solving performances of the different encodings (native SICStus Prolog, MiniZinc and Gecode code and the various compiled code). Section 5 summarizes the results and explains the future works we intend to achieve. In Appendix we report the Prolog codes of the test problems used.

2 The languages used

We briefly introduce the high-level declarative language CLP(FD), the medium-level declarative modelling language MiniZinc, and the constraint solving platform Gecode. We use the N-Queens problem to briefly explain the different programming paradigms.

The N-Queens problem *It is the problem of putting N chess queens on an $N \times N$ chessboard such that none of them is able to capture any other using the standard chess queen's moves.* A standard way to model this problem is to consider that on the chessboard there will be one queen for each chessboard column: so we need N variables $X_1 \dots X_N$ (one for each chessboard column), with domain $X_i = \{1, 2, \dots, N\}$, with $X_i = k$ meaning that *the queen of column i will be placed on row k .* With this model, vertical attacks are implicitly encoded; to avoid horizontal attacks we must impose that $X_i \neq X_j \forall i \neq j$; to avoid diagonal attacks, we must impose that $\forall_i \forall_{j>i}, X_i \neq X_j + (j - i) \wedge X_j \neq X_i + (j - i)$.

2.1 CLP(FD)

CLP(D) is a declarative programming paradigm, parametric on the constraint domain \mathcal{D} , first presented in 1986 [12] (see e.g. [13] for a popular review). Combinatorial problems are usually encoded using constraints over *finite domains* (namely, $\mathcal{D} = FD$), currently supported by all CLP systems based on Prolog (for example BProlog [17], GNUProlog [7], SICStus Prolog [1], ECLiPSe [16], just to name a few). We focused on the library `clpfd` of SICStus Prolog [1], but what we have done can be repeated for other systems embedding constraints on finite domains. We focus on the classical *constraint+generate* programming style, where a constraint definition phase anticipates a labeling stage. The SICStus primitives for defining an array of variables of length N (named `Queens`) and to set the domain of these variables in the integer range $1 \dots N$ are:

```
(1) length(Queens, N),
(2) domain(Queens, 1, N),
```

The avoiding-horizontal-attacks constrain can be easily posted using the built-in constrain that post inequalities for each pair of variables:

```
(3) all_distinct(Queens),
```

Then we impose that each queen (extracted by the recursive predicate *diagonal*) satisfies the avoiding-diagonal-attacks constrain (post with predicate *safe*):

```
(4) diagonal([]).
(5) diagonal([Q|Queens]) :-
(6)   safe(Q, 1, Queens),
(7)   diagonal(Queens).
(8) safe( _, _, []).
(9) safe(X, D, [Q|Queens]) :-
(10)  X + D #\= Q,
(11)  Q + D #\= X,
(12)  D1 is D + 1,
(13)  safe(X, D1, Queens).
```

The D variable represent the difference $(j - i)$ of the model explained above; the `#\=` symbol is the standard SICStus Prolog way to post an inequality constrain over

variables (it is sufficient to add the # character before any standard relational symbol, such as =, \=, <, =< and so on).

2.2 MiniZinc and FlatZinc

MiniZinc is a medium-level modeling language developed by the NICTA research group [15]. It allows to express most CP problems easily, supporting sets, arrays, user defined predicates, some automatic coercions and so on. But it is also low-level enough to be easily mapped onto existing solvers. It is a subset of the language Zinc.

To encode the N-Queens model in MiniZinc we must first declare an array (here named q) of N variables with domain in the integer range $1 \dots N$, using:

```
(14) array [1..n] of var 1..n: q;
```

Then we must post the constraints on all the couple of variables, using the code below:

```
(15) constraint
(16)   forall (i in 1..n, j in i+1..n) (
(17)     noattack(i, j, q[i], q[j])
(18)   );
```

The constraints are posted by the predicate `noattack` (line 21 deals with horizontal attacks, lines 22-23 deal with diagonal ones):

```
(19) predicate
(20)   noattack(int: i, int: j, var int: qi, var int: qj) =
(21)     qi != qj ^
(22)     qi + i != qj + j ^
(23)     qi - i != qj - j;
```

FlatZinc is a low-level solver-input language, and it is mostly a subset of MiniZinc. The NICTA research group provides a compiler from MiniZinc to FlatZinc that support all solver-supported global constraints. This way, a solver writer can support MiniZinc with the minimum effort of providing a simple FlatZinc frontend to the solver and combining it with the existing MiniZinc-to-FlatZinc translator. The NICTA team also provide its own solver that can read and execute a FlatZinc model.

2.3 Gecode

Gecode is an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications (see [11] for details). It is implemented in C++ and offers competitive performances w.r.t. both runtime and memory usage. It implements a lot of data structures, constraints definitions, and search strategies, allowing also the user to define his own ones. Its modeling language style is C++ like, and, thus, programmer should take care of several low-level details. We report an extract of the N-Queens problem encoded in Gecode, regarding the horizontal and diagonal constraints.

```

(24) for (int i = 0; i<n; i++){
(25)     for (int j = i+1; j<n; j++) {
(26)         post(this, q[i] != q[j]);
(27)         post(this, q[i]+i != q[j]+j);
(28)         post(this, q[i]-i != q[j]-j); } }

```

To simplify the using of the Gecode solver, its developers provide a FlatZinc frontend, i.e. an executable program that reads a FlatZinc model, solve it using the Gecode libraries and prints the solution (if any) on the standard output.

3 Translation

The translation from SICStus and MiniZinc programs to Gecode is carried on in two stages: first we translate the high-level code into an intermediate language (called CNT) that lists explicitly all the constraints. Then we generate C++ code from the CNT file, using static analysis to improve the second part of the compilation. After a brief introduction to the language CNT, we show the main lines of this two-steps translation.

3.1 CNT

The intermediate language CNT is used for listing the constraints, specifying some searching parameters and which variables should be printed. The CNT grammar is the one described in Table 1 (the grammar is also defined in the file `parser.y` [4]).

In the CNT language the variables are not typed, in sense that every variable is assumed to be an integer variable. Boolean variables are special variables that can assume only the values 0 and 1. This CNT language was realized to ease the listing of the constraints defined by a prolog program. For that reason we didn't add the possibility to declare variable arrays.

The semantics of the CNT language is intuitive. We only report the semantics of the non-terminal `<sentence>`:

rule 3: “domain $[x_1, \dots, x_n], n_1, n_2$ ” sets the domain $[n_1 \dots n_2]$ to the FD variables x_1, \dots, x_n .

rule 4: “in x, n_1, n_2 ” sets the domain $[n_1 \dots n_2]$ to the unique FD variable x

rule 5: “sum $[x_1, \dots, x_n], op, x$ ” posts the constraint: $\sum_{i=1}^n x_i \sim_{op} x$

rule 6: “scalar_product $[n_1, \dots, n_k], [x_1, \dots, x_k], op, x$ ” posts the constraint: $\sum_{i=1}^k n_i * x_i \sim_{op} x$

rule 7: “all_different $[x_1, \dots, x_n]$ ” adds the all_different global constraint between the variables x_1, \dots, x_n (viewed in binary form: $\forall i \forall j 1 \leq i < j \leq n. x_i \neq x_j$)

rule 8: “element $[x_1, \dots, x_n], y, z$ ” sets the constraint $x_y \sim_{eq} z$

rule 9: bool expression b sets the (reified) constraint $b \sim_{eq} true$

rule 10: “write_string_or_variable” print on the standard output the strings and variables passed as arguments

rule 11: “branch $[x_1, \dots, x_n]$ ” adds $\{x_1, \dots, x_n\}$ to the set of the variables to search

<S>	:= <sentence>	(rule 1)
	<S> <sentence>	(rule 2)
<sentence>	:= domain <array of el> , NUMBER , NUMBER	(rule 3)
	in <el> , NUMBER , NUMBER	(rule 4)
	sum <array of el> , <relOp> , <el>	(rule 5)
	scalar_product <array of int> , <array of el> , <relOp> , <el>	(rule 6)
	all_different <array of el>	(rule 7)
	element <array of el> , <el> , <el>	(rule 8)
	<boolE>	(rule 9)
	write <string or variable list>	(rule 10)
	branch <array of el>	(rule 11)
<boolE>	:= <el>	(rule 12)
	not <boolE>	(rule 13)
	(<boolE> <boolOp> <boolE>)	(rule 14)
	(<arithE> <relOp> <arithE>)	(rule 15)
<arithE>	:= <el>	(rule 16)
	abs(<arithE>)	(rule 17)
	max(<arithE> , <arithE>)	(rule 18)
	min(<arithE> , <arithE>)	(rule 19)
	(<arithE> <arithOp> <arithE>)	(rule 20)
<boolOp>	:= and or xor eqv imp reverse_imp	(rules 21-26)
<relOp>	:= > < <= >= == !=	(rules 27-32)
<arithOp>	:= + - * / mod	(rule 33-37)
<el>	:= NUMBER variable	(rules 38-39)
<variable>	:= _NUMBER	(rule 40)
<array of el>	:= [<list of el>]	(rule 41)
<list of el>	:= <el>	(rule 42)
	<el> , <list of el>	(rule 43)
<array of int>	:= [<list of int>]	(rule 44)
<list of int>	:= NUMBER	(rule 45)
	NUMBER , list of int	(rule 46)
<string or variable list>	:= <string or variable>	(rule 47)
	<string or variable> <string or variable list>	(rule 48)
<string or variable>	:= STRING variable	(rules 49-50)

Table 1. CNT grammar specification.

An example of CNT code is the following:

```
(29) domain [_1, _2, _3, _4], 1, 4;  
(30) all_different [_1, _2, _3, _4];  
(31) ((_1 + 1) != _2);  
(32) ((_2 + 1) != _1);  
(33) ((_1 + 2) != _3);  
(34) ((_3 + 2) != _1);  
(35) ((_1 + 3) != _4);  
(36) ((_4 + 3) != _1);  
(37) ((_2 + 1) != _3);  
(38) ((_3 + 1) != _2);  
(39) ((_2 + 2) != _4);  
(40) ((_4 + 2) != _2);  
(41) ((_3 + 1) != _4);  
(42) ((_4 + 1) != _3);
```

The CNT language can be seen as a subset of FlatZinc and in the future we hope to bypass the use of the CNT language and produce for every program a FlatZinc file.

3.2 CLP(FD) into CNT

For translating SICStus to CNT, we automatically create a new SICStus program where constraints definition is replaced by a printing stage. For instance consider the following code in which we use the constraints “domain” and “all_different”:

```
(43) test(X,N) :-  
(44)     length(X,N),  
(45)     domain(X,1,N),  
(46)     all_different(X).
```

To obtain the modified program we change all the predicates that add the constraints into the predicate “format” for printing informations. The previous code can therefore be converted into the following program:

```
(47) test(X,N) :-  
(48)     length(X,N),  
(49)     format("domain ~q, ~q, ~q;\n", [X,1,N]),  
(50)     format("all_different ~q;\n", [X]).
```

Some problems arise when there is a unification. In fact in some programs the logic variables are known to be FD-variables only at runtime and therefore every time in the program there is a unification we have to add some equality constraints. We developed some particular cases to cope with this and other minor technical problems.

The execution of the modified SICStus code instead of adding constraints simply prints all the constraints in the CNT form. Thus the execution of the modified program generates the CNT (flat) code. For instance, the execution of the modified version of the SICStus N-Queens code (1)–(13), with $N = 4$ generates the CNT code (29)–(42).

3.3 MiniZinc (FlatZinc) into CNT

We took advantage of the existing compiler from MiniZinc to FlatZinc [15] and thus focus on the translation from FlatZinc to CNT. As we said before the CNT language can be seen as a subset of FlatZinc, and thus, the subset of the FlatZinc programs that encode an integer CSP problem can be translated straightforwardly. Consider, for example, the following constraints in FlatZinc:

```
(51) array[0 .. 2] of var 0 .. 2: v;  
(52) constraint int_eq(v[0], 0);  
(53) constraint all_different([v[0], v[1], v[2]]);
```

These constraint can be defined in CNT in the following way:

```
(54) domain [_0, _1, _2], 0, 2;  
(55) (_0 == 0);  
(56) all_different [_0, _1, _2];
```

3.4 CNT into Gecode

We have developed a compiler from CNT to C++/Gecode. Before the compilation we execute some simple simplifications to the CNT code. First of all we precompute numerical expressions and then we use the equality constraints to reduce the number of variables (using this expedient we are able to get rid of the useless constraints added in the prolog to CNT translation).

Moreover, using static analysis, the compiler groups the constraints that can be defined within a `for` cycle to reduce the final length of the C++ program. In some cases this lead to a dramatic reduction of time needed by Gecode for the compilation of the `.cc` file with its libraries. For instance, the Gecode file obtained by the instance 100-Queens with this optimization has a size of 53 KB and is compiled with the Gecode libraries in 5.8s, while the code obtained by “flat” CNT has a size of 1.5MB and requires 13 hours and 40 minutes for the compilation.¹

Precisely, we have defined a syntactic notion of *matching* between constraints. Intuitively, two constraints match if they are based on the same predicate symbol and differ only in the index of some variables or in the other numbers involved. Two constraints that match can be defined in a “for” cycle (the definition of the matching function is correct but not complete in the sense that some constraints that could be defined in the same “for” cycle don’t match). After partitioning the set of the constraints using the matching function, we analyzed each subset of the partition trying to group the constraints to declare each constraint with the minimum amount of “for” cycles. We realized that, due to the complexity of the problem, this task can be unfeasible. For this reason we realized a procedure that use the order of the definitions of the constraint to group them in a good solution. This idea tries to exploit the fact that a program declares the constraints following some principles.

¹ To be honest, we have been negatively impressed by the time required for compiling a Gecode specification on a Linux 64bit machine with 2.2 GHz. We think that Gecode developers should work on this problem.

To see how constraints can be defined in “for” cycles consider the following code in which for the first 9 elements in a list of length 10 we add the constraints that every element should be minor and the predecessor of the following element in the list.

```
(57) example :- length(X, 10),
(58)     domain(X, 1,10),
(59)     constraint(X),
(60)     labeling([ff], X),
(61)     write(X).
(62)
(63) constraint([_]):- !.
(64) constraint([X,Y|Xs]):-
(65)     X #< Y,
(66)     X + 1 #= Y,
(67)     constraint([Y|Xs]).
```

If you launch the tool SICStus to CNT you will obtain the following CNT code:

```
(68) domain [_1, _2, _3, _4, _5, _6, _7, _8, _9, _10], 1, 10;
(69) (_1 < _2);
(70) ((_1 + 1) == _2);
(71) (_2 < _3);
(72) ((_2 + 1) == _3);
(73) (_3 < _4);
(74) ((_3 + 1) == _4);
(75) (_4 < _5);
(76) ((_4 + 1) == _5);
(77) (_5 < _6);
(78) ((_5 + 1) == _6);
(79) (_6 < _7);
(80) ((_6 + 1) == _7);
(81) (_7 < _8);
(82) ((_7 + 1) == _8);
(83) (_8 < _9);
(84) ((_8 + 1) == _9);
(85) (_9 < _10);
(86) ((_9 + 1) == _10);
(87) branch [_1, _2, _3, _4, _5, _6, _7, _8, _9, _10];
(88) write "[" _1 " ", " _2 " ", " _3 " ", " _4 " ", " _5 " ", " _6 " ", "
    _7 " ", " _8 " ", " _9 " ", " _10 "]" ;
```

The lines (68)–(86) define the constraints used when we lunch the command “example.”. The line (87) shows what are the variables to search while the last line shows what information should be printed. The lines (68)–(87) are translated in Gecode in the following way:

```
(89) IntVarArgs intVarArray2(10);
(90) for(int int3= 0; int3<10; int3++)
(91)     intVarArray2[0+int3]= array[0 + int3 * 1];
(92) dom(this, intVarArray2, 1, 10, opt.ic1);
```

```

(93) for(int int4= 0; int4<9; int4++) {
(94)     rel(this,array[0+int4*1],IRT_LE,array[1+int4*1],opt.ic1);
(95) }
(96) for(int int5= 0; int5<9; int5++) {
(97)     IntVar intVar6(this, 1, 1);
(98)     IntVar intVar7=plus(this,array[0+int5*1],intVar6,opt.ic1);
(99)     rel(this, intVar7, IRT_EQ, array[1+int5*1],opt.ic1);
(100) }
(101) IntVarArgs intVarArray1(10);
(102) for(int int2= 0; int2<10; int2++)
(103)     intVarArray1[0+int2]= array[0 + int2 * 1];
(104) branch(this, intVarArray1, BVAR_SIZE_MIN, BVAL_MIN);

```

The lines (89)–(92) and (101)–(104) are used to define respectively the instructions defined in lines 68 and 87. The constraint with the path “ $((i + 1) == _ (i+1))$ ” and “ $(i < _ (i+1))$ ” are defined using two “for” cycles in lines (93)–(95) and (96)–(100).

The programming languages used are the following ones:

- SICStus Prolog 4.0.1 is used for the SICStus to CNT tool
- c is used for the FlatZinc to CNT tool
- Haskell (<http://www.haskell.org>) is used for the CNT to Gecode tool

For the parsing tasks we also use the parsing generators Bison (<http://www.gnu.org/software/bison/>) and happy (<http://www.haskell.org/happy/>) and the lexical analyser generators Flex (<http://flex.sourceforge.net/>) and alex (<http://www.haskell.org/alex/>)

4 Experimental Results

We considered instances of four well-known problems, i.e. N-Queens, Sudoku, Golomb Rulers, and Knapsack.

Sudoku 16x16 instances are taken from <http://www.live-sudoku.com/play-online/geant>, and 25x25 ones are taken from <http://www.eleves.ens.fr/home/frisch/sudoku.html>; instances for N-Queens problems range from $N = 100$ to $N = 115$; we launched Golomb rulers instances of order from 6 to 13, with two different lengths (the biggest satisfiable and the shorter unsatisfiable) for each order; knapsack instances are the same used in [8].

We modeled each problem in SICStus Prolog, MiniZinc, and Gecode. When available, we used the modeling offered by languages libraries. We also consider their translation with the tools described in the paper. We report the running times of the various versions in Table 2. The columns of the table are: *SICS*: pure SICStus Prolog 4.0.1 model; *SICS2GEC*: Gecode 1.3.1 model obtained compiling the SICStus Prolog model; *MZN2GEC*: Gecode 1.3.1 model obtained from the MiniZinc model compiled (by our tool) into a Gecode model; *MZN2FZNNI*: a FlatZinc model obtained from the MiniZinc model and run with the utility provided by NICTA research group; *MZN2FZNGE*: a FlatZinc model obtained from the MiniZinc model and run with the utility provided

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
16-0	0,05	0,01	0,01	75,68	594,27	0,07
16-1	0,00	0,00	0,00	0,32	0	0,00
16-2	0,12	0,06	0,03	1,24	20839,66	0,06
16-3	0,14	0,05	0,00	409,23	184,36	0,12
16-4	0,05	0,02	0,01	7,75	24,71	0,15
16-5	0,05	0,02	0,00	610,61	1204,99	0,07
16-6	0,10	0,03	0,04	24,16	666,97	0,08
16-7	0,89	0,55	0,27	12651	-	3,07
25-0	-	303206	164013	-	-	109799
25-1	-	669	879	-	-	395
25-2	-	52788	223	-	-	-
25-3	-	186362	57566	-	-	-
25-4	-	347	137	-	-	2547
25-5	-	89914	3560	-	-	-
25-6	-	44012	7123	-	-	-
25-7	-	147594	57620	-	-	-

Running times (s) for SUDOKU instances

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
100	0,13	0,87	0,02	4,59	-	0,00
101	60,67	0,95	14,03	25,81	-	3,83
102	0,16	1,41	0,02	5,13	-	0
103	0,13	468,73	0,02	5,64	-	0
104	0,12	0,14	0,06	6,00	-	0,02
105	8,71	186,42	1,93	8,89	-	0,55
106	0,22	35,79	0,06	5,72	-	0,01
107	0,2	0,19	0,09	6,26	-	0,02
108	2747	0,54	0,03	973,49	-	164,32
109	0,17	0,32	598,37	6,19	-	0
110	1704	0,15	403,67	801,28	-	119,78
111	0,34	0,18	0,22	7,56	-	0,05
112	0,21	93,11	0,08	7,32	-	0,02
113	0,43	-	0,08	6,97	-	0,02
114	0,17	3170	0,03	6,49	-	0,00
115	-	-	-	-	-	-

Running times (s) for N-QUEENS instances

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
6-sat	0,01	0,00	0,00	0,17	0,00	0,00
6-uns	0,00	0,00	0,00	0,18	0,01	0,00
7-sat	0,02	0,00	0,00	0,24	0,05	0,00
7-uns	0,06	0,01	0,02	0,23	0,09	0,00
8-sat	0,19	0,01	0,00	1,13	0,06	0,00
8-uns	0,67	0,13	0,18	0,92	0,27	0,03
9-sat	1,52	0,16	0,08	12,30	0,31	0,02
9-uns	5,96	1,26	2,19	10,66	1,87	0,24
10-sat	11,22	1,46	1,23	215,40	2,93	0,18
10-uns	46,73	10,51	26,70	166,81	27,01	1,95
11-sat	255,37	20,43	33,69	9404	294,05	0,91
11-uns	1406	316,77	1070	8342	1250	44,24
12-sat	2286	1181	9509	78616	5429	89,33
12-uns	8693	2134	16324	-	-	302,24
13-sat	56492	25560	-	-	-	1566
13-uns	-	-	-	-	-	6782

Running times (s) for GOLOMB RULERS instances

Instance	SICS	SICS2GEC	MZN2GEC	MZN2FZNNI	MZN2FZNGE	Gecode
0-sat	0,01	0,01	0,00	0,17	0,01	0,01
1-uns	0,01	0,01	0,00	0,16	0,27	0,01
2-sat	0,16	0,05	0,05	0,21	0,30	0,60
3-uns	0,15	0,06	0,05	0,21	11,76	0,06
4-sat	3,74	1,28	1,23	1,05	13,37	1,27
5-uns	3,72	1,28	1,22	1,05	889,21	1,25
6-sat	156,88	53,35	51,08	26,89	1008,96	52,32
7-uns	155,41	53,42	50,58	25,65	123120	52,11

Running times (s) for KNAPSACK instances

– means that the execution didn't terminate after 4 days of execution time.

Table 2. Running times comparison of different encoding styles for various problems

by the Gecode Team that use Gecode 2.0 libraries; *Gecode*: pure Gecode 1.3.1 model. All codes and instances are available at [4].

There are two kinds of compile times: time of the compilation from high level code to Gecode C++ file and time needed by Gecode for internal compilation and libraries linking. With the proposed automatic detection of “`for`” loops both of them are rather low (the order of some seconds). Of course, for some small instances this time cannot be ignored w.r.t. execution time, but it becomes negligible for difficult instances.

Except for some 25x25 Sudoku instances, native Gecode code is always the fastest one, and this is a reasonable result, because native Gecode is the lower level way to encode the problems.

SICS2GEC often speeds-up SICStus native (because the SICStus is the higher-level encoding), except for some instances of N-Queens. Moreover it has, in average, comparable times with Gecode native code. Let us observe, however, that it solves all the Sudoku instances, while native Gecode does not.

The behavior of *MZN2GEC* is substantially equivalent with *SICS2GEC*, while in average it outperforms *MZN2FZNNI* and *MZN2FZNGE*, which are the standard ways to run MiniZinc (FlatZinc) models. Precisely, for Sudoku, N-Queens and Golomb rulers *MZN2GEC* is faster than *MZN2FZNNI* of various order of magnitude, while it is slightly slower in the case of Knapsack. We presume that the *MZN2GEC* performances are better than *MZN2FZNNI* and *MZN2FZNGE* ones, because when translating CNT models into C++ codes we perform precomputations and static analysis (see paragraph 3.4) that simplifies the variables domains and the set of constraints, w.r.t the execution of the flatzinc models. However, the utility provided by NICTA research group and the one of the Gecode Team directly execute the flatzinc files, without returning any intermediate encoding, so we can't perform an exhaustive comparison of the encodings.

We executed all tests on AMD Opteron 280 at 2.2GHz, Linux CentOS machine.

5 Conclusion and Future work

As a first step for the definition and implementation of a unified declarative modeling tool for combinatorial problems, we developed a compiler from SICStus and MiniZinc to Gecode. Although in its current preliminary version, is able to show that Constraint Programming can be done at high level using well-known languages and then executed in new paradigms as Gecode, since running times are comparable w.r.t. those of this latter paradigm. This way, we have ensured the feasibility of the three-phase constraint programming tool (modeling-translating-solving) we are developing. In fact, since the translating process can be done in reasonable time and the execution running times are comparable with native code ones, the user can benefit from both the flexibility and easiness of high-level modeling languages and the efficiency of the new low-level constraint solvers.

At this point a lot of work needs to be done. For instance:

- improving the static analysis of the generated code to further speed-up the overall process (compilation+execution)
- extending its compiling mechanism (up to now limited to CSP)
- porting the tool to the (new) Gecode 2.1.1, that should outperform the performances

- writing a front-end from CLP(FD) to FlatZinc: in this way, once we have the FlatZinc code of our model we can take advantage of the FlatZinc solvers

Moreover, as said above, this work is part of a more general project. We are currently working on hybridization between local search and constraint programming starting from Gecode and EasyLocal++ [6]. Basically, the solution's search of Gecode will be improved by this combined approach (in [3] we applied this hybrid approach to the protein structure prediction problem with interesting results). As a side effect, with the compiler described in this paper, one will be able, for instance, to program in Prolog and take automatically advantage of the hybrid search.

Acknowledgements The work is partially supported by MUR FIRB RBNE03B8KK and PRIN projects. We thank Luca Di Gaspero and Andrea Formisano for the useful discussions and the help in installing packages. We also thank Alberto Ghedin for releasing the first naive compiler from SICStus Prolog to Gecode.

References

- [1] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. *PLILP 1997*:191–206
- [2] R. Cipriano, L. Di Gaspero and A. Dovier. Hybrid Approaches for Rostering: A Case Study in the Integration of Constraint Programming and Local Search. *HM 2006*, LNCS 4030:110–123.
- [3] R. Cipriano, A. Dal Palù and A. Dovier. A hybrid approach mixing local search and constraint programming applied to the protein structure prediction problem. *WCB 2008*. <http://wcb08.dimi.uniud.it/accepted.html>
- [4] R. Cipriano, A. Dovier, and M. Jacopo. Tools for compiling SICStus and MiniZinc in Gecode. <http://www.dimi.uniud.it/dovier/MISIGE>
- [5] A. Dal Palù, A. Dovier, and E. Pontelli. Heuristics, optimizations, and parallelism for protein structure prediction in CLP(FD). *Proc. of PPDP 2005*: 230-241, 2005.
- [6] L. Di Gaspero and A. Schaerf. EasyLocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software Practice and Experience*, 33(8):733–765, 2003.
- [7] Daniel Diaz. GNU Prolog: a free Prolog compiler with constraint solving over finite domains. Available from <http://www.gprolog.org/>, 2007.
- [8] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. *ICLP 2005*, LNCS 3668:67–82.
- [9] A. Dovier, A. Formisano. and E. Pontelli. Multivalued Action Languages with Constraints in CLP(FD). *ICLP 2007* 4670:255–270.
- [10] Gecode Team. FlatZinc/Gecode: a parser for FlatZinc modelling language. Available from <http://www.gecode.org/flatzinc.html>, 2007.
- [11] Gecode Team. Gecode: Generic Constraint Development Environment. Available from <http://www.gecode.org>, 2006.
- [12] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. *Tech. rep.*, Department of Computer Science, Monash University, June 1986.
- [13] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [14] N. Nethercote. Specification of FlatZinc. Available from <http://www.g12.cs.mu.oz.au/minizinc/flatzinc-spec.pdf>.

- [15] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. *CP 2007*, LNCS 4741:529–543.
- [16] M. Wallace, S. Novello and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. *Technical report, IC-Parc, Imperial College, London, 1997.*
- [17] Neng-Fa Zhou. B-Prolog: a versatile and efficient constraint logic programming (CLP) system. Available from <http://www.probp.com/>, 2006.

A SICStus Prolog codes

A.1 Sudoku model

```

sudoku(Cells) :-
    input(N, Cells),
    domain(Cells, 1, N),
    row_constraints(Cells, N),
    column_constraints(Cells, N),
    square_constraints(Cells, N),
    labeling([ff, bisect], Cells).

%%% instance from www.repubblica.it, April 7 2008

input(9, [3, 5, _, _, _, 6, _, _,
         _, _, 8, _, _, _, 4, _,
         _, 7, 2, _, 5, _, 3, 9,
         _, _, 1, 9, _, _, _,
         4, _, 5, _, _, 2, _, 1,
         _, _, _, 6, 2, _, _,
         9, 3, _, 4, _, 8, 1, _,
         _, 8, _, _, 1, _, _,
         _, 1, _, _, _, 7, 5]).

row_constraints([], _).
row_constraints(Cells, N) :-
    extract(N, Cells, Row, Rest),
    all_distinct(Row, [on(dom), consistency(global)]),
    row_constraints(Rest, N).

column_constraints(Cells, N) :-
    column_constraints(Cells, 1, N).
column_constraints(_, M, N) :-
    M > N, !.
column_constraints(Cells, M, N) :-
    column(Cells, M, N, Col),
    all_distinct(Col, [on(dom), consistency(global)]),
    M1 is M + 1,
    column_constraints(Cells, M1, N).

square_constraints(Cells, N) :-

```

```

    M is integer(sqrt(N)),
    square_constraints(Cells,0,N,M).
square_constraints(_,N,N,_):-!.
square_constraints(Cells,I,N,M):-
    square(Cells,I,M,N),
    I1 is I + 1,
    square_constraints(Cells,I1,N,M).

square(Cells,I,M,N) :-
    Start is (I//M)*(N*M) + (I mod M)*M,
    extract(Start,Cells,_,MyCells),
    pick_square(MyCells,0,M,N,Square),
    all_distinct(Square,[on(dom),consistency(global)]).

pick_square( _,M,M,_,[]) :-!.
pick_square(Cells,I,M,N,Square):-
    extract(M,Cells,Sq,_),
    (extract(N,Cells,_,Rest), !;
     true),
    I1 is I + 1,
    pick_square(Rest,I1,M,N,Uare),
    append(Sq,Uare,Square).

column([],_,_,[]).
column(Code,M,N,[A|R]) :-
    element(M,Code,A),
    extract(N,Code,_,RestCode),
    column(RestCode,M,N,R).

extract(N,Code,First,Last) :-
    length(First,N),
    append(First,Last,Code).

```

A.2 N-Queens model

```

queens(N, Queens) :-
    length(Queens, N),
    domain(Queens,1,N),
    constrain(Queens),
    labeling([ff], Queens).

constrain(Queens) :-
    all_distinct(Queens),
    diagonal(Queens).

diagonal([]).
diagonal([Q|Queens]) :-
    safe(Q, 1, Queens),
    diagonal(Queens).

```

```

safe( _,_, []).
safe(X,D,[Q|Queens]) :-
    nonattacK(X,Q,D),
    D1 is D+1,
    safe(X,D1,Queens).

nonattacK(X,Y,D) :-
    X + D #\= Y,
    Y + D #\= X.

```

A.3 Golomb rulers model

```

golomb( Order, Length, Marks, Order ):-
    length( Marks, Order ),
    DiffNumber is (Order*(Order-1))/2,
    length( Diff, DiffNumber ),
    domain( Marks, 0, Length ),
    domain( Diff, 0, Length ),
    constrain_differences( Diff, Marks ),
    all_different( Diff ),
    break_simmetries( Marks ),
    labeling( [ffc], Marks ).
golomb( _, _, fail, _ ).

constrain_differences( Diff, [Hm|Tm] ):-
    constrain_mi( Diff, Hm, Tm, RemainingDiff ),
    constrain_differences( RemainingDiff, Tm ).
constrain_differences( [], _ ).

constrain_mi( [Hd|Td], Mi, [Mj|Tm], RemainingDiff ):-
    Hd #= Mj - Mi,
    constrain_mi( Td, Mi, Tm, RemainingDiff ).
constrain_mi( Td, _, [], Td ).

break_simmetries( Tm ):-
    Tm = [0|_],
    increasingMarks( Tm ),
    first_last_marks( Tm ).

increasingMarks( [ H1m , H2m | Tm ] ):-
    H1m #< H2m,
    increasingMarks( [ H2m | Tm ] ).
increasingMarks( [ _ ] ).

first_last_marks( Marks ):-
    findFirsts( Marks, F1, F2 ),
    findLasts( Marks, L1, L2 ),
    F2 - F1 #< L2 - L1.

findFirsts( [F1,F2|_] , F1, F2 ).

```

```
findLasts( Marks, L1, L2 ):-  
    reverse( Marks, Rmarks),  
    findFirsts( Rmarks, L2, L1).
```

A.4 Knapsack model

```
knapsack(Space,Profit) :-  
    inputdata(Weights,Costs),  
    length(Weights,N),  
    length(Vars,N),  
    domain(Vars,0,Space),  
    scalar_product(Weights,Vars,#=<,Space),  
    scalar_product(Costs,Vars,#>=,Profit),  
    labeling([ff],Vars),  
    write(Vars),nl.
```

Generalizing Finite Domain Constraint Solving

Federico Bergenti, Alessandro Dal Palù, and Gianfranco Rossi

Dipartimento di Matematica
Università degli Studi di Parma
Viale G. P. Usberti, 53/A
43100 Parma, Italy

`{federico.bergenti,alessandro.dalpalu,gianfranco.rossi}@unipr.it`

Abstract. This paper summarizes a constraint solving technique that can be used to reason effectively in the scope of a constraint language that supersedes common finite domain languages available in the literature. The first part of this paper motivates the presented work and introduces the constraint language, namely *Hereditarily Finite Sets (HFS)* language. Then, the proposed constraint solver is detailed in terms of a set of rewrite rules which exploit finite domain reasoning within the HFS language. The presented approach achieves good efficiency without losing the desired correctness and completeness properties that other solvers for HFS provide.

1 Introduction and Motivation

Finite Domain (FD) [5] constraint solvers have been effectively applied to a great variety of problems in different application domains. Unfortunately, FD constraint solving typically suffers from an inherent weakness from the point of view of *(i)* expressive power and *(ii)* required a priori knowledge.

From the point of view of expressive power, FD constraint languages force the domains of variables to *bounded intervals* (subset of a discrete universe, often \mathbb{Z}), which may lead to severe limitations in real-world applications where domains of variables are often sparse sets, possibly of structured entities [12]. As a matter of fact, the use of intervals introduces an approximation in the representation of domains that may result in reduced effectiveness of constraint solving. Some constraint solvers address this issue and provide a means to deal with non-interval domains, e.g., *(i)* FD set terms [20] of SICStus allow modeling domains in terms of finite unions of intervals, and *(ii)* GNU Prolog transparently switches between sparse and interval domain representations [5].

Furthermore, forcing domains in \mathbb{Z} inhibits a natural representation of structured knowledge. The recent introduction of *Finite Set (FS)* constraints [11] aims at overcoming such limitations because FS constraints model domains as collections of known sets. Such domains are normally expressed in terms of *set intervals* $[l..u]$ where l and u are known sets, usually of integers. A set interval $[l..u]$ represents the set of all subsets of u that contain l , i.e., $[l..u] = \{x : x \subseteq u \wedge l \subseteq x\}$. Many constraint solvers (including many CLP systems) now offer FS constraints, e.g., ECLiPSe [14], Oz [19] and B-Prolog [22].

FS constraints are explicitly derived from FD constraints and therefore they retain the appreciated efficiency of the latter, while inheriting many of their problems and limitations. In particular, available FS constraint solvers, e.g., Conjunto [11] and Cardinal [2], treat efficiently only interval domains that are defined as the convex closure of a collection of elements. For example, given a variable whose domain is $\{\{a, b\}, \{a, c\}, \{d\}\}$, the corresponding interval domain is the set interval $[\{\}\dots\{a, b, c, d\}]$.

Besides the mentioned expressive power limitations, it is also worth discussing another important limitation of FD-like constraint languages. The practice of using FD-like reasoning in knowledge representation has shown that domains are often unknown prior to reasoning and an important side effect of reasoning is revealing the shape of domains. In many real-world cases, domains are not available prior to computation, and they have to be acquired and/or computed; anyway, constraints over them are often known in advance. Mentioned situations are not really tractable with FD-like languages because their solvers typically attach a possibly implicit (and redundant) domain to each variable before processing constraints.

The difficulties connected with required a priori knowledge in FD-like reasoning are explicitly tackled by [10], which extends $\text{CLP}(\mathcal{FD})$ to deal with incomplete knowledge on domains. The proposed system is based on the *Interactive CSP* approach [6] and it uses domains as communication channels between the $\text{CLP}(\mathcal{FD})$ solver and an acquisition system.

The mentioned limitations in terms of expressive power and required a priori knowledge do not reduce the notable importance of FD-like constraint languages and solvers for their proved effectiveness in handling a great variety of problems. Nonetheless, the urge for expressive power to support modeling of complex domains that real-world applications require justifies the definition of constraint languages and solvers that trade-off efficiency with expressive power and completeness, e.g., $\text{CLP}(\mathcal{SET})$ [9] and CLPS [4].

In particular, $\text{CLP}(\mathcal{SET})$ provides a constraint language that subsumes most of the mentioned FD-like languages while delivering a correct and complete constraint solver, at the cost of notable inefficiency. More in details, $\text{CLP}(\mathcal{SET})$, and its Java porting JSetL [18], support modeling domains in terms of generic extensional sets, usually called *Hereditarily Finite Sets (HFS)*, that contain any kind of object (and nested finite sets in particular). Moreover, such sets can be constructed dynamically by means of common set operations and it is worth mentioning that constraint solving can take place even over partially or totally unspecified sets.

Unfortunately, the constraint solver of $\text{CLP}(\mathcal{SET})$ does not exploit the information that the domain of variables provides, and it does not even explicitly express that a variable may have a domain. This leads to a very weak form of propagation and most of constraint solving is basically a *generate & test*. For example, given the following problem $x \in \{1, 2, 3, 4, 5\} \wedge x \neq 10$, the $\text{CLP}(\mathcal{SET})$ solver enumerates all possible values of x before asserting that $x \neq 10$ holds.

While the expressive power and the computational features of $\text{CLP}(\mathcal{SET})$ make it a good candidate to deliver a constraint language capable of addressing the mentioned issues of FD-like languages and solvers, the inefficiencies of available implementations of $\text{CLP}(\mathcal{SET})$ prohibit its instant use in many problems where FD-like solvers proved their relevance. A step towards a more effective use of the $\text{CLP}(\mathcal{SET})$ language is represented by $\text{CLP}(\mathcal{SET}, \mathcal{FD})$ [7] which integrates FD constraints into $\text{CLP}(\mathcal{SET})$ thus allowing efficient processing of the former through the use of an embedded FD solver. Advantages, however, are limited to FD constraints only: $\text{CLP}(\mathcal{SET}, \mathcal{FD})$ still retains the overall constraint solving technique of $\text{CLP}(\mathcal{SET})$, with no possibility to exploit domain information except for the handling of FD constraints.

The overall goal of this paper is to combine the flexibility and expressive power of $\text{CLP}(\mathcal{SET})$ -like languages with the efficiency of FD-like languages, in a more general manner. In order to achieve this goal, we propose to integrate FD, FS and $\text{CLP}(\mathcal{SET})$ constraints into a single, uniform constraint language and to extend the domain-driven constraint solving techniques of FD and FS to the whole solver. For this reason, we develop a new (set) constraint solver that replaces the standard $\text{CLP}(\mathcal{SET})$ solver and that exploits the information on domains of variables to obtain improved efficiency.

The resulting language and solver can be viewed as a generalization of the languages of FD and FS constraints, in which domains are allowed to be general sets, containing elements of any kinds, possibly nested and partially specified.

On the other hand, the work described in this paper can be viewed as a generalization of [7], that allows retaining the expressive power of $\text{CLP}(\mathcal{SET})$ while permitting the efficient processing of a larger class of $\text{CLP}(\mathcal{SET})$ constraints.

The paper is organized as follows. Next section presents the language of HFS-constraints, focusing on the notion of variable domain. In Section 3, we first present the architecture of the proposed constraint solver, and then we detail the solver in terms of a set of rewrite rules which exploit FD-like reasoning within the HFS language. Finally, in Section 4 we summarize the current state of the implementation of our proposal and we point out some future work.

2 The HFS Constraint Language

This section details the syntax and semantics of the constraint language that we consider in this work. It also highlights the peculiar features that the language provides for treating domains of variables, which we exploited in the constraint solver reported in the next section.

The language described here is a minor extension of the constraint language provided by the CLP language $\text{CLP}(\mathcal{SET})$, and the Java library JSetL, which delivers (most of) $\text{CLP}(\mathcal{SET})$ facilities for set solving in an object-oriented language framework.

2.1 Syntax and semantics

As usual, the syntax of the language is defined by its *signature* Σ that is a triple $\langle \mathcal{V}, \mathcal{F}, \Pi \rangle$ where \mathcal{V} is a fixed finite set of variables, \mathcal{F} is the set of constant and function symbols, and Π is the set of *constraint* predicate symbols allowed in the language.

The set \mathcal{F} of constant and function symbols is

$$\{\emptyset, \text{ins}, \text{int}\} \cup Z \cup F_Z \cup F_U$$

where: \emptyset , ins , and int are the *set constructors*; Z is the denumerable set of constants representing the integer numbers, i.e., $Z = \{0, -1, 1, -2, 2, \dots\}$; F_Z is a set of function symbols representing operations over integer numbers, such as $+$, $-$, $*$, div , mod ; F_U is a (possibly empty) set of user-defined constant and function symbols.

The set Π of constraint predicate symbols is

$$\{=\} \cup \Pi_S \cup R \cup \{\text{set}, \text{integer}\} \cup \text{Neg}$$

where: Π_S is a set of predicate symbols representing the usual set-theoretic operations, such as \in , \subseteq , union , inters , $\|$, diff , size ; R is a set of predicate symbols representing the usual comparison relations over integer numbers, such as \leq , $>$; Neg is a set of predicate symbols representing the negated counterparts of most of other predicate symbols, such as \neq , \notin , not_integer .

A *primitive HFS-constraint* is any atomic predicate built using the symbols from the signature. A non-primitive HFS-constraint is a conjunction of primitive HFS-constraints.

The intuitive semantics of the various symbols is as follows¹. The symbol \emptyset represents the empty set. $\text{ins}(t, s)$ represents the set composed of the element t union the elements of the set s , i.e., $\{t\} \cup s$. For example, $\text{ins}(1, s)$, where s is an uninitialized variable, represents the (unbounded) set $\{1\} \cup s$. $\text{int}(a, b, s)$ represents the set composed of the elements in the interval $[a, b]$ union the elements of the set s . If a, b are integer constants, $[a, b]$ is the set of integers $\{x : x \geq a \wedge x \leq b\}$; if a, b are ground terms denoting sets, $[a, b]$ is the set of sets $\{x : a \subseteq x \wedge x \subseteq b\}$, that is the bounded lattice induced by the subset relation \subseteq having a as its greatest lower bound and b as its least upper bound.

It is worth noting that ins and int are different constructors for terms denoting the same notion of set. Thus, terms constructed using ins and int can be tested, e.g. for equality, and they can be combined. For example, $\text{ins}(1, \text{int}(10, 20, \text{ins}(100, \emptyset)))$ represents the set containing all integers between 10 and 20 with the integers 1 and 100. We call a term built using the set constructors ins and int an *extended set term*.

The predicates $=$ and \in represent the equality and the membership relationships, respectively; the predicate union represents the union relation: $\text{union}(r, s, t)$

¹ A formal discussion of these concepts is out of the scope of this paper and can be found in [7]

holds iff $t = r \cup s$; the predicate \parallel represents the disjoint relationship between two sets: $s \parallel t$ holds iff $s \cap t = \emptyset$; and so on.

Symbols in Z are mapped to the elements of \mathbb{Z} , while functions in F_Z and the predicate symbols \leq, \geq , etc. are mapped to functions and relations over \mathbb{Z} in the natural way.

We say that a HFS-constraint containing only predicates taken from $\{=, \neq, \in, \notin, \text{union}, \parallel, \leq, \text{size}, \text{set}, \text{integer}\}$ is in *canonical form*. As a matter of fact, [9] shows that all other predicates in Π can be defined as non-primitive constraints using the above together with the HFS theory.

Sets are defined by means of the set constructors `ins` and `int` as shown above. For the sake of simplicity, however, we will often use the following more convenient notations:

$$\{t_1 \mid t\}$$

as a shorthand for

$$\text{ins}(t_1, t)$$

and

$$\{t_1..t_2 \mid t\}$$

as a shorthand for

$$\text{int}(t_1, t_2, t)$$

writing $\{t_1\}$ and $\{t_1..t_2\}$, respectively, when t is the empty set. This notation is easily extended to the case of n elements (possibly mixing the `ins` and `int` constructors, e.g., $\{1, 10..20, 100\}$). With a little abuse of terminology, we call a syntactic object of the form $t_1..t_2$ a *range term*.

Various concrete language facilities are provided to make set definition simpler. For example, in JSetL, we can provide an array of elements when creating a set object; furthermore, an interval of integers can be specified at set creation by giving the lower and upper bounds of the interval. In $\text{CLP}(\mathcal{SET})$, a special syntax is provided to denote a set obtained by n applications of the set constructor `ins` as a list of n elements with curly brackets, as in the usual mathematical notation.

Elements of a set can be values of any type (not necessarily homogeneous), including logical variables and other sets. Hence, sets can be *nested* at any level, e.g., $\{1, \{\emptyset, \{a\}\}, \{\{\{b\}\}\}$. Moreover, sets can be *partially specified*, i.e., they can contain uninitialized logical variables in place of single elements.

It is worth noting that we assume that the constraints of the language deal naturally with set terms made of single elements and intervals, possibly mixed together. For example:

- $13 \in \{1..10, 15, 20..100\}$
- $\text{union}(\{1, 5, 7\}, \{3..6\}, R)$
- $\{1, 3, x\} = \{1..3\}$

are all admissible HFS-constraints, whose solved forms are: `false`, $R = \{1, 3..7\}$, and $x = 2$, respectively. The availability of extended set terms is particularly useful to represent non-interval sets, e.g., the ones obtained from the constraint $\text{diff}(\{1..10\}, \{5\}, R)$, whose solved form is $R = \{1..4, 6..10\}$.

Actually, extended set term management represents a straightforward extension of the set representation and manipulation facilities provided by CLP(\mathcal{SET}), JSetL, and other available FD/FS constraint solvers.

2.2 Domains

Logical variables can range over the domain of HFS (*set variables*), as well as over the domain of integers (*integer variables*), and, for equality and membership constraints only, over the domain of all possible values.

Domains can be specified as HFSs and associated to variables through membership constraints, e.g., $x \in d$, where d , the domain of x , is an extended set term. This turns domains into first class abstractions of the language and naturally equips them with common set operations, thus generalizing domains from bounded intervals to very general, unbounded and potentially sparse sets.

More in detail, domains can be specified either as sets or as intervals. In the first case, the domain can be given either by enumerating all its values or it can be constructed as the result of some set operation. Moreover, the domain can be partially specified. For example, $x \in \{1, 3, 5, 7, 9\}$ states that the domain of the integer variable x is the set of the odd natural numbers less than 10, whereas $x \in \{1 \mid s\}$, with s a set variable, states that the domain of x is an unbounded set containing the value 1 plus something else not yet specified. Similarly, $r \in \{\{1\}, \{2\}, \{3\}, \{1, 2, 3\}\}$ precisely defines the domain of the set variable r as a set of four different sets.

When the domain is specified as an interval, the interval itself can be given either by specifying its lower and upper bounds, $\text{int}(a, b)$, or it can be constructed as the result of some set operation, where a and b can be either integer constants (for integer variables), or known sets (for set variables). For example, the constraint $r \in \{\{1\}.. \{1, 2, 3\}\}$ states that the values of r can be: $\{1\}$, $\{1, 2\}$, $\{1, 3\}$, $\{1, 2, 3\}$.

A domain d is said a *closed domain* if any element t in d is completely known (i.e., it is denoted by a ground term) or all variables possibly occurring in t have a closed domain attached; otherwise, d is an *open domain*. For instance, $\{1, x\}$, where x is an integer variable with domain $\{1..100\}$, is a closed domain. Conversely, $\{1, x\}$, with $x > 10$, and $\{1 \mid s\}$, with s a set variable with no attached domain, are (resp., bounded and unbounded) open domains. A closed domain whose elements are integer numbers is said an *integer domain*, while a closed domain whose elements are known sets is said a *set domain*.

3 Constraint Solving

3.1 The Overall Solver Architecture

In order to enhance effectiveness and efficiency of the constraint solving process, we decided to structure the proposed constraint solver into a *layered architecture*. Each layer achieves a different level of consistency at a different cost. Lower levels

are considered more effective in proving inconsistency and reducing the search space than upper levels.

More precisely the solver is structured as follows:

- Level 1 – Single constraints are processed to generate *canonical constraints* by means of deterministic rewrite rules only. Rules are triggered by the type of the primitive constraint to be handled. A very weak form of propagation of constraints is performed at this level: variable substitution only.
- Level 2 – Deterministic rules involving pairs of constraints are applied. These are all propagation rules that exploit the knowledge about variable domains, whenever possible, to reduce the size of domains or to reveal an inconsistency, according to the FD and FS approaches.
- Level 3 – Nondeterministic and labeling rules are applied. This level is highly nondeterministic and it includes the labeling of variables whose domains are known. Such level 3 rules are mostly taken from the available implementations of CLP(\mathcal{SET}) solvers.

The overall solving process repeatedly exploits all applicable rules at a given level until a fixpoint is reached, before passing to an higher level. If no new constraints are added to the constraint store, the process steps forward to an higher lever, otherwise it restarts from level 1.

The results of levels 1 and 2 are *simplified forms* of the original constraint that we cannot prove satisfiable, much like the output of FD-like constraint solvers. Conversely, the output of level 3 enjoys the same desirable characteristics of constraint solving in CLP(\mathcal{SET}), i.e., correctness and completeness, provided that all variables in the CSP have a closed domain attached. In fact, the result of level 3 is a finite collection $\{C_1, \dots, C_k\}$ of constraints in *solved form* [9] which is guaranteed to be satisfiable. Moreover, the disjunction of all the constraints in solved form generated by the solver is equisatisfiable to the original constraint.

The achieved improvement of effectiveness mainly derives from the fact that the proposed layered architecture allows the user deciding which level of consistency he/she wants to achieve. As a matter of fact, the user is free to choose to (i) stop at any of the first two levels, with no satisfiability warrants, or (ii) select which (if any) variable to label if the corresponding domain is known, or optionally (iii) skip level 2 in order to avoid constraint propagation, thus letting more time to be spent in searching the solution space rather than in enforcing consistency. This last choice is what current implementations of CLP(\mathcal{SET}) provides, i.e., they skip over most of the rules that we now put in level 2.

The foreseen improvement of efficiency mainly relies on the decision of postponing all (costly) nondeterministic rules at the end of the constraint solving cycle. Generally speaking, the split of rules into deterministic and nondeterministic promotes efficiency over, e.g., CLP(\mathcal{SET}) and JSetL, because the solver is given more chance to detect inconsistency before branching any nondeterministic choice.

Finally, it is worth noting that the rewrite rules used by FD-like constraint solvers to process constraints can be entirely expressed in terms of set constraints

over (the sets that represent) the domains of variables. For example, domain reduction associated to a constraint like $x \in d \wedge x \in d'$ can be expressed by the CLP(\mathcal{SET}) constraint $x \in D \wedge \text{inters}(d, d', D)$. Thus, we can concentrate on providing efficient implementations of set constraints especially in cases where sets are completely specified, e.g., where sets represent FD or FS domains. The same set constraints, however, apply naturally also to all other cases (e.g., non-interval domains), though they are treated, in general, less efficiently.

3.2 Constraint Solving Rules

The rest of this section presents a taxonomy of the rewrite rules we employed for constraint solving. For each level of the solver, we detail the classes of rewrite rules that we used at that level, and we exemplify some rule for each class.

Many of the rewrite rules of our solver are directly derived from CLP(\mathcal{SET}), and they have been (i) adapted to cope with extended set terms, and (ii) specialized to exploit interesting properties of special cases (e.g. ground sets). Moreover, some new rules have been introduced to deal with integer and set domains with FD- and FS-like approaches. Finally, some other rules have been introduced to handle cardinality constraints, much like in Cardinal [2].

Most of the rules are direct application of the classic set theory adapted to support HFS [9].

The rules are presented as deterministic rewrite rules that operate when respective pre-conditions are met:

$$\frac{\text{pre-conditions}}{\{C_1, \dots, C_n\} \rightarrow \{C'_1, \dots, C'_m\}}$$

where C_1, \dots, C_n and C'_1, \dots, C'_m ($n, m \geq 0$) are primitive HFS-constraints in the constraint store, and $\{C_1, \dots, C_n\} \rightarrow \{C'_1, \dots, C'_m\}$ represents the changes in the constraint store caused by the application of the rule.

In the rest of this section, we adopt the following notation:

- t, t_i : any term (either ground or not);
- c, c_i : any ground term (either integer or not);
- i, i_i : integer constants;
- v, v_i : any non-ground term;
- X, Y, Z, N, M, \dots (uninitialized) variables; and
- s, r, s_i : any set (either ground or not).

We also introduce the function $\text{dom}(X)$, which returns the closed domain currently associated with X , or $-\infty.. +\infty$ if the domain of X is open or X has no domain attached. The details on how $\text{dom}(\cdot)$ is concretely implemented is out of the scope of this paper, but it is worth noting that the domain of a variable can be equally stored as an attribute of the variable itself, or as an appropriate constraint that links a variable to its domain.

$\text{dom}(X)$ is defined for any X and we need to complement it with two other predicates to perform the common task of deciding whether X is an integer variable or a set variable. In particular, $\text{int_dom}(d)$ holds if d is an *integer domain*, and $\text{set_dom}(d)$ holds if d is a *set domain*.

Level 1

This level performs a simplification of constraints by means of six classes of rewrite rules.

Ground cases – Solve constraints whose arguments are all ground terms. For example:

– *set membership test*

$$\frac{c \neq c_1 \text{ and } \dots \text{ and } c \neq c_n, c_i \text{ not range terms}}{\{c \in \{c_1, \dots, c_n \mid X\}\} \rightarrow \{c \in X\}} \quad (1)$$

– *interval membership test*

$$\frac{i \geq i_1, i \leq i_2}{\{i \in \{i_1..i_2\}\} \rightarrow \{\}} \quad (2)$$

Similar rules apply to other constraints that deal with both integer and set intervals, such as union, intersection and difference. These operations are fundamentals for the implementation of the FD and FS constraint solving techniques as illustrated for instance in [11] and [2]. For this reason it is crucial that these rules are implemented as efficiently as possible. This is the case, e.g., of $\text{inters}(1..10, 5..20, D)$, which is easily solved because both intervals are completely known. Conversely, a constraint like $\text{inters}(\{1, X\}, \{2, Y\}, D)$ is managed with the more general, though less efficient, rules of level 3.

Special cases – Identify simple cases, in which not all terms are necessarily ground, e.g, one term is ground and/or one term is a set whose elements are all variables and/or terms are singleton sets. For example:

– *empty set*

$$\overline{\{t \in \emptyset\}} \rightarrow \text{fail} \quad (3)$$

– *singleton set*

$$\frac{t_2 \text{ not a range term}}{\{t_1 \in \{t_2\}\} \rightarrow \{t_1 = t_2\}} \quad (4)$$

Generation of canonical constraints – Primitive constraints that are not in a canonical form and that can not be further simplified are rewritten to semantically equivalent canonical constraints. For example:

– *subset*

$$\overline{\{X \subseteq Y\}} \rightarrow \{\text{union}(X, Y, Y)\} \quad (5)$$

Inference of types – Provide additional constraints to ensure type coherence. For example:

– *integer variable*

$$\frac{\text{int_dom}(s)}{\{X \in s\} \rightarrow \{X \in s, \text{integer}(X)\}} \quad (6)$$

Inference of cardinalities – Provide additional constraints to ensure coherence of the cardinality of sets. For example:

– *set variable*

$$\frac{\text{set_dom}(\{s_1 .. s_2\}), |s_1| = a, |s_2| = b}{\{X \in \{s_1 .. s_2\}\} \rightarrow \{X \in \{s_1 .. s_2\}, \text{size}(X, N), N \in \{a .. b\}\}} \quad (7)$$

Inference of domains – Gather information on domains that is not explicitly provided. For example:

– *cardinality of a partially specified set*

$$\frac{\begin{array}{l} m = \text{number of distinct ground terms in } \{t_1, \dots, t_n\} \text{ or} \\ m = 1 \text{ if } t_1, \dots, t_n \text{ are all non-ground terms} \end{array}}{\{\text{size}(\{t_1, \dots, t_n\}, N)\} \rightarrow \{N \in \{m .. n\}\}} \quad (8)$$

For example, $\text{size}(\{X, Y, 1\}, N)$ generates $N \in \{1..3\}$, while $\text{size}(\{1, 2, 3\}, N)$ is rewritten to $N \in \{3..3\}$, i.e., $N = 3$.

All cases that are not tackled by the mentioned classes of rules are considered *irreducible* at level 1 and they are therefore shipped to level 2. In particular, constraints like $t \in \{t_1, \dots, t_n \mid s\}$, with $n \geq 1$, s either variable or \emptyset , and either t or t_1, \dots, t_n non-ground terms are passed unaltered to level 2 (note that constraints of the form $X \in \{t_1, \dots, t_n \mid s\}$ represent sort of domain declarations that are managed at level 2).

Remark 1. The outcome of level 1 is in canonical form, which eases the tasks of level 2 because it can address only pairs of constraints in canonical form. While this characteristic of the output of level 1 is crucial to limit the potential explosion of rules at level 2, on the other hand it may introduce some inefficiencies in the overall constraint solving process. As a matter of fact, the relation represented by a non-canonical constraint may become no longer evident after the constraint have been replaced by the (equivalent) conjunction of primitive constraints in canonical form. Hence, some optimizations and/or inferences that apply to the original constraint are hardly applicable to the transformed constraint. For example, the non-canonical constraint $\text{inters}(X, Y, Z)$ is replaced by the equivalent canonical constraint $\text{union}(A, Z, X), \text{union}(B, Z, Y), A \parallel B$; if after this replacement, X, Y, Z get instantiated to ground set terms, the solver can not apply the efficient processing rules provided for dealing with intersection of ground sets. The identification of the appropriate tradeoff between reducing the number of cases to be managed by the solver's upper levels and delaying as long as possible elimination of non-canonical constraints is left as an open problem for future work.

Level 2

Rules at this level are mainly intended to perform an FD-like reasoning on domains of variables. This level provides also rules to perform simple consistency checks between pairs of constraints (e.g., between type constraints).

Domains management – Add/remove elements from the domains of variables, merge sparse domain information of single variables, test for domain membership. For example:

$$- \text{domain update} \quad \frac{\text{set_dom}(d), \text{update_dom}(d, c) = d'}{\{c \in X, X \in d\} \rightarrow \{X \in d'\}} \quad (9)$$

where $\text{update_dom}(d, c)$ returns the subset d' of the set domain d whose elements contain the constant element c , i.e.,

$$d' = \{s : s \in d \wedge c \in s\}.$$

For example:

- $\text{update_dom}(\{\emptyset..1, 2, 3\}, 1) = \{\{1\}..1, 2, 3\}$
- $\text{update_dom}(\{\emptyset..1, 2, 3\}, 4) = \emptyset$
- $\text{update_dom}(\{\{1, 2\}, \{1, 3\}, \{2, 3\}\}, 1) = \{\{1, 2\}, \{1, 3\}\}$
- $\text{update_dom}(\{\{1, 2\}, a\}, 1) = \{\{1, 2\}\}$.

Similarly, the processing of disequalities may cause elements to be removed from a domain:

$$\frac{s, t \text{ ground}}{\{X \neq t, X \in s\} \rightarrow \{X \in D, \text{diff}(s, \{t\}, D)\}} \quad (10)$$

In particular, when s is an integer interval $\{i_1..i_2\}$ and t is an integer belonging to the interval, distinct from i_1 and i_2 , the resulting new domain D is $\{i_1..t-1, t+1..i_2\}$.

– *domain merge*

$$\frac{r, s \text{ ground}}{\{X \in s, X \in r\} \rightarrow \{X \in D, \text{inters}(s, r, D)\}} \quad (11)$$

Rules (10 and (11) are applicable for any (ground) s and r , which requires diff and inters to work equally with interval and non-interval sets. As an example, $\text{inters}(\{\{1\}..1, 2, 3\}, \{\{1, 2\}, \{2, 3\}\}, X)$ holds for $X = \{\{1, 2\}\}$.

– *domain test*

$$\frac{c \notin \text{dom}(X_k) \text{ and } \dots \text{ and } c \notin \text{dom}(X_n)}{\{c \in \{X_1, \dots, X_{k-1}, X_k, \dots, X_n\}\} \rightarrow \{c \in \{X_1, \dots, X_{k-1}\}\}} \quad (12)$$

This rule removes all variables whose domains do not contain c . It is worth noting that this rule has an interesting special case for $k = 1$ that leads to fail via $\{c \in \{X_1, \dots, X_n\}\} \rightarrow c \in \emptyset$.

As an example of application of rule (12), which shows some improvements of our approach on $\text{CLP}(\mathcal{SET})$ (and $\text{CLP}(\mathcal{SET}, \mathcal{FD})$), let us consider the following HFS-constraint:

$$R = \{X1, X2, X3, X4, X5\}, 15 \in R, R \subseteq \{1, 3, 5, 7, 9, 11\}$$

After propagating equality for R over the other two constraints through variable substitution, the solution of the \subseteq constraint forces all variables $X1, X2, X3, X4, X5$ to get the set $\{1,3,5,7,9,11\}$ as their domain; then the constraint $15 \in \{X1, X2, X3, X4, X5\}$ is reduced by rule (12) to $15 \in \emptyset$, hence to **false**. Note that $\text{CLP}(\mathcal{SET})$ addresses this constraint with a *generate & test* approach—generates all possible values for R and then test them using the \in constraint.

Remark 2. The condition that the sets involved in rules (10) and (11) must be ground is not strictly necessary. Allowing these rules to be applied even to more general cases, however, may cause troubles to arise:

- with termination of the constraint solving algorithm;
- with efficiency of the rewriting process, since the solution of constraints like diff and inters applied to general sets may generate (through non-determinism) a large number of (possibly redundant) solutions.

For these reasons we prefer for now to restrict applicability of these rules to the ground case only. A more precise analysis of cases in which it may be convenient to allow non-ground cases to be dealt with is left for future work. As an example of one such case, consider the constraint $X \in \{1, 3, 5, Y\}$ which states that the domain of X contains the values 1, 3, and 5, and possibly another value Y which is left unspecified for the moment. Assume that the constraint store contains also the constraint $X \neq 3$. If we remove the groundness condition of rule (10), then we can apply this rule and we get the new constraints $X \in \{1, 5, Y\}, Y \neq 3$. Thus, application of rule (10) allows us to get a reduced domain for X although it is not completely specified.

Type clash – Identify clashes in type constraints, e.g., $\text{set}(X) \wedge \text{integer}(X)$ immediately fails.

Size management – Make sure the cardinality of a set is always unique:

- *size uniqueness*

$$\frac{}{\{\text{size}(s, N), \text{size}(s, M)\} \rightarrow \{\text{size}(s, N), N = M\}} \quad (13)$$

Besides the mentioned cases $X \in s \wedge X \in r$ and $X \in s, X \neq t$, level 2 deals with all combinations of canonical constraints not eliminated at level 1 that share some variable. In particular, level 2 treats the combinations of $X \in s$ with:

- $X \notin r$;
- $X \leq e, \text{size}(s, X)$ (X integer variable);
- $\text{union}(X, Y, Z), \text{union}(Y, Z, X), X \parallel Y$ (X set variable).

Moreover, it is worth noting that the case $X \in s \wedge X \leq e$, with e arithmetic expression, fires the application of common FD-like rules that cause an FD-like propagation.

Finally, the rules of level 2 can exploit an interesting optimization of implementation. Most of reasonable implementations of the proposed constraint solving approach would likely store all knowledge about the domain of X in an attribute of X itself. Therefore, rules of level 2 may exploit this to avoid scanning of the entire constraint store while looking for constraints of the form $X \in d$ in order to associate X with its domain.

Level 3

This level groups all rewrite rules that involve nondeterminism. In particular rules dealing with membership constraints of the form $X \in s$ allow forcing assignment to the specified variables of values from their domains, leading to a chronological backtracking search of the space of solutions.

Nondeterministic choices – Open nondeterministic branches. For example:

– (*set inequality*)

$$\frac{}{\{s \neq r\} \rightarrow \{Z \notin s, Z \in r\} \text{ or } \{s \neq r\} \rightarrow \{Z \in s, Z \notin r\}} \quad (14)$$

where s and r are (any) sets.

Enumeration – Label variables with appropriate values. For example:

– (*set membership*)

$$\frac{t_1 \text{ not a range term}}{\{X \in \{t_1 \mid s\}\} \rightarrow \{X = t_1\} \text{ or } \{X \in \{t_1 \mid s\}\} \rightarrow \{X \in s\}} \quad (15)$$

Note that other forms of solution enumeration are obtained from the treatment of other constraints such as $\text{union}(X, Y, \{1, 2, 3\})$ which assigns to X and Y all possible subsets whose union yields $\{1, 2, 3\}$.

From a pragmatic point of view, the treatment of this kind of constraints, that leads to a generalized notion of labeling, could be explicitly activated or deactivated by the user on request.

4 Discussion

The presented work has been mainly performed on a theoretical basis and just some preliminary experiments are available. During the preliminary inclusion of HFS in a practical solver, we faced some technical difficulties.

The architecture of the solver that we sketched in Section 3 clearly shows that the solver embeds FD and FS constraint solving as particular cases. Actually, one of the main duties of level 2 is to identify FD and FS sub-problems and treat them accordingly. At the implementation level, this can be obtained either by a single solver with specialized subparts or via solver cooperation [13], e.g. using a master/slave architecture as discussed in [3], where a master solver invokes one or more existing slave solvers when needed.

The master/slave approach has been adopted in the $\text{CLP}(\mathcal{SET}, \mathcal{FD})$ proposal [7] where FD constraint solving is made available within $\text{CLP}(\mathcal{SET})$ by exploiting the facilities of an existing FD solver (specifically the FD solver of SICStus in the current implementation). Conversely, the integration of FD constraints within JSetL, our Java implementation of $\text{CLP}(\mathcal{SET})$, is based on a single solver in which rewrite rules dealing with FD constraints have been developed from scratch and embedded in the general solver that deals with all other (set) constraints.

The constraint technique described in this paper has not been fully implemented yet. However, previous experiences with the integration of FD and CLP(\mathcal{SET}), both within the Prolog interpreter `{log}` [17] and within the Java library JSetL, gave us some feedback about the feasibility and the possible performance improvements of this approach. In particular, the current version of JSetL, named JSetL(\mathcal{FD}) [1, 16], provides most of the CLP(\mathcal{SET}) facilities together with basic FD constraint solving facilities. It also supports the constraint size and it provides an implementation of the constraint `all_different` based on the well known approach of Hall sets [15].

The preliminary experiments that we performed using this implementation show that the use of FD techniques within our solver strongly enhances its performances on problems that can modeled as simple FD problems. This confirms the encouraging results obtained with CLP(\mathcal{SET} , \mathcal{FD}) and documented in [8].

Table 1 shows the results we obtained using JSetL(\mathcal{FD}) on the standard n -queens problem. On this problem JSetL(\mathcal{FD}) exhibits performances that are comparable to (or even better than) those of existing FD solvers, in particular the one provided by SWI Prolog [21].

n	Count of solutions	JSetL(\mathcal{FD})	JSetL(\mathcal{FD})	SWI Prolog
		Binary decomposition	<code>all_different</code>	
7	40	250ms	187ms	510ms
8	92	1,109ms	442ms	2,750ms
9	352	5,703ms	1,640ms	13,810ms
10	724	30,375ms	5,687ms	75,520ms
11	2,680	243,047ms	20,750ms	43,150ms
12	14,200	> 3 min	105,391ms	> 3 min

Table 1. Enumeration of all solutions of the n -queens problem

For the near future, we plan to extend the current implementation JSetL(\mathcal{FD}) to completely include the constraint solving technique described in this paper. This will be achieved by:

- Extending the current representation of sets by allowing JSetL(\mathcal{FD}) to support extended set terms that mix single set elements and intervals;
- Providing efficient implementation of all constraints in the ground cases, including those intended to handle integer and set intervals;
- Implementing rules for FS constraints to effectively manage set domains, as provided, e.g., by `Conjunto` and `Cardinal`; and
- Structuring the whole solver in terms of the layered architecture described in Section 3.

References

1. AMADINI, R. (2007) *Definizione e trattamento del vincolo di cardinalità insiemistica nella libreria JSetL* Tesi di Laurea, Dipartimento di Matematica, Università di Parma.
2. AZEVEDO, F. (2007) Cardinal: A Finite Sets Constraint Solver, *Constraints*, 12(37):93–129.
3. BERGENTI, F., PANEGAI, E., AND ROSSI, G. (2006) A Master-Slave Architecture to Integrate Sets and Finite Domains in Java, Presented at CILC'06 – Convegno Italiano di Logica Computazionale, Bari.
4. BOUQUET, F., LEGEARD, B., AND PEUREUX, F. (2002) CLPS-B - a constraint solver for B. In *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 2280 of LNCS, Springer Verlag, pp. 188–204.
5. CODOGNET, P., AND DIAZ, D. (1996) Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3):185–226.
6. CUCCHIARA, R., GAVANELLI, M., LAMMA, E., MELLO, P., MILANO, M., PICCARDI, M. (online) Extending the CSP Model to Cope With Partial Information. Available at: <http://lia.deis.unibo.it/Research/Areas/icsp>
7. DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. (2003) Integrating Finite Domain Constraints and CLP with Sets. In D. Miller, ed., *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229.
8. DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. (2006) Constraint Logic Programming Language for Effective Programming with Sets and Finite Domains. Research Report “Quaderno del Dipartimento di Matematica”, 437, Università di Parma.
9. DOVIER, A., PIAZZA, C., PONTELLI, E., AND ROSSI, G. (2000) Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931.
10. GAVANELLI M., LAMMA E., MELLO P., AND MILANO, M. (2005) Dealing with incomplete knowledge on CLP(FD) variable domains, *ACM Transactions on Programming Languages and Systems*, 27(2):236–263.
11. GERVET, C. (1997) Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244.
12. GERVET, C. (2006) Constraints over Structured Domains In Rossi, F. van Beek, P., and Walsh, T. (Eds.), *Handbook of Constraint Programming*, Elsevier.
13. HOFSTEDT, P. (2000) Cooperating Constraint Solvers. In Dechter, R. (Ed.), *International Conference on Principle and Practice of Constraint Programming*, Vol. 1894 of LNCS, Springer Verlag, 520–524.
14. IC PARC (2003) *The ECLiPSe Constraint Logic Programming System*. London. www.icparc.ic.ac.uk/eclipse/.
15. LECONTE, M. (1996) A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning*, pp. 19–28
16. PANDINI, D. (2008) *Progettazione e realizzazione in Java di un risolutore di vincoli su domini finiti* Tesi di Laurea, Dipartimento di Matematica, Università degli Studi di Parma.
17. ROSSI, G. (2005) *The {log} Constraint Logic Programming Language*. prmat.math.unipr.it/~gianfr/setlog.Home.html.

18. ROSSI, G., PANEGAI, E., AND POLEO, E. (2007) JSetL: A Java Library for Supporting Declarative Programming in Java *Software-Practice & Experience*, 37:115-149.
19. VAN ROY, P. (ED.) (2005) Multiparadigm Programming in Mozart/Oz Lecture Notes in Computer Science 3389 Springer 2005, ISBN 3-540-25079-4.
20. SWEDISH INSTITUTE OF COMPUTER SCIENCE. *The SICStus Prolog Home Page*. www.sics.se.
21. WIELEMAKER, J. (2004) *SWI-Prolog Reference Manual (Version 5.4)*. University of Amsterdam.
22. ZHOU, N-F. (2005) *B-Prolog User's Manual (Version 6.8)*. Afany Software.