

Atti del convegno CILC 09

24-esimo Convegno Italiano di Logica Computazionale

Polo Scientifico-Tecnologico – Facoltà di Ingegneria

Università di Ferrara

Marco Gavanelli Fabrizio Riguzzi (ed.)

25 e 26 giugno 2009

Contents

PrettyProlog: A Java Interpreter and Visualizer of Prolog Programs <i>Alessio Stalla, Viviana Mascardi and Maurizio Martelli</i>	7
Exploiting Prolog and NLP Techniques for Matching Ontologies and for Repairing Correspondences <i>Viviana Mascardi, Angela Locoro, and Fabrizio Larosa</i>	12
Satisfiability Procedures for Combination of Theories Sharing Integer Offsets <i>Enrica Nicolini, Christophe Ringeissen, and Michaël Rusinowitch</i>	27
A Logic-Based, Reactive Calculus of Events <i>Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni</i>	42
A Non-monotonic Description Logic of Typicality <i>Laura Giordano, Valentina Gliozzi, Nicola Olivetti, and Gian Luca Pozzato</i>	57
Permissive nominal terms and their unification <i>Gilles Dowek, Murdoch J. Gabbay, and Dominic Mulligan</i>	70
HyLMoC A Model Checker for Hybrid Logic <i>Alessandro Mosca, Luca Manzoni, and Daniele Codecasa</i>	85
A certification of Lagrange’s theorem with the proof assistant ÆtnaNova/ Referee <i>Domenico Cantone, Salvatore Cristofaro, Marianna Nicolosi Asmundo</i>	101
Incremental Learning from Positive Examples <i>Grazia Bombini, Nicola Di Mauro, Floriana Esposito, and Stefano Ferilli</i>	119
Constraint based implementation of a PDDL-like language with static causal laws and time fluents <i>Agostino Dovier and Jacopo Mauro</i>	131
Towards the use of Simplification Rules in Intuitionistic Tableaux <i>Mauro Ferrari, Camillo Fiorentini, and Guido Fiorino</i>	150
Exploiting Semantic Technology in Computational Logic-based Service Contracting <i>Marco Alberti, Massimiliano Cattafti, Marco Gavanelli, and Evelina Lamma</i>	162
Multi-Agent Planning in CLP <i>Agostino Dovier, Andrea Formisano, and Enrico Pontelli</i>	173

Extending and implementing RASP <i>Andrea Formisano and Davide Petturiti</i>	189
Transformational Verification of Linear Temporal Logic <i>Alberto Pettorossi, Maurizio Proietti, and Valerio Senni</i>	208
Graded Alternating-Time Temporal Logic <i>Marco Faella, Margherita Napoli, and Mimmo Parente</i>	225
A CLP engine for a general purpose configuration tool <i>Andrea Calligaris, Dario Campagna, Christian De Rosa, Agostino Dovier, Angelo Montanari, and Carla Piazza</i>	240
Bottom-up Evaluation of Finitely Recursive Queries <i>Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone</i>	255
Extending Soft Arc Consistency Algorithms to Non-Invertible Semirings, with an Application to Multi-Criteria Problems <i>Stefano Bistarelli, Fabio Gadducci, Emma Rollon, and Francesco Santini</i>	269
Constraint Based Languages for Biological Reactions <i>Stefano Bistarelli, Marco Bottalico, and Francesco Santini</i>	283
fn2dlp: A Normal Form Nested Programs Compiler <i>Annamaria Bria, Wolfgang Faber, and Nicola Leone</i>	289
Solving CSPs with Naming Games <i>Stefano Bistarelli, and Giorgio Gosti</i>	294
An Heuristics for Load Balancing and Granularity Control in the Parallel Instantiation of Disjunctive Logic Programs <i>Simona Perri, Francesco Ricca, and Marco Sirianni</i>	309
A Magic Set Implementation for Disjunctive Logic Programming with Function Symbols <i>Marco Marano, Francesco Ricca, and Giovambattista Ianni</i>	315
A note on constructive semantics for description logics <i>Loris Bozzato, Mauro Ferrari, and Paola Villa</i>	321
IDUM a Logic-Based System for e-Tourism <i>Giuliano Candreva, Gianfranco De Franco, Dino De Santo, Carmine Donato, Antonella Dimasi, Giovanni Grasso, Salvatore Maria Ielpa, Salvatore Iiritano, Nicola Leone, and Francesco Ricca</i>	334
A Fair Extension of (Soft) Concurrent Constraint Languages <i>Stefano Bistarelli, and Paola Campli</i>	347
Translating Natural Language Sentences into ASP theories using SE-DCG grammars and Lambda Calculus <i>Stefania Costantini and Alessio Paolucci</i>	354

PrettyProlog: A Java Interpreter and Visualizer of Prolog Programs

Poster Paper

Alessio Stalla Viviana Mascardi Maurizio Martelli

DISI - Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.
alessiostalla@gmail.com, {mascardi, martelli}@disi.unige.it

Abstract. Many years of teaching demonstrated that one of the hardest concepts for students facing Prolog is the construction of the SLD tree. For this reason one student and the teachers of the Artificial Intelligence Course held at the Computer Science Department, University of Genova, developed PrettyProlog: an interpreter for a subset of Prolog, written in Java, born for didactic use. PrettyProlog features a GUI which allows the user to visualize the inner functioning of the interpreter, namely the construction of stack and SLD tree, and can be used to implement and graphically trace sophisticated programs involving cut, negation as failure, meta-programming.

1 Introduction

PrettyProlog originates from the collaboration between one student and the teachers of the Artificial Intelligence Course held at the Computer Science Department, University of Genova, for providing concrete answers to demands raised by Prolog novices. Many years of teaching demonstrated that one of the hardest concepts for students facing Prolog, is the construction of the SLD tree. A common mistake that students make is to draw one child for each atom in a goal, instead of one child for each clause whose head unifies with the selected atom. The illegitimate children consist, in the best case, in the body of the first clause usable for the corresponding atom, but more bizarre solutions are proposed almost any year. However, since SLD trees are usually large and would require a careful organisation of the space for being properly represented, drawing them on-the-fly on a blackboard, may bother experienced teachers too: typos are just behind the corner. The way bindings of variables are propagated is also difficult to understand, and the link between the SLD tree, which is something static, and the dynamics of the Prolog interpreter and of the Prolog Stack, remains often obscure.

For these reasons, we developed PrettyProlog: an interpreter for Prolog written in Java, born for didactic use, supporting list management, cut, negation as failure and meta-programming. Currently, PrettyProlog does not support the “is” predicate. PrettyProlog is freely available under the General Public License and can be downloaded from <http://alessiostalla.altervista.org/software/prettyprolog/index.php>. It features a GUI which allows the user to visualize the inner functioning of the interpreter, namely the construction of stack and SLD

tree. It has been developed from scratch, without reusing any existing Prolog implementation. Several open-source prologs were examined including TuProlog, <http://www.alice.unibo.it/xwiki/bin/view/Tuprolog/>, and many others. However, either they were too simple, making it worthless to spend time in carefully studying them in order to extend them, or they were too complex, targeted to professional users and containing many additional features and optimizations that made them quite complicated and unsuitable for our didactic purpose. Also, most of them did not manage the stack in an explicit way, due to efficiency concerns. Since none satisfied completely the need for a stack-based, easily expandable and customizable Prolog system, we designed and developed PrettyProlog as a new and really “open” piece of software, in terms of customization and re-usability.

PrettyProlog has been designed to be simple, modular, and easily expandable. For example, the core engine has an event-driven interface that makes it easy to deeply control the solving process without sub-classing the engine itself or re-writing engine methods. Also, as a design choice, simplicity has been favoured against strong encapsulation when needed. So for example, nothing prevents the programmer to corrupt the engine’s stack during goal solving. The documentation explicitly states when it is unsafe to modify a data structure returned by some method.

This paper is organized in the following way: Section 2 describes the GUI offered by PrettyProlog for didactic use (stack and SLD viewers), Section 3 describes related and future work, and concludes.

2 Teaching with PrettyProlog: The Stack and SLD Tree Viewers

The PrettyProlog GUI contains various panels: the Theory panel shows the current theory. When PrettyProlog starts, this panel is empty. To load a theory, the File→Load theory menu item should be used. When PrettyProlog is used as an applet, it cannot access the local file system for security reasons. In this case a window appears where the theory can be typed in or pasted from another program. Otherwise, when used as a standalone application, the file system can be browsed and the file containing the theory can be selected. Once the theory has been loaded, it is possible to solve goals by typing them into the input field in the lower part of the window and pressing Enter or clicking Solve. If the step-by-step mode is enabled, it will be necessary to press Enter again, or click Continue, every time a frame is pushed or popped on/off the stack. Else, PrettyProlog’s engine will allow the user to continue only after it has found a solution to the goal. It is also possible to stop solving the goal using the Stop button. If the goal contains free variables, it will be possible to continue until there are no more substitutions that make it true. If the goal is ground, though, as soon as a solution is found the solving process is stopped.

The SLD tree viewer panel shows as a tree the series of steps the engine has performed. Each branch represents the selection of a clause from the theory, while leaves are either solutions or dead ends, i.e. goals that could not be solved. Near each node there is the substitution that was valid at that point. Also, the SLD tree shows which frames are removed from the stack as the effect of a cut, by printing them with a different font and icon.

PrettyProlog implements basic data types (integer and real numbers, lists, strings), and offers meta-programming facilities that, combined with the “cut” predicate, make the definition of negation as failure possible. Thanks to these features, sophisticated programs may be implemented.

Figure 1 shows the SLD tree of a Prolog program that implements a classical instance of a search problem: that of moving from a city in Romania (Arad, in our case) to Bucharest [4, Chapter 3]. A Stack Viewer is also available; we cannot show it for space constraints. We implemented a depth first search with control of cycles, as well as the auxiliary `not` and `member` predicates:

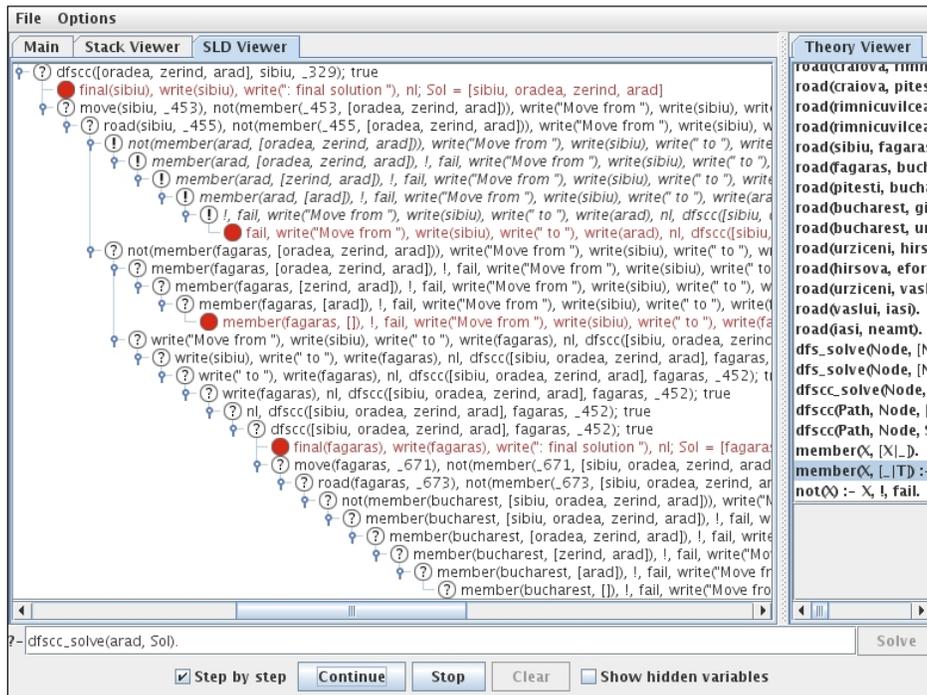


Fig. 1. SLD Viewer.

```
dfsc_solve(Node, Solution) :-
    dfsc([], Node, Solution).

dfsc(Path, Node, [Node|Path]) :-
    final(Node),
    write(Node), write(": final solution "), nl.

dfsc(Path, Node, Solution) :-
    move(Node, Node1),
```

```

    not(member(Node1, Path)),
    write("Move from"), write(Node), write("to"), write(Node1), nl,
    dfsc([Node|Path], Node1, Solution).

member(X, [X | _]).      member(X, [_ | T]) :- member(X, T).

not(X) :- X, !, fail.    not(X).

```

The problem formulation is given by the following state graph:

```

final(bucharest).

move(St1, St2) :- road(St1, St2).

road(arad, zerind).
road(arad, timisoara).
road(sibiu, arad).
road(zerind, oradea).
road(oradea, sibiu).
road(sibiu, fagaras).
...

```

Besides showing what happens both to the stack and to the SLD tree (while it is built), PrettyProlog correctly visualizes the effect of a “cut” on the SLD tree. In the upper part of Figure 1 there are goals written in italic (from *not(member(arad, [oradea, zerind, arad]))*, ... to *!, fail, write(...)*). These nodes are cut after the execution of the *!* in the first clause defining *not*, called with *member(arad, [oradea, zerind, arad])* as argument. PrettyProlog SLD viewer keeps the cut goals for didactic purposes, but shows them in a different font to emphasise that they no longer belong to the tree. The system predicates supported by PrettyProlog, although limited, include simple predicates for input-output, such as *write* and *nl*.

The stack viewer shows each frame pushed onto the stack. When the user clicks on a frame, its content is displayed: the goal that still had to be solved at the time the frame was pushed on the stack; the substitution that is the partial solution to such goal at this point; the clause that has been used to obtain the goal; the index from where, on backtracking, the engine will search for the next clause.

When the PrettyProlog engine solves a goal step-by-step, the clause used in each resolution step is highlighted in the theory panel.

3 Related and Future Work

Research on visualization of the execution of Prolog programs has a long history (just to make some examples, [5,2,6] and many papers collected in [1]). Nevertheless, nowadays few Prolog implementations offer an on-line Stack Viewer and an SLD tree visualizer as graphical means for debugging. The open-source implementations that provide these facilities are even fewer. Among them, we acknowledge SWI-Prolog (<http://www.swi-prolog.org/>) that offers a debugging

window showing current bindings; a diagrammatic trace of the call history; and a highlighted source code listing (<http://www.cs.bris.ac.uk/Teaching/Resources/COMS30106/labs/tracer.htm>). No SLD tree visualization is given. SLDNF-Draw [3], downloadable from <http://www.ing.unife.it/software/sldnfDraw/>, is a classical meta-interpreter that executes a Prolog program and, during execution of a node, saves the Latex instructions that generate the tree. Being based on Latex, the graphical result is definitely better than ours, but, on the other side, SLDNF-Draw provides no support to run-time drawing of the tree. Also, the stack is not visualized.

On the other hand, many Java implementations of a Prolog interpreter exist, starting from W-Prolog, <http://goanna.cs.rmit.edu.au/%7Ewinikoff/wp/>, the first prolog interpreter written in Java, and reaching more than 20 current implementations of either Prolog interpreters written in Java, or Java/Prolog interfaces <http://kaminari.scitec.kobe-u.ac.jp/logic/jprolog.html>. We already explained in the Introduction the reasons why we felt the need of implementing our own Prolog interpreter in Java.

Although at a prototypical stage, PrettyProlog presents three features that, to the best of our knowledge, cannot be found together in any other Prolog implementation:

1. it provides Stack and SLD Tree visualizers; 2. it is open source; 3. it is written in Java, and fully compliant with Java ME CDC application framework.

As far as the last point is concerned, the light implementation of PrettyProlog and its compliance with the Java ME CDC application framework make it suitable for other than mere educational purposes. In particular, PrettyProlog might be run on PDAs. We are currently designing its extension for allowing engines running on interconnected PDAs to exchange knowledge and to exploit exogenous facts in successive derivations.

A further step in the development of PrettyProlog could be the adoption of either JGraph, <http://www.jgraph.com/>, a free Java library that allows the developer to easily draw graphs with arcs and nodes and to place the elements without intersections, or Eclipse GEF <http://www.eclipse.org/gef/>. In the latter case, PrettyProlog should be extended and embedded into the Eclipse IDE as a plug-in. The adoption of some existing graphic library should greatly improve the results of our SLD tree visualizer whose user-friendliness is currently limited.

References

1. M. Ducassé, A-M Emde, T. Kusalik, and J. Levy, editors. *Logic Programming Environments, ICLP'90 Preconference Workshop*. 1990. ECRC Technical Report IR-LP-31-25.
2. M. Eisenstadt and M. Brayshaw. A fine-grained account of Prolog execution for teaching and debugging. *Instructional Science*, 19(4/5):407–436, 1990.
3. M. Gavanelli. SLDNF-Draw: a visualisation tool of prolog operational semantics. In G. Fiunara, M. Marchi, and A. Proveti, editors, *CILC 2007*, 2007.
4. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Second Edition*. Prentice Hall, 2003.
5. H. Shinomi. Graphical representation and execution animation for Prolog programs. In *MIV*, pages 181–186. IEEE Computer Society, 1989.
6. D. E. Tamir, R. Ananthakrishnan, and A. Kandel. A visual debugger for pure Prolog. *Inf. Sci. Appl.*, 3(2):127–147, 1995.

Exploiting Prolog and NLP Techniques for Matching Ontologies and for Repairing Correspondences*

Viviana Mascardi¹, Angela Locoro², and Fabrizio Larosa¹

¹DISI, Università degli Studi di Genova, Via Dodecaneso 35, 16146, Genova, Italy
viviana.mascardi@unige.it, 2003s140@educ.disi.unige.it

²DIBE, Università degli Studi di Genova, Via All'Opera Pia 11a, 16145 Genova, Italy,
angela.locoro@unige.it

Abstract. Providing efficient ontology matching algorithms is one of the means for pursuing semantic interoperability. In this paper we discuss an algorithm that exploits natural language processing techniques for matching ontologies and that post-processes the obtained alignment in order to find semantic inconsistencies. The algorithm has been entirely implemented in Prolog, whose usefulness was mainly evident in the post-processing phase. A careful analysis of the recent state-of-the-art witnesses the originality of our matching algorithm which is based on the “Adapted Lesk Algorithm” for word sense disambiguation. The experiments we carried out, although in their early stages, are encouraging.

Keywords. Computational Logic, Semantic Interoperability, Natural Language Processing, Ontology Matching, Correspondence Repair

1 Introduction

Semantic interoperability, namely “*the ability of two or more computer systems to exchange information and have the meaning of that information automatically interpreted by the receiving system accurately enough to produce useful results*”¹, is one of the most relevant and lively research fields of the last fifteen years. The advent of ontologies in computer science in the early nineties [14], the settlement of Web Services in the beginning of the new millennium, the combination of both into Semantic Web Services, all witness the fervid activity of academia and industry for finding algorithms, languages, tools, and infrastructures aimed at realising the “semantic interoperability dream”.

The reason for the interest in this topic is easy to explain. In a recent paper [26], Jan Walker et al. assess the value of information exchange and interoperability in the domain of health care, and state that a fully standardised system supporting information exchange and interoperability could yield a net value of 77.8 billion dollars per year once implemented. Other studies [10,5] confirm these impressive economic advantages in implementing solutions for semantic interoperability.

Since knowledge is more and more represented by means of ontologies, and different ontologies must be matched in order to make knowledge exchange possible, one

* Partially supported by the Italian project “Iniziativa Software CINI-FinMeccanica”.

¹ Wikipedia, http://en.wikipedia.org/wiki/Semantic_interoperability, accessed March, 15, 2009.

of the means for pursuing semantic interoperability is that of providing efficient ontology matching algorithms. The first formalisation of the matching problem dates back to 2000 [2]. Today, due to the ever increasing availability of many ontologies in very different domains, ontology matching is becoming one of the most important activities in the Semantic Web area.

This paper deals with the problem of finding semantically correct mappings between couples of ontologies. It exploits an algorithm used for word sense disambiguation originally proposed by Michael Lesk [16] and adapted by Satanjeev Banerjee and Ted Pedersen [1]. The obtained alignment is then post-processed in order to further refine it by finding semantic inconsistencies.

The paper is organised in the following way: Section 2 provides some background knowledge on WordNet, the Lesk algorithm, ontology matching, and alignment repair. Section 3 discusses the algorithms that we have implemented, whereas Section 4 describes their Prolog implementation and discusses the preliminary results we have obtained. Section 5 concludes and highlights future directions of our work.

2 Background

In this section we provide a short background on the topics upon which our research roots: WordNet, the Lesk algorithm for word sense disambiguation, ontology matching, and alignment repair.

2.1 WordNet

WordNet [9] is a large lexical database of English, developed under the direction of George A. Miller. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. A synset or synonym set is defined as a set of one or more synonyms that are interchangeable in some context without changing the truth value of the proposition in which they are embedded.

Most synsets are connected to other synsets via a number of semantic relations that include:

Semantic relations between nouns. *Hypernyms*: Y is a hypernym of X if every X is a (kind of) Y ; *hyponyms*: Y is a hyponym of X if every Y is a (kind of) X ; *coordinate terms*: Y is a coordinate term of X if X and Y share a hypernym; *holonym*: Y is a holonym of X if X is a part of Y ; *meronym*: Y is a meronym of X if Y is a part of X .

Semantic relations between verbs. *Hypernym*: the verb Y is a hypernym of the verb X if the activity X is a (kind of) Y ; *troponym*: the verb Y is a troponym of the verb X if the activity Y is doing X in some manner; *entailment*: the verb Y is entailed by X if by doing X you must be doing Y ; *coordinate terms*: those verbs sharing a common hypernym.

While semantic relations apply to all members of a synset because they share a meaning but are all mutually synonyms, words can also be connected to other words through lexical relations, including antonyms (opposites of each other) and derivationally related, as well.

2.2 Lesk Algorithm

The Lesk algorithm is an algorithm for word sense disambiguation introduced by Michael E. Lesk in 1986 [16]. It applies to short sentences and uses the number of words common to the definitions, or *glosses*, of each word in the sentence as a measure representing how much that gloss correctly defines the word. For example, if the sentence to disambiguate contains *bass* and *fishing*, and both words are defined by one gloss that contains *fish* (suppose gloss “a fish with a spiny dorsal fin, related to or resembling the perch”² for *bass*, and gloss “the sport, industry, or occupation of catching fish”³ for *fishing*), then these two glosses have some chance to be the right ones for *bass* and *fishing* respectively. The original Lesk algorithm uses dictionaries like Oxford Advanced Learner’s to retrieve glosses.

In 2002, Satanjeev Banerjee and Ted Pedersen adapted Lesk algorithm by using WordNet as the source of glosses [1]. They also adapted the algorithm in order to disambiguate each word in a sentence, whatever the sentence length. The algorithm operates in the following way:

Input: a sentence s ; a target word tw to disambiguate within that sentence; the dimension $2n + 1$ of the window to consider as a context for tw

Output: a gloss that provides the right definition for tw

begin

1. create the list *context* that contains the n words that belong to WordNet and that come before tw in s , the n words that belong to WordNet and that come after tw in s , and tw ;
2. if tw is near the beginning or the end of the sentence, take additional WordNet words from the other direction;
3. for each word w in *context*
 - (a) retrieve its synonyms, hypernyms, hyponyms, holonyms, meronyms, troponyms, and attributes from WordNet;
 - (b) if the part of speech (noun, adjective, verb, ...) p that w plays in the sentence is known, than take into account only relations and synsets associated with w as a p , otherwise consider all possible relations and synsets for w ;
 - (c) retrieve glosses for each word in the list of synonyms and senses related to w obtained in the previous step: be it $g_1(w), g_2(w), \dots, g_n(w)$;
 - (d) the score of each gloss in the list associated with w , $score(g_i(w))$, is initially set to 0;
4. for each pair of words wa, wb in *context*
 - (a) for each pair of glosses $g_j(wa)$ belonging to the list of synonyms of and sensed related to wa and $g_k(wb)$ belonging to the list of synonyms of and sensed related to wb
 - i. define an overlap between $g_j(wa)$ and $g_k(wb)$ as the longest sequence of common consecutive words: this overlap results into a score stating how much $g_j(wa)$ is the right definition for wa , and $g_k(wb)$ is the right definition for wb ;

² AskOxford.com, <http://www.askoxford.com/>.

³ Msn Encarta Dictionary, <http://encarta.msn.com/encnet/features/dictionary/dictionaryhome.aspx>.

- ii. update $score(g_j(wa))$ and $score(g_k(wb))$ by adding the scores obtained in the previous step;
 - 5. to disambiguate the target word tw , select the gloss in $g_1(tw), g_2(tw), \dots, g_m(tw)$ with highest score, and return it
- end**

2.3 Ontology Matching

This section is based on [8] and adopts definitions similar to those used there, eventually simplified for sake of clarity.

Definition 1: Correspondence. A correspondence between an entity e belonging to ontology o and an entity e' belonging to ontology o' is a 5-tuple $\langle id, e, e', R, conf \rangle$ where:

- id is a unique identifier of the correspondence;
- e and e' are the entities (e.g. properties, classes, individuals) of o and o' respectively;
- R is a relation such as “equivalence”, “more general”, “disjointness”, “overlapping”, holding between the entities e and e' .
- $conf$ is a confidence measure (typically in the $[0, 1]$ range) holding for the correspondence between the entities e and e' ;

In our algorithm we only consider classes as entities and equivalence as relation.

Definition 2: Alignment. An alignment a of ontologies o and o' is a set of correspondences between entities of o and o' .

Definition 3: Matching Process. A matching process can be seen as a function f which takes two ontologies o and o' , a set of parameters p and a set of oracles and resources r , and returns an alignment a between o and o' (adapted from [3]).

Matching techniques. An exhaustive survey on matching approaches can be found in [8]. Among the matching techniques, we just discuss those that fall under the “*Granularity / Input Interpretation*” classification, based on the granularity of the matcher and on the interpretation of the input information. In particular, we consider string-based methods and language-based ones.

1. **String-based methods.** These methods measure the similarity of two entities just looking at the strings (seen as mere sequences of characters) that label them. They include *Levenshtein Distance*, defined as the minimum number of insertions, deletions and substitutions of characters required to transform one string into another [17], and *SMOA Measure* which is a function of their commonalities (in terms of substrings) as well as of their differences [24].

2. **Language-based methods.** Language-based methods exploit natural language processing techniques to find the similarity between two strings seen as meaningful

pieces of text rather than sequences of characters. Some of them exploit external resources like WordNet, and exploit the semantic relations that it offers to compute the similarity. Language-based techniques include

- Tokenisation: names of entities are parsed into sequences of tokens. For example, *RedWine* becomes $\langle red, wine \rangle$ [12].
- Stemming: the strings underlying tokens are morphologically analysed in order to find all their possible basic forms. For example, *goes* becomes *go* [12].

These techniques are applied to names of entities before running string-based or lexicon-based techniques in order to improve their results [23].

Linguistic resources, such as common knowledge or domain specific thesauri are used in order to match words resulting from tokenisation and stemming of ontology entities based on linguistic relations between them (e.g., synonyms, hyponyms). For example, WordNet may be used to obtain meaning of words used in ontologies and relations between ontology entities can be computed in terms of bindings between WordNet senses. CtxMatch [4] represents the first instantiation of a language-based approach of this kind, discussed in [11].

Other matchers exploit the structural properties of thesauri, e.g., WordNet hierarchies. Hierarchy-based matchers measure the distance, for example, by counting the number of arcs traversed, between two concepts in a given hierarchy [13]. Several other distance measures for thesauri have been proposed in the literature [22,21].

The recent paper “A Survey of Exploiting WordNet in Ontology Matching” [18] discusses language- and WordNet-based matching techniques in detail. According to that paper and to many others that we have considered while analysing the state-of-the-art, no approaches based on the Adapted Lesk algorithm for ontology matching have been proposed so far. The algorithm we discuss in Section 3.1 is, to the best of our knowledge, an original contribution to the ontology matching research field.

2.4 Repairing Ontology Correspondences

As we will see in Section 4.2, where experimental results will be discussed, alignment methods may both produce wrong correspondences (the methods are not correct), and not produce right correspondences (they are not complete). For this reason, repairing wrong correspondences in an ontology alignment is a very pressing need. Very few attempts to solve this problem exist [15,19,6]. Among them, the approach that inspired our work and that we will briefly discuss here, is “Repairing Ontology Mappings” by C. Meilicke, H. Stuckenschmidt, and A. Tamilin [19].

The approach followed there is to interpret the problem of identifying wrong correspondences in an ontology alignment as a diagnosis task. Meilicke, Stuckenschmidt, and Tamilin formalise correspondences as “bridge rules” in distributed description logics and analyse the impact of each correspondence on the ontologies it connects. The basic assumption is that a correspondence that correctly states the semantic relations between ontologies should not cause inconsistencies in any of the ontologies. The encoding in distributed description logics allows the authors of [19] to detect these inconsistencies which are treated as symptoms caused by an incorrect correspondence. They

then compute sets of correspondences that jointly cause a symptom and repair each symptom by removing correspondences from these sets. The set of correspondences remaining after this process can be regarded as an approximation of the set of correct correspondences.

For example, consider two ontologies o and o' characterised respectively by axiom ax and axiom ax'

$$\begin{aligned} ax &: author \sqsubseteq person \\ ax' &: person \sqsubseteq \neg authorization \end{aligned}$$

Suppose that the alignment to repair is the following:

$$\{ \langle id_h, person, person, \equiv, 1.00 \rangle, \langle id_k, author, authorization, \sqsupseteq, 0.46 \rangle \}$$

The alignment is inconsistent with respect to *authorization*. In fact, it is possible to derive by distributed reasoning, that *authorization* \sqsubseteq *person* has to hold. At the same time, *authorization* and *person* are defined as disjoint concepts in ontology o' . In particular, this makes *authorization* unsatisfiable with respect to the global interpretation.

3 Matching Ontologies with Lesk and WordNet, and Repairing the Obtained Correspondences

In this section we describe our approach for generating ontology alignments by exploiting Lesk algorithm and WordNet and for repairing them in a supervised way, by detecting inconsistencies.

3.1 Exploiting “Lesk + WordNet” Algorithm to Match Ontologies

Given two ontologies o and o' to match, our language-based algorithm uses the adapted Lesk algorithm described in [1] (with some simplifications that have a very limited impact on the results it gives) to find the correct definition, or gloss, of any concept $c \in o$ and $c' \in o'$. The glosses found in this way are then compared to decide whether c corresponds to c' or not. Instead of taking the words before and after the target word to disambiguate in a given text, we take the neighboring concepts of c (those related by any kind of relation in the ontology) in the given ontology o , and we use these concepts as the context for disambiguating c .

The algorithm generates the $|o| \times |o'|^4$ couples of concepts $\langle c \in o, c' \in o' \rangle$ and calls the function **is_correct_correspondence**(c, c') on each of them. We limit ourselves at considering the equivalence relation, thus we drop it from correspondences. Also, we drop correspondence identifiers for sake of readability. Thus, our correspondences are triples made by one concept in o , one concept in o' , and a confidence in their equivalence. For the moment, the confidence is a binary value in $\{0, 1\}$: 1 means that, according to the adapted Lesk algorithm, the concepts in the couple have exactly the same gloss; 0 means the gloss is not the same. Finer grained values may be given to the confidence, taking some similarity measure between glosses into account, as well as hyponymy, hypernymy, meronymy, and holonymy relations between the words they define.

The function **is_correct_correspondence** is defined by the following algorithm:

⁴ $|o|$ is the number of concepts in o .

Input: an ontology o ; an ontology o' ; a concept $c \in o$; a concept $c' \in o'$; the WordNet Thesaurus

Output: an integer in $\{0, 1\}$

begin

1. if at least one between c and c' , eventually after stemming, does not belong to WordNet, then return 0; otherwise
2. find the context of $c \in o$ by retrieving all the concepts $related(c) \in o$ that are related to c by any kind of relation apart from “disjoint”;
3. stem c and any $related(c)$ found;
4. put the stemmed words (including c or its stem) that belong to WordNet into a list l , and discard those that do not belong to WordNet;
5. disambiguate the stem of c with respect to the context l using the “Lesk + WordNet” algorithm: be $disambiguated(c)$ the found gloss;
6. repeat steps from 2 to 5 for $c' \in o'$;
7. if $disambiguated(c)$ and $disambiguated(c')$ are the same gloss, then return 1; otherwise
8. return 0

end

The function **lesk_wordnet_matching** just calls **is_correct_correspondence** for each couple $\langle c, c' \rangle \in o \times o'$; only those correspondences for which the function returns 1 are kept in the generated alignment. The function is defined by the following algorithm:

Input: an ontology o ; an ontology o' ; the WordNet Thesaurus

Output: an alignment a

begin

1. $a = \{\}$
2. for each couple $\langle c, c' \rangle \in o \times o'$
 - (a) if **is_correct_correspondence**($o, o', c, c', \text{WordNet}$)
 - (b) then $a = a \cup \{\langle c, c', 1 \rangle\}$

end

3.2 Supervised Reasoning on Inheritance and Disjointness

The Lesk algorithm applied to the ontology matching process does not allow to detect semantic inconsistencies like the one described in Section 2.4. For this reason, we propose to involve the user in the process of repairing the alignment generated by the **lesk_wordnet_matching** function (or by any other matching method), in order to remove those correspondences that cause inconsistencies in the hierarchies of concepts induced by the inheritance relation. The inheritance relation is named *subClass* in OWL [25]; we translated into an *is_a* predicate, when we moved from the OWL representation of ontologies, to the Prolog one. Our repair algorithm identifies the couples of correspondences that raise an inconsistency, but cannot decide which one is to be removed. Hence, the user is asked to perform the choice of whether keeping or discarding a “suspicious” correspondence, after providing him/her with a careful explanation on the inconsistency raised.

Our idea is that the disjointness of concepts, that also propagates to sub-concepts, must be respected by the alignment.

Suppose that the alignment a to be repaired contains the correspondences $\langle c_i \in o, c'_i \in o', conf_i \rangle$ and $\langle c_j = ancestor(c_i) \in o, c'_j \in o', conf_j \rangle$. This means that $c_i \equiv c'_i$ and that $c_j \equiv c'_j$ to some extent given by $conf_i$ and $conf_j$.

Suppose that there is one ancestor of $c'_i \in o'$, let's name it $ancestor(c'_i)$, and that $ancestor(c'_i)$ is disjoint with c'_j . Since c'_i is a sub-concept of $ancestor(c'_i)$, c_i must be a sub-concept of $ancestor(c'_i)$ too (recall that c_i and c'_i are equivalent, according to the alignment). But c_i is also a sub-concept of $c_j = ancestor(c_i)$, and $ancestor(c'_i)$ is disjoint with $c_j = ancestor(c_i)$. In other words, c_i descends from two disjoint concepts $ancestor(c'_i)$, c_j , which is not consistent. This situation is depicted in Figure 1. The symmetric situation, where disjoint concepts are in o and not in o' , must also be considered. The inconsistency might be due to either $\langle c_i \in o, c'_i \in o', conf_i \rangle$ or $\langle c_j = ancestor(c_i) \in o, c'_j \in o', conf_j \rangle$. Our algorithm cannot easily tell the wrong correspondence, thus, when a couple of inconsistent correspondences is detected, the user is prompted, and he/she can select the correspondence to remove, if any.

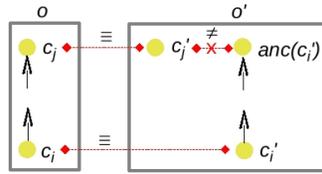


Fig. 1. Inconsistent correspondences.

The algorithm for reasoning about inconsistencies can be better described in Prolog than in an imperative language. The Prolog program accesses an atom table where the representations of the two ontologies o and o' , and of the alignment a have been previously loaded.

Ontologies are defined by the $is_a(OntologyId, SubConc, SuperConc)$ predicate defining inheritance relations, by the $rel(OntologyId, RelName, Concept, RelatedConcept)$ predicate defining domain-dependent relations, and by the $disjointWith(OntologyId, Concept, DisjointConcept)$ predicate that corresponds to the OWL $owl:disjointWith$ element. An alignment consists of the correspondences that belong to it, represented by $map(ConceptInO1, ConceptInO2, Confidence)$ atoms.

Ontologies o and o' , and alignment a represented as above, are thus the input for the algorithm.

First of all, we define $disjoint$ as a symmetric relation:

```

disjoint(O, C1, C2) :-
    disjointWith(O, C1, C2).
disjoint(O, C1, C2) :-
    disjointWith(O, C2, C1).

```

Then, we define the notion of ancestor as the transitive closure of the *is_a* predicate:

```

transitive_closure_is_a(_, C, C).
transitive_closure_is_a(Onto, C, SuperC):-
    is_a(Onto, C, Super),
    transitive_closure_is_a(Onto, Super, SuperC).

```

The **get_inconsistencies/5** predicate unifies the variable *Inconsistencies* with a term that contains the causes of the inconsistency raised by the mapping between *C1* in *Onto1* and *C2* in *Onto2* as shown in Table 1.

```

get_inconsistencies(Onto1, Onto2, C1, C2, Inconsistencies) :-
    findall(inconsistency(
        transitive_closure_is_a(Onto1, C1, SuperC1),
        transitive_closure_is_a(Onto2, C2, SuperC2),
        map(SuperC1, MapSuperC1, Con),
        disjoint(Onto2, MapSuperC1, SuperC2)),
        (map(C1, C2, Confidence),
        transitive_closure_is_a(Onto1, C1, SuperC1),
        transitive_closure_is_a(Onto2, C2, SuperC2),
        map(SuperC1, MapSuperC1, Con),
        disjoint(Onto2, MapSuperC1, SuperC2)),
        List1),
    findall(inconsistency(
        transitive_closure_is_a(Onto1, C1, SuperC1),
        transitive_closure_is_a(Onto2, C2, SuperC2),
        map(InverseMapSuperC2, SuperC2, Con),
        disjoint(Onto1, InverseMapSuperC2, SuperC1)),
        (map(C1, C2, Confidence),
        transitive_closure_is_a(Onto1, C1, SuperC1),
        transitive_closure_is_a(Onto2, C2, SuperC2),
        map(InverseMapSuperC2, SuperC2, Con),
        disjoint(Onto1, InverseMapSuperC2, SuperC1)),
        List2),
    append(List1, List2, Inconsistencies).

```

Table 1. The **get_inconsistencies/5** predicate.

The above Prolog predicate means that, given two ontology identifiers *Onto1* and *Onto2* and two concepts $C1 \in \text{Onto1}$ and $C2 \in \text{Onto2}$, the correspondence $\text{map}(C1, C2, \text{Confidence})$ raises an inconsistency if

- $\exists \text{SuperC1} \in \text{Onto1}$, and *SuperC1* is an ancestor of *C1*
- $\exists \text{SuperC2} \in \text{Onto2}$, and *SuperC2* is an ancestor of *C2*
- $\exists \text{MapSuperC1} \in \text{Onto2}$ such that there is a correspondence $\text{map}(\text{SuperC1}, \text{MapSuperC1}, \text{Con}) \in a$
- *MapSuperC1* and *SuperC2* are disjoint in *Onto2*

or

- $\exists \text{SuperC1} \in \text{Onto1}$, and *SuperC1* is an ancestor of *C1*
- $\exists \text{SuperC2} \in \text{Onto2}$, and *SuperC2* is an ancestor of *C2*
- $\exists \text{InverseMapSuperC2} \in \text{Onto1}$ such that there is a correspondence $\text{map}(\text{InverseMapSuperC2}, \text{SuperC2}, \text{Con}) \in a$
- *InverseMapSuperC2* and *SuperC1* are disjoint in *Onto1*

4 Prolog Implementation and Experimental Results

4.1 Implementation

Both the adapted Lesk algorithm and the supervised reasoning algorithm are implemented in Sicstus Prolog 3.11 and can be downloaded from <http://www.disi.unige.it/person/MascardiV/Software/LeskRepairMatching.html>.

Our adapted Lesk algorithm accesses the Prolog version of WordNet 3.0 available at <http://wordnet.princeton.edu/3.0/WNprolog-3.0.tar.gz>. It operates as described in Section 3.1. With respect to Banerjee and Pedersen’s proposal [1], we have added a stemming stage implemented by the *recognize_words(ListOfWords, StemsOfWords)* predicate that translates nouns in plural form, and verbs in a form different from the infinite one, into the singular and infinite form, respectively. Also, as already explained, the context that we take into account depends on the ontology structure given by the *is_a* and *rel* predicates. The *get_concept_context(C, Onto, Context)* predicate is defined in the following way (*appendall* appends lists in a list, in the order they appear):

```
get_concept_context(C, O, Context):-
    findall(Conc, is_a(O, C, Conc), L1), findall(Conc, is_a(O, Conc, C), L2),
    findall(Conc, rel(O, _, C, Conc), L3), findall(Conc, rel(O, _, Conc, C), L4),
    appendall([L1, L2, L3, L4], Context).
```

In order to overcome a limitation of the 3.11 release of Sicstus Prolog, namely a very limited atom table that fills up just after loading two or three files of WordNet relations, we had to manually implement an “atom table cleaning” mechanism. Our attempts to load all WordNet files in the working memory and use them failed due to an “atom table full” error. This happened both using a Sicstus release for Windows and for Linux, and on different machines. For this reason we had to load each WordNet file at a time, extract all the atoms that we needed from it, according to the concepts we had to deal with in our ontologies, unload it, and explicitly call the atom garbage collector to make room for new atoms loaded from another WordNet file.

The goal to call in order to match ontologies $O1$ and $O2$, whose Prolog representation is stored in file $OntoFile$, is `lesk_match(O1, O2, OntoFile, MappingFile, LogFile)`. $MappingFile$ is the file where the resulting alignment will be saved, whereas $LogFile$ is the file where all the activities of the program will be traced.

As far as the supervised reasoning algorithm is concerned, its Prolog core has been already described in Section 3.2. Besides it, we have implemented a textual interface that allows the user to interact with the program.

The goal to call in order to repair the alignment stored in $MappingFile$ is `repair(O1, O2, OntoFile, MappingFile, RepairedMappingFile, LogFile)`. $O1$ and $O2$ are the ontology identifiers, and $OntoFile$ is the file containing the Prolog representation of both of them. The alignment resulting after the repair process and a trace of the performed reasoning are written to two output files.

4.2 Results

In order to evaluate the feasibility of the adapted Lesk algorithm for matching ontologies, we have applied it to the ontologies represented in Figure 2. The ontologies have been created using Protégé, <http://protege.stanford.edu/>, have been exported into OWL, and have been manually translated into Prolog. The reference alignment between o and o' , namely the correct and complete alignment computed by a domain expert, is

```
map(being, organism, 1).          map(nonliving, inanimate, 1).
map('human being', human, 1).
```

We have run SMOA, Levenshtein, and WordNet-based matching algorithms described in Section 2.3, and implemented by the Alignment API, <http://alignapi.gforge.inria.fr/>, on o and o' .

Any ontology matching algorithm, be it string- or language-based, produces the correspondence $\langle bank, bank, 1 \rangle$. However, $bank$ in o has not the same meaning than $bank$ in o' : in fact, the first one refers to “a building in which the business of banking transacted” whereas the second one refers to a “sloping land (especially the slope beside a body of water)” (definitions are taken from WordNet 3.0). Since the Lesk-based disambiguation process of $bank$ in the context [*building, banker*] provided by ontology o and $bank$ in the context [*geological formation, river*] provided by ontology o' leads to two unrelated glosses, our algorithm classifies the correspondence as incorrect. The same happens with $bass \in o$ and $bass \in o'$, whose context in the respective ontologies allows Lesk algorithm to understand that the first one is the professional singer, whereas the second one is the fish.

All the ontology matching algorithms that we have run, rate the correspondence between *airplane* and *plane* with a very high value. However, the context of *plane* in o' is [*object, carpenter*]. This is enough for Lesk algorithm decide that it refers to “a carpenter’s hand tool with an adjustable blade for smoothing or shaping wood”, and not to “an aircraft that has a fixed wing and is powered by propellers or jets”. In the end, the output of our adapted Lesk algorithm is

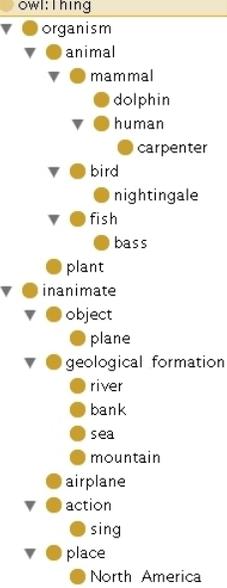
 <pre> owl:Thing ├── being │ ├── human_being │ │ ├── banker │ │ └── singer │ │ └── bass │ └── nonliving │ ├── airplane │ └── building │ ├── bank │ └── theatre │ └── formation </pre>	 <pre> owl:Thing ├── organism │ └── animal │ ├── mammal │ │ ├── dolphin │ │ └── human │ │ └── carpenter │ └── bird │ └── nightingale │ └── fish │ └── bass │ └── plant ├── inanimate │ └── object │ ├── plane │ └── geological_formation │ ├── river │ ├── bank │ ├── sea │ ├── mountain │ └── airplane ├── action │ └── sing └── place └── North_America </pre>
<pre> rel(o, performs_in, singer, theatre). rel(o, works_in, banker, bank). rel(o, fly_in, airplane, formation). </pre>	<pre> rel(o', has_habitat, bass, river). rel(o', performs_action, bird, sing). rel(o', lives_in, bass, 'North America'). rel(o', uses_tool, carpenter, plane). rel(o', is_near, bank, river). </pre>
<pre> disjointWith(o, being, nonliving). </pre>	<pre> disjointWith(o', animate, inanimate). disjointWith(o', dolphin, fish). </pre>

Fig. 2. Ontologies o and o' .

$map(being, organism, 1)$. $map(nonliving, inanimate, 1)$.
 $map('human\ being', human, 1)$. $map(formation, 'geological\ formation', 1)$.

On the given input, the algorithm is complete, but not correct. The tricky correspondences that the SMOA, Levenshtein, and WordNet-based matching algorithms find, are correctly discarded by Lesk's one. Instead, both *formation* in the context [*nonliving, airplane*], and *geological formation* in the context [*inanimate, river, bank, sea, mountain*], are given the same sense “(geology) the geological features of the earth”. The intended sense of *formation* was, instead, “an arrangement of people or things acting as a unit”. In this case, the context was not informative enough for disambiguating the word properly.

Below we summarise the results obtained by our matching algorithm and by the SMOA, Levenshtein, and WordNet-based ones in terms of precision, recall and F-measure adapted for ontology alignment evaluation [7], obtained by exploiting the *GroupEval* method offered by the Align API.

	SMOA	Levenstein	WordNet	Our algorithm
Precision	0.10	0	0.11	0.75
Recall	0.33	0	0.33	1
F-measure	0.15	Not a Number	0.17	0.86

Precision is the number of correctly found correspondences with respect to the reference alignment (true positives), divided by the total number of found correspondences (true positives and false positives), and recall is the number of correctly found correspondences (true positives) divided by the total number of expected correspondences (true positives and false negatives). F-measure is the harmonic mean of precision and recall.

The table shows that on the given ontologies, our adapted Lesk algorithm gives the best results. In fact, SMOA and the WordNet-based algorithms only find the $\langle 'human\ being', human \rangle$ correct correspondence, whereas Levenshtein finds no correct correspondences. On the other hand, all the algorithms find many wrong correspondences.

To experiment our alignment repair algorithm, we have modified ontology o' by changing *sing* into *singer*. The adapted Lesk matching algorithm finds the correspondence $\langle singer, singer, 1 \rangle$. However, this would cause *singer* in o to descend from both *being* and *nonliving*, which are defined as disjoint concepts. The repair algorithm correctly detects this problem and prompts the user:

Checking the consistency of (singer in o, singer in o') with confidence 1
Concept singer in o descends from concept being; concept singer in o' descends from
concept inanimate; concept nonliving in o is equivalent to concept inanimate in o'
with confidence 1; concepts nonliving and being are disjoint in o; this raises an in-
consistency!
What should I do with the (singer in o, singer in o') mapping?
What should I do with the (nonliving in o, inanimate in o') mapping?

5 Conclusions and Future Work

In this paper, we described our algorithms for ontology matching and supervised mapping repair and their implementation, and we discussed the preliminary results obtained with them.

These results are extremely encouraging but our work is open to many improvements: the experiment discussed in Section 4.2 has been carried out on two artificial ontologies, created ad hoc for containing concepts that can be found in WordNet and whose context might help Lesk algorithm to find their correct meaning. Our matching algorithm works well only on ontologies whose concepts belong to WordNet. This is a strong requirement, barely satisfied by most real ontologies.

Together with the students of the Artificial Intelligence course at DISI, however, we are currently working for overcoming this problem: the individual assignment for the academic year 2008/2009 requires to implement an algorithm in Prolog for matching ontology concepts to sets of WordNet words according to some well known distance

metric on strings to be chosen by the student. Students have to tokenise ontology concepts first. After tokenisation, the algorithm has to find correspondences between each concept token and one or more WordNet words. In the end, each ontology concept will be tagged with a set of WordNet words, based just on string metrics, and these tags will be used instead of the original ontology concepts by the Adapted Lesk algorithm, solving the main current limitation of our approach.

Another direction that we are exploring for partly overcoming this problem is to use the SUMO Upper Ontology [20] as a bridge between the ontology o whose concepts must be disambiguated, and WordNet. SUMO has been entirely mapped to WordNet 3.0 by hand: mapping o (or at least, those concepts of o that cannot be found in WordNet) to SUMO using some existing matching algorithm would give a mapping between o and WordNet for free, allowing us to run the adapted Lesk algorithm on the WordNet translation of o .

A significant improvement to our algorithm would be refining the confidence returned by our matching algorithm by setting it to some value < 1 if the concepts are not synonyms, but they are related by some semantic relation.

After these improvements will be implemented, we will start to test our algorithms on real ontologies. This experimentation will provide us with hints on how making our algorithms suitable for matching whatever ontologies, still keeping high precision and recall values. Optimisation issues will also be taken under consideration: we expect that performances will dramatically drop on large ontologies, and the tests' outcomes will help us in identifying some viable approach for balancing efficiency, precision and recall.

References

1. S. Banerjee and T. Pedersen. An adapted lesk algorithm for word sense disambiguation using WordNet. In A. F. Gelbukh, editor, *3rd International Conference Computational Linguistics and Intelligent Text Processing, CICLing 2002, Proceedings*, volume 2276 of *Lecture Notes in Computer Science*, pages 136–145. Springer, 2002.
2. Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.
3. P. Bouquet, J. Euzenat, E. Franconi, L. Serafini, G. Stamou, and S. Tessaris. Specification of a common framework for characterizing alignment. Technical Report D2.2.1, NoE Knowledge Web project, 2004.
4. P. Bouquet, L. Serafini, and S. Zanobini. Semantic coordination: A new approach and an application. In D. Fensel, K. P. Sycara, and J. Mylopoulos, editors, *2nd International Semantic Web Conference, ISWC 2003, Proceedings*, volume 2870 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2003.
5. S. B. Brunnermeier and S. A. Martin. Interoperability cost analysis of the u.s. automotive supply chain. Technical Report NIST 99-1, U.S. Department of Commerce, National Institute of Standards and Technology, 1999.
6. H. Stuckenschmidt C. Meilicke and A. Tamin. Reasoning support for mapping revision. *Journal of Logic and Computation*, 2008.
7. H. H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In A. B. Chaudhri, M. Jeckle, E. Rahm, and R. Unland, editors, *Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, 2002, Revised Papers*, volume 2593 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2002.

8. J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.
9. C. Fellbaum, editor. *WordNet – An Electronic Lexical Database*. The MIT Press, 1998.
10. M. P. Gallaher, A. C. O’Connor, J. L. Dettbarn Jr., and L. T. Gilday. Cost analysis of inadequate interoperability in the u.s. capital facilities industry. Technical Report NIST GCR 04-867, U.S. Department of Commerce, National Institute of Standards and Technology, 2004.
11. F. Giunchiglia and P. Shvaiko. Semantic matching. *Knowl. Eng. Rev.*, 18(3):265–280, 2003.
12. F. Giunchiglia, P. Shvaiko, and M. Yatskevich. Semantic schema matching. In *International Conference on Cooperative Information Systems, CoopIS, Proceedings*, pages 347–365, 2005.
13. F. Giunchiglia and M. Yatskevich. Element level semantic matching. In *Meaning Coordination and Negotiation Workshop at the International Semantic Web Conference ISWC 2004, Proceedings*, 2004.
14. T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220, 1993.
15. S. H. Haeri, B. B. Hariri, and H. Abolhassani. Coincidence-based refinement of ontology matching. In *Joint 3rd International Conference on Soft Computing and Intelligent Systems and 7th International Symposium on advanced Intelligent Systems, SCIS+ISIS 2006, Proceedings*, 2006.
16. M. Lesk. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *5th Annual International Conference on Systems Documentation, SIGDOC ’86, Proceedings*, pages 24–26. ACM, 1986.
17. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady akademii nauk SSSR*, 163(4):845–848, 1965. In Russian. English Translation in Soviet Physics Doklady 10(8), 707-710, 1966.
18. F. Lin and K. Sandkuhl. A survey of exploiting wordnet in ontology matching. In M. Bramer, editor, *Artificial Intelligence in Theory and Practice II, IFIP 20th World Computer Congress*, volume 276 of *IFIP*, pages 341–350. Springer, 2008.
19. C. Meilicke, H. Stuckenschmidt, and A. Taminin. Repairing ontology mappings. In *22nd AAAI Conference on Artificial Intelligence, AAAI 2007, Proceedings*, pages 1408–1413. AAAI Press, 2007.
20. I. Niles and A. Pease. Towards a standard upper ontology. In C. Welty and B. Smith, editors, *FOIS 2001, 2nd International Conference on Formal Ontology in Information Systems, Proceedings*, pages 2–9. ACM Press, 2001.
21. R. Rada, H. Mili, E. Bicknell, and M. Blettner. Development and application of a metric on semantic nets. *IEEE Transactions on Systems, Man and Cybernetics*, 19(1):17–30, 1989.
22. P. Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *J. Artif. Intell. Res. (JAIR)*, 11:95–130, 1999.
23. P. Shvaiko. Iterative schema-based semantic matching. Technical Report DIT-06-102, DIT - University of Trento, 2005.
24. G. Stoilos, G. B. Stamou, and S. D. Kollias. A string metric for ontology alignment. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *4th International Semantic Web Conference, ISWC 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 624–637. Springer, 2005.
25. W3C. OWL Web Ontology Language Overview – W3C Recommendation 10 February 2004, 2004.
26. J. Walker, E. Pan, D. Johnston, J. Adler-Milstein, D. W. Bates, and B. Middleton. The value of healthcare information exchange and interoperability. *Health Affairs*, 2005. Web Exclusive, 19 January 2005.

Satisfiability Procedures for Combination of Theories Sharing Integer Offsets^{*}

Enrica Nicolini, Christophe Ringeissen, Michaël Rusinowitch

LORIA & INRIA Nancy Grand Est, France
E-mail: `FirstName.LastName@loria.fr`

Abstract. We present a novel technique to combine satisfiability procedures for theories that model some data-structures and that share the integer offsets. This procedure extends the Nelson-Oppen approach to a family of non-disjoint theories that have practical interest in verification. The result is derived by showing that the considered theories satisfy the hypotheses of a general result on non-disjoint combination. In particular, the capability of computing logical consequences over the shared signature is ensured in a non trivial way by devising a suitable complete superposition calculus.

1 Introduction

Satisfiability procedures for fragments of Arithmetics and data structures such as arrays and lists are at the core of many state-of-the-art verification tools, and their design and correct implementation is a hard task [5]. To overcome this difficulty, there is an obvious need for developing general and systematic methods to build decision procedures. Two important approaches have been investigated based respectively on combination and rewriting.

The *combination approach* for the satisfiability problem has been initiated in [13,16]. The methodology is to combine existing decision procedures for component theories in order to get a decision procedure for the union of the theories. In particular, the combination à la Nelson-Oppen is the core of many verification tools, even if the implementations often exploit ideas quite far from the original schema (see, e.g. [11,4]). This method assumes that component theories have disjoint signatures. An extension to the non-disjoint case has been proposed in [8,9], where the cooperation between the decision procedures relies on their capabilities of computing logical consequences built over the shared signature.

The *rewriting approach* allows us to flexibly build satisfiability procedures [2,1] based on a general calculus for automated deduction, namely the superposition calculus [15]. Hence, to obtain satisfiability procedures becomes easy by using an (almost) off-the-shelf theorem prover implementing superposition.

^{*} This paper is a presentation-only version of the paper “Satisfiability Procedures for Combination of Theories Sharing Integer Offsets” that appeared in the proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '09 [14].

These two approaches are complementary for two main reasons. First, combination techniques allow us to incorporate theories which are difficult to handle using rewriting techniques, such as Linear Arithmetics. Second, rewriting techniques are of prime interest to design satisfiability procedures which can be efficiently plugged into the disjoint combination framework [10]. In some particular cases, the rewriting approach is an alternative to the combination approach by allowing us to build superposition-based satisfiability procedures for combinations of finitely axiomatized theories, including the theory of Integer Offsets [1,3], but these theories must be over *disjoint* signatures.

In this paper, we show how to apply a superposition calculus to build decision procedures that can be plugged into the *non-disjoint* combination framework. We focus on theories sharing Integer Offsets. We present a superposition calculus dedicated to this theory and show the soundness of this new calculus for several *non-disjoint* extensions of this theory. The interest of combining counter arithmetic and uninterpreted functions in verification is advocated in [6], where uninterpreted functions are used for abstracting data and Integer Offsets allows us to express counters and a form of pointers, thanks to the successor function s and 0 . For instance, the possibility of using Integer Offsets enables us to consider (and combine) several models of lists:

- We can use the classical model of lists à la LISP, using cons , car , cdr operators, augmented with a length function ℓ defined as follows: $\ell(\mathit{cons}(e, x)) = s(\ell(x))$ and $\ell(\mathit{nil}) = 0$. In general, lists are over arbitrary elements but we may use also lists over integer elements.
- We can consider lists defined as records with two fields, the first one for the list itself, and the second one to store its length. Let us consider the operator $\mathit{rselect}_i$ to access to the i -th field of a record, $\mathit{rcons}(e, r)$ denotes the record obtained by adding an element e to the list of r , and rnil denotes the record corresponding to the empty list, we have the following axiomatization:

$$\begin{array}{ll} \mathit{rselect}_1(\mathit{rcons}(e, r)) = \mathit{cons}(e, \mathit{rselect}_1(r)) & \mathit{rselect}_1(\mathit{rnil}) = \mathit{nil} \\ \mathit{rselect}_2(\mathit{rcons}(e, r)) = s(\mathit{rselect}_2(r)) & \mathit{rselect}_2(\mathit{rnil}) = 0 \end{array}$$

This model of lists can be seen as a refinement of the first model in which one has a direct access to its “cardinality”.

The combination framework presented in the paper can be applied to decide the satisfiability of ground formulas expressed in the union of these two models of lists (provided both models use distinct names for list operators). Roughly speaking, such combination is useful to verify for instance that two programs written using different models of lists are “equivalent”.

Plan of the paper. After this introduction, Section 2 gives the main concepts and notations related to first-order theories. Section 3 recalls the non-disjoint combination framework of [9]. In Section 4, we present a superposition calculus dedicated to the theory of Integer Offsets. In Section 5, we give some examples of theories for which this superposition calculus can be turned into decision procedures. In Section 6, we show that this superposition calculus can be also

applied to deduce logical shared consequences. Moreover, all the requirements for applying the non-disjoint combination framework of [9] are satisfied by the extensions of Integer Offsets we are interested in. Finally, Section 7 concludes with some final remarks and some hints of future work. For lack of space, proofs are omitted and can be found in [14].

2 Preliminaries

A *signature* Σ is a set of functions and predicate symbols (each endowed with the corresponding arity). We assume the binary equality predicate symbol ‘=’ to be always present in any signature Σ (so, if $\Sigma = \emptyset$, then Σ does not contain other symbols than equality). The signature obtained from Σ by adding a set \underline{a} of new constants (i.e., 0-ary function symbols) is denoted by $\Sigma^{\underline{a}}$. Σ -*atoms*, Σ -*literals*, Σ -*clauses*, and Σ -*formulae* are defined in the usual way. A set of Σ -literals is called a Σ -*constraint*. Terms, literals, clauses and formulae are called *ground* whenever no variable appears in them; *sentences* are formulae in which free variables do not occur. Given a function symbol f , a f -rooted term is a term whose top-symbol is f .

From the semantic side, we have the standard notion of a Σ -*structure* $\mathcal{M} = (M, \mathcal{I})$: this is a support set M endowed with an arity-matching interpretation \mathcal{I} of the function and predicate symbols from Σ . The truth of a Σ -formula in \mathcal{M} is defined in any one of the standard ways. If $\Sigma_0 \subseteq \Sigma$ is a subsignature of Σ and if \mathcal{M} is a Σ -structure, the Σ_0 -*reduct* of \mathcal{M} is the Σ_0 -structure $\mathcal{M}|_{\Sigma_0}$ obtained from \mathcal{M} by forgetting the interpretation of function and predicate symbols from $\Sigma \setminus \Sigma_0$.

A collection of Σ -sentences is a Σ -theory, and a Σ -theory T admits *quantifier elimination* iff for every formula $\varphi(\underline{x})$ there is a quantifier-free formula (over the same free variables \underline{x}) $\varphi'(\underline{x})$ such that $T \models \varphi(\underline{x}) \leftrightarrow \varphi'(\underline{x})$.

In this paper, we are concerned with the (*constraint*) *satisfiability problem* for a theory T , also called the T -satisfiability problem, which is the problem of deciding whether a Σ -constraint is satisfiable in a model of T (and, if so, we say that the constraint is T -satisfiable). Notice that a constraint may contain variables: since these variables may be equivalently replaced by free constants, we can reformulate the constraint satisfiability problem as the problem of deciding whether a finite conjunction of ground literals in a simply expanded signature $\Sigma^{\underline{a}}$ is true in a $\Sigma^{\underline{a}}$ -structure whose Σ -reduct is a model of T .

3 Background on Combining Theories

We are interested in applying the general method developed in [9] for the combination of satisfiability procedures in unions of non-disjoint theories. This method extends the Nelson-Oppen combination schema known for unions of signature-disjoint theories, and leads to the following result:

Theorem 1. [9] *Consider two theories T_1, T_2 in signatures Σ_1, Σ_2 and suppose that:*

1. both T_1, T_2 have decidable constraint satisfiability problem;
2. there is some theory T_0 in the signature $\Sigma_1 \cap \Sigma_2$ such that:
 - T_0 is universal;
 - T_1, T_2 are both T_0 -compatible;
 - T_0 is Noetherian;
 - T_1, T_2 are both effectively Noetherian extensions of T_0 .

Then the $(\Sigma_1 \cup \Sigma_2)$ -theory $T_1 \cup T_2$ also has decidable constraint satisfiability problem.

The decidability result of Theorem 1 is obtained by relying on the available decision procedures for T_1 and T_2 , and cooperating them through an exchange of information over the shared signature $\Sigma_1 \cup \Sigma_2$. There are three crucial points in this schema: first of all, one should identify conditions sufficient to guarantee the correctness of the resulting procedure: this has been addressed requiring that the component theories must be both “compatible” with respect to a common sub-theory. Secondly, one should ensure the capability of computing the information to be exchanged: this issue is encoded into the requirement that the two theories T_1 and T_2 are “effectively Noetherian extensions” of a common sub-theory T_0 . Finally, one should guarantee that the exchange process eventually halts: the termination of the whole procedure is ensured thanks to the so called Noetherianity of T_0 . Let us explain in more details what the aforementioned requirements are.

Definition 1 (T_0 -compatibility [8]). Let T be a theory in the signature Σ and let T_0 be a universal theory in a subsignature $\Sigma_0 \subseteq \Sigma$. We say that T is T_0 -compatible iff $T_0 \subseteq T$ and there is a Σ_0 -theory T_0^* such that

- (i) $T_0 \subseteq T_0^*$;
- (ii) T_0^* has quantifier elimination;
- (iii) every Σ_0 -constraint which is satisfiable in a model of T_0 is satisfiable also in a model of T_0^* ;
- (iv) every Σ -constraint which is satisfiable in a model of T is satisfiable also in a model of $T_0^* \cup T$.

The requirements (i) to (iii) make the theory T_0^* unique, provided it exists (T_0^* is the so-called *model completion* of T_0). These requirements are a generalization of the stable infiniteness requirement of the Nelson-Oppen combination procedure: in fact, if T_0 is the empty theory in the empty signature, T_0^* is the theory axiomatizing an infinite domain, so that (iii) holds trivially and (iv) is precisely stable infiniteness.

Example 1. Let us consider the theory of Integer Offsets T_I :

$\boxed{T_I}$ rules the behaviour of the successor function s and the constant 0. T_I has the mono-sorted signature $\Sigma_I := \{0 : \text{INT}, s : \text{INT} \rightarrow \text{INT}\}$, and it is axiomatized as follows:

$$\begin{aligned}
& \forall x \ s(x) \neq 0 \\
& \forall x, y \ s(x) = s(y) \rightarrow x = y \\
& \forall x \ x \neq t(x) \quad \text{for all the terms } t(x) \text{ over } \Sigma_I \text{ that properly contain } x
\end{aligned}$$

T_I is a universal theory that admits model completion: indeed, if we add to T_I the axiom $\forall x(x \neq 0 \rightarrow \exists y x = s(y))$, we obtain a theory T_I^* that admits quantifier elimination (see, e.g. [7]) and such that every constraint that is satisfiable in a model of T_I is satisfiable also in a model of T_I^* . To justify the last claim, it is sufficient to observe that each model of T_I can be extended to a model of T_I^* simply by adding recursively to each element different from (the interpretation of) 0 a “predecessor”. Since this operation does not affect the truth of any constraint, we obtain that the condition (iii) is satisfied.

Now, for any theory $T \supseteq T_I$ over a signature $\Sigma \supseteq \Sigma_I$ the T_I -compatibility requirement simply reduces to the following condition: every constraint Γ that is satisfiable in a model of T must be satisfiable also in a model of $T \cup \forall x(x \neq 0 \rightarrow \exists y x = s(y))$.

The method for cooperating the satisfiability procedures makes use of the capability of deducing logical consequences over the shared signature. In order to ensure the termination when deducing those logical consequences, we rely on Noetherian theories. Intuitively, a theory is Noetherian if there exists only a finite number of atoms that are not redundant when reasoning modulo T_0 .

Definition 2 (Noetherian Theory [9]). *A Σ_0 -theory T_0 is Noetherian if and only if for every finite set of free constants \underline{a} , every infinite ascending chain*

$$\Theta_1 \subseteq \Theta_2 \subseteq \dots \subseteq \Theta_n \subseteq \dots$$

of sets of ground $\Sigma_0^{\underline{a}}$ -atoms is eventually constant modulo T_0 , i.e. there is an n such that $T_0 \cup \Theta_n \models A$, for every natural number m and atom $A \in \Theta_m$.

Example 2. (Example 1 continued). Many examples of Noetherian theories come from the formalization of algebraic structures, but an interesting class of Noetherian theories consists in all the theories whose signature contains only constants and one unary function symbol [17]. Thus, the theory of Integer Offsets T_I enjoys this property.

Let us consider now a theory $T \supseteq T_0$ with signature $\Sigma \supseteq \Sigma_0$, and suppose we want to discover, given an arbitrary set of ground clauses Θ over Σ , a “complete set” of logical positive consequences of Θ over Σ_0 , formalized by the notion of T_0 -basis.

Definition 3 (T_0 -basis). *Given a finite set Θ of ground clauses (built out of symbols from Σ and possibly further free constants) and a finite set of free constants \underline{a} , a T_0 -basis for Θ w.r.t. \underline{a} is a set Δ of positive ground $\Sigma_0^{\underline{a}}$ -clauses such that*

- (i) $T \cup \Theta \models C$, for all $C \in \Delta$ and
- (ii) if $T \cup \Theta \models C$ then $T_0 \cup \Delta \models C$, for every positive ground $\Sigma_0^{\underline{a}}$ -clause C .

Notice that in the definition of a basis we are interested only in positive ground clauses: the exchange of positive information is sufficient to ensure the

completeness of the resulting procedure. The interest in Noetherian theories lies in the fact that, for every set of Σ -clauses Θ and for every finite set \underline{a} of constants, a finite T_0 -basis for Θ w.r.t. \underline{a} always exists (Proposition 3.22 in [9]). Unfortunately, a basis for a Noetherian theory needs not to be computable; this motivates the following definition corresponding to the last hypothesis of Theorem 1:

Definition 4 ([9]). *Given a finite set \underline{a} of free constants, a T -residue enumerator for T_0 w.r.t. \underline{a} is a computable function $Res_{T_0}^{\underline{a}}(\Gamma)$ mapping a set of Σ -clauses Γ to a finite T_0 -basis for Γ w.r.t. \underline{a} ¹. A theory T is an effectively Noetherian extension of T_0 if and only if T_0 is Noetherian and there exists a T -residue enumerator for T_0 w.r.t. every finite set \underline{a} of free constants.*

Now we are ready to give a more detailed picture of the procedure that is the core of Theorem 1, and that extends the Nelson-Oppen combination method to theories over non disjoint signatures.

Algorithm 1 Extending Nelson-Oppen

Step 1. Purify the finite **input** set of ground $(\Sigma_1 \cup \Sigma_2)^{\underline{b}}$ -literals Γ , thus producing a finite set Γ_1 of ground $\Sigma_1^{\underline{a}}$ -literals and finite set Γ_2 of ground $\Sigma_2^{\underline{a}}$ -literals s.t. $\Gamma_1 \cup \Gamma_2$ is $T_1 \cup T_2$ -equisatisfiable with Γ .

Step 2. Using the T_i -residue enumerator $Res_{T_i}^{\underline{a}}$, check the output of $Res_{T_i}^{\underline{a}}(\Gamma_i)$:

If $Res_{T_i}^{\underline{a}}(\Gamma_i) = \Delta_i$ and $\Delta_i \neq \perp$ for each $i \in \{1, 2\}$, then

Step 2.1. For each $D \in \Delta_i$ such that $T_j \cup \Gamma_j \not\models D$, ($i \neq j$), add D to Γ_j

Step 2.2. If Γ_1 or Γ_2 has been changed in **Step 2.1**, then rerun **Step 2**

Else **return** “unsatisfiable”

Step 3. If this step is reached, **return** “satisfiable”.

In the following we will show how to discover theories that are amenable to be combined via the above schema and that share the theory of Integer Offsets. More in detail, we will focus on a particular extension of the superposition calculus that will proved to be a decision procedure for theories extending T_I and that will provide residue enumerators for T_I .

4 Superposition Calculus for Integer Offsets

Recent literature has focused on the possibility of using the superposition calculus in order to decide the satisfiability of ground formulae modulo the theory of Integer Offsets and some disjoint extensions [1,3]. Contrary to those papers, we are interested in a superposition-based calculus to deal with non-disjoint extensions of Integer Offsets, being able to constraint the successor symbol with additional axioms.

¹ If Γ is T -unsatisfiable, then without loss of generality a residue enumerator can always return the singleton set containing the empty clause.

Let us consider the axiomatization of the theory of Integer Offsets T_I defined in Example 1. Our aim is to develop a calculus able to take into account the axioms of T_I into a framework based on superposition. To this aim, let us consider a presentation of the superposition calculus specialized for reasoning over sets of literals, whose rules are described in Figures 1 and 2, augmented with the four rules over ground terms presented in Figure 3. as usual, we assume a term reduction ordering \prec which is total on ground terms.

$$\begin{array}{c}
\text{Superposition} \quad \frac{l[u'] = r \quad u = t}{(l[t] = r)\sigma} \quad (i), (ii) \\
\hline
\text{Paramodulation} \quad \frac{l[u'] \neq r \quad u = t}{(l[t] \neq r)\sigma} \quad (i), (ii) \\
\hline
\text{Reflection} \quad \frac{u' \neq u}{\perp} \\
\hline
\end{array}$$

where σ is the most general unifier of u and u' , u' is not a variable in *Superposition* and *Paramodulation*, L is a literal, \perp is the syntactic sign used to denote the inconsistency and the following hold: (i) $u\sigma \not\prec t\sigma$, (ii) $l[u']\sigma \not\prec r\sigma$.

Fig. 1. Expansion Inference Rules.

Let us adapt the standard definition of *derivation* to the calculus we are interested in:

Definition 5. Let \mathcal{SP}_I be the calculus depicted in Figures 1, 2 and 3. A *derivation* (δ) with respect to \mathcal{SP}_I is a (finite or infinite) sequence of sets of literals $S_1, S_2, S_3, \dots, S_i, \dots$ such that, for every i , it happens that:

- (i) S_{i+1} is obtained from S_i adding a literal obtained by the application of one of the rules in Figures 1, 2 and 3 to some literals in S_i ;
- (ii) S_{i+1} is obtained from S_i removing a literal according to one of the rules in Figures 2 or to the rule R1 or R2.

If we focus on the rules of Simplification, R1 and R2, we notice that the effects of the application of any of these rules involve two steps in the derivation: in the former a new literal is added, and in the latter a literal is deleted.

If S is a set of literals, let GS be the set of all the ground instances of S . A literal L is said to be *redundant* with respect to a set of literals S if, for all the ground instances $L\sigma$ of L , it happens that $\{E \mid E \in GS \ \& \ E \prec L\sigma\} \models L\sigma$. We notice that in our derivations only redundant literals are deleted:

Fact. If in a derivation S_{i+1} is equal to $S_i \setminus \{L\}$, then L is redundant with respect to S_i .

Proof. The claim above is well known if S_{i+1} is obtained from S_i applying one of the rules in Figure 2, and it follows immediately in the case we are applying R1 or R2.

<i>Subsumption</i>	$\frac{S \cup \{L, L'\}}{S \cup \{L\}}$	if $L\vartheta \equiv L'$ for some substitution ϑ
<i>Simplification</i>	$\frac{S \cup \{L[l'], l = r\}}{S \cup \{L[r\vartheta], l = r\}}$	if $l' \equiv l\vartheta$, $r\vartheta \prec l\vartheta$, and ($l\vartheta = r\vartheta$) $\prec L[l\vartheta]$
<i>Deletion</i>	$\frac{S \cup \{t = t\}}{S}$	

where L and L' are literals and S is a set of literals.

Fig. 2. Contraction Inference Rules.

<i>R1</i>	$\frac{S \cup \{s(u) = s(v)\}}{S \cup \{u = v\}}$	if u and v are ground terms
<i>R2</i>	$\frac{S \cup \{s(u) = t, s(v) = t\}}{S \cup \{s(v) = t, u = v\}}$	if u, v and t are ground terms and $s(u) \succ t$, $s(v) \succ t$ and $u \succ v$
<i>C1</i>	$\frac{S \cup \{s(t) = 0\}}{S \cup \{s(t) = 0\} \cup \perp}$	if t is a ground term
<i>C2</i>	$\frac{S \cup \{s^n(t) = t\}}{S \cup \{s^n(t) = t\} \cup \perp}$	if t is a ground term and $n \in \mathbb{N}$

where S is a set of literals and \perp is the symbol for the inconsistency.

Fig. 3. Ground Reduction Inference Rules.

So, as usual, we label with S_∞ the set of literals generated during a derivation δ (in symbols, $S_\infty = \bigcup_i S_i$), and with S_ω the set of persistent literals of δ : $S_\omega = \bigcup_i \bigcap_{j>i} S_j$. We adopt the standard definition for a rule π of the calculus being *redundant* with respect to a set of clauses S whenever, for every ground instance of the rule $\pi\sigma$ it happens that $\{E \mid E \in GS \ \& \ E \prec C_m\sigma\} \models D\sigma$, where $C_m\sigma$ is the maximal clause in the antecedent, and $D\sigma$ is the consequent of the rule. According to this definition, a derivation w.r.t. \mathcal{SP}_I is *fair* if, for every literal $L_1, L_2, \dots, L_m \in S_\omega$, every rule that has L_1, \dots, L_m as premises is redundant w.r.t. S_∞ .

Suppose now to take into account a fair derivation δ . We notice that, if a literal L is added at a certain step of the derivation, say S_{i+1} , then L is either a logical consequence of some literals in S_i , or it is a consequence of some literals in S_i and the axioms of the theory T_I . Thus:

Proposition 1. *If the set of persistent literals S_ω contains \perp , then S_ω is unsatisfiable in any model of T_I .*

On the other hand, since the reduction rules we can apply during the derivation satisfy the general requirements about the redundancy, we have that:

Proposition 2. *If the set of persistent literals S_ω does not contain \perp , then S_ω is satisfiable.*

What remains to show is that this calculus is *refutationally complete* with respect to the models of T_I (namely the structures in which the function \mathfrak{s} is injective, acyclic and such that 0 does not belong to the image of \mathfrak{s}). We want to identify in the following at least one case in which the calculus in Figures 1, 2 and 3 is not only refutationally complete w.r.t. T_I , but it is complete, too.

Remark 1. Since the satisfiability of S_ω is equivalent to the satisfiability of S_∞ , and since the satisfiability of each step S_{i+1} in the derivation implies the satisfiability of S_i , we have in particular that if S_ω is satisfiable, then S_0 is satisfiable. Moreover, it is immediate to check that the unsatisfiability in the models of T_I of S_ω implies the unsatisfiability of S_0 in the same class of structures. So, in case it happens that the calculus described in Figures 1, 2 and 3 is complete, we can proceed as usual when considering procedures based on saturation methods: an initial set of literals S_0 will be satisfiable (in a model of T_I) if and only if its saturation S_ω does not contain \perp .

4.1 Completeness

From now on, we assume that the ordering we consider when performing any application of \mathcal{SP}_I is T_I -good:

Definition 6. *We say that an ordering \succ over terms on a signature containing Σ_I is T_I -good whenever it satisfies the following requirements:*

- (i) \succ is a simplification ordering that is total on ground terms;
- (ii) 0 is minimal;
- (iii) whenever two terms t_1 and t_2 are not \mathfrak{s} -rooted it happens that $\mathfrak{s}^{n_1}(t_1) \succ \mathfrak{s}^{n_2}(t_2)$ iff either $t_1 \succ t_2$ or $(t_1 \equiv t_2$ and n_1 is bigger than $n_2)$.

Proposition 3. *Assuming T_I -good ordering \succ over terms, if the set of persistent literals S_ω satisfies the following assumptions:*

- S_ω does not contain \perp ,
- S_ω does not contain equation whose maximal term is a variable of sort INT, and \mathfrak{s} -rooted terms can be maximal just in ground equations.

then S_ω is satisfiable in a model of T_I .

Collecting all the results obtained so far, we can conclude that:

Theorem 2. *Let T be a Σ -theory presented as a finite set of unit clauses such that $\Sigma \supseteq \Sigma_I$, and assume to put an ordering over terms that is T_I -good. \mathcal{SP}_I induces a decision procedure for the constraint satisfiability problem w.r.t. $T \cup T_I$ if, for any set G of ground literals:*

- the saturation of $Ax(T) \cup G$ w.r.t. \mathcal{SP}_I is finite,
- the saturation of $Ax(T) \cup G$ w.r.t. \mathcal{SP}_I does not contain non-ground equations whose maximal term is \mathfrak{s} -rooted, or equations whose maximal term is a variable of sort INT.

4.2 Termination

Proposition 4. *For any set G of ground literals over a signature extending Σ_I , any saturation of G w.r.t. \mathcal{SP}_I is finite.*

Proof. Each step either adds a literal that is smaller than (at least) one literal already present in the saturation, or delete one literal, hence the multiset of literals decreases according to the well-founded ordering $((\succ)^{mul})^{mul}$.

Corollary 1. *\mathcal{SP}_I induces a decision procedure for the constraint satisfiability problem w.r.t. the union of T_I and the theory of equality.*

5 Examples of Integer Offsets Extensions

We investigate theories sharing symbols of T_I in a specific way, thanks to axioms of the form $g(f(\dots, x, \dots)) = s(g(x))$ where f, g are function symbols not occurring in Σ_I . Despite this restricted form of axioms, we are already able to consider interesting examples of Integer Offsets extensions.

5.1 Lists with Length

Let us consider T_{LLI} , the theory of lists endowed with length. T_{LLI} can be axiomatized as the union of the theories T_L , T_ℓ and T_I , where T_I is the theory of Integer Offsets of Example 1 and:²

T_L has the multi-sorted signature of the theory of lists: Σ_L is the set of function symbols $\{\text{nil} : \text{LISTS}, \text{car} : \text{LISTS} \rightarrow \text{ELEM}, \text{cdr} : \text{LISTS} \rightarrow \text{LISTS}, \text{cons} : \text{ELEM} \times \text{LISTS} \rightarrow \text{LISTS}\}$ plus the predicate symbol $\text{atom} : \text{LISTS}$, and it is axiomatized as follows:

$$\begin{array}{ll} \text{car}(\text{cons}(x, y)) = x & \neg \text{atom}(x) \rightarrow \text{cons}(\text{car}(x), \text{cdr}(x)) = x \\ \text{cdr}(\text{cons}(x, y)) = y & \neg \text{atom}(\text{cons}(x, y)) \\ & \text{atom}(\text{nil}) \end{array}$$

T_ℓ is the theory that gives the axioms for the function length $\ell : \text{LISTS} \rightarrow \text{INT}$:

$$\begin{array}{l} \ell(\text{nil}) = 0 \\ \ell(\text{cons}(x, y)) = s(\ell(y)) \end{array}$$

We want to show that the constraint satisfiability problem for T_{LLI} is decidable via the calculus described in the previous section.

² All the axioms should be considered as universally quantified.

First: reduction We start addressing the problem of checking the satisfiability of a constraint w.r.t. T_{LLI} . Let G be a set of ground literals over $\Sigma_{T_{LLI}}$; we can associate to G the set of formulae G' obtained by replacing all the literals in $G \cup \{\text{atom}(\text{nil})\}$ in the form $\neg\text{atom}(t)$ and $\text{atom}(t')$ with respectively $t = \text{cons}(sk_1, sk_2)$ and $\forall x_0, x_1 t' \neq \text{cons}(x_0, x_1)$, where t and t' are ground terms of sort LISTS and sk_1, sk_2 are fresh constants of the appropriate sort (this is the same reduction used in [2]).

Let now $T_{L'}$ be the subtheory of T_L whose axioms are just the (equational) axioms in the left column of the presentation of T_L . We have that:

Proposition 5. *G is satisfiable w.r.t. T_{LLI} if and only if G' is satisfiable w.r.t. $T_{L'} \cup T_\ell \cup T_I$.*

Second: saturation According to Proposition 5 and applying at most some standard steps of flattening, we can focus our attention to sets of literals of the following kinds (x is a variable of sort ELEM, y is a variable of sort LISTS, $h, l, a, f, g, l_1, l_2, e, d, e_1, e_2, i, i_1, i_2$ are constants of the appropriate sorts and the symbol \bowtie is a shortening for both $=$ and \neq), and the left-hand side of all the literals is the maximal one.

- | | |
|---|--|
| i.) equational axioms for lists | b) $\text{cdr}(f) = g$; |
| a) $\text{car}(\text{cons}(x, y)) = x$; | c) $l_1 \bowtie l_2$; |
| b) $\text{cdr}(\text{cons}(x, y)) = y$; | v.) ground literals over the sort ELEM |
| ii.) reduction for $\neg\text{atom}$ | a) $\text{car}(h) = d$; |
| a) $\text{cons}(x, y) \neq h$; | b) $e_1 \bowtie e_2$; |
| b) $\text{cons}(x, y) \neq \text{nil}$; | vi.) ground literals over the sort INT |
| iii.) axioms for the length | a) $\ell(a) = s^m(i)$; |
| a) $\ell(\text{nil}) = 0$; | b) $s^m(i_1) \neq s^n(i_2)$; |
| b) $\ell(\text{cons}(x, y)) = s(\ell(y))$; | c) $s^n(i_1) = i_2$; |
| iv.) ground literals over the sort LISTS | d) $i_1 = s^n(i_2)$. |
| a) $\text{cons}(e, l) = c$; | |

Let us choose, as ordering over the terms, a LPO ordering \succ whose underlying precedence over the symbols of the signature respects the following requirements:

- $\text{cons} > \text{cdr} > \text{car} > c > e > \ell$ for every constant c of sort LISTS and every constant e of sort ELEM;
- $\ell > i > 0 > s$ for every constant i of sort INT;

These requirements over the precedence guarantee that every compound term of sort LISTS is bigger than any constant, any compound term over the sort ELEM is bigger than any constant, and that \succ is a T_I -good ordering.

We require that the rules in Figures 2 and 3 are applied, whenever possible, before the rules in Figure 1 (in other words we require that the contraction rules have a higher priority).

Proposition 6. *For any set G of ground literals, any saturation of $Ax(T_{LLI}) \cup G$ w.r.t. \mathcal{SP}_I is finite.*

The key observations, in order to prove termination, are that the non-ground set of literals is already saturated, every equation obtained by the application of a rule to ground factors is smaller in the ordering w.r.t. the biggest factor in the antecedent of the rule, and every application of a rule of the calculus to a ground and a non-ground literal produces a ground literal that is smaller than the ground factor. In other terms, every literal produced during the saturation phase is ground and it is strictly smaller than the biggest ground literal in the input set. Since the ordering on the literals is the multiset extension of a terminating ordering, it is terminating too.

Moreover, since in the saturation no non-ground equation whose maximal term is s -rooted is generated, we can conclude by Theorem 2 that \mathcal{SP}_I is a decision procedure for the constraint satisfiability problem w.r.t. T_{LLI} .

5.2 Lists over Integer Elements

Let us consider now lists whose elements are integers. The reduction of Section 5.1 works without any changes, so we can check if the calculus developed in Figures 1, 2 and 3 is still a decision procedure for the constraint satisfiability problem of lists with length and integer elements. We can apply at most some standard steps of flattening and we focus our attention to sets of literals of the kinds i—iv) defined in Section 5.1 plus the new following one which merges the kinds v—vi) of Section 5.1:

v.) ground literals over the sort INT

- | | |
|-------------------------------|-----------------------|
| a) $\text{car}(h) = s^n(i)$; | d) $s^n(i_1) = i_2$; |
| b) $\ell(a) = s^m(i)$; | |
| c) $s^m(i_1) \neq s^n(i_2)$; | e) $i_1 = s^n(i_2)$. |

Let us put over the symbols of the signature an order that respects the same requirements we have asked in Section 5.1. The same remarks about termination and the shape of the saturated set of the previous section apply also to this case, guaranteeing that \mathcal{SP}_I provides a decision procedure.

5.3 Records with Increment

Let us consider records in which all the attribute identifiers are associated to the same sort INT, and suppose we want to be able to increment by a unity every value stored into the record. To formalize this situation, we can choose a signature as follows: let $Id = \{id_1, id_2, \dots, id_n\}$ a set of attribute identifiers and let us name REC the sort of records; for every attribute identifier id_1, id_2, \dots, id_n we have a couple of functions $\text{rselect}_i : \text{REC} \rightarrow \text{INT}$ and $\text{rstore}_i : \text{REC} \times \text{INT} \rightarrow \text{REC}$; moreover, there is also the increment function $\text{incr} : \text{REC} \rightarrow \text{REC}$. The axioms of the theory of integer records with increment, T_{IRI} , are the following:

$\boxed{T_{IRI}}$: for every i, j such that $1 \leq i, j \leq n, i \neq j$

$$\begin{aligned} \text{rselect}_i(\text{rstore}_i(x, y)) &= y \\ \text{rselect}_j(\text{rstore}_i(x, y)) &= \text{rselect}_j(x) \\ \bigwedge_{i=1}^n (\text{rselect}_i(x) = \text{rselect}_i(y)) &\rightarrow x = y && \text{(extensionality)} \\ \text{rselect}_i(\text{incr}(x)) &= \text{s}(\text{rselect}_i(x)) \end{aligned}$$

In order to check the satisfiability of a set of ground literals w.r.t. T_{IRI} , we notice that every literal of the kind $r_1 \neq r_2$ is equivalent to a clause of the kind $\bigvee_{i=1}^n \text{rselect}_i(r_1) \neq \text{rselect}_i(r_2)$, so can we substitute every disequation between records with the corresponding clause and then check the satisfiability of the resulting set of clauses by case split.

So we can restrict our attention to sets of literals in which no disequation between records appears. In this case, following the same argument used in [1], it is possible to check the satisfiability forgetting the extensionality axioms (the presence of the function incr does not affect the argument). Thus we are reduced to consider the saturation of sets of literals of the following kind:

- | | |
|---|---|
| i.) equational axioms for records | c) $\text{incr}(r_1) = r_2$; |
| a) $\text{rselect}_i(\text{rstore}_i(x, y)) = y$; | iii.) ground literals over the sort INT |
| b) $\text{rselect}_j(\text{rstore}_i(x, y)) = \text{rselect}_j(x)$; | a) $\text{rselect}_i(r) = \text{s}^n(k)$; |
| c) $\text{rselect}_i(\text{incr}(x)) = \text{s}(\text{rselect}_i(x))$; | b) $\text{s}^n(k_1) = k_2$; |
| ii.) ground literals over the sort REC | c) $k_1 = \text{s}^n(k_2)$; |
| a) $r_1 = r_2$; | d) $\text{s}^n(k_1) \neq \text{s}^m(k_2)$. |
| b) $\text{rstore}_i(r_1, \text{s}^n(k)) = r_2$; | |

where x is a variable of sort REC, y is a variable of sort INT, and r, r_1, r_2, k, k_1, k_2 are constants of appropriate sorts. As usual, let us consider a LPO ordering over terms such that the underlying precedence over the symbols in the signature satisfies the following requirements: for all i, j in $\{1, \dots, n\}$, $\text{incr} > \text{rstore}_i$, $\text{rstore}_i > \text{rselect}_j$, $\text{rselect}_i > c$ for every constant c and every constant c is such that $c > 0 > \text{s}$.

Proposition 7. *For any set G of ground literals, any saturation of $Ax(T_{IRI}) \cup G$ w.r.t. \mathcal{SP}_I is finite.*

The completeness of the calculus can be shown relying on the observation that no non-ground literals involving the function symbol s are generated, and that the chosen ordering is a T_I -good one.

6 Combination of Theories Sharing Integer Offsets

In the previous section we have collected examples of theories extending the theories of the Integers Offsets T_I and whose constraint satisfiability problem is decidable. We have already noticed that T_I admits a model completion T_I^* and that it is a Noetherian theory; to guarantee that these theories can be combined together it is sufficient to show that they satisfy the requirement of being T_I -compatible and effectively Noetherian extensions of T_I .

6.1 T_I -Compatibility

Being for a theory $T \supseteq T_I$ a T_I -compatible theory means that every constraint that is satisfiable w.r.t. T is satisfiable also in a model in which the axiom $\forall x(x \neq 0 \rightarrow \exists y x = s(y))$ holds. To see that actually it is the case for all the theories considered in Section 5, it is sufficient to check that any model of that theories can always be extended, if needed, adding recursively to each element that is different from (the interpretation of) 0 its predecessor and, in case it is needed, modifying accordingly the remaining part of the structure; and to check that this enlargement does not affect the validity both of the constraints that are verified in the structure and of the axioms of the theory. For example, we consider in the Appendix the case of the theory of lists over integer elements with length. Using similar (or simpler) arguments as the ones for this case, it is possible to verify that all the theories in Section 5 are T_I -compatible.

6.2 Derivation of T_I -bases

We have considered Horn Σ' -theories $T' = T \cup T_I$ extending T_I with some theories T axiomatized by unit clauses and we have shown under which assumptions the Superposition Calculus \mathcal{SP}_I is complete w.r.t. T' -satisfiability problems. Let us show that \mathcal{SP}_I allows us to derive T_I -basis. Assume that $G(\underline{a}, \underline{b})$ is a set of ground literals over an expansion of Σ' with the finite sets of fresh constants $\underline{a}, \underline{b}$. Our claim is the following: if S_ω is the saturation of $Ax(T) \cup G(\underline{a}, \underline{b})$ and assuming a T_I -good order over the terms in the signature $\Sigma' \cup \{\underline{a}, \underline{b}\}$ such that every term over the subsignature $\Sigma_T^{\underline{a}}$ is smaller than any term that contains a symbol in $(\Sigma' \setminus \Sigma_I) \cup \{\underline{b}\}$, then the subset of OGS_ω over the signature $\Sigma_T^{\underline{a}}$, denoted by $\Delta(\underline{a})$, is a T_I -basis. Since T' is a Horn theory, it is convex and so we can focus our attention just over equations instead of positive (ground) clauses.

Proposition 8. *If $l = r$ is an equation over $\Sigma_T^{\underline{a}}$ implied by $T' \cup G(\underline{a}, \underline{b})$, then $l = r$ is already implied by $T_I \cup \Delta(\underline{a})$, whenever S_ω is (i) finite, (ii) does not \perp , and such that (iii) s -rooted terms can be maximal just in ground equations in S_ω and (iv) variables of sort INT are never the maximal term into the equations.*

7 Conclusion

We have shown how to apply a superposition calculus to build decision procedures for some theories sharing Integer Offsets. These theories and the related decision procedures satisfy all the requirements for their applications in a non-disjoint combination framework. To the best of our knowledge, this paper is the first contribution showing the interest of a superposition calculus for non-disjoint combinations. This paper paves the way of using non-disjoint combinations (with a shared fragment of Arithmetics) in the context of verification. There are several research directions we want to investigate. Currently, the soundness of the superposition calculus is proved manually for each theory considered in the paper. It would be very interesting to have an automatic proof mechanism using

for instance a meta-saturation calculus [12]. Moreover, the considered fragment of Arithmetics is not very expressive and we have some limitations on the form of axioms we are able to handle. Further works are needed to go beyond these restrictions.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1).
2. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
3. M. P. Bonacina and M. Echenim. On variable-inactivity and polynomial T -satisfiability procedures. *Journal of Logic and Computation*, 18(1):77–96, 2008.
4. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient theory combination via boolean search. *Information and Computation*, 204(10):1493–1525, 2006.
5. A. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.
6. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of CAV 2002*, volume 2404 of *LNCS*, pages 78–92, Copenhagen (Denmark), 2002. Springer-Verlag.
7. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York-London, 1972.
8. S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3-4):221–249, 2004.
9. S. Ghilardi, E. Nicolini, and D. Zucchelli. A comprehensive combination framework. *ACM Transactions on Computational Logic*, 9(2):1–54, 2008.
10. H. Kirchner, S. Ranise, C. Ringeissen, and D.-K. Tran. On superposition-based satisfiability procedures and their combination. In *Proc. of ICTAC 2005*, volume 3722 of *LNCS*, pages 594–608, Hanoi (Vietnam), 2005. Springer-Verlag.
11. S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *Proc. of TACAS 2007*, volume 4424 of *LNCS*, pages 618–631, Braga (Portugal), 2007. Springer.
12. C. Lynch and D.-K. Tran. Automatic decidability and combinability revisited. In *Proc. of CADE-21*, volume 4603 of *LNCS*, pages 328–344, Bremen (Germany), 2007. Springer.
13. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transaction on Programming Languages and Systems*, 1(2):245–257, 1979.
14. E. Nicolini, C. Ringeissen, and M. Rusinowitch. Satisfiability procedures for combination of theories sharing integer offsets. In *Proc. of TACAS'09*, volume 5505 of *LNCS*, pages 428–442. Springer, 2009. Extended version as INRIA Report RR-6697.
15. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
16. R. E. Shostak. Deciding combinations of theories. *J. of the ACM*, 31:1–12, 1984.
17. D. Zucchelli. *Combination Methods for Software Verification*. PhD thesis, Università degli Studi di Milano and Université Henri Poincaré - Nancy 1, 2008.

A Logic-Based, Reactive Calculus of Events

Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni

DEIS, University of Bologna,
Viale Risorgimento 2, 40136 - Bologna, Italy
name.surname@unibo.it

Abstract. Since its introduction, the Event Calculus (\mathcal{EC}) has been recognized for being an excellent framework to reason about time and events, and it has been applied to a variety of domains. However, its use inside logic-based frameworks has been mainly a-posteriori, based on specific queries and backward reasoning. This has somehow limited its applicability in dynamic environments. We fill this gap by proposing a Reactive and logic-based implementation of the \mathcal{EC} , called $\mathcal{RE}\mathcal{C}$. We give an axiomatization of $\mathcal{RE}\mathcal{C}$ inside the SCIFF Abductive Logic Programming framework, and study its formal properties.

1 Introduction

More than 20 years ago, Kowalski and Sergot [8] introduced the Event Calculus (\mathcal{EC}) as a general framework to reason about time and events which overcomes limitations of other previous approaches, such as the situation calculus.

The event calculus has many interesting features. Among them: an extremely simple and compact representation, symmetry of past and future, generality with respect to time orderings, executability and direct mapping with computational logic frameworks, modeling of concurrent events, immunity from the frame problem, and explicit treatment of time and events. It has therefore been applied in a variety of domains. Among them, planning, via Abductive Logic Programming (ALP), as suggested by Eshghi [5], using backward, goal-oriented and hypothetical reasoning.

A decade later, following a different line of research, Kowalski and Sadri [7] proposed to use ALP as a way to reconcile backward with forward reasoning inside an intelligent agent architecture. However, beside planning, ALP has not been used in combination with the \mathcal{EC} . Nor are we aware of other logical frameworks that implement the \mathcal{EC} in a reactive way: i.e., a logical based implementation of \mathcal{EC} that dynamically reacts to happening events is missing.

For that reason, we only find reactive \mathcal{EC} implementations outside of logical frameworks, or else logic-based implementations of the \mathcal{EC} that do not exhibit any reactive feature. As a consequence, large application domains such as runtime monitoring and event processing have been tackled so far by \mathcal{EC} -inspired methods but only based on ad-hoc methods without a strong formal basis. In particular, it is very difficult to understand and prove the formal properties of current reactive \mathcal{EC} implementations.

In this work, we show how to overcome this limitation. We identify a new feature, not encompassed by the original \mathcal{EC} framework, named irrevocability. We deem such a feature necessary for monitoring applications. Building on Kowalski et al.’s work, we equip the \mathcal{EC} framework with the reactive features of a powerful, general purpose ALP language and proof-procedure named SCIFF. We obtain a reactive version of the calculus, which we call Reactive Event Calculus (\mathcal{REC}).

\mathcal{REC} exhibits irrevocability. It draws inspiration from Chittaro and Montanari’s work [3], in which they introduce the concept of *maximum validity intervals* (MVIs). These are the maximum time intervals in which fluents hold, according to the known events. Chittaro and Montanari propose a mechanism, called *Cached Event Calculus* (\mathcal{CEC}), to cache the outcome of the inference process every time the knowledge base is updated by a new event. Also \mathcal{REC} uses MVIs.

We demonstrate the advantage of \mathcal{REC} with respect to existing \mathcal{EC} implementations and investigate in depth the \mathcal{REC} ’s formal properties. We identify a class of *well-formed* specifications, which ensure that \mathcal{REC} is a viable method for runtime, event-based tracking of a system’s properties.

2 The Event Calculus

The Event Calculus (\mathcal{EC}) was introduced as a logic programming framework for representing and reasoning about events and their effects [8]. Basic concepts are that of *event*, happening at a point in time, and *fluent*, holding during time intervals. Fluents are initiated/terminated by events. Given an event narrative (a set of events), the \mathcal{EC} theory and domain-specific axioms together (“ \mathcal{EC} axioms”) define what fluents hold at each time. There are many different formulations of these axioms [4]. One possibility is given by the following axioms ec_1 , ec_2 (P stands for *Fluent*, E for *Event*, and T represents time instants):

$$\begin{aligned} holds_at(P, T) \leftarrow & initiates(E, P, T_{Start}) \\ & \wedge T_{Start} < T \wedge \neg clipped(T_{Start}, P, T). \end{aligned} \quad (ec_1)$$

$$\begin{aligned} clipped(T_1, P, T_3) \leftarrow & terminates(E, P, T_2) \\ & \wedge T_1 < T_2 \wedge T_2 < T_3. \end{aligned} \quad (ec_2)$$

$$\begin{aligned} initiates(E, P, T) \leftarrow & happens_at(E, T) \wedge holds_at(P_1, T) \\ & \wedge \dots \wedge holds_at(P_M, T). \end{aligned} \quad (ec_3)$$

$$\begin{aligned} terminates(E, P, T) \leftarrow & happens_at(E, T) \wedge holds_at(P_1, T) \\ & \wedge \dots \wedge holds_at(P_N, T). \end{aligned} \quad (ec_4)$$

Axioms (ec_3 , ec_4) are schemas for defining the domain-specific axioms: a certain fluent P is initiated/terminated at a time instant T if an event E happened at the same time, and if some other fluents P_i hold at that time. Sometimes *initially*(P) is used to define fluents that hold at the beginning of time. Dual axioms and predicates can be added to define when fluents *do/do not* hold [10]: e.g., an axiom can be added to define *declipped*/3 (an event has made a certain fluent holding).

The \mathcal{EC} framework has been extensively used in the past to carry out two main reasoning tasks: deductive *narrative verification*, to check whether a certain fluent holds given a narrative (set of events), [8], and abductive *planning*, to simulate a possible narrative which satisfies some requirements [11]. These tasks take place after or prior to execution, but not during execution. The reason is that each time an event occurs, the \mathcal{EC} enables a straightforward update of the theory (it suffices to add *happens_at* facts), but it incurs a substantial increase of the query time, since backward reasoning has to be restarted from scratch. However, runtime reasoning tasks, such as monitoring, would greatly benefit from the \mathcal{EC} 's expressive power. For this reason, some propose to cache the outcome of the inference process every time the knowledge base is updated by a new event. The *Cached Event Calculus* (\mathcal{CEC}) [3] computes and stores fluents' *maximum validity intervals* (MVIs), which are the maximum time intervals in which fluents hold, according to the known events. The set of cached validity intervals is then extended/revised as new events occur or get to be known.

3 The Reactive Event Calculus

The \mathcal{EC} can be elegantly formalized in logic programming, but as we said above, that would be suitable for top-down, “backward” computation. Runtime monitoring, intended as checking the behaviour of interacting entities, capturing at run time (as soon as possible) violations, and possibly raising alarms, requires reactive \mathcal{EC} implementations. For this reason, we resort to a framework which reconciles backward with forward reasoning: the SCIFF language and proof-procedure [1].

SCIFF is an extension of Fung and Kowalski's IFF proof-procedure for abductive logic programming [6]. It has two primitive notions: events (mapped as \mathbf{H} atoms) and expectations (mapped as \mathbf{E}/\mathbf{EN} atoms). $\mathbf{H}(Ev, T)$ means that an event Ev has occurred at time T , and it is a ground atom. $\mathbf{H}(Ev, T)$ is similar to Chittaro and Montanari's *happens_at* atom. Instead $\mathbf{E}(Ev, T)$ and $\mathbf{EN}(Ev, T)$ can contain variables with domains and CLP constraints, and they denote in the first case that an event unifying with Ev is expected to occur at some time in the range of T (T existentially quantified), and in the second case that all events unifying with Ev are expected to not occur, at all times in the range of T (i.e., T is considered as universally quantified over its CLP range). SCIFF accommodates existential and universal variable quantification and quantifier restriction, CLP constraints, dynamic update of event narrative and it has a built-in runtime protocol verification procedure. A SCIFF specification is composed of a knowledge base \mathcal{P} , a set of ICs (integrity constraints) \mathcal{IC} , and a goal \mathcal{G} . \mathcal{P} consists of backward rules $head \leftarrow body$ (see ax_1 below), whereas the ICs in \mathcal{IC} are forward implications $body \rightarrow head$ (see ax_2). ICs are interpreted in a reactive manner; the intuition is that when the body of an IC becomes true (i.e., the involved events occur), then the rule fires, and the expectations in the head are generated by abduction. For example, $\mathbf{H}(a, T) \rightarrow \mathbf{EN}(b, T')$ defines a relation between events a and b , saying that if a occurs at time T , b should not

occur at any time; $\mathbf{H}(a, T) \rightarrow \mathbf{E}(b, T') \wedge T' \leq T + 300$ says that if a occurs, then an event b should occur no later than 300 time units after a .

To exhibit a correct behavior, given a goal \mathcal{G} and a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$, a set of abduced expectations must be *fulfilled* by corresponding events. The SCIFF semantics [1] is given for a given specifications and narrative, denoted by **HAP** (a set of **H** atoms), and it intuitively states that \mathcal{P} , together with the abduced literals, must entail $\mathcal{G} \wedge \mathcal{IC}$, **E** expectations must have a corresponding matching happened event, and **EN** expectations must not have a corresponding matching event. Moreover, **E** and **EN** atoms must be consistent and not contradictory (this is formalized in [1] via the **E**-consistency notion).

The SCIFF axiomatization of \mathcal{EC} that follows draws inspiration from Chit-taro and Montanari’s \mathcal{CEC} and on their idea of MVIs. Events and fluents are terms and times are integer (CLP) variables, 0 being the “initial” time. The main distinctive feature of our implementation is its reactivity: fluents are initiated and terminated by dynamically occurring events. Thus its name, “Reactive Event Calculus” (\mathcal{REC}). \mathcal{REC} uses the abduction mechanism to generate MVIs and define their persistence. It has a fully declarative axiomatization (Axioms ax_1 through ax_7): no operational specifications are needed. It uses two special internal events (denoted by the reserved *clip/declip* words, differently from generic external events, taht are “incapsulated” into the reserved term *event*) to model that a fluent is terminated/initiated, respectively. The expressive power of \mathcal{REC} is the same as the one of \mathcal{CEC} , specifically it enables the definition of a context. A use case will be shown below.

Axiom 1 (Holding of fluent) *A fluent F holds at time T if a MVI containing T has been abduced for F .*¹

$$\text{holds_at}(F, T) \leftarrow \mathbf{mvi}(F, [T_s, T_e]) \wedge T > T_s \wedge T \leq T_e. \quad (ax_1)$$

Axiom ax_1 is a backward rule (clause), as well as Axiom ax_7 , whereas Axiom ax_2 through Axiom ax_6 are forward implications (ICs). Such a mixture of backward and forward inference rules is enabled by ALP [7] and it represents the backbone of \mathcal{REC} ’s reactive behaviour.

Axiom 2 (MVI semantics) *If $(T_s, T_e]$ is a MVI for F , then F must be de-clipped at time T_s and clipped at time T_e , and no further declipping/clipping must occur in between.*

$$\begin{aligned} & \mathbf{mvi}(F, [T_s, T_e]) \\ & \rightarrow \mathbf{E}(\text{declip}(F), T_s) \wedge \mathbf{E}(\text{clip}(F), T_e) \\ & \quad \wedge \mathbf{EN}(\text{declip}(F), T_d) \wedge T_d > T_s \wedge T_d \leq T_e \\ & \quad \wedge \mathbf{EN}(\text{clip}(F), T_c) \wedge T_c \geq T_s \wedge T_c < T_e. \end{aligned} \quad (ax_2)$$

¹ A fluent F does not hold at the time it is declipped, but holds at the time it is clipped, i.e., MVIs are left-open and right-closed.

Axiom 3 (Initial status of fluents) *If a fluent initially holds, a corresponding declipping event is generated at time 0.*

$$\text{initially}(F) \rightarrow \mathbf{H}(\text{declip}(F), 0). \quad (ax_3)$$

Operationally, Axiom ax_3 enforces the generation of a set of \mathbf{H} events that are needed for the correct extension of MVIs.

Axiom 4 (Fluents initiation) *If an event Ev occurs at time T which initiates fluent F , either F already holds or it is declipped.*

$$\begin{aligned} & \mathbf{H}(\text{event}(Ev), T) \wedge \text{initiates}(Ev, F, T) \\ & \rightarrow \mathbf{H}(\text{declip}(F), T) \\ & \vee \mathbf{E}(\text{declip}(F), T_d) \wedge T_d < T \\ & \wedge \mathbf{EN}(\text{clip}(F), T_c) \wedge T_c > T_d \wedge T_c < T. \end{aligned} \quad (ax_4)$$

(ax_4) does not use the *holds_at* predicate and it does not incur a new MVI.

Axiom 5 (Impact of initiation) *The happening of a declip(F) event causes fluent F to start to hold.*

$$\mathbf{H}(\text{declip}(F), T_s) \rightarrow \mathbf{mvi}(F, [T_s, T_e]) \wedge T_e > T_s. \quad (ax_5)$$

Axiom 6 (Fluents termination) *If an event Ev occurs which terminates a fluent F , F is clipped.*

$$\begin{aligned} & \mathbf{H}(\text{event}(Ev), T) \\ & \wedge \text{terminates}(Ev, F, T) \rightarrow \mathbf{H}(\text{clip}(F), T). \end{aligned} \quad (ax_6)$$

Axiom 7 (Final clipping of fluents) *All fluents are terminated by the special complete event.*

$$\text{terminates}(\text{complete}, _, F). \quad (ax_7)$$

4 \mathcal{REC} illustrated: a personnel monitoring facility

The following real-world case study has been proposed to us by a local medium-sized enterprise. A company wants to monitor its personnel's time-sheets. Each employee punches the clock when entering or leaving the office. The system recognizes two events:

- *check_in*(E): employee E has checked in;
- *check_out*(E): employee E has checked out.

The following requirements on employee behaviour require monitoring:

(R0) after check in, an employee must check out within 8 hours;

(R1) as soon as a deadline expiration is detected, a dedicated alarm fires at an operator’s desk. It reports the employee ID, and an indication of the time interval elapsed between deadline expiration and its detection. The alarm is turned off when the operator decides to handle it.

We assume that the following actions are available to the operator:

- *handle(E)* states that the operator wants to handle the situation concerning the employee identified by *E*;
- *tic* is used to take a snapshot of the current situation of the system, by updating the current time.

We capture requirements (R0) and (R1), using three fluents:

- *in(E)*: *E* is currently in;
- *should_leave(E, T_d)*: *E* is expected to leave her office by *T_d*;
- *alarm(delay(E, D))*: *E* has not left the office in time – *D* represents the difference between the time a deadline expiration is detected and the deadline expiration time itself.

It is possible to model such requirements declaratively using *initiates* and *terminates* predicate definitions. We assume hour time granularity.

Let us first focus on the *in(E)* fluent. *E* is “in” as of the time she checks in. She ceases to be “in” as of the time she checks out:

$$\textit{initiates}(\textit{check_in}(E), \textit{in}(E), -). \quad (1)$$

$$\textit{terminates}(\textit{check_out}(E), \textit{in}(E), T) \leftarrow \textit{holds_at}(\textit{in}(E), T). \quad (2)$$

When *E* checks in at *T_c*, a *should_leave* fluent is activated, expressing that *E* is expected to leave the office by *T_c + 8*:

$$\textit{initiates}(\textit{check_in}(E), \textit{should_leave}(E, T_d), T_c) \leftarrow T_d \textit{ is } T_c + 8. \quad (3)$$

(note that *T_c* is ground at *body* evaluation time, due to *ax₄*).

Such a fluent can be terminated in two ways: either *E* correctly checks out within the 8-hour deadline, or the deadline expires. In the latter case, termination is imposed at the next *tic* action.

$$\textit{terminates}(\textit{check_out}(E), \textit{should_leave}(E, T_d), T_c) \leftarrow \quad (4)$$

$$\textit{holds_at}(\textit{should_leave}(E, T_d), T_c) \wedge T_c \leq T_d.$$

$$\textit{terminates}(\textit{tic}, \textit{should_leave}(E, T_d), T) \leftarrow \quad (5)$$

$$\textit{holds_at}(\textit{should_leave}(E, T_d), T) \wedge T > T_d.$$

The same *tic* action also causes an alarm to go off:

$$\textit{initiates}(\textit{tic}, \textit{alarm}(\textit{delay}(E, D)), T) \leftarrow \quad (6)$$

$$\textit{holds_at}(\textit{should_leave}(E, T_d), T) \wedge D \textit{ is } T - T_d.$$

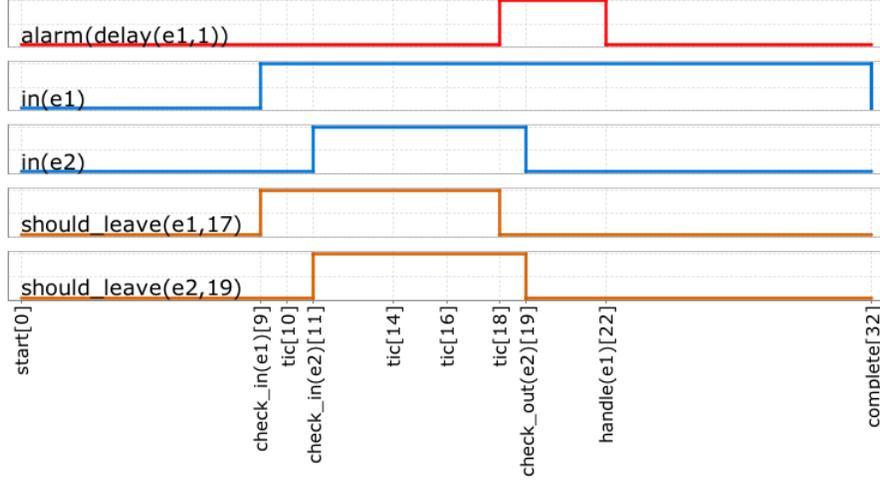


Fig. 1. Fluents tracking with \mathcal{REC} .

Note that in these equations the time of event termination/start (T_c and T) is the same time present in their respective body's *holds_at* atoms. This is perfectly normal, but it is not a requirement. In particular, times could be different, as long as the times of *holds_at* atoms do not follow event termination/start times. That again would be allowed, but it would amount to define fluents that depend on future events: fluents that are thus not suitable for runtime monitoring. For that reason, we assume that *well-formed* theories do not contain such kind of clauses. More details on this matter will be given below when we discuss the *irrevocability* property in a formal way.

Finally, an alarm is turned off when the operator handles it:

$$\text{terminates}(\text{handle}(E), \text{alarm}(\text{delay}(E, D)), T) \leftarrow \text{holds_at}(\text{delay}(E, D), T). \quad (7)$$

Based on such a theory, \mathcal{REC} becomes able to dynamically reason from the employees' flow inside the company. In particular, \mathcal{REC} tracks the status of each employee, and generates an alarm as soon as a *tic* action detects a deadline expiration.

Let us consider an event narrative (*execution trace*) involving two employees e_1 and e_2 , where e_2 does respect the required deadline while e_1 does not:

$$\begin{aligned} & \mathbf{H}(\text{event}(\text{check_in}(e_1)), 9), & \mathbf{H}(\text{event}(\text{tic}), 10), & \mathbf{H}(\text{event}(\text{check_in}(e_2)), 11), \\ & \mathbf{H}(\text{event}(\text{tic}), 14), & \mathbf{H}(\text{event}(\text{tic}), 16), & \mathbf{H}(\text{event}(\text{tic}), 18). \end{aligned}$$

Figure 1 shows the global state of fluents at 18, when \mathcal{REC} generates an alarm because e_1 was expected to leave the office no later than 17, but she has not left yet. The operator can check all pending alarms, and pick an employee to handle

in case. The execution now proceeds as follows:

$$\begin{aligned} & \mathbf{H}(\text{event}(\text{check_out}(e_2)), 19), & \mathbf{H}(\text{event}(\text{handle}(e_1)), 22), \\ & \mathbf{H}(\text{event}(\text{check_out}(e_1)), 23). \end{aligned}$$

e_2 correctly leaves the office within the deadline, bringing her corresponding *in* and *should_leave* fluents to termination. At 22 the operator handles an alarm involving e_1 , who eventually leaves her office at 23.

In general, a monitoring application is usable only if it provides stable, deterministic outputs, which do not flutter due to algorithmic issues but only change as a sensible response to the inputs. The reasons of an undesired fluttering behaviour could be of two types: a bad set of specifications, or an unsuitable underlying reasoning machinery.

As an example of the former, imagine to replace Eq. (1) by

$$\text{initiates}(\text{check_in}(e_1), \text{in}(E_2), -), \quad (8)$$

in which E_2 represents a generic employee. This is an ambiguous specification since it does not clearly state which employee should change status as a consequence of e_1 checking in. Another different source of ambiguity could lay hidden in an alternative formulation of Eq. (6) such as the following:

$$\begin{aligned} & \text{initiates}(\text{tic}, \text{alarm}(\text{delay}(E, D), T) \leftarrow & (9) \\ & \text{holds_at}(\text{should_leave}(E, T_d), T_1) \wedge D \text{ is } T - T_d, T_1 > T. \end{aligned}$$

The meaning would be that an action is a consequence of an alarm which has not fired yet. Only speculations are possible in that case, and no runtime monitoring algorithm could provide deterministic answers (save freezing until the alarm fires, but in that case the application would no longer be called “runtime”).

Thus we need to isolate “good” sets of specifications. Once we have them, we must ensure that the reasoning machinery does not make unjustified retractions. In other words, we must guarantee irrevocability.

5 Formal properties of \mathcal{REC}

\mathcal{REC} is implemented on top of SCIFF. As a consequence, it inherits soundness and completeness properties of the SCIFF’s operational semantics with respect to the declarative semantics.

Theorem 1 (Soundness and completeness of \mathcal{REC}). *\mathcal{REC} is sound and complete. Specifically, the SCIFF proof-procedure will derive all and only the answers defined by its corresponding declarative semantics, augmented with the \mathcal{REC} axioms ax_1 – ax_7 .*

Proof. It follows from SCIFF’s soundness and completeness properties [1]. \square

Hence, the operational behaviour of the \mathcal{REC} is faithful to its axiomatization. We are unaware of other implementations of the \mathcal{EC} that provide such a guarantee. The next results concern uniqueness and irrevocability, and they are instead relative to the special kind of reasoning needed for the monitoring applications. We thus need to introduce some information about the SCIFF's operational behaviour.

5.1 Open, closed and semi-open reasoning

In general, SCIFF features two main forms of inference, called *open* and *closed derivation*. A derivation starts from a (possibly empty) goal and \mathcal{IC} , and it generates a list of nodes according to the operational semantics [1], i.e., by applying a SCIFF transition rule at a time. It terminates when no transition is applicable. Open and closed derivations differ by one such transition.

Given a specification \mathcal{S} and two *execution traces* (sets of \mathbf{H} events) \mathcal{H}^i and $\mathcal{H}^f \supseteq \mathcal{H}^i$, if there exists an *open successful derivation* [2] for a goal \mathcal{G} that leads from \mathcal{H}^i to \mathcal{H}^f we write $\mathcal{S}_{\mathcal{H}^i} \vdash_{\Delta}^{\mathcal{H}^f} \mathcal{G}$, where Δ is the computed abductive explanation.² If \mathcal{S} is a \mathcal{REC} specification, Δ includes the abduced MVIs. When SCIFF executes an open derivation, it assumes that the acquired execution trace is partial. Thus \mathbf{E} atoms without a matching \mathbf{H} atom are not considered as violated but only as *pending*: further events may still occur to fulfill them. \mathbf{EN} atoms can instead be evaluated, because they must never have a matching \mathbf{H} atom. This approach is used when SCIFF is used for runtime verification, with events occurring dynamically, and the narrative is incomplete.

SCIFF can also perform *closed derivations*, to reason from narratives known to be complete, or to close the inference process when a dynamic execution comes to an end. In that case, both \mathbf{E} and \mathbf{EN} atoms are evaluated: a CWA is made about the collected execution trace, and pending expectations are considered as violated, because no further event will occur to fulfill them.

SCIFF is sound and complete independently of the order of events. However, there are many important domains in which we can safely assume that events are acquired in increasing order of time. In that case, reasoning is *partially open*: open on the future, when events may still occur, but closed on the past. Expectations on the past can thus be evaluated immediately. To enable this form of *semi-open* reasoning, the SCIFF proof-procedure is equipped with an additional rule, which states that if the execution trace has reached time t , then all pending expectations must be fulfilled at a time $t' \geq t$. We denote such a *semi-open* derivation by \vdash_{\sim} .

5.2 Irrevocability of \mathcal{REC}

Monitoring applications enable semi-open derivation. It is required that the generated MVIs are never retracted, but only extended or terminated as new events

² Note that Δ only depends on \mathcal{H}^f , because \mathcal{H}^f includes \mathcal{H}^i . Therefore, in the following we will omit \mathcal{H}^i when possible.

occur. If that is the case, the reasoning process is called *irrevocable*. Some target applications need irrevocable tracking procedures, which can give a continuously updated view of the status of all fluents. Fluttering behaviours must be by all means avoided. This is true, e.g., when the modeled fluents carry a normative meaning. It would be undesirable, for instance, to attach a certain obligation to an agent at runtime, and see it disappear later only because the calculus revises its computed status.

In the remainder of this section, we first define a class of \mathcal{REC} theories, then we show that semi-open reasoning on the resulting \mathcal{REC} specifications is irrevocable. Note that when monitoring the execution, \mathcal{REC} is used with goal *true*.

Definition 1 (Well-formed \mathcal{REC} theory). *A well-formed \mathcal{REC} theory \mathcal{T} is a set of clauses of the type*³

$$\begin{aligned} \text{initiates}(Ev, F, T) &\leftarrow \text{body}. \\ \text{terminates}(Ev, F, T) &\leftarrow \text{body}. \end{aligned}$$

which satisfies the following properties:

1. *negation is not applied to holds_at predicates;*
2. *for initiates/3 clauses, fluent F must always be resolved with a ground substitution.*
3. *$\forall \text{holds_at}(F_2, T_2)$ predicate used in body, $T_2 \leq T$.*

In the previous section our illustration was modeled by a well-formed \mathcal{REC} theory. Using the same example, we have discussed the consequences of ill-formed specifications in the \mathcal{REC} theory in a concrete case. Definition 1 identifies in the general case the three possible sources of non irrevocability. In particular, 1. ensures monotonicity, 2. prevents non-determinism due to case analysis and 3. restricts us to reasoning on stable, past conditions. \mathcal{REC} theories that violate 2. and 3. would introduce choice points that hinder irrevocability.

Definition 2 (\mathcal{REC} specification). *Given a well-formed \mathcal{REC} theory \mathcal{T} , the corresponding \mathcal{REC} specification $\mathcal{R}^{\mathcal{T}}$ is defined as the SCIFF specification.*⁴

$$\mathcal{R}^{\mathcal{T}} \equiv \langle \text{KB}_{\mathcal{REC}} \cup \mathcal{T}, \{\mathbf{E}, \mathbf{EN}, \mathbf{mvi}\}, \text{IC}_{\mathcal{REC}} \rangle \quad (10)$$

where $\text{KB}_{\mathcal{REC}} = \{(ax_1), (ax_7)\}$ and $\text{IC}_{\mathcal{REC}} = \{(ax_2), (ax_3), \dots, (ax_6)\}$.

The following three lemmas establish interesting properties of \mathcal{REC} , defining the link between MVIs and the internal events used to clip and declip them.

³ The *body* is a conjunction of *holds_at* predicates and CLP constraints. It can be omitted when *true*.

⁴ Throughout the paper, we will simply use \mathcal{R} to identify a generic \mathcal{REC} specification. We will also state that a \mathcal{REC} specification is well-formed as a shortcut to state that its theory is.

Lemma 1 (Groundedness of MVIs' starting times). *For each well-formed REC theory \mathcal{T} , for each execution trace \mathcal{H} and given the goal true, the abduced MVIs always have a ground starting time, i.e.*

$$\forall \Delta, \mathcal{T} \vdash_{\Delta}^{\mathcal{H}} \text{true} \Rightarrow \forall \text{ mvi}(F, [T_s, T_e]) \in \Delta, T_s \in \mathbb{N}$$

Proof. Axiom (ax_5) is the only IC in which the abducible representing a MVI appears in the head. The starting time T_s is bound to the time at which the *declip* event in the body happens. This event is not contained in the execution trace, but is instead generated by applying axiom (ax_3) or (ax_4). By axiom (ax_3), a *declip* event is generated at time 0, whereas by axiom (ax_4), a *declip* event is generated using the same time of the body's external **H** event. Since all happened events contained in an execution trace are ground, then also the *declip* event is abduced to happen at a ground time. Therefore, also the starting time T_s of the corresponding MVI is ground. \square

Lemma 2 (Relationship between clipping events and MVIs). *The expectation about the clipping of a MVI can be fulfilled by exactly one happened event, in particular the nearest which occurs after the declipping of the MVI.*

Proof. Let us consider by absurdum that there exists $\text{ mvi}(f, [t_s, T_e])$ triggering Axiom (ax_2) and generating an expectation about *clip*(f) which can potentially be fulfilled by two distinct event happening, say, at time $t_{e1} > t_s$ and $t_{e2} > t_{e1}$, are able to fulfill it. If $\mathbf{E}(\text{clip}(f), T_e)$ is fulfilled at time t_{e2} , then Axiom (ax_2) also imposes $\mathbf{EN}(\text{clip}(f), T_c)$ between t_s and t_{e2} . However, this time interval also includes t_{e1} , leading to inconsistency. \square

Lemma 3 (Interleaving between declipping and clipping events). *For each fluent, between two declipping events at least one clipping event must occur.*

Proof. Let us suppose that there exists a fluent f for which two *declip*(f) events occur, say, at time t_1 and $t_2 > t_1$, without having a *clip*(f) inbetween. An MVI is generated for f starting at t_1 (thank to Axiom ax_5); this MVI will be clipped by the first consequent time at which a *clip*(f) occurs (see Lemma 2). The hypothesis which states that this *clip*(f) event must occur after t_2 is however inconsistent, because Axiom ax_2 would state that between t_1 and this time (thus t_2 included) no further *declip*(f) event can occur. \square

We are now ready to state the following:

Theorem 2 (Uniqueness of derivation). *For each well-formed REC theory \mathcal{T} and for each execution trace \mathcal{H} , there exists exactly one successful semi-open derivation computed by SCIFF for the goal true, i.e. $\exists! \Delta$ s.t. $\mathcal{T} \vdash_{\Delta}^{\mathcal{H}} \text{true}$.*

Proof. First, we prove that at most one computed explanation exists. Different explanations are computed when inclusive disjunctions are contained in the head of some integrity constraint, or if there are different ways to fulfill a positive expectation.⁵

⁵ The other possibilities of having multiple explanations are ruled out by the fact that *initiates* predicate are resolved with a ground fluent, thus MVIs are always ground.

Let us first consider the case of disjunctions in the head. The only integrity constraint containing a disjunctive head is (ax_4) ;⁶ however, we prove that these two disjuncts are mutually exclusive. Let us consider the second disjunct. It states that fluent F has been already declipped at a certain past time (let us denote it with t_d), and that between t_d and T no clipping event has been generated: thus F still holds at time T . In fact, when $declip(F)$ happened at time t_d , a *mvi* abducible was generated by applying rule (ax_5) ; due to the negative expectation contained in the second disjunct of (ax_5) 's head, this maximal validity interval must be clipped at a time greater than T (say, t_c). The application of rule (ax_2) , in turn, states that it is forbidden to declip F between t_d and t_c , i.e., also at time T . Therefore, the first disjunct of (ax_5) 's head cannot be true at time T . A further important observation concerns the semi-open nature of the derivation. Even if the first disjunct of Axiom (ax_5) did not lead to violation, an open derivation would open a choice point, waiting for a suitable past declipping event to fulfill the expectation in the second disjunct. As we have just proven, this second possibility is impossible, because the two disjuncts are mutually exclusive. A semi-open derivation is immediately able to detect this situation, because the second disjunct refers to the past, and the first one to the present. Therefore, no choice point is left open.

Let us now consider the reasons to fulfill positive expectations, which are present in axioms (ax_2) and (ax_4) . $\mathbf{E}(declip(F), T_s)$ in Axiom (ax_2) can be fulfilled in one way, because both F and T_s are ground, the former because \mathcal{T} is well-defined by hypothesis, the latter as stated in Lemma 1. By Lemma 2, also the expectation about the clipping of the fluent can be fulfilled by exactly one event, in particular by the first clipping occurring after $declip$. Finally, the positive expectation in Axiom (ax_4) can be fulfilled by only one $declip$ event; the proof is obtained by combining the negative expectation about the clipping of the fluent in Axiom ax_4 and the result proven in Lemma 3.

Now we prove that there always exists a computed explanation, i.e. that, when the goal is *true*, all execution traces comply with the \mathcal{REC} specifications. The axioms imposing expectations are (ax_2) and (ax_4) . If the positive expectation in the second disjunct of (ax_4) 's head cannot be fulfilled, then the involved fluent is not holding, and therefore it can be declipped by choosing the first disjunct. When the goal is *true*, Axiom (ax_2) only fires if Axiom (ax_5) fires, and therefore the positive expectation about the declipping of the fluent has a corresponding matching event (exactly the one which has caused Axiom (ax_5) to fire). The positive expectation about the clipping of fluent is eventually fulfilled by the special *complete* event, which is able to terminate all fluents (see Axiom (ax_7)). Finally, the negative expectations contained in Axioms (ax_2) and (ax_4) are simply used to select the “nearest” declipping/clipping, but are not used to rule out executions. \square

Theorem 2 ensures that exactly one Δ is produced by a semi-open derivation of SCIFF. This, in turn, means that there exists exactly one “configuration” for

⁶ The presence of negated abducibles in the body of a rule would also produce inclusive disjunctive head [6], but Definition 1 forbids negated *holds_at* predicates.

the MVIs of each fluent. We give a precise definition of this notion of state, which is the one of interest when evaluating the irrevocability of the reasoning process, and define the notion of progressive extension between states, which formally define irrevocability.

Definition 3 (Current time). *The current time of an execution trace \mathcal{H} , $ct(\mathcal{H})$, is the latest time of its events:*

$$ct(\mathcal{H}) \equiv \max(t \mid \mathbf{H}(\text{event}(\cdot), t) \in \mathcal{H})$$

Definition 4 (MVI State). *Given a \mathcal{REC} specification \mathcal{R} and an execution trace \mathcal{H} the resulting MVI state at time $ct(\mathcal{H})$ is the set of **mvi** abducibles contained in the computed explanation generated by SCIFF with goal true:*

$$\text{MVI}(\mathcal{R}_{\mathcal{H}}) \equiv \{ \mathbf{mvi}(F, [T_s, T_e]) \in \Delta \}, \text{ where } \mathcal{R} \vdash_{\Delta}^{\mathcal{H}} \text{true}$$

Definition 5 (State sub-sets). *Given a \mathcal{REC} specification \mathcal{R} and a (partial) execution trace \mathcal{H} , the current state $\text{MVI}(\mathcal{R}_{\mathcal{H}})$ is split into two sub-sets:*

- $\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}})$, is the set of (closed) MVIs, terminating at a ground time:

$$\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}}) = \{ \mathbf{mvi}(F, [s, e]) \in \text{MVI}(\mathcal{R}_{\mathcal{H}}) \mid s, e \in \mathbb{N} \};$$

- $\text{MVI}_{\lceil}(\mathcal{R}_{\mathcal{H}})$, is the set of (open) MVIs, terminating at a variable time:

$$\text{MVI}_{\lceil}(\mathcal{R}_{\mathcal{H}}) = \{ \mathbf{mvi}(F, [s, T]) \in \text{MVI}(\mathcal{R}_{\mathcal{H}}) \mid s \in \mathbb{N} \}.$$

Definition 6 (Trace extension). *Given two execution traces \mathcal{H}^1 and \mathcal{H}^2 , \mathcal{H}^2 is an extension of \mathcal{H}^1 , written $\mathcal{H}^1 < \mathcal{H}^2$, iff*

$$\forall \mathbf{H}(e, t) \in \mathcal{H}^2 / \mathcal{H}^1, t > ct(\mathcal{H}^1)$$

Definition 7 (State progressive extension). *Given a well-formed \mathcal{REC} specification \mathcal{R} and two execution traces \mathcal{H}^1 and \mathcal{H}^2 , the state of $\mathcal{R}_{\mathcal{H}^2}$ is a progressive extension of the state of $\mathcal{R}_{\mathcal{H}^1}$, written $\text{MVI}(\mathcal{R}_{\mathcal{H}^1}) \preceq \text{MVI}(\mathcal{R}_{\mathcal{H}^2})$, iff*

1. the set of closed MVIs is maintained in the new state: $\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^1}) \subseteq \text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^2})$
2. if the set of MVIs is extended with new MVIs, these are declipped after the maximum time of \mathcal{H}^1 : $\forall \mathbf{mvi}(f, [s, t]) \in \text{MVI}(\mathcal{R}_{\mathcal{H}^2}) / \text{MVI}(\mathcal{R}_{\mathcal{H}^1}), s > ct(\mathcal{H}^1)$
3. $\forall \mathbf{mvi}(f, [s, T_e]) \in \text{MVI}_{\lceil}(\mathcal{R}_{\mathcal{H}^1})$, either
 - (a) it remains untouched in the new state: $\mathbf{mvi}(f, [s, T_e]) \in \text{MVI}_{\lceil}(\mathcal{R}_{\mathcal{H}^2})$, or
 - (b) it is clipped after the maximum time of \mathcal{H}^1 : $\mathbf{mvi}(f, [s, e]) \in \text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^2}), e > ct(\mathcal{H}^1)$.

Progressive extensions capture the intuitive notion that a state extends another one if it keeps the already computed closed MVIs and affects the status of fluents only at later times w.r.t. the time the first state was recorded. The extension is determined by adding new MVIs and by clipping fluents which held at the previous state. We can state the main result leading to irrevocability, namely that extending a trace results in a progressive extension of the MVI state.

Lemma 4 (Trace extension leads to a state progressive extension).
Given a well-formed \mathcal{REC} specification \mathcal{R} and two execution traces \mathcal{H}^1 and \mathcal{H}^2 ,

$$\mathcal{H}^1 \prec \mathcal{H}^2 \Rightarrow \text{MVI}(\mathcal{R}_{\mathcal{H}^1}) \trianglelefteq \text{MVI}(\mathcal{R}_{\mathcal{H}^2})$$

Proof. Let us consider Definition 7, showing that $\text{MVI}(\mathcal{R}_{\mathcal{H}^1})$ and $\text{MVI}(\mathcal{R}_{\mathcal{H}^2})$ obey to its three requirements:

1. Let us consider an element of $\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^1})$, say, $\mathbf{mvi}(f, [s, e])$. The presence of this element is evidence that an event $\in \mathcal{H}^1$ occurring at time e exists s.t. fluent f is declipped. Since this event belongs to \mathcal{H}^1 , $e < ct(\mathcal{H}^1)$. From the hypotheses that $\mathcal{H}^1 \prec \mathcal{H}^2$, all the events that belong to $\mathcal{H}^2/\mathcal{H}^1$ happen at a time greater than $ct(\mathcal{H}^1)$. Lemma 2 thus ensures that none of these events can change $\mathbf{mvi}(f, [s, e])$, which is maintained untouched in $\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^2})$.
2. As pointed out in the proof of Lemma 1, a happened event can start a new MVI at the time it happens. Since $\mathcal{H}^1 \prec \mathcal{H}^2$, each MVI generated by events belonging to $\mathcal{H}^2/\mathcal{H}^1$ will always have a starting time greater than $ct(\mathcal{H}^1)$. The only unfortunate case would be that there exists an event associated to an *initiates* predicate that can be evaluated as true *after* the time at which the event occurs. This case arises when the *initiates* predicate is defined in terms of *holds_at* predicates which refer to the future.⁷ However, Definition 1 rules out theories of this kind.
3. Axiom ax_6 is the rule which regulates the way fluents are clipped; a happened event can cause the termination of a MVI exactly at the time at which it occurs. From the hypotheses that $\mathcal{H}^1 \prec \mathcal{H}^2$, if a MVI is open at time $ct(\mathcal{H}^1)$ (i.e., it belongs to $\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^1})$), it is impossible for an event belonging to $\mathcal{H}^2/\mathcal{H}^1$ to clip it before time $ct(\mathcal{H}^1)$. Indeed, as in the case of *initiates* predicates, theories which lead to violate this property are not well-formed. Two possible cases for such an MVI may then arise:
 - (a) no event $\in \mathcal{H}^2/\mathcal{H}^1$ is able to terminate the fluent associated to the MVI, which is therefore maintained untouched in $\text{MVI}_{\perp}(\mathcal{R}_{\mathcal{H}^1})$;
 - (b) there exists at least one event $\in \mathcal{H}^2/\mathcal{H}^1$ able to terminate the fluent associated to the MVI; the first one (see Lemma 2) will shift the MVI from the open to the closed set and respect the requirement that the MVI termination time must be greater than $ct(\mathcal{H}^1)$. \square

Theorem 3 (Irrevocability of \mathcal{REC}). *Given a well-formed \mathcal{REC} specification with goal true and a temporally ordered narrative, each time SCIFF processes a new event, the new MVI state is a progressive extension of the previous one.*

Proof. Let us suppose that the current execution trace is \mathcal{H}^1 , and that a new happened event $\mathbf{H}(e, t)$ is acquired by SCIFF. Let us denote the new execution trace with \mathcal{H}^2 , having $\mathcal{H}^2 = \mathcal{H}^1 \cup \{\mathbf{H}(e, t)\}$. If the execution of the system always grows by increasing times, then $t > ct(\mathcal{H}^1)$. Therefore, from Definition 6 it holds that $\mathcal{H}^1 \prec \mathcal{H}^2$ and, in turn, Lemma 4 ensures that $\text{MVI}(\mathcal{R}_{\mathcal{H}^1}) \trianglelefteq \text{MVI}(\mathcal{R}_{\mathcal{H}^2})$, i.e. that the new state is a progressive extension of the previous one. \square

⁷ For example, if the user states that “event e initiates fluent f at time T if fluent f_2 holds at time $T + 2$ ”, then it could be the case that at time $T + 2$ f_2 indeed holds, causing f to be declipped in the past and violating condition 2 of Definition 7

6 Conclusions

The \mathcal{EC} is a powerful framework for reasoning about time and events. We identified the problem applying the \mathcal{EC} to runtime monitoring and tracking systems. We observed that there is no satisfactory solution to it in the state of the art. Specifically, related approaches mainly boil down to ad-hoc, tailored procedures that are not easily modifiable and whose formal properties are not easy to determine. Moreover, they do not guarantee properties that we deem fundamental in this domain, namely irrevocability. We therefore provided the first formal and operational approach to the problem, a \mathcal{REC} implementation in SCIFF.

We proved that \mathcal{REC} enjoys soundness, completeness and irrevocability: in other words, it is a reactive version of the \mathcal{EC} which matches the domain requirements. Irrevocability holds for a class of well-formed \mathcal{REC} theories. Soundness and completeness instead hold for the broader class of theories defined by well-formed SCIFF theories [1]. Thus \mathcal{REC} can also reason on non well-formed \mathcal{REC} theories, but in that case the value of fluents in the past may change as events in the future are processed, which is not at all surprising.

In a companion paper [9] we show how to realize a commitment tracking framework for multi-agent systems using \mathcal{REC} . Future work will focus on performance evaluation and on the integration of \mathcal{REC} with the other forms of reasoning enabled by SCIFF, mainly with abduction.

References

1. M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic*, 9(4):1–43, 2008.
2. F. Chesani. *Specification, execution and verification of interaction protocols: an approach based on computational logic*. PhD thesis, University of Bologna, 2007. Available at <http://amsdottorato.cib.unibo.it/392/>.
3. L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12:359–382, 1996.
4. L. Chittaro and A. Montanari. Temporal representation and reasoning in artificial intelligence: Issues and approaches. *AMAI*, 28(1-4):47–106, 2000.
5. K. Eshghi. Abductive planning with event calculus. In *Proc. 5th ICSLP*:562–579. MIT Press, 1988.
6. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, Nov. 1997.
7. R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. *Logic in Databases*, LNCS 1154:137–149. Springer, 1996.
8. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
9. M. Montali, F. Chesani, P. Mello, and P. Torroni. Commitment tracking via the reactive event calculus, 2009. Accepted at IJCAI 2009. Available upon request.
10. M. Shanahan. The event calculus explained. In *Artificial Intelligence Today*, LNAI 1600:409–430, Springer, 1999.
11. M. Shanahan. An abductive event calculus planner. *J. Log. Program.*, 44(1-3):207–240, 2000.

A Non-monotonic Description Logic of Typicality

Laura Giordano¹, Valentina Gliozzi², Nicola Olivetti³, Gian Luca Pozzato⁴

1. Dipartimento di Informatica - Università del Piemonte Orientale “A. Avogadro”
viale Teresa Michel, 11 - 15121 - Alessandria - Italy - E-mail: laura@mf.n.unipmn.it

2. Dipartimento di Informatica - Università degli Studi di Torino
C.So Svizzera, 185 - 10149 Torino (Italy) - E-mail: gliozzi@di.unito.it

3. LSIS - UMR CNRS 6168 Université “P. Cézanne” (Aix-Marseille 3), Campus de
S. Jérôme - Avenue Escadrille Normandie-Niemen 13397 Marseille Cedex 20 (France)
E-mail : nicola.olivetti@lsis.org, nicola.olivetti@univ-cezanne.fr

4. Dipartimento di Informatica - Università degli Studi di Torino
C.So Svizzera, 185 - 10149 Torino (Italy) - E-mail: pozzato@di.unito.it

Abstract. We present a nonmonotonic extension of the Description Logic \mathcal{ALC} for reasoning about prototypical properties and inheritance with exception. The logic \mathcal{ALC} is extended by introducing a typicality operator \mathbf{T} which is intended to select the “most normal” instances of a concept. A minimal model semantics is defined for $\mathcal{ALC} + \mathbf{T}$, in order to perform nonmonotonic inferences. The resulting logic $\mathcal{ALC} + \mathbf{T}_{min}$ is able to infer defeasible properties of explicit and implicit individuals. The paper presents a tableau calculus for deciding $\mathcal{ALC} + \mathbf{T}_{min}$ entailment. The work was partially supported by Regione Piemonte, Project ICT4Law. Main contribution of this paper have been also presented at the 11th European Conference on Logics in Artificial Intelligence “JELIA 2008” [10].

1 Introduction

In Description Logics (DLs) the need of representing prototypical properties and of reasoning about defeasible inheritance of such properties naturally 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 [2–4, 7, 8, 15]. However, finding a suitable nonmonotonic extension for inheritance with exceptions is far from being obvious.

In this work, we propose a new nonmonotonic logic $\mathcal{ALC} + \mathbf{T}_{min}$ for defeasible reasoning in description logic. Our starting point is the monotonic logic $\mathcal{ALC} + \mathbf{T}$ introduced in [9], obtained by adding a typicality operator \mathbf{T} to \mathcal{ALC} . The intended meaning of the operator \mathbf{T} , for any concept C , is that $\mathbf{T}(C)$ 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” [4] are represented by $\mathbf{T}(Student) \sqsubseteq \neg TaxPayer$ and $\mathbf{T}(User) \sqsubseteq \neg \exists HasAccess.ConfidentialFile$. As shown in [9], the operator \mathbf{T} is characterised by a set of postulates that are essentially a reformulation of KLM [14] axioms of preferential logic \mathbf{P} , namely the assertion $\mathbf{T}(C) \sqsubseteq P$ is equivalent to the conditional assertion $C \vdash P$ of \mathbf{P} . It turns out that the semantics of the typicality operator can be equivalently specified by a suitable modal logic.

The idea underlying the modal interpretation is that there is a global preference relation (a strict partial order) $<$ on individuals, so that typical instances of a concept C can be defined as the instances of C that are minimal with respect to $<$. In this modal logic, $<$ works as an accessibility relation R with $R(x, y) \equiv y < x$, so that we can define $\mathbf{T}(C)$ as $C \sqcap \Box \neg C$. The preference relation $<$ does not have infinite descending chains as we adopt the so-called Smoothness condition or Limit Assumption of conditional logics. As a consequence, the corresponding modal operator \Box has the same properties as in Gödel-Löb modal logic \mathbf{G} of arithmetic provability.

In our setting, we assume that a knowledge base (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} . For instance, let the KB $(*)$ contain:

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

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. Suppose that the ABox contains, alternatively, the following facts about *john*:

1. $Student(john)$
2. $Student(john), Worker(john)$
3. $Student(john), Worker(john), \exists HasChild.\top(john)$

We would like to infer the expected (defeasible) conclusion about *john* in each case:

1. $\neg TaxPayer(john)$
2. $TaxPayer(john)$
3. $\neg TaxPayer(john)$

Moreover, we would like to infer (defeasible) properties also of individuals implicitly introduced by existential restrictions, for instance, if the ABox further contains

$$\exists HasChild.(Student \sqcap Worker)(jack)$$

it should derive (defeasibly) the “right” conclusion

$$\exists HasChild.TaxPayer(jack)$$

in the latter. Finally, adding irrelevant information should not affect the conclusions. Given the KB as above, one should be able to infer as well

1. $\mathbf{T}(Student \sqcap SportLover) \sqsubseteq \neg TaxPayer$
2. $\mathbf{T}(Student \sqcap Worker \sqcap SportLover) \sqsubseteq TaxPayer$

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 adding *SportLover(john)* to the ABox.

The monotonic logic $\mathcal{ALC} + \mathbf{T}$ is not sufficient to perform the kind of defeasible reasoning illustrated above. Concerning the example, we get for instance that: $KB \cup \{Student(john), Worker(john)\} \not\models TaxPayer(john)$; $KB \not\models \mathbf{T}(Student \sqcap SportLover) \sqsubseteq \neg TaxPayer$. In order to derive the conclusion about *john* we should know (or assume) that *john* is a typical working student, but we do not dispose of this information. Similarly, in order to derive that also a typical student who loves sport must not pay taxes, we must be able to infer or assume that a typical “student loving sport” is also a “typical student”, since there is no reason why it should not be the case; this cannot be derived by the logic itself given the nonmonotonic nature of \mathbf{T} . The basic monotonic logic $\mathcal{ALC} + \mathbf{T}$ is then too weak to enforce these extra assumptions, so that we need an additional mechanism to perform defeasible inferences. In [9], we proposed a completion of the KB that adds, for each individual, the assumption that the individual is a typical member of the *most specific concept* to which it belongs. However, this solution presents some difficulties: (i) it is not clear how to take into account implicit individuals, (ii) the completion might be inconsistent, so that we must consider alternative maximal completions, (iii) it is not clear whether and how the completion has to take into account concept instances that are inferred from previous typicality assumptions introduced by the completion itself (this would require a kind of fixpoint definition).

In this work we follow another approach, rather than defining an ad-hoc mechanism to perform defeasible inferences or making nonmonotonic assumptions, we strengthen the semantics of the logic by proposing a minimal model semantics. Intuitively, the idea is to restrict our consideration to models that maximise typical instances of a concept. In order to define the preference relation on models we take advantage of the modal semantics of $\mathcal{ALC} + \mathbf{T}$: the preference relation on models (with the same domain) is defined by comparing, for each individual, the set of modal (or more precisely \Box -ed) concepts containing the individual in the two models. Similarly to circumscription, where we must specify a set of minimised predicates, here we must specify a set of concepts \mathcal{L}_T of which we want to maximise the set of typical instances (it may just be the set of all concepts occurring in the knowledge base). We call the new logic $\mathcal{ALC} + \mathbf{T}_{min}$ and we denote by $\models_{min}^{\mathcal{L}_T}$ semantic entailment determined by minimal models. Taking the KB of the examples above we obtain, for instance, $KB \cup \{Student(john), Worker(john)\} \models_{min}^{\mathcal{L}_T} TaxPayer(john)$; $KB \cup \{\exists HasChild.(Student \sqcap Worker)(jack)\} \models_{min}^{\mathcal{L}_T} \exists HasChild.TaxPayer(jack)$ and $KB \models_{min}^{\mathcal{L}_T} \mathbf{T}(Student \sqcap SportLover) \sqsubseteq \neg TaxPayer$. As the second example shows, we are able to infer the intended conclusion also for the implicit individuals.

We provide a decision procedure for checking satisfiability and validity in $\mathcal{ALC} + \mathbf{T}_{min}$. Our decision procedure has the form of tableaux calculus, with a two-step tableau construction. The idea is that the top level construction generates open branches that are candidates to represent minimal models, whereas the auxiliary construction checks whether a candidate branch represents indeed a minimal model. Termination is ensured by means of a standard blocking mechanism. Our procedure can be used to determine constructively an upper bound of the complexity of $\mathcal{ALC} + \mathbf{T}_{min}$. Namely we obtain that checking query entailment for $\mathcal{ALC} + \mathbf{T}_{min}$ is in $\text{co-NEXP}^{\text{NP}}$.

2 Reasoning About Typicality in \mathcal{ALC}

In this section, we summarize the features of the original $\mathcal{ALC} + \mathbf{T}$, which is an extension of \mathcal{ALC} by a typicality operator \mathbf{T} . We then illustrate the nonmonotonic extension $\mathcal{ALC} + \mathbf{T}_{min}$.

Given 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 of \mathcal{L} , then C and $\mathbf{T}(C)$ are extended concepts of \mathcal{L} , and all the boolean combinations of extended concepts are extended concepts of \mathcal{L} . A knowledge base is a pair $(\text{TBox}, \text{ABox})$. TBox contains subsumptions $C \sqsubseteq D$, where $C \in \mathcal{L}$ is an extended concept of the form C' or $\mathbf{T}(C')$, 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 $a, b \in \mathcal{O}$.

Definition 1 (Semantics of $\mathcal{ALC} + \mathbf{T}$). *A model \mathcal{M} is any structure $\langle \Delta, <, I \rangle$, where Δ is the domain; $<$ is a strict partial order over Δ . For all $S \subseteq \Delta$, we define $\text{Min}_{<}(S) = \{a : a \in S \text{ and } \nexists b \in S \text{ s.t. } b < a\}$. We say that $<$ satisfies the Smoothness Condition i.e., for all $S \subseteq \Delta$, for all $a \in S$, either $a \in \text{Min}_{<}(S)$ or $\exists b \in \text{Min}_{<}(S)$ such that $b < a$. I is the extension function that maps each extended concept C to $C^I \subseteq \Delta$, and each role R to a $R^I \subseteq \Delta^I \times \Delta^I$. For concepts (built from operators of \mathcal{ALC}), C^I is defined in the usual way. For the \mathbf{T} operator: $(\mathbf{T}(C))^I = \text{Min}_{<}(C^I)$. A model satisfying a Knowledge Base $(\text{TBox}, \text{ABox})$ is defined as usual. We assume the unique name assumption.*

Notice that the meaning of \mathbf{T} can be split into two parts: for any a of the domain Δ , $a \in (\mathbf{T}(C))^I$ just in case (i) $a \in C^I$, and (ii) there is no $b \in C^I$ such that $b < a$. In order to isolate the second part of the meaning of \mathbf{T} (for the purpose of the calculus that we will present later), we introduce a new modality \square . The basic idea is simply to interpret the preference relation $<$ as an accessibility relation. By the Smoothness Condition, it turns out that \square has the properties as in 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}). The interpretation of \square in \mathcal{M} is as follows:

Definition 2. $(\Box C)^I = \{a \in \Delta \mid \text{for every } b \in \Delta, \text{ if } b < a \text{ then } b \in C^I\}$

We have that a is a typical instance of C ($a \in (\mathbf{T}(C))^I$) iff $a \in (C \sqcap \Box \neg C)^I$. Since we only use \Box to capture the meaning of \mathbf{T} , in the following we will always use the modality \Box followed by a negated concept, as in $\Box \neg C$.

The logic $\mathcal{ALC} + \mathbf{T}$ allows one to reason about typicality. As a difference with respect to standard \mathcal{ALC} , 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. This can be consistently expressed by the KB $(*)$ of the Introduction. Assume that *john* is an instance of the concept $Student \sqcap Worker \sqcap \exists HasChild.\top$. What can we conclude about *john*? If the ABox contains the assertion $(*) \mathbf{T}(Student \sqcap Worker \sqcap \exists HasChild.\top)(john)$, then, in $\mathcal{ALC} + \mathbf{T}$, we can conclude that $\neg TaxPayer(john)$. However, in the absence of $(*)$, we cannot derive $\neg TaxPayer(john)$.

We would like to infer that individuals are typical instances of the concepts they belong to, if consistent with the KB. In order to maximize the typicality of instances, we define a preference relation on models, and we introduce a semantic entailment determined by minimal models. Informally, we prefer a model \mathcal{M} to a model \mathcal{N} if \mathcal{M} contains more typical instances of concepts than \mathcal{N} .

Given a KB, we consider a finite set \mathcal{L}_T of concepts occurring in the KB, the typicality of whose instances we want to maximize. The maximization of the set of typical instances will apply to individuals explicitly occurring in the ABox as well as to implicit individuals. We assume that the set \mathcal{L}_T contains at least all concepts C such that $\mathbf{T}(C)$ occurs in the KB.

We have seen that a is a typical instance of a concept C ($a \in (\mathbf{T}(C))^I$) when it is an instance of C and there is not another instance of C preferred to a , i.e. $a \in (C \sqcap \Box \neg C)^I$. In the following, in order to maximize the typicality of the instances of C , we minimize the instances of $\Box \neg C$. Notice that this is different from maximising the instances of $\mathbf{T}(C)$. We have adopted this solution since it allows to maximise the set of typical instances of C without affecting the extension of C (whereas maximising the extension of $\mathbf{T}(C)$ would imply maximising also the extension of C).

We define the set $\mathcal{M}_{\mathcal{L}_T}^{\Box \neg}$ of negated boxed formulas holding in a model, relative to the concepts in \mathcal{L}_T . Given a model $\mathcal{M} = \langle \Delta, <, I \rangle$, let $\mathcal{M}_{\mathcal{L}_T}^{\Box \neg} = \{(a, \Box \neg C) \mid a \in (\Box \neg C)^I, \text{ with } a \in \Delta, C \in \mathcal{L}_T\}$.

Let KB be a knowledge base and let \mathcal{L}_T be a set of concepts occurring in KB.

Definition 3 (Preferred and minimal models). *Given a model $\mathcal{M} = \langle \Delta_{\mathcal{M}}, <_{\mathcal{M}}, I_{\mathcal{M}} \rangle$ of KB and a model $\mathcal{N} = \langle \Delta_{\mathcal{N}}, <_{\mathcal{N}}, I_{\mathcal{N}} \rangle$ of KB, we say that \mathcal{M} is preferred to \mathcal{N} with respect to \mathcal{L}_T , and we write $\mathcal{M} <_{\mathcal{L}_T} \mathcal{N}$, if the following conditions hold: $\Delta_{\mathcal{M}} = \Delta_{\mathcal{N}}$ and $\mathcal{M}_{\mathcal{L}_T}^{\Box \neg} \subset \mathcal{N}_{\mathcal{L}_T}^{\Box \neg}$. A model \mathcal{M} is a minimal model for KB (with respect to \mathcal{L}_T) if it is a model of KB and there is no a model \mathcal{M}' of KB such that $\mathcal{M}' <_{\mathcal{L}_T} \mathcal{M}$.*

Definition 4 (Minimal Entailment in $\mathcal{ALC} + \mathbf{T}_{min}$). *A query F (see section below) is minimally entailed from a knowledge base KB with respect to \mathcal{L}_T if it holds in all models of KB minimal with respect to \mathcal{L}_T . We write $\text{KB} \models_{min}^{\mathcal{L}_T} F$.*

While the original $\mathcal{ALC} + \mathbf{T}$ is *monotonic* (see [9]), $\mathcal{ALC} + \mathbf{T}_{min}$ is *nonmonotonic*.

It is worth noticing that, in general, a knowledge base KB may have no minimal model or more than one minimal model, with respect to a given \mathcal{L}_T .

3 A Tableau Calculus for Minimal Entailment in $\mathcal{ALC} + \mathbf{T}_{min}$

We present a tableau calculus for deciding whether a formula (*query*) F is minimally entailed from a knowledge base (TBox, ABox). We introduce a labelled tableau calculus called $\mathcal{TAB}_{min}^{\mathcal{ALC} + \mathbf{T}}$, which extends the calculus $T^{\mathcal{ALC} + \mathbf{T}}$ presented in [9], and allows to reason about minimal models.

$\mathcal{TAB}_{min}^{\mathcal{ALC} + \mathbf{T}}$ performs a two-phase computation in order to check whether a query F is minimally entailed from the initial KB. In particular, the procedure tries to build an open branch representing a minimal model satisfying $\text{KB} \cup \{\neg F\}$. A query F is either a formula of the form $C(a)$ or a subsumption relation $C \sqsubseteq D$ such that, for all $\mathbf{T}(C')$ occurring in F , $C' \in \mathcal{L}_T$. In the first phase, a tableau calculus, called $\mathcal{TAB}_{PH1}^{\mathcal{ALC} + \mathbf{T}}$, simply verifies whether $\text{KB} \cup \{\neg F\}$ is satisfiable in an $\mathcal{ALC} + \mathbf{T}$ model, building candidate models. In case F has the form $C \sqsubseteq D$, then $\neg F$ corresponds to $(C \sqcap \neg D)(x)$, where x does not occur in KB. In the second phase another tableau calculus, called $\mathcal{TAB}_{PH2}^{\mathcal{ALC} + \mathbf{T}}$, checks whether the candidate models found in the first phase are *minimal* models of KB, i.e. for each open branch of the first phase, $\mathcal{TAB}_{PH2}^{\mathcal{ALC} + \mathbf{T}}$ tries to build a “smaller” model of KB, i.e. a model whose individuals satisfy less formulas $\neg \Box \neg C$ than the corresponding candidate model. The whole procedure $\mathcal{TAB}_{min}^{\mathcal{ALC} + \mathbf{T}}$ is described at the end of this section (Definition 8).

$\mathcal{TAB}_{min}^{\mathcal{ALC} + \mathbf{T}}$ is based on the notion of a *constraint system*. We consider a set of *variables* drawn from a denumerable set \mathcal{V} . $\mathcal{TAB}_{min}^{\mathcal{ALC} + \mathbf{T}}$ 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. A constraint of the form $x \xrightarrow{R} y$ says that the object denoted by label x is related to the object denoted by y by means of role R ; a constraint $x : C$ says that the object denoted by x is an instance of the concept C .

Let us now separately analyze the two components of the calculus $\mathcal{TAB}_{min}^{\mathcal{ALC} + \mathbf{T}}$, starting with $\mathcal{TAB}_{PH1}^{\mathcal{ALC} + \mathbf{T}}$.

A tableau of $\mathcal{TAB}_{PH1}^{\mathcal{ALC} + \mathbf{T}}$ 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 $x \xrightarrow{R} y$, whereas 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 from its immediate predecessor $\langle S_{i-1} \mid U_{i-1} \rangle$ by applying a rule of $\mathcal{TAB}_{PH1}^{\mathcal{ALC} + \mathbf{T}}$, 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. A tableau is closed if all its branches are closed.

In order to check the satisfiability of a KB, we build the corresponding constraint system $\langle S \mid U \rangle$, and we check its satisfiability.

Definition 5 (Corresponding constraint system). *Given a knowledge base $\text{KB}=(\text{TBox},\text{ABox})$, 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 $U = \{C \sqsubseteq D^0 \mid C \sqsubseteq D \in \text{TBox}\}$.*

Definition 6 (Model satisfying a constraint system). *Let \mathcal{M} be a model as defined in Definition 1. 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 every constraint in S under α and that, for all $C \sqsubseteq D^L \in U$ and for all x occurring in S , we have that if $\alpha(x) \in C^I$ then $\alpha(x) \in D^I$.*

It can be easily shown that, given a knowledge base, it is satisfiable if and only if its corresponding constraint system is satisfiable.

To verify the satisfiability of $\text{KB} \cup \{\neg F\}$, we use $\mathcal{TAB}_{PH1}^{ALC+T}$ to check the satisfiability of the constraint system $\langle S \mid U \rangle$ obtained by adding the constraint corresponding to $\neg F$ to S' , where $\langle S' \mid U \rangle$ is the corresponding constraint system of KB. To this purpose, the rules of the calculus $\mathcal{TAB}_{PH1}^{ALC+T}$ are applied until either a contradiction is generated (Clash) or a model satisfying $\langle S \mid U \rangle$ can be obtained from the resulting constraint system.

As in the calculus proposed in [9], 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 $x : \neg C \sqcup D$ (*unfolding*). As mentioned above, each formula $C \sqsubseteq D$ is equipped with a list L of labels in which it has been unfolded in the current branch. This is needed to avoid multiple unfolding of the same subsumption by using the same label, generating infinite branches.

Before introducing the rules of $\mathcal{TAB}_{PH1}^{ALC+T}$ we need some more definitions. First, as in [5], we define an ordering relation \prec to keep track of the temporal ordering of insertion of labels in the tableau, 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 represent the same element in the model built by $\mathcal{TAB}_{PH1}^{ALC+T}$. Last, we define $S_{x \rightarrow y}^M = \{y : \neg C, y : \Box \neg C \mid x : \Box \neg C \in S\}$.

The rules of $\mathcal{TAB}_{PH1}^{ALC+T}$ are presented in Figure 1. Rules (\exists^+) and (\Box^-) are called *dynamic* since they introduce a new variable in their conclusions. The other rules are called *static*. The side condition on (\exists^+) is introduced in order to ensure a terminating proof search, by implementing the standard *blocking*

$\langle S, x : \neg C, x : C \mid U \rangle$ (Clash)	$\frac{\langle S \mid U, C \sqsubseteq D^L \rangle}{\langle S, x : \neg C \sqcup D \mid U, C \sqsubseteq D^{L,x} \rangle}$ (Unfold) <small>if x occurs in S and $x \notin L$</small>	$\frac{\langle S, x : \neg C \mid U \rangle}{\langle S, x : C \mid U \rangle}$ $(-)$	$\frac{\langle S, x : \mathbf{T}(C) \mid U \rangle}{\langle S, x : C, x : \Box \neg C \mid U \rangle}$ (\mathbf{T}^+)
$\frac{\langle S, x : \neg \mathbf{T}(C) \mid U \rangle}{\langle S, x : \neg C \mid U \rangle}$ (\mathbf{T}^-)	$\frac{\langle S \mid U \rangle}{\langle S, x : \Box \neg C \mid U \rangle}$ (cut) <small>If $x : \Box \neg C \notin S$ and $x : \neg \Box \neg C \notin S, C \in \mathcal{L}_T$ x occurs in S</small>	$\frac{\langle S, x : \forall R.C, x \xrightarrow{R} y \mid U \rangle}{\langle S, x : \forall R.C, x \xrightarrow{R} y, y : C \mid U \rangle}$ (\forall^+) <small>if $y : C \notin S$</small>	
$\langle S, x : \Box \neg C \mid U \rangle$			
$\frac{\langle S, x : \Box \neg C, y : C, y : \Box \neg C, S_{x \rightarrow y}^M \mid U \rangle \quad \langle S, x : \Box \neg C, v_1 : C, v_1 : \Box \neg C, S_{x \rightarrow v_1}^M \mid U \rangle \quad \cdots \quad \langle S, x : \Box \neg C, v_n : C, v_n : \Box \neg C, S_{x \rightarrow v_n}^M \mid U \rangle}{\langle S, x : \Box \neg C \mid U \rangle}$ (\Box^-) <small>If $\exists u$ s.t. $u : C \in S, u : \Box \neg C \in S$, and $S_{x \rightarrow u}^M \subseteq S$. $\forall v_i$ occurring in $S, x \neq v_i, y$ new</small>			
$\langle S, x : \exists R.C \mid U \rangle$			
$\langle S, x : \exists R.C, x \xrightarrow{R} y, y : C \mid U \rangle$	$\langle S, x : \exists R.C, x \xrightarrow{R} v_1, v_1 : C \mid U \rangle$	\cdots	$\langle S, x : \exists R.C, x \xrightarrow{R} v_n, v_n : C \mid U \rangle$ (\exists^+) <small>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 $u : C \in S$ $\forall v_i$ occurring in $S, x \neq v_i, y$ new</small>

Fig. 1. The calculus $\mathcal{TAB}_{PH1}^{ALC+T}$. To save space, we omit the standard rules for \sqcup and \Box , as well as the rules (\forall^-) and (\exists^-) , dual to (\exists^+) and (\forall^+) , respectively.

technique described below. The (cut) rule ensures that, given any concept $C \in \mathcal{L}_T$, an open branch built by $\mathcal{TAB}_{PH1}^{ALC+T}$ contains either $x : \Box \neg C$ or $x : \neg \Box \neg C$ for each label x : this is needed in order to allow $\mathcal{TAB}_{PH2}^{ALC+T}$ to check the minimality of the model corresponding to the open branch, as we will discuss later.

The rules of $\mathcal{TAB}_{PH1}^{ALC+T}$ are applied with the following *standard strategy*: 1. apply a rule to a label x only if no rule is applicable to a label y such that $y \prec x$; 2. apply dynamic rules ((\Box^-) first) only if no static rule is applicable. This strategy ensures that the labels are considered one at a time according to the ordering \prec . The calculus so obtained is sound and complete with respect to the semantics in Definition 6.

Theorem 1 (Soundness and Completeness of $\mathcal{TAB}_{PH1}^{ALC+T}$ [10]). *Given a constraint system $\langle S \mid U \rangle$, it is unsatisfiable iff it has a closed tableau in $\mathcal{TAB}_{PH1}^{ALC+T}$.*

To ensure termination, we adopt the standard loop-checking machinery known as *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 . By virtue of 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. 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 \prec are allowed). The (cut) rule does not affect termination, since it is applied only to the finitely many formulas belonging to \mathcal{L}_T .

$\langle S, x : C, x : \neg C \mid U \mid K \rangle$ (Clash)	$\langle S \mid U \mid \emptyset \rangle$ (Clash) ₀	$\langle S, x : \neg \Box \neg C \mid U \mid K \rangle$ (Clash) _{\Box^-} if $x : \neg \Box \neg C \notin K$
$\langle S \mid U, C \sqsubseteq D^L \mid K \rangle$ (Unfold)	$\langle S, x : \exists R.C \mid U \mid K \rangle$	
$\langle S, x : \neg C \sqcup D \mid U, C \sqsubseteq D^L, x \mid K \rangle$ $x \in \mathcal{D}(\mathbf{B})$ and $x \notin L$	$\langle S, x \xrightarrow{R} v_1, v_1 : C \mid U \mid K \rangle$	$\langle S, x \xrightarrow{R} v_2, v_2 : C \mid U \mid K \rangle \cdots \langle S, x \xrightarrow{R} v_n, v_n : C \mid U \mid K \rangle$ (\exists^+) If $\exists u \in \mathcal{D}(\mathbf{B})$ s.t. $x \xrightarrow{R} u \in S$ and $u : C \in S, \forall v_i \in \mathcal{D}(\mathbf{B})$
$\langle S, x : \neg \Box \neg C \mid U \mid K, x : \neg \Box \neg C \rangle$		
$\langle S, v_1 : C, v_1 : \Box \neg C, S_{x \rightarrow v_1}^M \mid U \mid K \rangle$	$\langle S, v_2 : C, v_2 : \Box \neg C, S_{x \rightarrow v_2}^M \mid U \mid K \rangle$	$\cdots \langle S, v_n : C, v_n : \Box \neg C, S_{x \rightarrow v_n}^M \mid U \mid K \rangle$ (\Box^-) $\forall v_i \in \mathcal{D}(\mathbf{B}), x \neq v_i$

Fig. 2. The calculus $\mathcal{TAB}_{PH2}^{ALCC+T}$.

Theorem 2 (Termination of $\mathcal{TAB}_{PH1}^{ALCC+T}$ [10]). *Let $\langle S \mid U \rangle$ be a constraint system, then any tableau generated by $\mathcal{TAB}_{PH1}^{ALCC+T}$ is finite.*

Since $\mathcal{TAB}_{PH1}^{ALCC+T}$ is sound and complete (Theorem 1), and since a KB is satisfiable iff its corresponding constraint system is satisfiable, from Theorem 2 above it follows that checking whether a given KB (TBox, ABox) is satisfiable is a decidable problem. It is possible to prove that, with the calculus above, the satisfiability of a KB can be decided in nondeterministic exponential time in the size of the KB.

Let us now introduce the calculus $\mathcal{TAB}_{PH2}^{ALCC+T}$ which, for each open branch \mathbf{B} built by $\mathcal{TAB}_{PH1}^{ALCC+T}$, verifies if it is a minimal model of the KB.

Definition 7. *Given an open branch \mathbf{B} of a tableau built from $\mathcal{TAB}_{PH1}^{ALCC+T}$, we define: (i) $\mathcal{D}(\mathbf{B})$ as the set of objects occurring on \mathbf{B} ; (ii) $\mathbf{B}^{\Box^-} = \{x : \neg \Box \neg C \mid x : \neg \Box \neg C \text{ occurs in } \mathbf{B}\}$.*

A tableau of $\mathcal{TAB}_{PH2}^{ALCC+T}$ is a tree whose nodes are triples of the form $\langle S \mid U \mid K \rangle$, where $\langle S \mid U \rangle$ is a constraint system, whereas K contains formulas of the form $x : \neg \Box \neg C$, with $C \in \mathcal{L}_T$.

The basic idea of $\mathcal{TAB}_{PH2}^{ALCC+T}$ is as follows. Given an open branch \mathbf{B} built by $\mathcal{TAB}_{PH1}^{ALCC+T}$ and corresponding to a model $\mathcal{M}^{\mathbf{B}}$ of $\text{KB} \cup \{-F\}$, $\mathcal{TAB}_{PH2}^{ALCC+T}$ checks whether $\mathcal{M}^{\mathbf{B}}$ is a minimal model of KB by trying to build a model of KB which is preferred to $\mathcal{M}^{\mathbf{B}}$. Checking (un)satisfiability of $\langle S \mid U \mid \mathbf{B}^{\Box^-} \rangle$, where $\langle S \mid U \rangle$ is the corresponding constraint system of the initial KB, allows to verify whether the candidate model $\mathcal{M}^{\mathbf{B}}$ is minimal. More in detail, $\mathcal{TAB}_{PH2}^{ALCC+T}$ tries to build an open branch containing all the objects appearing on \mathbf{B} , i.e. those in $\mathcal{D}(\mathbf{B})$. To this aim, the dynamic rules use labels in $\mathcal{D}(\mathbf{B})$ instead of introducing new ones in their conclusions. The additional set K of a tableau node, initialized with \mathbf{B}^{\Box^-} , is used in order to ensure that any branch \mathbf{B}' built by $\mathcal{TAB}_{PH2}^{ALCC+T}$ is preferred to \mathbf{B} , that is \mathbf{B}' only contains negated boxed formulas occurring in \mathbf{B} and there exists at least a $x : \neg \Box \neg C$ that occurs in \mathbf{B} and *does not occur* in \mathbf{B}' . The rules of $\mathcal{TAB}_{PH2}^{ALCC+T}$ are shown in Figure 2. $\mathcal{TAB}_{PH2}^{ALCC+T}$ also contains analogues of the following rules from $\mathcal{TAB}_{PH1}^{ALCC+T}$ in Figure 1: (\neg), (\mathbf{T}^+), (\mathbf{T}^-), (*cut*), (\forall^+), where the rules in $\mathcal{TAB}_{PH2}^{ALCC+T}$ include the additional component K .

More in detail, the rule (\exists^+) is applied to a constraint system containing a formula $x : \exists R.C$; it introduces $x \xrightarrow{R} y$ and $y : C$ where $y \in \mathcal{D}(\mathbf{B})$, instead of y being a new label. The choice of the label y introduces a branching in the tableau construction. The rule (Unfold) is applied in the same way as in $\mathcal{TAB}_{PH1}^{ACC+T}$ to all the labels of $\mathcal{D}(\mathbf{B})$ (and not only to those appearing in the branch). The rule (\Box^-) is applied to a node $\langle S, x : \neg\Box\neg C \mid U \mid K \rangle$, when $x : \neg\Box\neg C \in K$, i.e. when the formula $x : \neg\Box\neg C$ also belongs to the open branch \mathbf{B} . In this case, the rule introduces a branch on the choice of the individual $v_i \in \mathcal{D}(\mathbf{B})$ which is preferred to x and is such that C and $\Box\neg C$ hold in v_i . In case a tableau node has the form $\langle S, x : \neg\Box\neg C \mid U \mid K \rangle$, and $x : \neg\Box\neg C \notin K$, then $\mathcal{TAB}_{PH2}^{ACC+T}$ detects a clash, called $(\text{Clash})_{\Box^-}$: this corresponds to the situation in which $x : \neg\Box\neg C$ does not belong to \mathbf{B} , while $S, x : \neg\Box\neg C$ is satisfiable in a model \mathcal{M} only if \mathcal{M} contains $x : \neg\Box\neg C$, and hence only if \mathcal{M} is *not* preferred to the model represented by \mathbf{B} .

The calculus $\mathcal{TAB}_{PH2}^{ACC+T}$ also contains the clash condition $(\text{Clash})_{\emptyset}$. Since each application of (\Box^-) removes the principal formula $x : \neg\Box\neg C$ from the set K , when K is empty all the negated boxed formulas occurring in \mathbf{B} also belong to the current branch. In this case, the model built by $\mathcal{TAB}_{PH2}^{ACC+T}$ satisfies the same set of negated boxed formulas (for all individuals) as \mathbf{B} and, thus, it is not preferred to the one represented by \mathbf{B} .

Theorem 3 (Soundness and completeness of $\mathcal{TAB}_{PH2}^{ACC+T}$ [10]). *Given a KB and a query F , let $\langle S' \mid U \rangle$ be the corresponding constraint system of KB, and $\langle S \mid U \rangle$ the corresponding constraint system of $\text{KB} \cup \{\neg F\}$. An open branch \mathbf{B} built by $\mathcal{TAB}_{PH1}^{ACC+T}$ for $\langle S \mid U \rangle$ is satisfiable by an injective mapping in a minimal model of KB iff the tableau in $\mathcal{TAB}_{PH2}^{ACC+T}$ for $\langle S' \mid U \mid \mathbf{B}^{\Box^-} \rangle$ is closed.*

$\mathcal{TAB}_{PH2}^{ACC+T}$ always terminates. Indeed, only a finite number of labels can occur on the branch (only those in $\mathcal{D}(\mathbf{B})$ which is finite). Moreover, the side conditions on the application of the rules copying their principal formulas in their conclusion(s) prevent the uncontrolled application of the same rules.

It is possible to show that the problem of verifying that a branch \mathbf{B} represents a minimal model for KB in $\mathcal{TAB}_{PH2}^{ACC+T}$ is in NP in the size of \mathbf{B} .

The overall procedure $\mathcal{TAB}_{min}^{ACC+T}$ is defined as follows:

Definition 8. *Let KB be a knowledge base whose corresponding constraint system is $\langle S \mid U \rangle$. Let F be a query and let S' be the set of constraints obtained by adding to S the constraint corresponding to $\neg F$. The calculus $\mathcal{TAB}_{min}^{ACC+T}$ checks whether a query F can be minimally entailed from a KB by means of the following procedure: (phase 1) the calculus $\mathcal{TAB}_{PH1}^{ACC+T}$ is applied to $\langle S' \mid U \rangle$; if, for each branch \mathbf{B} built by $\mathcal{TAB}_{PH1}^{ACC+T}$, either (i) \mathbf{B} is closed or (ii) (phase 2) the tableau built by the calculus $\mathcal{TAB}_{PH2}^{ACC+T}$ for $\langle S \mid U \mid \mathbf{B}^{\Box^-} \rangle$ is open, then $\text{KB} \models_{min}^{\mathcal{L}_T} F$, otherwise $\text{KB} \not\models_{min}^{\mathcal{L}_T} F$.*

$\mathcal{TAB}_{min}^{ACC+T}$ is therefore a sound and complete decision procedure for verifying if a formula F can be minimally entailed from a KB. We can also prove that:

Theorem 4 (Complexity of $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ [10]). *The problem of deciding whether $\text{KB} \models_{min}^{\mathcal{L}_T} F$ is in $\text{co-NEXP}^{\text{NP}}$.*

4 Conclusions, related works, future issues

We have presented a nonmonotonic extension called $\mathcal{ALC} + \mathbf{T}_{min}$ of \mathcal{ALC} for reasoning about prototypical properties in DLs. The extension is obtained by adding to \mathcal{ALC} a typicality operator, originally defined in [9]. In the logic $\mathcal{ALC} + \mathbf{T}$, prototypical properties can be directly express by inclusions like $\mathbf{T}(C) \sqsubseteq P$, whose meaning is that “the typical instances of concept C have the property P ”. However, $\mathcal{ALC} + \mathbf{T}$ is not sufficient to perform defeasible reasoning. For this reason, we introduce a preferential semantics, called $\mathcal{ALC} + \mathbf{T}_{min}$, where preferred or minimal models maximise typical instances of concepts. This nonmonotonic extension allows one to perform defeasible reasoning in particular in the context of inheritance with exceptions to some extent. We have then developed a procedure for deciding query-entailment in $\mathcal{ALC} + \mathbf{T}_{min}$. The procedure has the form of a two-phase tableau calculus for generating $\mathcal{ALC} + \mathbf{T}_{min}$ minimal models. The procedure is sound, complete, and terminating, whereby giving a decision procedure for deciding $\mathcal{ALC} + \mathbf{T}_{min}$ entailment in $\text{co-NEXP}^{\text{NP}}$.

As mentioned in the Introduction, the problem of representing prototypical properties and of reasoning about defeasible inheritance of such properties in DLs has been largely addressed in the literature. Several approaches are based on the integration of DLs with some kind of nonmonotonic reasoning mechanism. [2] proposes the extension of DL with Reiter’s default logic. 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 [15, 3]. A more general approach is undertaken in [7], where an extension of DL is proposed with two epistemic operators. This extension, called $\mathcal{ALCK}_{\mathcal{NF}}$, allows one to encode Reiter’s default logic as well as to express epistemic concepts and procedural rules. [13] extends the work in [7] by providing a translation of an $\mathcal{ALCK}_{\mathcal{NF}}$ KB to an equivalent *flat* KB and by defining a simplified tableau algorithm for flat KBs, which includes an optimized minimality check. In [4] an extension of DL with prioritized circumscription is proposed to express prototypical properties with exceptions, by introducing “abnormality” predicates whose extension is minimized. The authors provide algorithms for checking satisfiability, subsumption and instance checking which are proved to have an optimal complexity, but are based on massive nondeterministic guessing.

We plan to extend this work in several directions. First of all, we plan to optimize our tableau procedure, for instance by making the calculus $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, dealing with the monotonic logic $\mathcal{ALC} + \mathbf{T}$, more efficient. In particular, our goal is to obtain an EXP decision procedure, matching the known complexity of the logic $\mathcal{ALC} + \mathbf{T}$ [11]. From the point of view of knowledge representation, a limit of our logic is the inability to handle inheritance of multiple properties in case of exceptions as in the example: $\mathbf{T}(\textit{Student}) \sqsubseteq \neg \textit{HasIncome}$,

$\mathbf{T}(\textit{Student}) \sqsubseteq \exists \textit{Owns.LibraryCard}$, $\textit{PhDStudent} \sqsubseteq \textit{Student}$, $\mathbf{T}(\textit{PhDStudent}) \sqsubseteq \textit{HasIncome}$. Our semantics does not support the inference $\mathbf{T}(\textit{PhDStudent}) \sqsubseteq \exists \textit{Owns.LibraryCard}$, that is PhDStudents typically own a library card, as we might want to conclude. Moreover, we intend to establish a precise relation of $\mathcal{ALC} + \mathbf{T}_{min}$ with other formalisms for nonmonotonic reasoning, first of all with circumscription. Furthermore, we aim to extend our approach based on the typicality operator to *Low Complexity Description Logics*, such as the logics of the families \mathcal{EL} and $DL - Lite$ (see [1, 6]). These families of DLs have been identified and investigated to meet the needs of semantic web applications. Preliminary results on extending low complexity DLs with the \mathbf{T} operator are given in [12], where an extension $\mathcal{EL}^\perp + \mathbf{T}$ of \mathcal{EL}^\perp is presented and it is shown that the problem of deciding entailment in $\mathcal{EL}^\perp + \mathbf{T}$ is in CO-NP.

References

1. F. Baader, S. Brandt, and C. Lutz. Pushing the \mathcal{EL} envelope. In *Proc. of IJCAI'05, Professional Book Center* pp. 364-369, 2005.
2. F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. *J. Autom. Reasoning*, 14(1):149–180, 1995.
3. F. Baader and B. Hollunder. Priorities on defaults with prerequisites, and their application in treating specificity in terminological default logic. *J. of Automated Reasoning (JAR)*, 15(1):41–68, 1995.
4. P. A. Bonatti, C. Lutz, and F. Wolter. Description logics with circumscription. In *Proc. of KR*, pages 400–410, 2006.
5. M. Buchheit, F. M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. Artif. Int. Research (JAIR)*, 1:109–138, 1993.
6. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *dl-lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.
7. 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.
8. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *KR 2004*, 141-151.
9. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Preferential Description Logics. In *LPAR 2007. LNCS(LNAI)*, vol. 4790, pp. 257-272, 2007.
10. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Reasoning About Typicality in Preferential Description Logics. In *JELIA 2008. LNCS(LNAI)*, vol. 5293, pp. 192-205, 2008.
11. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. On Extending Description Logics for Reasoning About Typicality: a First Step. Technical Report 116/09, Dipartimento di Informatica, Università degli Studi di Torino, Italy, December 2009.
12. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Reasoning About Typicality in Low Complexity Description Logics: Preliminary Results. Technical Report 121/09, Dipartimento di Informatica, Università degli Studi di Torino, Italy, December 2009.
13. P. Ke and U. Sattler. Next Steps for Description Logics of Minimal Knowledge and Negation as Failure. In *DL2008*, 2008.

14. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
15. U. Straccia. Default inheritance reasoning in hybrid kl-one-style logics. In *Proc. of IJCAI*, pages 676–681, 1993.

Permissive nominal terms and their unification

Gilles Dowek Murdoch J. Gabbay Dominic Mulligan

Abstract. We introduce permissive nominal terms. Nominal terms extend first-order terms with binding. They lack properties of first- and higher-order terms: Terms must be reasoned on in a context of ‘freshness assumptions’; it is not always possible to ‘choose a fresh variable symbol’ for a nominal term; and it is not always possible to ‘alpha-convert a bound variable symbol’. Permissive nominal terms recover these ‘always fresh’ and ‘always alpha-rewrite’ properties. Freshness contexts are elided and the notion of unifier is based on substitution alone rather than on nominal terms’ notion of substitution plus freshness conditions. We prove that all expressivity of nominal unification is retained.

1 Introduction

Many formal languages feature *variable binding*: examples include λ -abstraction, quantification, and sets comprehension $\{x \mid \phi(x)\}$. Binding is ubiquitous, because variables are there to be bound or substituted.

Variables cannot be bound in first-order terms (constructed from variables and term-formers). This motivates extensions of first-order syntax; logics where variables can be bound by function or predicate symbols [DHK02], rewriting on terms with binders [KvOvR93, MN98, FG07], and languages with datatypes with binders [PE88, Mil91, SPG03, HHP87, Pau90, CU04].

We will study nominal terms [UPG04]. These preserve the flavour of first-order terms and extend them to represent informal statements like

“If $y \notin fv(t)$ then $\lambda x.t$ is α -equivalent with $\lambda y.[y/x]t$ ” and
“How can we choose t and u to make $\lambda x.\lambda y.(y t)$ equal to $\lambda x.\lambda x.(x u)$?”.

(We write $fv(t)$ for the free variables of t ; e.g. $fv(\lambda x.(xy)) = \{y\}$.)

Nominal terms use a characteristic combination of features: two levels of variable (*atoms* and *unknowns*), *freshness* conditions, *permutations*, and a distinctive theory of α -equivalence (see Section 6 or [UPG04] for details). The first statement above is rendered in nominal terms as the equality judgement $b\#X \vdash [a]X = [b](b a) \cdot X$. Here a and b denote atoms, which represent the ‘ x ’ and ‘ y ’; X denotes an unknown, it represents the ‘ t ’; $b\#X$ is a *freshness side-condition*, it represents the ‘ $y \notin fv(t)$ ’; $(b a)$ is a *permutation* meaning ‘map a to b and b to a ’, it represents the ‘ $[y/x]$ ’ (we assumed $y \notin fv(t)$, so this is possible).

Yet nominal terms from [UPG04] have some less attractive properties too:

- Freshness contexts are not fixed so we must often prove properties of *terms-in-freshness context*. This is harder than reasoning just about terms.

– We cannot always pick a fresh variable symbol and α -rename a bound variable. For X in the empty freshness context, there is — by definition of the empty freshness context — no a such that $a\#X$.¹

We propose *permissive nominal terms*. An unknown takes the form X^S where S is a permission sort (Definition 3). Permission sorts are sets of atoms that are both infinite and co-infinite (see Definition 2; $S \subseteq X$ is co-finite when $X \setminus S$ is infinite). Thus, we can always choose a fresh atom, always α -convert (Definition 8, Lemma 12), α -equivalence is independent of a freshness context (there is no freshness context), and the notion of unifier is based just on substitution rather than (as for nominal terms) a freshness context and a substitution.

We illustrate this with two examples; definitions are in the paper. $comb$ is an infinite, co-infinite set of atoms. Suppose $a, b \in comb$. Assume a binary term-former (function-symbol) for application. Then:

– α -equivalence is ‘if $y \notin fv(u)$ then $\lambda x.u = \lambda y.(u[x/x])$ ’.

In nominal terms this becomes $b\#Z \vdash [a]Z = [b](b a) \cdot Z$.² In permissive nominal terms it becomes $[a]Z^{comb \setminus \{b\}} = [b](b a) \cdot Z^{comb \setminus \{b\}}$.

It is not possible to α -convert a in the nominal term $\emptyset \vdash [a]Z$; to do this, we must enrich the freshness context \emptyset .

In permissive nominal terms, it is always possible to α -convert a in $[a]Z^{comb \setminus \{b\}}$; we can convert to any of the atoms not in $comb \setminus \{b\}$.

– ‘Unify $\lambda x.\lambda y.(yt)$ with $\lambda x.\lambda x.(xu)$ ’ becomes in nominal terms ‘ $[a][b](bX) \stackrel{?}{=} [a][a](a Y)$ ’ and has solution $(b\#Y, [X := (b a) \cdot Z, Y := Z])$.

In permissive nominal terms it becomes ‘ $[a][b](bX^{comb}) \stackrel{?}{=} [a][a](aY^{comb})$ ’ and has solution $[X^{comb} := Z^{comb \setminus \{b\}}, Y^{comb} := Z^{comb \setminus \{b\}}]$.

Unification is the basis of rewriting and logic programming. Nominal terms have served as the basis of both [FG07,CU04]. Therefore, and following that work, in this paper we explore permissive nominal unification.

2 Permissive nominal terms

Definition 1 Fix a countably infinite set A of **atoms**. We use a **permutative convention** that a, b, c, \dots will range over *distinct* atoms. Fix a set of **term-formers**. f, g, h will range over distinct term-formers.

Examples of term-formers are symbols like `lam`, `app`, and `forall`. In the presence of rewrites or equational axioms we can make these term-formers ‘do’ something; in this paper, they are just ways of building terms with suggestive names.

¹ ‘Freshness contexts’ sounds like ‘typing contexts’, but the terminology misleads. Extending a typing context makes more terms typable and, intuitively, extends the universe of discourse with extra typable terms. Extending a freshness context makes more terms *equal*. This is a more drastic event because it is more likely to change the behaviour of existing elements; they may now be equal to other elements with quite different behaviour.

² In the body of the paper we put dots on atoms and unknowns in nominal terms, to emphasise the difference from permissive nominal terms. Here, we do not bother.

Definition 2 Call $S \subseteq \mathbb{A}$ co-infinite when $\mathbb{A} \setminus S$ is infinite. Fix an infinite, co-infinite set $comb \subseteq \mathbb{A}$. A **permission sort** has the form $(comb \cup A_1) \setminus A_2$ for finite sets $A_1 \subseteq \mathbb{A}$ and $A_2 \subseteq \mathbb{A}$. S, S', T will range over permission sorts.

If S and T are permission sorts, so are $S \cup T$ and $S \cap T$, and both S and $\mathbb{A} \setminus S$ are infinite.

Definition 3 For each S fix a disjoint countably infinite set of **unknowns** of sort S . X^S, Y^S, Z^S , will range over distinct unknowns of sort S . If $S \neq S'$ then there is no particular connection between X^S and $X^{S'}$. \mathcal{V} will range over finite sets of unknowns (we use this from Section 4 onwards).

Definition 4 A **permutation** is a bijection on atoms such that $\{a \mid \pi(a) \neq a\}$ is finite. π and π' will range over permutations (not necessarily distinct).

Write id for the **identity** permutation such that $id(a) = a$ always. Write $(a\ b)$ for the **swapping** which maps a to b , b to a , and all other c to themselves.

Definition 5 Define (**permissive nominal**) terms by:

$$r, s, t, \dots ::= a \mid f(r_1, \dots, r_n) \mid [a]r \mid \pi \cdot X^S$$

We write \equiv for syntactic identity; $r \equiv s$ when r and s denote identical terms.

Definition 6 Define a **permutation action** by:

$$\pi \cdot a \equiv \pi(a) \quad \pi \cdot (f(r_1, \dots, r_n)) \equiv f(\pi \cdot r_1, \dots, \pi \cdot r_n) \quad \pi \cdot [a]r \equiv [\pi(a)](\pi \cdot r) \quad \pi \cdot (\pi' \cdot X^S) \equiv (\pi \circ \pi') \cdot X^S$$

Definition 7 If $S \subseteq \mathbb{A}$, define the **pointwise** action by: $\pi \cdot S = \{\pi(a) \mid a \in S\}$. Define **free atoms** $fa(r)$ and **unknowns** $fV(r)$ by:

$$\begin{aligned} fa(a) &= \{a\} & fa(f(r_1, \dots, r_n)) &= fa(r_1) \cup \dots \cup fa(r_n) & fa([a]r) &= fa(r) \setminus \{a\} & fa(\pi \cdot X^S) &= \pi \cdot S \\ fV(a) &= \emptyset & fV(f(r_1, \dots, r_n)) &= fV(r_1) \cup \dots \cup fV(r_n) & fV([a]r) &= fV(r) & fV(\pi \cdot X^S) &= \{X^S\} \end{aligned}$$

An intuition for $fa(r)$ is ‘possible free atoms after instantiation’.

Definition 8 Let $\pi|_S$ denote the partial function ‘ π restricted to S ’. Define α -**equivalence** $=_\alpha$ inductively by:

$$\begin{aligned} \frac{}{a =_\alpha a} (=_\alpha \mathbf{aa}) & \quad \frac{r_1 =_\alpha s_1 \quad \dots \quad r_n =_\alpha s_n}{f(r_1, \dots, r_n) =_\alpha f(s_1, \dots, s_n)} (=_\alpha \mathbf{f}) & \quad \frac{r =_\alpha s}{[a]r =_\alpha [a]s} (=_\alpha \mathbf{[a]}) \\ \frac{(b\ a) \cdot r =_\alpha s \quad (b \notin fa(r))}{[a]r =_\alpha [b]s} (=_\alpha \mathbf{[b]}) & \quad \frac{(\pi|_S = \pi'|_S)}{\pi \cdot X^S =_\alpha \pi' \cdot X^S} (=_\alpha \mathbf{X}) \end{aligned}$$

Lemma 9 $id \cdot r \equiv r$ and $\pi' \cdot (\pi \cdot r) \equiv (\pi' \circ \pi) \cdot r$.

Lemma 10 $\pi \cdot fa(r) = fa(\pi \cdot r)$.

Theorem 11 $=_\alpha$ is transitive, reflexive, and symmetric.

Lemma 12 For any r , there exist infinitely many b such that $b \notin fa(r)$. As a corollary, for any r and a there exists infinitely many fresh b (so $b \notin fa(r)$) such that for some s , $[a]r =_\alpha [b]s$.

Proof. The first part is by an easy induction on r . We consider only the case $r \equiv \pi \cdot X^S$: $fa(\pi \cdot X^S) = \pi \cdot S$, and infinitely many b are such that $b \notin \pi \cdot S$. The corollary follows using $(=_\alpha \mathbf{[b]})$.

3 Substitutions

Definition 13 A **substitution** θ is a function from unknowns to terms such that $fa(\theta(X^S)) \subseteq S$ always (so S in X^S describes the ‘permission’ we have to instantiate X^S , namely to terms with free atoms in S). $\theta, \theta', \theta_1, \theta_2$, will range over substitutions.³

Write id for the **identity** substitution mapping X^S to $id \cdot X^S$ always. It will always be clear whether id means the identity substitution or permutation.

Suppose $fa(t) \subseteq S$. Write $[X^S := t]$ for the substitution such that $[X^S := t](X^S) \equiv t$ and $[X^S := t](Y^T) \equiv id \cdot Y^T$ for all other Y^T .

Definition 14 Define a **substitution action** of substitutions on terms by:

$$a\theta \equiv a \quad f(r_1, \dots, r_n)\theta \equiv f(r_1\theta, \dots, r_n\theta) \quad ([a]r)\theta \equiv [a](r\theta) \quad (\pi \cdot X^S)\theta \equiv \pi \cdot \theta(X^S)$$

Theorem 15 $fa(r\theta) \subseteq fa(r)$.

Proof. By induction on r . We consider the case $r \equiv \pi \cdot X^S$. By definition $fa(\pi \cdot X^S) = \pi \cdot S$. By assumption in Definition 13, $fa(\theta(X^S)) \subseteq S$. Using Lemma 10, it follows that $fa(\pi \cdot \theta(X^S)) \subseteq \pi \cdot S$.

Lemma 16 $\pi \cdot (r\theta) \equiv (\pi \cdot r)\theta$.

Proof. By a routine induction on r using the definitions and Lemma 9.

Theorem 17 If $\theta(X^S) =_{\alpha} \theta'(X^S)$ for all $X^S \in fV(r)$, then $r\theta =_{\alpha} r\theta'$.

Definition 18 Define **composition** $\theta \circ \theta'$ by $(\theta \circ \theta')(X^S) \equiv (\theta(X^S))\theta'$.

Theorem 19 $(r\theta)\theta' \equiv r(\theta \circ \theta')$.

Proof. By induction on r . We consider the case of $\pi \cdot X^S$, and use Lemma 16: $(\pi \cdot X^S)(\theta \circ \theta') \equiv \pi \cdot (\theta \circ \theta')(X^S) \equiv \pi \cdot (\theta(X^S))\theta' \equiv (\pi \cdot \theta(X^S))\theta' \equiv ((\pi \cdot X^S)\theta)\theta'$

4 Support inclusion problems

Recall from Definition 13 that $fa(\theta(X^S)) \subseteq fa(X^S) = S$, and from Theorem 15 that instantiation must reduce the set of free atoms. We will exhibit an algorithm which, intuitively, solves the problem “please make $fa(r\theta) \subseteq T$ true” (Definition 27 and Lemma 25). In fact the algorithm calculates solutions that are most general, in a sense made formal in Theorem 32.

Definition 20 A **support inclusion** is a pair $r \sqsubseteq T$ of a term and a permission sort. A **support inclusion problem** is a finite multiset of support inclusions; Inc will range over support inclusion problems. Call θ a **solution** to Inc when $fa(r\theta) \subseteq T$ for every $r \sqsubseteq T \in Inc$. Write $Sol(Inc)$ for the solutions of Inc .

³ ‘ $fa(\theta(X^S)) \subseteq S$ ’ looks absent in nominal terms theory ([UPG04, Definition 2.13], [FG07, Definition 4]), yet it is there: see the conditions ‘ $\nabla' \vdash \theta(\nabla)$ ’ in Lemma 2.14, and ‘ $\nabla \vdash a\#\theta(t)$ ’ in Definition 3.1 of [UPG04]. More on this in Section 6.

Definition 21 Define a **simplification** rewrite relation by:

$$\begin{array}{ll}
(\sqsubseteq \mathbf{a}) & a \sqsubseteq T, Inc \implies Inc \quad (a \in T) \\
(\sqsubseteq \mathbf{f}) & f(r_1, \dots, r_n) \sqsubseteq T, Inc \implies r_1 \sqsubseteq T, \dots, r_n \sqsubseteq T, Inc \\
(\sqsubseteq \mathbf{[]}) & [a]r \sqsubseteq T, Inc \implies r \sqsubseteq T \cup \{a\}, Inc \\
(\sqsubseteq \mathbf{X}) & \pi \cdot X^S \sqsubseteq T, Inc \implies id \cdot X^S \sqsubseteq \pi^{-1} \cdot T, Inc \quad (S \not\subseteq \pi^{-1} \cdot T, \pi \neq id) \\
(\sqsubseteq \mathbf{X}') & \pi \cdot X^S \sqsubseteq T, Inc \implies Inc \quad (S \subseteq \pi^{-1} \cdot T)
\end{array}$$

Lemma 22 If $Inc \implies Inc'$ then $Sol(Inc) = Sol(Inc')$.

Proof. We consider $(\sqsubseteq \mathbf{X})$. Suppose $S \not\subseteq \pi^{-1} \cdot T$ and $\pi \neq id$. Suppose $\theta \in Sol(\pi \cdot X^S \sqsubseteq T, Inc)$. Then $fa((\pi \cdot X^S)\theta) \subseteq T$. By Lemma 16 $(\pi \cdot X^S)\theta \equiv \pi \cdot (X^S\theta)$. By Lemma 10 $fa(\pi \cdot (X^S\theta)) = \pi \cdot fa(X^S\theta)$. So $\pi \cdot fa(X^S\theta) \subseteq T$. π is a bijection, so $fa(X^S\theta) \subseteq \pi^{-1} \cdot T$ and $\theta \in Sol(X^S \sqsubseteq \pi^{-1} \cdot T, Inc)$. Conversely, if $\theta \in Sol(X^S \sqsubseteq \pi^{-1} \cdot T, Inc)$ then $\theta \in Sol(\pi \cdot X^S \sqsubseteq T, Inc)$ follows by a similar argument.

Lemma 23 Support inclusion problem simplification is confluent and terminating. Write $nf(Inc)$ for the unique \implies -normal form of Inc .

Proof.(Sketch) Support inclusion problem simplification is strongly confluent (see, for instance [BN98]), hence it is confluent. It is not hard to define a notion of size on problems such that rewrite rules strictly reduce the size of the problem they act on; it follows that simplification is terminating.

Definition 24 Call Inc **consistent** when $a \sqsubseteq T \notin nf(Inc)$ for all atoms a and permission sorts T . Call Inc **solvable** when $Sol(Inc) \neq \emptyset$. Call Inc **non-trivial** when $nf(Inc) \neq \emptyset$.

Lemma 25 If Inc is consistent then all $inc \in nf(Inc)$ have the form $X^S \sqsubseteq T$ where $S \not\subseteq T$.

Definition 26 Define $fV(Inc)$ by $fV(Inc) = \bigcup \{fV(r) \mid \exists T. r \sqsubseteq T \in Inc\}$.

Definition 27 Let \mathcal{V} range over finite sets of unknowns.

Suppose Inc is consistent. For every $X^S \in \mathcal{V}$ make a fixed but arbitrary choice of $X'^{S'}$ such that $X'^{S'} \notin \mathcal{V}$ and $S' = S \cap \bigcap \{T \mid X^S \sqsubseteq T \in nf(Inc)\}$.

We make our choice injectively; for distinct $X^S \in fV(Inc)$ and $Y^T \in fV(Inc)$, we choose $X'^{S'}$ and $Y'^{T'}$ distinct. It will be convenient to write $\mathcal{V}_{Inc}^{\mathcal{V}}$ for the set of our choices $\{X'^{S'} \mid X^S \in \mathcal{V}\}$. Define a substitution $\rho_{Inc}^{\mathcal{V}}$ by:

$$\rho_{Inc}^{\mathcal{V}}(X^S) \equiv id \cdot X'^{S'} \text{ if } X^S \in \mathcal{V} \quad \rho_{Inc}^{\mathcal{V}}(Y^T) \equiv id \cdot Y^T \text{ otherwise.}$$

Lemma 28 If Inc is consistent then $\rho_{Inc}^{\mathcal{V}} \in Sol(Inc)$. ($\rho_{Inc}^{\mathcal{V}}$ solves Inc .)

Proof. Suppose Inc is a \implies -normal form. If $X^S \sqsubseteq T \in Inc$ then $\rho_{Inc}^{\mathcal{V}}(X) = id \cdot X'^{S'}$ for an S' which, by construction, satisfies $S' \subseteq T$. The result follows.

More generally, if Inc is not a \implies -normal form then note that by Lemma 22 $Sol(Inc) = Sol(nf(Inc))$; we use the previous paragraph.

Theorem 29 *Inc is consistent if and only if Inc is solvable.*

Proof. By Lemma 22 $Sol(Inc) = Sol(nf(Inc))$, so it suffices to show the result for the special case that Inc is a \implies -normal form.

Suppose Inc is inconsistent, so $nf(Inc)$ contains some support inclusions of the form $a \sqsubseteq T$ where $a \notin T$. $a\theta \equiv a$ always, so there is no substitution θ such that $a\theta \sqsubseteq T$. Conversely, if Inc is consistent the result follows by Lemma 28.

Definition 30 Suppose that Inc is consistent, $fV(Inc) \subseteq \mathcal{V}$, and $\theta \in Sol(Inc)$. Define a substitution $\theta\text{-}\rho_{Inc}^\mathcal{V}$ by:

- $(\theta\text{-}\rho_{Inc}^\mathcal{V})(X^{S'}) \equiv \theta(X^S)$ if $X^S \in \mathcal{V}$ and $\rho_{Inc}^\mathcal{V}(X^S) \equiv id \cdot X^{S'}$.
- $(\theta\text{-}\rho_{Inc}^\mathcal{V})(X^S) \equiv \theta(X^S)$ if $X^S \notin \mathcal{V}$.

Lemma 31 *Suppose $\theta \in Sol(Inc)$ and $fV(Inc) \subseteq \mathcal{V}$. Then $\rho_{Inc}^\mathcal{V}$ exists, and $\theta\text{-}\rho_{Inc}^\mathcal{V}$ is a substitution.*

Proof. If $\theta \in Sol(Inc)$ then Inc is solvable (Definition 20). By Theorem 29, Inc is consistent. Therefore $\rho_{Inc}^\mathcal{V}$ from Definition 27 exists. We now show that $fa((\theta\text{-}\rho_{Inc}^\mathcal{V})(X^{S'})) \subseteq S$ always. We reason by cases:

- The case that $id \cdot X^{S'} \equiv \rho_{Inc}^\mathcal{V}(X^S)$ for $X^S \in \mathcal{V}$.

It is not hard to prove that $fV(nf(Inc)) \subseteq fV(Inc)$ always, so $fV(nf(Inc)) \subseteq \mathcal{V}$.

There are two sub-cases:

- The case $X^S \notin fV(nf(Inc))$. Then $S = S'$ and $(\theta\text{-}\rho_{Inc}^\mathcal{V})(X^{S'}) = \theta(X^S)$.

By assumption $fa(\theta(X^S)) \subseteq S$.

- The case $X^S \in fV(nf(Inc))$. $\theta \in Sol(Inc)$ so by Lemma 22 $\theta \in Sol(nf(Inc))$ and so $fa(\theta(X^S)) \subseteq T$ for every T such that $X^S \sqsubseteq T \in nf(Inc)$. By definition $S' = \bigcap \{T \mid X^S \sqsubseteq T \in nf(Inc)\}$ and the result follows.

- Otherwise, $(\theta\text{-}\rho_{Inc}^\mathcal{V})(X^S) \equiv \theta(X^S)$. By assumption $fa(\theta(X^S)) \subseteq S$.

Theorem 32 *Suppose $\theta \in Sol(Inc)$ and suppose $fV(Inc) \subseteq \mathcal{V}$. Then $\theta(X^S) \equiv (\rho_{Inc}^\mathcal{V} \circ (\theta\text{-}\rho_{Inc}^\mathcal{V}))(X^S)$ for every $X^S \in \mathcal{V}$.*

Proof. $\rho(X^S) \equiv id \cdot X^{S'}$ for some fresh $X^{S'} \notin \mathcal{V}$, and $(\theta\text{-}\rho_{Inc}^\mathcal{V})(X^{S'}) \equiv \theta(X^S)$. The result follows by Lemma 9.

5 Permissive nominal unification problems

5.1 Problems, solutions, the unification algorithm

Definition 33 An **equality (problem)** is a pair $r \stackrel{?}{=} s$. A **problem** Pr is a finite multiset of equalities. Define $Pr\theta$ by $Pr\theta = \{r\theta \stackrel{?}{=} s\theta \mid r \stackrel{?}{=} s \in Pr\}$.

Definition 34 θ **solves** Pr when $r \stackrel{?}{=} s \in Pr$ implies $r\theta =_\alpha s\theta$. Write $Sol(Pr)$ for the set of solutions to Pr . Call Pr **solvable** when $Sol(Pr)$ is non-empty.

A solution to Pr ‘makes the equalities valid’, as for first- and higher-order unification. This simplifies the nominal unification notion of solution (Definition 66 or [UPG04, Definition 3.1]) based on ‘a substitution + a freshness context’.

Lemma 35 $\theta \circ \theta' \in \text{Sol}(Pr)$ if and only if $\theta' \in \text{Sol}(Pr\theta)$.

Proof. Suppose $\theta \circ \theta' \in \text{Sol}(Pr)$ and $r \stackrel{?}{=} s \in Pr$. We reason using Theorem 19: $(r\theta)\theta' \equiv r(\theta \circ \theta') =_{\alpha} s(\theta \circ \theta') \equiv (s\theta)\theta'$. The reverse implication is similar.

Definition 36 If Pr is a problem, define a support inclusion problem Pr_{\sqsubseteq} by: $Pr_{\sqsubseteq} = \{r \sqsubseteq fa(s), s \sqsubseteq fa(r) \mid r \stackrel{?}{=} s \in Pr\}$.

Definition 37 Define a **simplification** rewrite relation $\mathcal{V}; Pr \Longrightarrow \mathcal{V}'; Pr'$ by:

$$\begin{array}{llll}
(\stackrel{?}{=} \mathbf{a}) & \mathcal{V}; a \stackrel{?}{=} a, Pr & \Longrightarrow & \mathcal{V}; Pr \\
(\stackrel{?}{=} \mathbf{f}) & \mathcal{V}; f(r_1, \dots) \stackrel{?}{=} f(s_1, \dots), Pr & \Longrightarrow & \mathcal{V}; r_1 \stackrel{?}{=} s_1, \dots, Pr \\
(\stackrel{?}{=} \mathbf{[a]}) & \mathcal{V}; [a]r \stackrel{?}{=} [a]s, Pr & \Longrightarrow & \mathcal{V}; r \stackrel{?}{=} s, Pr \\
(\stackrel{?}{=} \mathbf{[b]}) & \mathcal{V}; [a]r \stackrel{?}{=} [b]s, Pr & \Longrightarrow & \mathcal{V}; (b a) \cdot r \stackrel{?}{=} s, Pr \\
& & & (b \notin fa(r)) \\
(\stackrel{?}{=} \mathbf{X}) & \mathcal{V}; \pi \cdot X^S \stackrel{?}{=} \pi \cdot X^S, Pr & \Longrightarrow & \mathcal{V}; Pr \\
(\mathbf{I1}) & \mathcal{V}; \pi \cdot X^S \stackrel{?}{=} s, Pr & \xrightarrow{[X^S := \pi^{-1} \cdot s]} & \mathcal{V}; Pr[X^S := \pi^{-1} \cdot s] \\
& & & (X^S \notin fV(s), fa(s) \subseteq \pi \cdot S) \\
(\mathbf{I2}) & \mathcal{V}; r \stackrel{?}{=} \pi \cdot X^S, Pr & \xrightarrow{[X^S := \pi^{-1} \cdot r]} & \mathcal{V}; Pr[X^S := \pi^{-1} \cdot r] \\
& & & (X^S \notin fV(r), fa(r) \subseteq \pi \cdot S) \\
(\mathbf{I3}) & \mathcal{V}; Pr & \xrightarrow{\rho_{Pr_{\sqsubseteq}}^{\mathcal{V}}} & \mathcal{V} \cup \mathcal{V}_{Pr_{\sqsubseteq}}^{\mathcal{V}}; Pr \rho_{Pr_{\sqsubseteq}}^{\mathcal{V}} \\
& & & (Pr_{\sqsubseteq} \text{ consistent and non-trivial})
\end{array}$$

Call $(\stackrel{?}{=} \mathbf{a})$, $(\stackrel{?}{=} \mathbf{f})$, $(\stackrel{?}{=} \mathbf{[a]})$, $(\stackrel{?}{=} \mathbf{[b]})$, and $(\stackrel{?}{=} \mathbf{X})$ **non-instantiating rules**.

Call $(\mathbf{I1})$, $(\mathbf{I2})$, and $(\mathbf{I3})$ **instantiating rules**.

We insist Pr_{\sqsubseteq} is non-trivial (Definition 20) to avoid indefinite rewrites. We insist Pr_{\sqsubseteq} is consistent so $\rho_{Pr_{\sqsubseteq}}^{\mathcal{V}}$ exists. $\rho_{Pr_{\sqsubseteq}}^{\mathcal{V}}$ and $\mathcal{V}_{Pr_{\sqsubseteq}}^{\mathcal{V}}$ are defined in Definition 27.

Lemma 38 If $Pr \Longrightarrow Pr'$ by a non-instantiating rule then $\text{Sol}(Pr) = \text{Sol}(Pr')$.

Proof. The interesting case is $(\stackrel{?}{=} \mathbf{[b]})$. Suppose that $Pr = [a]r \stackrel{?}{=} [b]s, Pr''$ and $b \notin fa(r)$ and so $Pr \Longrightarrow (b a) \cdot r \stackrel{?}{=} s, Pr''$ with $(\stackrel{?}{=} \mathbf{[b]})$. Then:

– Suppose $([a]r)\theta =_{\alpha} ([b]s)\theta$. By Definition 14 $[a](r\theta) =_{\alpha} [b](s\theta)$. By the structure of the rules in Definition 8, $(b a) \cdot (r\theta) =_{\alpha} s\theta$. By Lemma 16 and Theorem 11, $((b a) \cdot r)\theta =_{\alpha} s\theta$.

– Suppose $((b a) \cdot r)\theta =_{\alpha} s\theta$. By Lemma 16 and Theorem 11, $(b a) \cdot (r\theta) =_{\alpha} s\theta$. By Theorem 15 $b \notin fa(r\theta)$. Therefore by $(=_{\alpha} \mathbf{[b]})$ $[a](r\theta) =_{\alpha} [b](s\theta)$. By Definition 14 $[a](r\theta) =_{\alpha} [b](s\theta)$, as required.

Definition 39 Define $fV(Pr) = \bigcup \{fV(r) \cup fV(s) \mid r \stackrel{?}{=} s \in Pr\}$.

Definition 40 Suppose \mathcal{V} is a set of unknowns. Define $\theta|_{\mathcal{V}}$ by:⁴

$$\theta|_{\mathcal{V}}(X) \equiv \theta(X) \text{ if } X \in \mathcal{V} \quad \theta|_{\mathcal{V}}(X) \equiv id \cdot X \text{ otherwise.}$$

⁴ We overload $|$, for technical convenience: $\pi|_S$ (Definition 8) is partial and $\theta|_{\mathcal{V}}$ is total.

Definition 41 If Pr is a problem, define a **unification algorithm** by:

1. Rewrite $fV(Pr); Pr$ using the rules of Definition 37 as much as possible, with top-down precedence (so apply $(\stackrel{?}{=}a)$ before $(\stackrel{?}{=}f)$, and so on down to **(I3)**).
2. If we reduce to $\mathcal{V}; \emptyset$, we succeed and return $\theta|_{\mathcal{V}}$ where θ is the functional composition of all the substitutions labelling rewrites (we take $\theta = id$ if there are none). Otherwise, we fail.

Lemma 42 *The algorithm of Definition 41 always terminates.*

Proof. By an easy argument using a notion of the size of a unification problem.

Lemma 43 *Suppose $\theta(X^S) =_{\alpha} \theta'(X^S)$ for all $X^S \in fV(Pr)$. Then $\theta \in Sol(Pr)$ if and only if $\theta' \in Sol(Pr)$.*

Proof. Unpacking Definition 34 it suffices to show that $r\theta =_{\alpha} s\theta$ if and only if $r\theta' =_{\alpha} s\theta'$, for every $r \stackrel{?}{=} s \in Pr$. This is easy using Theorem 17 and the fact by construction (Definition 39) that $fV(r) \subseteq fV(Pr)$ and $fV(s) \subseteq fV(Pr)$.

Definition 44 Write $\theta\text{-}X^S$ for the substitution such that:

$$(\theta\text{-}X^S)(X^S) \equiv id \cdot X^S \quad \text{and} \quad (\theta\text{-}X^S)(Y^T) \equiv \theta(Y^T) \text{ for all other } Y^T$$

Theorem 45 *Suppose $(id \cdot X^S)\theta =_{\alpha} s\theta$ and $X^S \notin fV(s)$. Then:*

$$\theta(X^S) =_{\alpha} ([X^S := s] \circ (\theta\text{-}X^S))(X^S) \quad \text{and} \quad \theta(Y^T) =_{\alpha} ([X^S := s] \circ (\theta\text{-}X^S))(Y^T)$$

Proof. We reason as follows:

$$\begin{aligned} ([X^S := s] \circ (\theta\text{-}X^S))(X^S) &\equiv s(\theta\text{-}X^S) && \text{Definition 14} \\ &\equiv s\theta && X^S \notin fV(s), \text{ Theorem 17} \\ &=_{\alpha} \theta(X^S) && \text{Assumption} \\ ([X^S := s] \circ (\theta\text{-}X^S))(Y^T) &\equiv (\theta\text{-}X^S)(Y^T) && \text{Definition 18} \\ &\equiv \theta(Y^T) && \text{Definition 44} \end{aligned}$$

5.2 Simplification rewrites calculate principal solutions

Definition 46 Write $\theta_1 \leq \theta_2$ when there exists θ' such that $X^S\theta_2 =_{\alpha} X^S(\theta_1 \circ \theta')$ always. We call \leq the **instantiation ordering**.

Definition 47 A **principal** (or **most general**) solution to a problem Pr is a solution $\theta \in Sol(Pr)$ such that $\theta \leq \theta'$ for all other $\theta' \in Sol(Pr)$.

Our main results are Theorems 48 — the unification algorithm from Definition 41 calculates a solution — and 53 — the solution it calculates, is principal.

Theorem 48 *Suppose $fV(Pr) \subseteq \mathcal{V}$. If $\mathcal{V}; Pr \xrightarrow{\theta} \mathcal{V}; \emptyset$ then $\theta|_{\mathcal{V}} \in Sol(Pr)$.*

Proof. By induction on the length of the path in $\xrightarrow{\theta}$. If it has length 0 then $Pr = \emptyset$ and $\theta \equiv id$ and the result follows. Otherwise, there are three cases:

– *The non-instantiating case.* Suppose $\mathcal{V}; Pr \implies \mathcal{V}; Pr'' \xRightarrow{\theta} \mathcal{V}'; \emptyset$. By easy calculations $fV(Pr'') \subseteq \mathcal{V}$. By inductive hypothesis $\theta|_{\mathcal{V}} \in \text{Sol}(Pr'')$. By Lemma 38, $\theta|_{\mathcal{V}} \in \text{Sol}(Pr)$.

– *The case of (I1) or (I2).* Suppose $\mathcal{V}; Pr \xrightarrow{X} \mathcal{V}; Pr\chi \xRightarrow{\theta'} \mathcal{V}'; \emptyset$. By easy calculations $fV(Pr\chi) \subseteq \mathcal{V}$. By inductive hypothesis $\theta'|_{\mathcal{V}} \in \text{Sol}(Pr\chi)$. It is a fact that $(\chi \circ \theta')|_{\mathcal{V}} = \chi \circ (\theta'|_{\mathcal{V}})$. By Lemma 35, $(\chi \circ \theta')|_{\mathcal{V}} \in \text{Sol}(Pr)$.

– *The case of (I3).* Suppose $\mathcal{V}; Pr \xrightarrow{\rho} \mathcal{V}'; Pr\rho \xRightarrow{\theta'} \mathcal{V}''; \emptyset$. By easy calculations $fV(Pr\rho) \subseteq \mathcal{V}'$. By inductive hypothesis $\theta'|_{\mathcal{V}'} \in \text{Sol}(Pr\rho)$. By Lemma 35, $\rho \circ (\theta'|_{\mathcal{V}'}) \in \text{Sol}(Pr)$. It is a fact that $\rho \circ (\theta'|_{\mathcal{V}'}) = (\rho \circ \theta')|_{\mathcal{V}'}$. By Lemma 43, $(\rho \circ \theta')|_{\mathcal{V}} \in \text{Sol}(Pr)$.

Lemma 49 *If $\theta_1 \leq \theta_2$ then $\theta \circ \theta_1 \leq \theta \circ \theta_2$.*

Proof. By Definition 46 θ' exists such that $X^S\theta_2 =_{\alpha} X^S(\theta_1 \circ \theta')$ always. We use Theorems 19 and 17: $X^S(\theta \circ \theta_2) \equiv (X^S\theta)\theta_2 =_{\alpha} (X^S\theta)(\theta_1 \circ \theta') \equiv X^S((\theta \circ \theta_1) \circ \theta')$

Lemma 50 *Suppose $X^S\theta_2 =_{\alpha} X^S\theta'_2$ always. Then $\theta_1 \leq \theta_2$ implies $\theta_1 \leq \theta'_2$.*

Proof. By a routine calculation unpacking Definition 46 and using Theorem 11.

Lemma 51 *If $r =_{\alpha} s$ then $fa(r) = fa(s)$.*

Lemma 52 *If $\theta \in \text{Sol}(Pr)$ (Definition 34) then $\theta \in \text{Sol}(Pr_{\perp})$ (Definition 20).*

Proof. Using Lemma 51.

Theorem 53 *Suppose $fV(Pr) \subseteq \mathcal{V}$. If $\mathcal{V}; Pr \xRightarrow{\theta} \mathcal{V}'; \emptyset$ then $\theta|_{\mathcal{V}}$ is a principal solution to Pr .*

Proof. By Theorem 48 $\theta|_{\mathcal{V}} \in \text{Sol}(Pr)$. We prove that $\theta|_{\mathcal{V}}$ is principal by induction on the path length of $\mathcal{V}; Pr \xRightarrow{\theta} \mathcal{V}'; \emptyset$.

– *Length 0.* So $Pr = \emptyset$ and $\theta = id|_{\mathcal{V}}$. $id|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}}$ is a fact of Definition 46.

– *Length $k + 1$.* We consider the rules in Definition 37.

– *The non-instantiating case.* Suppose $\mathcal{V}; Pr \implies \mathcal{V}; Pr' \xRightarrow{\theta} \mathcal{V}'; \emptyset$ where $\mathcal{V}; Pr \implies \mathcal{V}; Pr'$ is non-instantiating. By inductive hypothesis, $\theta|_{\mathcal{V}}$ is a principal solution of Pr' . By Lemma 38 $\theta|_{\mathcal{V}}$ is also a principal solution of Pr .

– *The case (I1).* Suppose $fa(s) \subseteq \pi \cdot S$ and $X^S \notin fV(s)$. Write $\chi = [X^S := \pi^{-1} \cdot s]$. Suppose $Pr = \pi \cdot X^S \stackrel{?}{=} s$, Pr'' so that $\mathcal{V}; \pi \cdot X^S \stackrel{?}{=} s$, $Pr'' \xrightarrow{X} \mathcal{V}; Pr''\chi$ and suppose $\mathcal{V}; Pr''\chi \xRightarrow{\theta''} \mathcal{V}'; \emptyset$ and $\theta'|_{\mathcal{V}} \in \text{Sol}(Pr)$.

By Theorem 48 $\theta''|_{\mathcal{V}} \in \text{Sol}(Pr''\chi)$. It is routine to check that $fV(Pr''\chi) \subseteq \mathcal{V}$. By Theorem 45 and Lemma 43, $\chi \circ (\theta''|_{\mathcal{V}} - X^S) \in \text{Sol}(Pr)$. It is a fact that $(\theta''|_{\mathcal{V}} - X^S) = (\theta'' - X^S)|_{\mathcal{V}}$ so by Lemma 35, $(\theta'' - X^S)|_{\mathcal{V}} \in \text{Sol}(Pr''\chi)$.

By inductive hypothesis $\theta''|_{\mathcal{V}} \leq (\theta'' - X^S)|_{\mathcal{V}}$. By Lemma 49 we have $\chi \circ (\theta''|_{\mathcal{V}}) \leq \chi \circ (\theta'' - X^S)|_{\mathcal{V}}$. Now by assumption $fV(s) \subseteq \mathcal{V}$ and $X \in \mathcal{V}$, and it follows that $\chi \circ (\theta''|_{\mathcal{V}}) = (\chi \circ \theta'')|_{\mathcal{V}}$. Also, it is a fact that $(\theta'' - X^S)|_{\mathcal{V}} = \theta''|_{\mathcal{V}} - X^S$. By Theorem 45 and Lemma 50, $(\chi \circ \theta'')|_{\mathcal{V}} \leq \theta''|_{\mathcal{V}}$ as required.

– *The case (I2)* is similar to the case of (I1).

– The case **(I3)**. Suppose Pr_{\sqsubseteq} is consistent and non-trivial. Write $\rho = \rho_{Pr_{\sqsubseteq}}^{\vee}$, so that $\mathcal{V}; Pr \xrightarrow{\rho} \mathcal{V}''; Pr\rho$ and $\mathcal{V}''; Pr\rho \xrightarrow{\theta''} \mathcal{V}'; \emptyset$ and suppose $\theta'|_{\mathcal{V}} \in Sol(Pr)$. By Theorem 48 $\theta''|_{\mathcal{V}''} \in Sol(Pr\rho)$. $\mathcal{V}'' = \mathcal{V} \cup \mathcal{V}_{Pr_{\sqsubseteq}}^{\vee}$, so $fV(Pr\rho) \subseteq \mathcal{V}''$. By Lemma 52 $\theta'|_{\mathcal{V}} \in Sol(Pr_{\sqsubseteq})$. By Theorem 32 and Lemma 43 it follows that $\rho \circ (\theta'|_{\mathcal{V}-\rho}) \in Sol(Pr)$. By Lemma 35, $\theta'|_{\mathcal{V}-\rho} \in Sol(Pr\rho)$. By inductive hypothesis $\theta''|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}-\rho}$. By Lemma 49 $\rho \circ \theta''|_{\mathcal{V}} \leq \rho \circ (\theta'|_{\mathcal{V}-\rho})$. It is a fact that $\rho \circ (\theta''|_{\mathcal{V}}) = (\rho \circ \theta'')|_{\mathcal{V}}$. By Theorem 32 and Lemma 50, $(\rho \circ \theta'')|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}}$ as required.

Lemma 54 *If $\mathcal{V}; Pr \xrightarrow{X} \mathcal{V}; Pr'$ with **(I1)** or **(I2)** then $\theta \in Sol(Pr)$ implies $\theta-\chi \in Sol(Pr')$. Similarly, if $\mathcal{V}; Pr \xrightarrow{\rho} \mathcal{V}'; Pr'$ with **(I3)** then $\theta \in Sol(Pr)$ implies $\theta-\rho \in Sol(Pr')$.*

Proof. Suppose $fa(s) \subseteq \pi \cdot S$ and $X^S \notin fV(s)$. Write $\chi = [X^S := \pi^{-1} \cdot s]$. Suppose $Pr = \pi \cdot X^S \stackrel{?}{=} s$, Pr'' so that $\mathcal{V}; \pi \cdot X^S \stackrel{?}{=} s$, $Pr'' \xrightarrow{X} \mathcal{V}; Pr''\chi$. Now suppose $\theta \in Sol(Pr)$. By Lemma 43 and Theorem 45, $\chi \circ (\theta-X^S) \in Sol(Pr)$. By Lemma 35, $\theta-X^S \in Sol(Pr\chi)$. It follows that $\theta-X^S \in Sol(Pr''\chi)$ as required.

Suppose Pr_{\sqsubseteq} is consistent and non-trivial. Write $\rho = \rho_{Pr_{\sqsubseteq}}^{\vee}$, so that $\mathcal{V}; Pr \xrightarrow{\rho} \mathcal{V}''; Pr\rho$. Now suppose $\theta \in Sol(Pr)$. By Lemma 43 and Theorem 32, $\rho \circ (\theta-\rho) \in Sol(Pr)$. By Lemma 35, $\theta-\rho \in Sol(Pr\rho)$ as required.

Theorem 55 *Given a problem Pr , if the algorithm of Definition 41 succeeds then it returns a principal solution; if it fails then there is no solution.*

Proof. If the algorithm succeeds we use Theorem 53. Otherwise, the algorithm generates an element of the form $f(r_1, \dots, r_n) \stackrel{?}{=} f(r'_1, \dots, r'_{n'})$ where $n \neq n'$, $f(\dots) \stackrel{?}{=} g(\dots)$, $f(\dots) \stackrel{?}{=} [a]s$, $f(\dots) \stackrel{?}{=} a$, $[a]r =_{\alpha} a$, $[a]r =_{\alpha} b$, $a \stackrel{?}{=} b$, a Pr such that Pr_{\sqsubseteq} is inconsistent, or $\pi \cdot X^S \stackrel{?}{=} r$ or $r \stackrel{?}{=} \pi \cdot X^S$ where $X^S \in fV(r)$. By arguments on syntax and size of syntax, no solution to the reduced problem exists. It follows by Lemma 54 that no solution to Pr exists.

6 Relation to nominal terms

In permissive nominal terms, freshness information is fixed once and for all. This is mentioned already as a design alternative in [UPG04, Remark 2.6], but there, we would obtain ‘permission sorts’ A such that A is infinite and $\mathbb{A} \setminus A$ is finite. Permission sorts of *co-infinite* sets of atoms are new, as far as we know.

We will now develop the intuitively clear connection with [UPG04] into a precise mathematical correspondence between nominal unification and permissive nominal unification. Definition 59 translates from ‘nominal’ to ‘permissive’. Theorems 62 and 63 express how this translation is sound and complete for respective notions of α -equivalence. Theorem 71 then shows the most interesting fact: that furthermore, solutions to unification problems are also preserved across the translation.

A feature of Definition 59 is that it maps nominal terms to permissive nominal terms with free atoms in *comb*. In nominal terms we may need to enrich the

freshness context (see [GM07, Figure 2, axiom (fr)] and [GM09b, e.g. Lemma 25 and Theorem 33]). One way to view the interpretation of Definition 59 is therefore this: *comb* is ‘the atoms we had available so far’ (any other permission sort would do as well) and $\mathbb{A} \setminus \text{comb}$ is ‘the atoms with which we will extend the freshness context, in the future’. Both these sets are countably infinite, and syntax is finite, so it is not absolutely necessary to explicitly separate them: permissive nominal terms do this, for each fixed permission sort S ; nominal terms do not.

Definition 56 Fix a countably infinite set of **nominal atoms**, \mathbb{A} . $\dot{a}, \dot{b}, \dot{c}, \dots$ will range over distinct nominal atoms. Fix a bijection ι between \mathbb{A} and *comb* (Definition 2). Fix a countably infinite set of **nominal unknowns**, $\dot{X}, \dot{Y}, \dot{Z}, \dots$ will range over distinct nominal unknowns. A **nominal permutation** is a bijection $\dot{\pi}$ on \mathbb{A} such that $\{\dot{a} \mid \dot{\pi}(\dot{a}) \neq \dot{a}\}$ is finite. $\dot{\pi}, \dot{\pi}', \dot{\pi}'', \dots$ will range over permutations.

Write $\dot{\pi}^{-1}$ for the inverse of $\dot{\pi}$, \dot{id} for the identity permutation, and $\dot{\pi} \circ \dot{\pi}'$ for function composition, as is standard. For example, $(\dot{\pi} \circ \dot{\pi}')(\dot{a}) = \dot{\pi}(\dot{\pi}'(\dot{a}))$

Definition 57 Define **nominal terms** by $\dot{r}, \dot{s}, \dot{t} ::= \dot{a} \mid \dot{\pi} \cdot \dot{X} \mid [\dot{a}]\dot{r} \mid \mathbf{f}(\dot{r}, \dots, \dot{r})$, with permutation action

$$\dot{\pi} \cdot \dot{a} \equiv \dot{\pi}(\dot{a}) \quad \dot{\pi} \cdot \mathbf{f}(\dot{r}_1, \dots) \equiv \mathbf{f}(\dot{\pi} \cdot \dot{r}_1, \dots) \quad \dot{\pi} \cdot [\dot{a}]\dot{r} \equiv [\dot{\pi}(\dot{a})](\dot{\pi} \cdot \dot{r}) \quad \dot{\pi} \cdot (\dot{\pi}' \cdot \dot{X}) \equiv (\dot{\pi} \circ \dot{\pi}') \cdot \dot{X}$$

Write \equiv for syntactic identity. \mathbf{f} ranges over term-formers (Definition 1).

Definition 58 A **freshness** is a pair $\dot{a} \# \dot{r}$. An **equality** is a pair $\dot{r} = \dot{s}$. A **freshness context** is a finite set of freshesses of the form $\dot{a} \# \dot{X}$. Define **derivable freshness** and **derivable equality** by:

$$\begin{array}{c} \frac{}{\Delta \vdash \dot{a} \# \dot{b}} (\# \dot{\mathbf{b}}) \quad \frac{\Delta \vdash \dot{a} \# \dot{r}_i \quad (1 \leq i \leq n)}{\Delta \vdash \dot{a} \# \mathbf{f}(\dot{r}_1, \dots, \dot{r}_n)} (\# \mathbf{f}) \quad \frac{}{\Delta \vdash \dot{a} \# [\dot{a}]\dot{r}} (\# [\dot{\mathbf{a}}]) \\ \frac{\Delta \vdash \dot{a} \# \dot{r}}{\Delta \vdash \dot{a} \# [\dot{b}]\dot{r}} (\# [\dot{\mathbf{b}}]) \quad \frac{(\dot{\pi}^{-1}(\dot{a}) \# \dot{X} \in \Delta)}{\Delta \vdash \dot{a} \# \dot{\pi} \cdot \dot{X}} (\# \dot{\mathbf{X}}) \\ \frac{}{\Delta \vdash \dot{a} = \dot{a}} (= \dot{\mathbf{a}}) \quad \frac{\Delta \vdash \dot{r}_i = \dot{s}_i \quad (1 \leq i \leq n)}{\Delta \vdash \mathbf{f}(\dot{r}_1, \dots, \dot{r}_n) = \mathbf{f}(\dot{s}_1, \dots, \dot{s}_n)} (= \mathbf{f}) \quad \frac{\Delta \vdash \dot{r} = \dot{s}}{\Delta \vdash [\dot{a}]\dot{r} = [\dot{a}]\dot{s}} (= [\dot{\mathbf{a}}]) \\ \frac{\Delta \vdash (\dot{b} \ \dot{a}) \cdot \dot{r} = \dot{s} \quad \Delta \vdash \dot{b} \# \dot{r}}{\Delta \vdash [\dot{a}]\dot{r} = [\dot{b}]\dot{s}} (= [\dot{\mathbf{b}}]) \quad \frac{(\dot{a} \# \dot{X} \in \Delta \text{ for every } \dot{\pi}(\dot{a}) \neq \dot{\pi}'(\dot{a}))}{\Delta \vdash \dot{\pi} \cdot \dot{X} = \dot{\pi}' \cdot \dot{X}} (= \dot{\mathbf{X}}) \end{array}$$

Definition 58 repeats [UPG04, Figure 2], up to differences in presentation. A full discussion of nominal terms is in [UPG04]. $\Delta \vdash \dot{a} \# \dot{r}$ corresponds with ‘ $a \notin fa(r)$ ’ (made formal in Lemma 61), and $\Delta \vdash \dot{r} = \dot{s}$ corresponds with ‘ $r =_{\alpha} s$ ’ (see Theorems 62 and 63). Δ plays the role of permission sorts, but is part of the judgement-form.

Definition 59 Define a mapping $\llbracket \dot{\pi} \rrbracket$ from nominal permutations to permissive nominal permutations by $\llbracket \dot{\pi} \rrbracket(\iota(\dot{a})) = \iota(\dot{\pi}(\dot{a}))$ and $\llbracket \dot{\pi} \rrbracket(c) = c$ for all $c \in \mathbb{A} \setminus \text{comb}$. Define an **interpretation** $\llbracket \dot{r} \rrbracket_\Delta$ by:

$$\begin{aligned} \llbracket \dot{a} \rrbracket_\Delta &\equiv \iota(\dot{a}) & \llbracket f(\dot{r}_1, \dots, \dot{r}_n) \rrbracket_\Delta &\equiv f(\llbracket \dot{r}_1 \rrbracket_\Delta, \dots, \llbracket \dot{r}_n \rrbracket_\Delta) & \llbracket [\dot{a}] \dot{r} \rrbracket_\Delta &\equiv [\iota(\dot{a})] \llbracket \dot{r} \rrbracket_\Delta \\ \llbracket \dot{\pi} \cdot \dot{X} \rrbracket_\Delta &\equiv \llbracket \dot{\pi} \rrbracket \cdot X^S & \text{where } S &= \text{comb} \setminus \{\iota(\dot{a}) \mid \dot{a} \# \dot{X} \in \Delta\} \end{aligned}$$

Here, we make a fixed but arbitrary choice of X^S for each \dot{X} , injectively so that $\llbracket \dot{X} \rrbracket_\Delta$ and $\llbracket \dot{Y} \rrbracket_\Delta$ are always distinct.

Lemma 60 $\llbracket \dot{\pi} \rrbracket \cdot \llbracket \dot{r} \rrbracket_\Delta \equiv \llbracket \dot{\pi} \cdot \dot{r} \rrbracket_\Delta$

Lemma 61 $\iota(\dot{a}) \notin \text{fa}(\llbracket \dot{r} \rrbracket_\Delta)$ if and only if $\Delta \vdash \dot{a} \# \dot{r}$. $b \notin \text{fa}(\llbracket \dot{r} \rrbracket_\Delta)$ if $b \in \mathbb{A} \setminus \text{comb}$.

Theorem 62 $\llbracket \dot{r} \rrbracket_\Delta =_\alpha \llbracket \dot{s} \rrbracket_\Delta$ implies $\Delta \vdash \dot{r} = \dot{s}$.

Proof. Induction on the derivation of $\llbracket \dot{r} \rrbracket_\Delta =_\alpha \llbracket \dot{s} \rrbracket_\Delta$. We consider some cases:

- The case $(=_\alpha \mathbf{[b]})$. Suppose $(\iota(\dot{b}) \iota(\dot{a})) \cdot \llbracket \dot{r} \rrbracket_\Delta =_\alpha \llbracket \dot{s} \rrbracket_\Delta$ and $\iota(\dot{b}) \notin \text{fa}(\llbracket \dot{r} \rrbracket_\Delta)$. By Lemmas 60 and 61 $\llbracket (\dot{b} \dot{a}) \cdot \dot{r} \rrbracket_\Delta =_\alpha \llbracket \dot{s} \rrbracket_\Delta$ and $\Delta \vdash \dot{b} \# \dot{r}$. By inductive hypothesis $\Delta \vdash (\dot{b} \dot{a}) \cdot \dot{r} = \dot{s}$. We use $(= \mathbf{[b]})$.
- The case $(=_\alpha \mathbf{X})$. Suppose $\llbracket \dot{\pi} \rrbracket|_S = \llbracket \dot{\pi}' \rrbracket|_S$ where $S = \text{comb} \setminus \{\iota(\dot{a}) \mid \dot{a} \# \dot{X} \in \Delta\}$. ι is injective, so $\dot{a} \# \dot{X} \in \Delta$ for all \dot{a} such that $\dot{\pi}(\dot{a}) \neq \dot{\pi}'(\dot{a})$. We use $(= \mathbf{X})$.

Theorem 63 If $\Delta \vdash \dot{r} = \dot{s}$ then $\llbracket \dot{r} \rrbracket_\Delta =_\alpha \llbracket \dot{s} \rrbracket_\Delta$.

Proof. Induction on the derivation of $\Delta \vdash \dot{r} = \dot{s}$. We consider some cases:

- The case $(= \mathbf{[b]})$. Suppose $\Delta \vdash (\dot{b} \dot{a}) \cdot \dot{r} = \dot{s}$ and $\Delta \vdash \dot{b} \# \dot{r}$. By inductive hypothesis and Lemma 60, $(\dot{b} \dot{a}) \cdot \llbracket \dot{r} \rrbracket_\Delta =_\alpha \llbracket \dot{s} \rrbracket_\Delta$. By Lemma 61 $\iota(\dot{b}) \notin \text{fa}(\llbracket \dot{r} \rrbracket_\Delta)$. We use $(\stackrel{?}{=} \mathbf{[b]})$.
- The case $(= \mathbf{X})$. Recall that $\llbracket \dot{\pi} \cdot \dot{X} \rrbracket_\Delta = \llbracket \dot{\pi} \rrbracket \cdot X^S$ and $\llbracket \dot{\pi}' \cdot \dot{X} \rrbracket_\Delta = \llbracket \dot{\pi}' \rrbracket \cdot X^S$ where $S = \text{comb} \setminus \{\iota(\dot{a}) \mid \dot{a} \# \dot{X} \in \Delta\}$. Suppose $\dot{\pi}(\dot{a}) \neq \dot{\pi}'(\dot{a})$ implies $\Delta \vdash \dot{a} \# \dot{X}$. Using Lemma 61, $\llbracket \dot{\pi} \rrbracket(\iota(\dot{a})) \neq \llbracket \dot{\pi}' \rrbracket(\iota(\dot{a}))$ implies $\iota(\dot{a}) \notin S$. We use $(\stackrel{?}{=} \mathbf{X})$.

Definition 64 A **substitution** $\dot{\theta}$ is a function from nominal unknowns to nominal terms such that $\{\dot{X} \mid \dot{\theta}(\dot{X}) \neq \text{id} \cdot \dot{X}\}$ is finite. $\dot{\theta}, \dot{\theta}', \dot{\theta}'', \dots$ will range over nominal substitutions. Write id for the **identity**, mapping \dot{X} to $\text{id} \cdot \dot{X}$.

Definition 65 Define a **substitution action** on nominal terms by:

$$\dot{a}\dot{\theta} \equiv \dot{a} \quad f(\dot{r}_1, \dots, \dot{r}_n)\dot{\theta} \equiv f(\dot{r}_1\dot{\theta}, \dots, \dot{r}_n\dot{\theta}) \quad ([\dot{a}]\dot{r})\dot{\theta} \equiv [\dot{a}](\dot{r}\dot{\theta}) \quad (\dot{\pi} \cdot \dot{X})\dot{\theta} \equiv \dot{\pi} \cdot \dot{\theta}(\dot{X})$$

Definition 66 Following [UPG04, Definition 3.1], a **unification problem** $\dot{P}r$ is a finite multiset of freshnesses and equalities. A **solution** to $\dot{P}r$ is a pair $(\Delta, \dot{\theta})$ such that $\Delta \vdash \dot{a} \# \dot{r}\dot{\theta}$ for every $\dot{a} \# \dot{r} \in \dot{P}r$, and $\Delta \vdash \dot{r}\dot{\theta} = \dot{s}\dot{\theta}$ for every $\dot{r} = \dot{s} \in \dot{P}r$.

Definition 67 We extend the interpretation of Definition 59 to solutions by:

$$\llbracket (\Delta, \dot{\theta}) \rrbracket(X^S) \equiv \llbracket \dot{\theta}(X) \rrbracket_\Delta \text{ if } \text{id} \cdot X^S \equiv \llbracket X \rrbracket_\Delta \quad \llbracket (\Delta, \dot{\theta}) \rrbracket(Y^T) \equiv \text{id} \cdot Y^T \text{ otherwise}$$

Lemma 68 $\llbracket \dot{r} \rrbracket_{\Delta} \llbracket (\Delta, \dot{\theta}) \rrbracket \equiv \llbracket \dot{r}\dot{\theta} \rrbracket_{\Delta}$.

Proof. By inductions on \dot{r} . We consider the case $\dot{\pi} \cdot \dot{X}$, and reason using Lemma 60: $\llbracket (\dot{\pi} \cdot \dot{X})\dot{\theta} \rrbracket_{\Delta} \equiv \llbracket \dot{\pi} \cdot \dot{\theta}(\dot{X}) \rrbracket_{\Delta} \equiv \llbracket \dot{\pi} \rrbracket \cdot \llbracket \dot{\theta}(\dot{X}) \rrbracket_{\Delta} \equiv \llbracket \dot{\pi} \rrbracket \cdot \llbracket \dot{\theta} \rrbracket (\llbracket \dot{X} \rrbracket_{\Delta})$

Definition 69 Define $\llbracket \dot{P}r \rrbracket_{\Delta}$ by mapping $\dot{r} = \dot{s}$ to $\llbracket \dot{r} \rrbracket_{\Delta} \stackrel{?}{=} \llbracket \dot{s} \rrbracket_{\Delta}$ and mapping $\dot{a}\#\dot{r}$ to $(b \iota(\dot{a})) \cdot \llbracket \dot{r} \rrbracket_{\Delta} \stackrel{?}{=} \llbracket \dot{r} \rrbracket_{\Delta}$, for some choice of fresh b (so $b \notin fa(\llbracket \dot{r} \rrbracket_{\Delta})$); in fact, it suffices to choose some $b \notin comb$.

Lemma 70 *Suppose $b \notin fa(r)$. Then $a \notin fa(r)$ if and only if $(b a) \cdot r =_{\alpha} r$.*

Theorem 71 $(\Delta, \dot{\theta})$ solves $\dot{P}r$ if and only if $\llbracket (\Delta, \dot{\theta}) \rrbracket$ solves $\llbracket \dot{P}r \rrbracket_{\Delta}$.

Proof. We sketch the necessary reasoning.

Suppose $\Delta \vdash \dot{r}\dot{\theta} = \dot{s}\dot{\theta}$. By Lemma 68 and Theorem 63, $\llbracket \dot{r} \rrbracket_{\Delta} \llbracket (\Delta, \dot{\theta}) \rrbracket =_{\alpha} \llbracket \dot{s} \rrbracket_{\Delta} \llbracket (\Delta, \dot{\theta}) \rrbracket$. The reverse direction uses Theorem 62.

Suppose $\Delta \vdash \dot{a}\#\dot{r}\dot{\theta}$. By Lemmas 61, 70, 68, and 16 $((b \iota(\dot{a})) \cdot \llbracket \dot{r} \rrbracket_{\Delta}) \llbracket (\Delta, \dot{\theta}) \rrbracket =_{\alpha} \llbracket \dot{r} \rrbracket_{\Delta} \llbracket (\Delta, \dot{\theta}) \rrbracket$. Here, we use the b chosen fresh in Definition 69. The reverse direction uses the same results.

7 Conclusions

Nominal contrasted with permissive nominal terms. It can be problematic that in nominal terms [GM09b,GM08] we often want to enrich the freshness context, e.g. to α -convert or in solving a unification problem. Also, it is unfortunate that we cannot ‘just quotient terms by α -conversion’ since we cannot apply nominal abstract syntax [GP01] ... to nominal terms. Permissive nominal terms let us do all these things, *and* they behave more like first-order terms; we recover Lemma 12, α -equivalence is a property of terms, and the notions of unification problem and solution involve just equality, not equality-and-freshness-context.

The step from permissive nominal terms to nominal terms makes a remarkably big difference. This has implications for developing nominal techniques further; we believe that in many situations, permissive nominal terms may be easier to work with and better-behaved. By Definition 59 and Theorem 71, there is no loss in expressivity.

Permissive nominal terms do not necessarily obsolete nominal terms; if we really do want to discuss ‘an arbitrary term’, then the nominal terms unknown \dot{X} from Section 6 may be more simply and directly useful than X^{comb} . One possibly interesting extension of permissive nominal terms, is with variables for permission sorts.

Note that the unification algorithm in [UPG04] solves freshnesses concurrently with equalities. We have factored the algorithm differently, so that problems to do with free atoms (Section 4) are solved separately from problems to do with equalities (Section 5). The link is rule (I3) in Definition 37.

Concerning implementation, permissive nominal terms are implementable. Sorts are infinite sets of atoms, but differ finitely from *comb* and so may be finitely represented in an implementation.

Related work not based on nominal techniques. Permissive nominal terms resemble first- and higher-order terms more than do nominal terms, but they are a special case of neither. Higher-order patterns also have good properties of first-order terms [Mil91]. A significant difference is that the notion of unification is based on capture-avoiding substitution rather than the (permissive) nominal term capturing substitution; this gives the system a different flavour. The reader is referred to a recent paper [LV08], which goes some way to teasing out formal connections between these two approaches.

Permission sorts can be compared with the types used in [DHK00] and fully developed in [NPP07] that indicate the atoms that may appear in a term.

Hamana’s β_0 unification of λ -terms with holes adds a capturing substitution [Ham01]. Level 2 variables (which are instantiated) are annotated with level 1 variable symbols that *may* appear in them; permissive nominal terms move in this direction in the sense that permission sorts also describe which level 1 variable symbols (we call them atoms in this paper) may appear in them, though with our permission sorts there are infinitely many that may, and infinitely many that may not. The treatment of α -equivalence in Hamana’s system is not nominal (not based on permutations) and Hamana’s system does not have most general unifiers. Similarly Qu-Prolog [NR96] adds level 2 variables, but does not manage α -conversion in nominal style and also, for better or for worse, it is more ambitious in what it expresses and loses mathematical properties (unification is semi-decidable, most general unifiers need not exist).

Future work. We have investigated how the instantiation ordering interacts with the interpretation of Section 6. Most general solutions of nominal unification problems in the sense of [UPG04, Definition 3.1] do translate to most general solutions of the translated problem, in the sense of Definition 46. It is also possible to leverage permissive nominal terms to improve on the correspondence between nominal unification and Miller’s pattern unification [Mil92, Mil91] reported on in [LV08]; in [LV08] only *solvability* is considered, but permissive nominal terms’ good properties can be applied to give a particularly neat translation of *solutions* between permissive nominal unification and higher-order pattern unification. These results with full proofs will be reported in a journal version of this paper.

We hypothesise that ‘permissive’ versions of nominal rewriting, logic programming, and algebra, should be possible, following previous work using nominal terms [FG07, CU03, GM07, GM09a]. On this topic, addressing a point by Fernández, suppose $c \notin \text{comb}$; then the rewrite $X^{\text{comb}} \rightarrow X^{\text{comb}}$ cannot trigger the rewrite $U^{\text{comb} \cup \{c\}} \rightarrow U^{\text{comb} \cup \{c\}}$ because there is no π such that $\pi \cdot \text{comb} = \text{comb} \cup \{c\}$, and likewise there is no substitution σ , because of the side-condition that $\text{fa}(\sigma(X^{\text{comb}})) \subseteq \text{comb}$. It may therefore be desirable to consider only permissions sorts of the form $\text{comb} \setminus A$ for $A \subseteq \mathbb{A}$ finite. This is not a problem for unification and the proofs in this paper because they do not ‘add’ atoms to permissions sorts (we only ever need to ‘subtract’ them, as in Section 4). Alternatively, as mentioned above, we could consider permissive nominal terms syntax with variables ranging over permissions sorts.

References

- BN98. Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, Great Britain, 1998.
- CU03. J. Cheney and C. Urban. System description: Alpha-Prolog, a fresh approach to logic programming modulo alpha-equivalence. In *UNIF'03*, pages 15–19. Universidad Politécnica de Valencia, 2003.
- CU04. James Cheney and Christian Urban. Alpha-prolog: A logic programming language with names, binding and alpha-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Proc. of the 20th Int'l Conf. on Logic Programming (ICLP 2004)*, number 3132 in Lecture Notes in Computer Science, pages 269–283. Springer, 2004.
- DHK00. G. Dowek, Th. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157, 2000.
- DHK02. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Binding logic: Proofs and models. In *LPAR '02: Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 130–144, London, UK, 2002. Springer.
- FG07. Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting (journal version). *Information and Computation*, 205(6):917–965, 2007.
- GM07. Murdoch J. Gabbay and Aad Mathijssen. A formal calculus for informal equality with binding. In *Proceedings of 14th Workshop on Logic, Language and Information in Computation (WoLLIC 2007)*, volume 4576 of *Lecture Notes in Computer Science*, pages 162–176, 2007.
- GM08. Murdoch J. Gabbay and Dominic P. Mulligan. One-and-a-halfth Order Terms: Curry-Howard for Incomplete Derivations. In *Proceedings of 15th Workshop on Logic, Language and Information in Computation (WoLLIC 2008)*, volume 5110 of *Lecture Notes in Artificial Intelligence*, pages 180–194, 2008.
- GM09a. Murdoch J. Gabbay and Aad Mathijssen. Nominal (universal) algebra: equational logic with names and binders. *Journal of Logic and Computation*, 2009. Accepted subject to revision.
- GM09b. Murdoch J. Gabbay and Dominic P. Mulligan. Two-and-a-halfth Order Lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 2009. To appear.
- GP01. Murdoch J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax with Variable Binding (journal version). *Formal Aspects of Computing*, 13(3–5):341–363, 2001.
- Ham01. Makoto Hamana. A logic programming language based on binding algebras. In *TACS'01*, volume 2215 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2001.
- HHP87. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proc. 2nd Annual IEEE Symposium on Logic in Computer Science, LICS'87*, pages 194–204. IEEE Computer Society Press, 1987.
- KvOvR93. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems. *Theoretical Computer Science*, 121:279–308, 1993.
- LV08. Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *Proceedings of RTA'08*, volume 5117 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2008.
- Mil91. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
- Mil92. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.
- MN98. Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.
- NPP07. Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007.
- NR96. Peter Nickolas and Peter J. Robinson. The Qu-Prolog unification algorithm: formalisation and correctness. *Theoretical Computer Science*, 169(1):81–112, 1996.
- Pau90. Lawrence C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
- PE88. F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI (Programming Language design and Implementation)*, pages 199–208. ACM Press, 1988.
- SPG03. M. R. Shinwell, A. M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with Binders Made Simple. In *ICFP'03*, volume 38, pages 263–274. ACM Press, 2003.
- UPG04. Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

HyLMoC

A Model Checker for Hybrid Logic

Alessandro Mosca, Luca Manzoni, Daniele Codecasa

Department of Computer Science, Systems and Communication (DISCO)
University of Milan - Bicocca
alessandro.mosca@disco.unimib.it

Abstract. The current technological trend depicts a scenario in which space, and more generally the environment in which the computation takes place, represents a key aspect that must be considered in order to improve systems context awareness. Reasoning about such context can be interpreted as spatial reasoning, which means not only to be able to carry out inferences about the space itself, but also inferences about spatially related information according to a given background knowledge. Past works have shown that hybrid modal logics are a powerful and rich formalism to model qualitative spatial reasoning and reasoning on information spread into graph-like structures. In this paper we present the preliminary results we obtained in designing and implementing HyLMoC, a model checking system for hybrid modal logics. The functionalities of the model checker are based on a backend module that accepts as inputs a list of hybrid modal formulas and the specification of a labeled graph structure that is compliant with the characteristics of a Kripke model. The current implementation of the backend allows to perform different reasoning tasks with respect to those inputs: *Local* and *Global model checking*, and *All-worlds model checking*.

1 Introduction

As shown in [1], Modal Logic, originally conceived as the logic of necessity and possibility, has developed into a powerful mathematical discipline that deals with (restricted) description languages for talking about various kinds of relational structures, as spatial and temporal localizations of entities (see [2–4]). Modal logic has been widely employed with respect to time dimension and various time interval logics have been developed (as shown in [5]). On the other hand, as far as the representation of space is concerned, within the logical approach there is a vast literature of which Davis presents a good overview in [6]. Interesting theoretical research in the area of temporalized spatial logics (with an accurate analysis of decidability and complexity issues), can be found in the works of Bennett, Cohn, Wolter, and Zakharyashev; a result is the PSTL, a two-dimensional logic capable of describing topological relationships that change over time (see [7]).

Hybrid languages extend multimodal languages (characterized by a set of modal operators $MOD = \{\langle \pi_0 \rangle, [\pi_0], \dots, \langle \pi_n \rangle, [\pi_n]\}$ and a set of propositional variables $PROP = \{p_0, \dots, p_n\}$) by adding: (i) a nonempty set of propositional symbols $NOM =$

$\{i_0, \dots, i_n\}$, disjoint from $PROP$, that are called *nominals*, and (ii) a *satisfaction operator* of the form $@_i$ for each nominal $i \in NOM$. Informally, we just recall that a Kripke model for hybrid logic is a triple $(W, \{R_\pi | \pi \in MOD\}, V)$ where $(W, \{R_\pi | \pi \in MOD\})$ is a frame and V is a hybrid valuation. Semantics of hybrid formulas is defined as usual for modal logics, but (i) nominals are interpreted to be true at one and only one world of the model (their *denotation*), and (ii) given a model \mathcal{M} and a world w in the model, formulas preceded by satisfaction operators are interpreted as follows:

$$M, w \models @_i \varphi \text{ iff } M, w' \models \varphi, \text{ where } w' \text{ is the denotation of } i$$

Hybrid logics allow to express in the language itself, by means of nominals and satisfaction operators, sentences about the satisfiability of formulas; formulas preceded by statements about specific states of the model [8, 9]. Suppose we deal with a hybrid language containing the nominal ‘living room’ and the proposition ‘alarm’, the formula

$$@_{\text{livingRoom}} \text{alarm}$$

states that at the state of the model denoted as living room (think for example to a smart home environment), it is true that an alarm has been triggered. Moreover, different properties of hybrid logic revealed to be extremely useful for correlation of information that are distributed over relational structures. In particular, modal-like logics help to focus on qualitative relational aspects of networked space (e.g. containment, orientation, and reachability relations among physical entities) requiring the specification of a family of modal operators, whose meaning goes to define the underlying relational structure. With respect to some peculiar traits of hybrid logic, the combination of modal perspective and hybrid expressions enhances representation and reasoning in this context. The modal local perspective over reasoning helps in fact to easily move through the graphs following operators and their respective relations. Hybrid expressions with nominals, satisfaction operators allow not only to name specific states within these relational structures, but also to reason over equality and inequality statements about them. As an example, the formula:

$$@_i(\text{alarm} \wedge \diamond_{N_I} k) \rightarrow (@_j(\text{alarm} \wedge \diamond_{N_I} k \wedge \neg i) \rightarrow @_k \text{alarm})$$

makes use of the negated nominals to explicitly states that the states denoted as i e j are distinct (the formula says that an alarm must be triggered only if two states an alarm has been triggered at two distinct states i and j , that are both contained in k). Furthermore, we can take advantage from the characteristics that are peculiar of hybrid logic: frame definability and modularity. In fact, the hybrid machinery allows to model quite specific conditions defining the frame of reference, and this would have not been possible within plain modal logic. We can use ‘pure’ formulas of the hybrid language, i.e. formulas that do not contain propositional symbols, to constraint the meaning of the accessibility relations in the Kripke model; the following formulas express irreflexivity and antisymmetry of a generic relation π :

$$\begin{array}{ll}
(irref) & @_i \neg \langle \pi \rangle i \\
(antisym) & @_i [\pi] (\langle \pi \rangle i \rightarrow i)
\end{array}$$

In the past years, our work has been mainly focused on the exploitation of hybrid logic expressivity and inferential mechanisms in order to perform commonsense spatial reasoning in the contexts of correlation of information for pervasive and ubiquitous computing, mobile systems, context aware agents and multi-agent systems design. Here we briefly introduce some pointers to our previous papers that provide an accurate introduction of our exploitation of hybrid logic formalism to correlation and spatial reasoning. [10] introduces the approach based on commonsense spatial representation and reasoning to model context aware reasoning. This approach has been further described and discussed in [11], together with the underlying knowledge based approach to information correlation (with the related pros and cons) and the comparison with other non knowledge based approach. Moreover the last paper discuss the choice of qualitative spatial models and qualitative spatial reasoning techniques which are similar to the reasons discussed in [12]. The formal characterization of the Hybrid Commonsense Spatial Logics (HCSLs) is given in [13], together with a calculus and the discussion of deduction examples in a Smart Home context; here the relationship between our approach and other prominent logics for qualitative spatial representation and reasoning is discussed. We refer to this last work and to [11] also for the comparison with other approaches to qualitative spatial representation and reasoning (QSRR). As for the consideration of spatio-temporal events, an example of spatio-temporal correlation (but where spatial representation is simplified up to the 1D) is presented in [14]. SAMOT, the system devoted to traffic monitoring over an highway based on this approach is described in [15].

The main reason for which we started thinking to HyLMoC one year ago was the intention to develop a system for automatizing the checking of hybrid logic formulas against graph-like and network structures. The existence of an efficient model checker for hybrid logic was in our mind a way to make executable the models we defined in the above mentioned application fields, on one hand, and to improve the reasoning capabilities of the developed systems, on the other. In particular, as the introduced examples suggest, we were interested to exploit hybrid logic model checking techniques to perform the following tasks: (i) The correlation of distributed qualitative information along a local as well as a global perspective¹; and (ii) the verification of consistency properties of the structures at hand (e.g. to verify that the specification of a given structure respects a set of formal properties - reflexivity, antisymmetry, etc. - we can define by means of pure hybrid formulas). In this paper we introduced the main characteristics of a model checker for hybrid logic we start developing with these aims in mind. To the best of our knowledge, there is only another example of a computational tool for model checking that exploits hybrid logic expressivity, that is the MCLITE-MCFULL model checker,

¹ With ‘local perspective’ we mean the possibility to check if some property is satisfied at the current state of evaluation - for example, the state at which a specific device is located - and with ‘global perspective’ the possibility to check if a property is satisfied at a specified state of the model - for example, the living room of a smart home. The second task was

whose algorithms have been designed by Massimo Franceschet and implemented by Luigi Dragone [16]. We briefly discuss at the end of the paper some preliminary, and still partially incomplete, results coming from a set of comparative tests we made about the execution time of these systems.

The paper is organized as follows: The next section introduces the way HyLMoC deals with the Kripke model definition, together with the formula syntax and the grammar according to which HyLMoC parses the input formulas. The main algorithmic structure of HyLMoC is introduced in Section 4, while Section 5 is dedicated to the introduction of the core backend algorithms of the system. Experimental results are introduced in Section 6, while some concluding remarks and notes on the planned future works end the paper.

2 The HyLMoC model checker

Given a formal specification of a model and a property, Model Checking can be defined as a reasoning task of checking whether the model satisfies the property [17–19]. The model is often specified as a relational structure (e.g. a labelled graph), and the property as a formula expressed in the language of some logic. Model checking thus consist in exploring the model, jumping from one node to another according to its internal structure, in order to verify the satisfiability of the formula itself. Usually this task is performed in order to automatically verify if a model, representing hardware and software systems, meets some given specifications, e.g. safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. We refer to [20] for a comprehensive introduction of the most important model checking techniques used in the field of systems verification, and to an introduction to the automated method based on temporal logic for this tasks. The use of modal and temporal logics for performing model checking tasks is well documented in the related literature and they have conducted to very efficient symbolic model checker implementations (see, for example, [21]).

The HyLMoC system is a model checker explicitly devoted to deal with hybrid logic formulas and Kripke models. This means that the properties one can verify using HyLMoC must be expressed using the language of hybrid logic and the relational structures representing the phenomena of interest must be compliant with the formal characteristics of Kripke models. The HyLMoC system was developed to perform the following tasks:

- **[Local model checking]** Taking a model \mathcal{M} , a state w , and a formula φ , the system outputs the answer 'yes' if φ is satisfied at w in \mathcal{M} , 'no', otherwise;
- **[Global model checking]** Taking a model \mathcal{M} and a formula φ , the system outputs the answer 'yes' if φ is satisfied at all the states in \mathcal{M} , 'no', otherwise;
- **[All-Global model checking]** Taking a model \mathcal{M} and a formula φ , the system stops its computation and outputs the answer 'yes' when it finds the first state at which φ is satisfied, 'no', if there is no state satisfying φ ; and,
- **[All-States model checking]** Taking a model \mathcal{M} and a formula φ , the system outputs the answer 'yes' if there exist at least a state w in \mathcal{M} that satisfies φ , and prints

out the set w_1, w_2, \dots, w_n of all the states in \mathcal{M} that satisfy φ (where $1 \leq n \leq |W|$, and $|W|$ is the cardinality of the set of states of the model).

Consider that in standard model checking, we usually have some kind of transition system, and we want to model check formulas in runs of such transition system: either over the set of all individual runs (using linear time logics like LTL), or in the tree obtained superposing all possible runs (using tree logics like CTL). This is not what HyLMoC does: HyLMoC checks whether a formula is true (globally true or locally true) in a given model, and the model has to be taken as a finite structure. Although this is not relevant in the case of hardware and software system verification, the presence of the global task becomes increasingly interesting when model checking is applied to different domains (e.g. to the query processing of semistructured data and to common-sense spatial reasoning cited above), where, it is mandatory to check if a given property is true at each state of the model.

In what follows, we report the requirements that have guided the design and implementation of the HyLMoC model checker during all the phases of its development:

- The grammar used for the specification of the formulas must support the addition of further modal operators. For example, the addition of a ternary modal operator must be possible without changing the parser module of the model checker.
- More than one method for local and global model checking should be added without changing the language used to specify the model or the formula. In particular, different application fields (e.g. spatial information correlation task) may require the use of hybrid modal languages with different expressivity; the treatment of the formulas defined on the basis of these languages and the respective Kripke models must be preserved, and the exploitation of new optimized model checking methods for them should be allowed without breaking the compatibility with other instances of the model checker.
- The model checker must scale on multiprocessor and multicore systems, especially when the computational effort mostly depends on the throughput (i.e. number of formulas that can be checked per unit of time) instead of on the checking time of a single formula.

The first two requirements are mandatory in order to make feasible the reusing of the already written code, for example, in order to support the implementation of specific procedures for new operators in the language, or to help the prototyping of new algorithms for formula model checking. As regards to the third requirement, there are at least two distinct but interrelated reasons that forced us to work in this direction: the first one obviously refers to the goal of increasing the final performances of the model checker system, while the second one arises from the consideration that parallel multiprocessing architectures are already available on the market of desktop computers and this provides the possibility to design and implement systems for final users that exploit this new computational power.

3 The Kripke models and formulas definition

Due to well known portability and compatibility issues, the data specifying the Kripke models the HyLMoC takes as inputs are stored in XML files, formatted according to a suitable DTD. In particular, the models used by HyLMoC can be defined by using the same DTD scheme of the Franceschet and Dragone [16] system (thus providing a way to exchange model specifications among these two systems), or by using an improved version of this DTD that allows the possibility to explicitly define the formal properties of the relations characterizing the model. This resolves to be extremely useful in providing the model checker with model definitions that are formally consistent, on one hand, and guarantees a significant reduction of effort for end users in the input definition phase, on the other. In Section 4, the implemented *model completion* and *consistency check* procedures, based on the relations formal properties specification, are briefly introduced. The following paragraphs furnish examples on how a model can be defined according to the implemented DTD scheme, and introduce the grammar according to which the formulas are parsed by the system.

A **state** of a Kripke model is specified by means of a identifier and by the introduction of one or more nominals denoting it.

```
<world label="w1"/>
<nominal label="i" truth-assignment="w1"/>
```

A binary accessibility textbfrelation is identified by a name, one or more formal properties, and a list (possibly incomplete) of the pairs of worlds for which the relation holds (the specification of the properties associated to the relation will be used by the system to complete the model).

```
<modality label="pil" property="reflexivity simmetry">
  <acc-pair to-world-label="w2" from-world-label="w1"/>
</modality>
```

The DTD scheme and the related parser provides the possibility to introduce n-ary relations in model description as follow:

```
<relation label="r">
  <relation-element vector="w1 w2 w3" />
</relation>
```

As for the valuation function of the Kripke model, a new name for each textbfpropositional symbol is introduced together with the set of worlds at which it is true.

```
<prop-sym label="p" truth-assignment="w1 w2"/>
```

Figure 1 shows a complete specification of a Kripke model that is compliant with the HyLMoC DTD.

The syntax used to express formulas in HyLMoC is a lisp-like one. It supports the definition of all the modal and hybrid operators of the language introduced above, and also the addition of new operators by using a special sequence of symbols (i.e. the

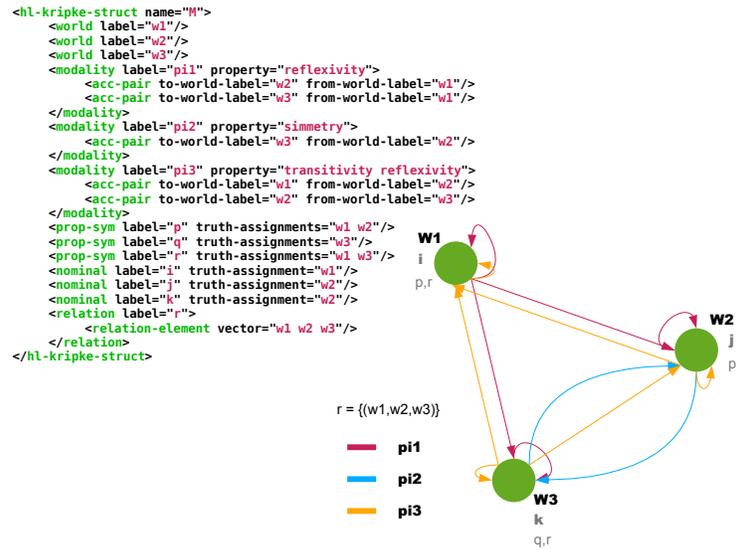


Fig. 1. An XML Kripke model specification and its diagrammatic representation.

operator name is between two question marks: $\langle ?operator_name? \rangle$). As an example, the formula $\langle r \rangle (p \rightarrow q)$ is translated as $\langle r \rangle (\text{imply } p \ q)$. The following is the grammar for the formula language the model checker is able to interpret:

```

formula ::= prop | nominal | variable
         | T | F
         | (not formula) | (imply formula formula)
         | (and formula formula) | (or formula formula)
         | (at id formula)
         | (bind variable formula)
         | (exists variable formula)
         | (forall variable formula)
         | <rel> formula | ([rel] formula)
         | <rel>- formula | ([rel]- formula)
         | (?rel? formula+)

id      ::= var | nominal
variable ::= [A-Z][a-zA-Z0-9_]*
prop    ::= [a-z0-9_][a-zA-Z0-9_]*
nominal ::= [a-z0-9_][a-zA-Z0-9_]*

```

4 The HyLMoC modules

The algorithmic architecture of HyLMoC can be reduced to four different modules: (i) The model parser; (ii) the formula parser; (iii) the backend manager; and (iv) the work unit dispatcher. In what follows we briefly introduce the characteristics of each module.

The model parser. The model parser handles two different tasks. The first one is related to the parsing of the XML specification of the model according to the introduced DTD scheme; the second one concerns the application of the structural transformations on the resulting model in order to verify that the model specification is consistent with respect to the formal properties of the contained accessibility relations. As regards to the second task, different properties can be defined for each of the relations of the Kripke model (actually, symmetry, transitivity, reflexivity and anti-reflexivity are implemented). Therefore, for each property there exists an *ad-hoc* module, we called *filter*, that performs the completion of the model with respect to its arcs (and generates warnings when inconsistencies are detected). Filters are used to manipulate the model and new filters can be added in a easily way. The presence of filters gives greater flexibility to the model checker and simplify the backends' implementation.

The formula parser. The formula parser creates a tree representation of the formula where every subtree is the representation of a well-formed formula of the implemented hybrid language. Since the semantic interpretation of possibly new operators is left to the backend, a specific node type, bringing information about the 'name and the 'arity' of the new operator, has been introduced. In order to check if the formula is well formed it is necessary to verify some properties that are contained in the model definition (for example, the first argument of an "at" must be a variable or a nominal, but not a propositional symbol). Every formula is parsed independently from the model where it will be checked; this choice minimizes the number of formulas' instances that must be present to HyLMoC when the same formula must be checked against different models.

The backend manager. The main function of the backend manager is to simplify and minimize the complexity of the backend implementation. To accomplish these goals the interface of the backend manager allows to specify the model checking task, and the underlying backends only have to check a formula at a single world of the model one at the time. Moreover, the backend manager may implement specific strategies to reduce the complexity of checking a formula; in the current implementation, the manager exploit a strategy to save resources in the case where the evaluation of a formula does not depend on the current world of evaluation. This strategy is done by checking if all the world-dependent sub-formulas have a parent that is an *at* operator.

The work unit dispatcher. The model checker can work on more than one formula or one model at the time, thus giving the possibility to exploit a certain degree of parallelism. The idea here is that it is possible to create self-contained units of work that can be executed concurrently without making the backend implementation difficult. We defined the work unit in order to contain: (i) A Kripke model; (ii) a formula; (iii) an instance of the selected backend; and (iv) a list of optional flags to specify what kind of task the model checker should be done (i.e. local, global, etc.). This kind of work unit can be executed concurrently if the backend instances do not make use of shared structures (in this case, the backend must have proper locking on these structures). Therefore, the model checker creates a queue of work units which are consumed by a pool of threads (one thread per processor available to the Java Virtual Machine).

5 Backend implementation

The formula evaluation process implemented by the HyLMoC backends follows the usual semantic conditions of the involved operators. This means that HyLMoC assumes a top-down approach in evaluating formulas, instead of the usual bottom-up. Since to check atomic formulas is the easiest task the model checker can do, the bottom-up approach is clearly the most efficient when modal propositional formulas are concerned (first check the leaves of the parse tree and identify the worlds at which the atomic sub-formulas are true, then reconstruct the entire modal formula by navigating the inverse of the respective accessibility relations). On the other hand, the same approach becomes unfeasible when binders are present, and the evaluation process should start from a variable x , as for example in $[r]\downarrow x.\langle r\rangle x$ (i.e. there no way to evaluate the variable x without any further information on the formula structure).

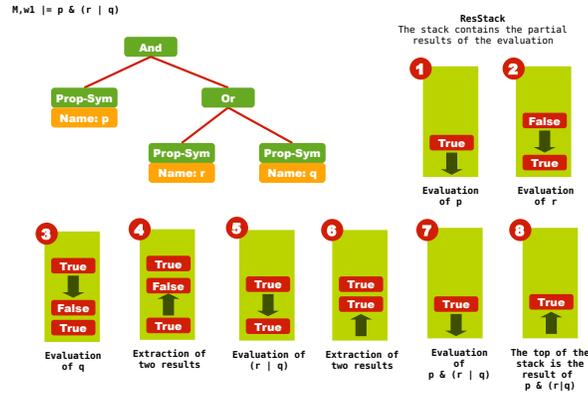


Fig. 2. An example of the use of the stack structure. The use of a pair of stacks for passing intermediate results simulate recursion on the parse tree.

Moreover, the top-down choice has supported a simple implementation of the backends, that is based on two stack data structures and a visitor pattern [22]. The first stack simply contains the intermediate results of the computation. For example, for evaluating the formula $p \wedge q$ the stack contains the truth vales of p and of q that will be used to find the truth value of the whole formula according to the semantic of the connective. The second stack contains the world where the evaluation takes place. For example, in checking if the formula $p \wedge @_i q$ is satisfiable at the world w , the stack contains w when evaluating the whole formula and p , but it contains i on top top when evaluating q . Moreover, the choice of following the semantic definition of the operators supported a simple implementation of the backends that is based on two stacks and a visitor pattern [22]. The first stack simply contains the intermediate results of the computation. For example, for evaluating the formula $p \wedge q$ the stack contains the truth vales of p and of q that will be used to find the truth value of the whole formula according to the

semantic of the connective. The second stack contains the world where the evaluation takes place. For example, in checking if the formula $p \wedge @_i q$ is satisfiable at the world w , the stack contains w when evaluating the whole formula and p , but it contains i on top top when evaluating q .

The use of the visitor pattern simplifies the maintenance of the code, as well as the creation of new backends based on the current implementation (only new operands must be defined). This design choice makes the implementation of (simple) backend a task that can be accomplished in a very low count of lines of codes (the simplest implementation which support all the modal and hybrid operators is under 300 lines). In what follows, the pseudocode implementation of the existential quantifier is introduced:

Algorithm 5.1: EXISTS($formula f, hash_table Variables$)

$$\left\{ \begin{array}{l} variable_name = first_operand[f] \\ world_list = worlds[model] \\ \mathbf{while} \ notempty[world_list] \\ \quad \left\{ \begin{array}{l} insert(variables, variable_name, dequeue[world_list]) \\ evaluate(second_operand[f], variables) \\ remove(variables, variable_name) \end{array} \right. \\ \mathbf{do} \quad \left\{ \begin{array}{l} intermediate_result = pop[results_stack] \\ \left\{ \begin{array}{l} \mathbf{if} \ intermediate_result = \mathbf{true} \\ \quad \mathbf{then} \ push[results_stack, \mathbf{true}] \\ \mathbf{return} \end{array} \right. \\ push[results_stack, \mathbf{false}] \end{array} \right. \end{array} \right.$$

The procedure changes the world where the evaluation will be done and then it starts the evaluation of the sub-formula. The procedure for the evaluation of the binder initializes the variable passed as first operand to refer to the current world.

Algorithm 5.2: BIND($formula f, hash_table Variables$)

$$\left\{ \begin{array}{l} actual_world = top[world_stack] \\ variable_name = first_operand[f] \\ insert(variables, variable_name, actual_world) \\ evaluate(second_operand[f], variables) \\ remove(variables, variable_name) \end{array} \right.$$

The pseudocode implementation of the diamond modal operator resembles the implementation of the existential quantifier because of its semantics (from a perspective close to the implementation, we consider it an existential quantification only on the neighbors of the current world):

Algorithm 5.3: POSSIBLE(*formula* *f*)

$$\left\{ \begin{array}{l} \text{actual_world} = \text{top}[\text{worlds_stack}] \\ \text{relation_name} = \text{relation}[f] \\ \text{world_neighbours_list} = \text{neighbours}[\text{actual_worlds}, \text{relation}] \\ \mathbf{while} \text{ notempty}[\text{world_neighbours_list}] \\ \quad \left\{ \begin{array}{l} \text{push}[\text{worlds_stack}, \text{dequeue}[\text{world_neighbours_list}]] \\ \text{evaluate}(f) \\ \text{pop}[\text{worlds_stack}] \\ \mathbf{do} \left\{ \begin{array}{l} \text{intermediate_result} = \text{pop}[\text{result_stack}] \\ \left\{ \begin{array}{l} \mathbf{if} \text{ intermediate_result} = \mathbf{true} \\ \quad \mathbf{then} \text{ push}[\text{result_stack}, \mathbf{true}] \\ \quad \mathbf{return} \end{array} \right. \end{array} \right. \\ \text{push}[\text{result_stack}, \mathbf{false}] \end{array} \right. \end{array} \right.$$

5.1 Backend improvements

Actually two backends have been implemented. The first one is a simple backend, called ‘basic backend’, that can be used as a baseline for comparison with other backends. The second one, called ‘advanced backend’ includes: (i) A cache for recently verified formulas; (ii) a module that makes statistical analyses on the model; and (iii) framework for adding operations that can reorder the structure of the formula without semantic changes (for example, to normalize all the formulas and to have an increase of the hit rate in the cache). Following the algorithmic architecture introduced above, the advanced backend provides optimization strategies that aim at decreasing the effort in time of the computation. A first kind of optimization is especially devoted to increasing the HyLMoC performances every time a set of long formulas have to be checked. It consists in using a cache that stores tuples in the form (φ, w) , where φ is a formula and w is a world of the input model. The cache has been implemented in a naive way, but further work will deal explicitly with the following two improvements. The first one concerns the normalization of the formula before its insertion in the cache; this is necessary to provide matches on the basis of semantic equivalence, and not only on the basis of the syntactic one. The second improvement aims at providing a better management of the space in the cache; in order to do this we are working in adapting the model of the ‘ARC cache’ [23] to work on sets of pairs (φ, w) instead of pages of memory.

The way we can improve the efficiency of HyLMoC is not only limited to the semantic and syntax of formulas, but also on the structural characteristics of the model against which the formulas have to be checked. A further optimization deals in fact with the use of structural properties of the model in order to decide the *order* of evaluation of the branches of the `and`, `or` and `imply` nodes. This can be done by combining two parameters that are computed for every node of a parsed formula: (i) *cost*, and (ii) *probability*. Intuitively, the cost parameter is computed for every node of a parsed formula and, since every node is the root of a tree that is a well-formed formula, it can be interpreted as an estimate of the time that is needed to check the formula represented by the selected node. On the other hand, the second introduced parameter aims at representing

the probability that the result of the evaluation of the given formula at the current world is equal to ‘true’. At the first step, the parameter is computed at each terminal node of the parsed formula by using statistical measures that come from an *ad-hoc* analysis of the model at which the formula has to be evaluated (e.g. number of outgoing relations per node, distribution of true propositions over the nodes). As a second step, the parameter is computed also in correspondence to each of the internal nodes of the tree, by combining this with the value obtained for the child nodes. The exploitation of both these parameters is essentially directed to perform syntactical transformations of a formula that have to be checked in order to obtain a semantically equivalent formula that can be checked in less time and with a lower computational effort.

6 Experimental results

In this section we present some preliminary results we obtained on the performances of HyLMoC once equipped with the ‘basic’ backend. The tests have been performed on a AMD Athlon 64 x2 5200+ (2.6GHz), with 4096 MB of RAM, running Ubuntu 8.04 and the JVM version 1.6.0.07. The design of an experimental campaign for a system like HyLMoC has not been immediate. On one hand, up to our knowledge there exists no available benchmark for hybrid logic model checking (consider that even if the MCLITE-MCFULL algorithms are described in publications, there is no accessible experimental results measuring their performances). On the other hand, we considered the choice of confining the tests to the formulas and models of a real-world project as too much restrictive and, in some sense, insufficient to provide the data variability we needed to test the system.

In order to design and partially automatize the experimental campaign, two ad-hoc algorithms generating random formulas and Kripke models have been implemented. The random formulas generator provides sets of formulas of growing complexity on the basis of the following fourth parameters: (i) formula depth; (ii) number of propositional symbols; (iii) number of modal operators; and (iv) number of nominals. A fifth parameter has been introduced just to fix the cardinality of the set of formulas one need to generate. The depth of a formula is understood here as the depth of the generated parse tree and, as such, it strictly depends on the introduced formula syntax (e.g. the formula `(not p4)` has depth 2 and 0 nested operators, while the formula

```
(and([r1] (<r3>-w82)) (forallY0 (atY0 (existsY1 (atY1w11))))),
```

has depth 6 and 4 nested operators). The formula generator algorithm is flexible enough to support user in generating sets of formulas with bounded complexity according to the input parameters. Furthermore, the whole campaign has benefited from the combination of the formula random generator with an similar algorithm for the Kripke models specification (the details about the XML-based syntax for the Kripke model specification have been introduced in Section ??). The algorithm takes as input the following parameters: (i) number of worlds; (ii) number of accessibility relations; and (iii) number of propositional symbols (this last parameter is necessary in order to set up the valuation function that is part of the model).

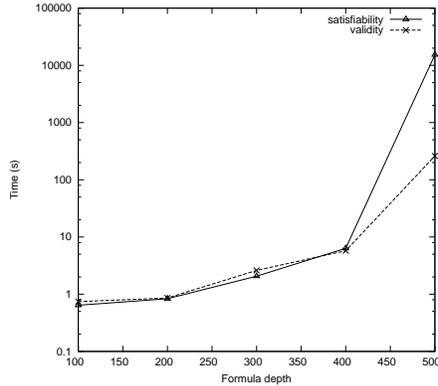


Fig. 3. ALL-GLOBAL and GLOBAL tasks with formulas of growing depth.

In particular, the curves in Figure 3 shows the performances of HyLMoC in performing the all-global and the global (blue line) tasks introduced in Section 2 with formulas of growing complexity. The structure against which the formulas have been checked is a Kripke model of 100 worlds, and 5 accessibility relations. On the other hand, the following experiments aim at verifying the performances of HyLMoC when the depth of the formulas is fixed. In particular, we used this kind of setting to compare the computational cost of different model checking tasks: Figure 5 and Figure 4 are about the results we obtained by testing HyLMoC on the all-global and global tasks respectively), while Figure 6 is about the all-worlds task.

All the present results have been produced by using batches of 50 formulas for each test. Moreover, due to the characteristic of Java libraries used in the implementation of HyLMoC, the process of scanning of graph nodes is not deterministic, and it may produce some unexpected result in terms of final observed performances of the system (e.g. the amount of time that is needed for checking if a formula containing existential quantifiers is satisfied at a state in model significantly depends on the order the graph nodes are explored). In order to reduce the undesirable effects of this indeterministic feature of the system, each test has been repeated 10 times and only the resulting average value has been recorded. The last set of results we present here is about a comparison between our basic backend and the MCLITE algorithm by Franceschet. At the present moment, MCLITE (and a still unstable version of MCFULL) has been implemented in HyLMoC and it is available as a further backend of the system. We already planned to produce a systematically comparison among these algorithms, and the results we introduce here must be considered as the first step in this direction. They concern the modal fragment of the full hybrid language and the MCLITE system, that is the algorithm devoted to deal with this fragment. With modal fragment we mean the language made by the hybrid apparatus of nominals and satisfaction operators, plus the diamond and box modal operators. The graphics show that the bottom-up approach of MCLITE work better than our top-down approach, especially when large models are considered. As one would

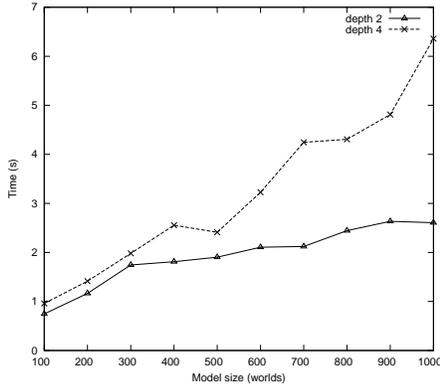


Fig. 4. GLOBAL

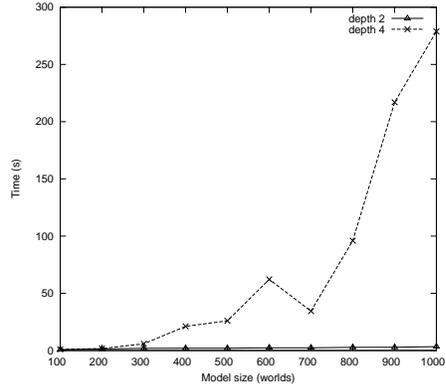


Fig. 5. ALL-GLOBAL

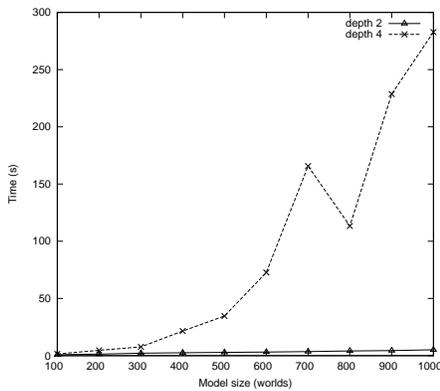


Fig. 6. ALL-WORLDS

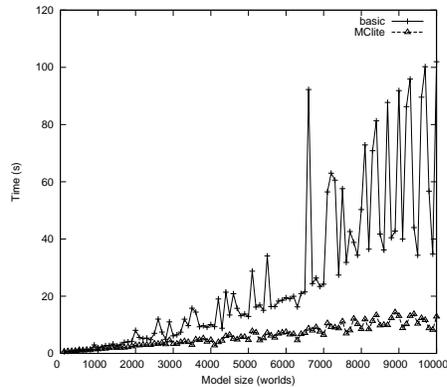


Fig. 7. ALL-GLOBAL, Formula Depth = 6, Language $\mathcal{H}_1(@, \langle \pi \rangle)$

expect, the performances of the top-down approach of HyLMoC are much more influenced by the order of visiting of the states than the bottom-up approach. In fact, starting from the leaf nodes of the formula parse trees (i.e. from the contained propositional symbols), and following backward the semantics conditions of the modal operators, the bottom-up approach is able to identify a model for a formula, if it exists, in a time span that is significantly minor than that needed by the top-down approach. Intuitively, the top-down approach is blind with respect to the possibility of finding a model for a formula (in a top-down approach every state of the model could be a good candidate for the final satisfiability of a formula containing modal operators), while the bottom-up approach resolves to be a “goal directed” process, a fact that produces an appreciable saving of computational resources.

7 Concluding remarks

Even if the present literature on model checking is truly vast, and a number of excellent computational tools and algorithms have been proposed [24], the implementation of model checkers for hybrid logic still remains a quite unexplored field of research. In this paper we introduced the main characteristics of a new model checker, called HyLMoC, that is able to deal with the expressivity of hybrid multimodal logic. The reasons why we started designing and implementing HyLMoC can be traced back to previous works on spatial reasoning and correlation of information we briefly introduced in the introduction of the paper. On the other hand, the first experimental results we obtained on HyLMoC, and the relevance the networks and graph-like structures have in the present scientific research, suggest to proceed further with its development and to try to improve its performances. In particular, the results confirm that the performances of HyLMoC are negatively affected much more by the increasing of the formula complexity than by the increasing of model dimension, and this justifies the attention we are paying in developing an advanced version of the backend. At the present moment, we are working on performing a set of tests explicitly devoted to compare the performances of the basic and the advanced backends. As for the future developments, we plan to extend our work along the following lines: (i) To provide a significant number of translations of XPath and XQuery queries into the hybrid language in order to obtain a measure of the efficiency of HyLMoC with respect to dedicated query/answering engines; (ii) to apply HyLMoC to RDF-XML graphs and compare our model checking technique with SPARQL query/answering (we believe that this study could be of some interest for the Semantic Web community); and (iii) to define a new (not yet provided in the hybrid logic literature) suite of benchmarks for hybrid logic model checking that could be useful to compare existing algorithms, as well as to develop new ones.

References

1. Blackburn, P., de Rijke, M., Venema, Y.: *Modal Logic*. Cambridge University Press (2000)
2. Prior, A.: *Past, Present and Future*. Oxford: Clarendon Press (1967)
3. Rescher, N., Urquhart, A.: *Temporal Logic*. Springer-Verlag, New York (1971)
4. Benthem, J.V.: *The logic of time*. Reidel, Dordrecht (1983)
5. Goranko, V., Montanari, A., Sciavicco, G.: A road map of interval temporal logics and duration calculi. *Journal of Applied Non-Classical Logics* **14**(1-2) (2004) 9–54
6. Davis, E.: *Representations of commonsense knowledge*. Morgan Kaufman, Los Altos CA, July, 1990, 515 pages (1990)
7. Bennett, B., Cohn, A.G., Wolter, F., Zakharyashev, M.: Multi-dimensional modal logic as a framework for spatio-temporal reasoning. *Applied Intelligence* **17** (2002) 2002
8. Blackburn, P., Seligman, J.: What are hybrid languages? CLAUS-Report 83, Universitt des Saarlandes, Saarbrcken (November 1996)
9. Blackburn, P., Seligman, J.: Hybrid languages. *Journal of Logic, Language and Information* **4** (1995) 251–272
10. Bandini, S., Mosca, A., Palmonari, M.: Commonsense spatial reasoning for context-aware pervasive systems. In: *Location- and Context-Awareness, First International Workshop, LoCA 2005*. Volume 3479 of LNCS., Springer-Verlag (2005) 180–188

11. Palmonari, M., Bandini, S.: Context-Aware Applications Enhanced with Commonsense Spatial Reasoning. Lecture Notes in Geoinformation and Cartography. In: Map-based Mobile Services. Springer (2008) 105–124
12. Cohn, A.G., Hazarika, S.M.: Qualitative spatial representation and reasoning: An overview. *Fundamenta Informaticae* **46**(1-2) (2001) 1–29
13. Bandini, S., Mosca, A., Palmonari, M.: Commonsense spatial reasoning for information correlation in pervasive computing. *Applied Artificial Intelligence* **21**(4&5) (2007) 405–425
14. Bandini, S., Mosca, A., Palmonari, M.: Intelligent alarm correlation and abductive reasoning. *Logic Journal of the IGPL* **14**(2) (2006) 347–362
15. Bandini, S., Bogni, D., Manzoni, S., Mosca, A.: St-modal logic to correlate traffic alarms on italian highways: project overview and example installations. In: IEA/AIE'2005: Proceedings of the 18th international conference on Innovations in Applied Artificial Intelligence, London, UK, Springer-Verlag (2005) 819–828
16. Franceschet, M., de Rijke, M.: Model checking hybrid logics (with an application to semistructured data). *J. Applied Logic* **4**(3) (2006) 279–304
17. Clarke, E.M., Grumberg, O., Peled, D., eds.: *Model Checking*. MIT Press (2000)
18. Clarke, E.M., Schlingloff, B.H.: *Model checking*. In Robinson, J.A., Voronkov, A., eds.: *Handbook of Automated Reasoning*. Elsevier and MIT Press (2001) 1635–1790
19. Clarke, E.M.: The birth of model checking. [24] 1–26
20. Emerson, E.: The beginning of model checking: A personal perspective. (2008) 27–45
21. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (2002) 359–364
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley Professional (January 1995)
23. Megiddo, N., Modha, D.S.: Arc: A self-tuning, low overhead replacement cache. In: FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, USENIX Association (2003) 115–130
24. Grumberg, O., Veith, H., eds.: 25 Years of Model Checking - History, Achievements, Perspectives. In Grumberg, O., Veith, H., eds.: 25 Years of Model Checking. Volume 5000 of *Lecture Notes in Computer Science*., Springer (2008)

A certification of Lagrange’s theorem with the proof assistant \mathcal{A} etnaNova/Referee ^{*}

Domenico Cantone, Salvatore Cristofaro, and Marianna Nicolosi Asmundo

Dipartimento di Matematica e Informatica, Università di Catania
Viale A. Doria 6, I-95125 Catania, Italy
e-mail: {cantone|cristofaro|nicolosi}@dmi.unict.it

Abstract. We report on the computerized verification of Lagrange’s theorem, carried out with the proof assistant \mathcal{A} etnaNova/Referee. The scenario starts with the basic definitions in group theory, such as the notions of subgroups and right cosets. Then, the proof of Lagrange’s theorem is formalized following the same approach present in most algebra textbooks.

The proof assistant \mathcal{A} etnaNova/Referee is grounded on a variant of Zermelo-Fraenkel set theory, designed for the automatic verification of proofs. Referee is provided with a powerful inferential and defining mechanism based on set theory, together with a notation very close to that commonly used in mathematics. We also compare our set theoretic approach in the formalization of elementary group theory, and in particular of Lagrange’s theorem, with those followed in other, more widespread, proof assistants.

Key words: automated proof verification, Lagrange’s theorem, set theory.

*In memory of Jacob (Jack) T. Schwartz
[January 9, 1930 - March 2, 2009]*

1 Introduction

Proof assistants aim at the mechanical verification, or certification, of the correctness of fully formalized theorem and program correctness proofs [25]. Their approach is different from that of automated theorem provers, as they are provided with inference mechanisms to check proofs already engineered by humans, rather than proof search procedures and heuristics.

Fully formalizing a mathematical proof and submitting it to a verifier results harder than just writing it by pencil and paper in the usual semi-formal style, since the user must develop the proof from scratch, specifying all details and including derivation steps, that are usually omitted in the semi-formal approach. However, once a formal proof has been checked by a proof assistant, it becomes

^{*} Work partially supported by MIUR project “*Large-scale development of certified mathematical proofs*” n. 2006012773.

certified *proofware* that can be reused in other proofs. Moreover, the great level of detail characterizing computerized proofs allows sometimes to point out certain aspects of the manipulated objects that in semi-formal proofs are usually hidden.

In this paper we present a formalization of Lagrange's theorem for algebraic group theory, which states that the order of every subgroup S of any finite group G divides the order of G (see [21] and Section 3). It turns out that our formalization closely parallels the semi-formal presentations of most algebra textbooks, such as, for instance [21]. After introducing the basic definitions of group identity and of the inverse operation, we prove some of their algebraic properties. Subsequently, the notions of subgroup and right coset of a subgroup are formalized and some related properties are proved as well. Finally, using such results, Lagrange's theorem is proved.

The verifier we used to check the above outlined proof scenario is *ÆtnaNova/Referee* [18], which is grounded on a variant of Zermelo-Fraenkel set theory designed for the automatic verification of proofs.¹ *Referee* comprises a powerful inferential and defining mechanism based on set theory. Scenarios are written in a set theoretic language very close to that one commonly uses in mathematics. The current implementation of *Referee* has been developed together with a challenging proof scenario, not yet complete, that starting from the barest rudiments of set theory aims at proving Cauchy's Integral theorem for analytic functions.

The adoption of set theory to formalize mathematical notions and to reason on them makes *Referee* different from most of the proof assistants currently in use, usually based on a logic calculus on which mathematical theories (such as set theory itself) are defined. The verifiers *HOL* [13] and *Isabelle* [17], for instance, are based on higher-order logic, *Mizar* [15] on first-order logic and set theory, and the proof assistant *Coq* [2] on a variant of the lambda calculus called the Calculus of Inductive Constructions.

Several researchers working on the development of the proof systems mentioned above have been interested in the formalization of group theory. In [1], group theory has been formalized from scratch up to the proof of Lagrange's theorem. *Mizar*'s library contains a number of articles treating several results in group theory [16]. The certification of the first Sylow's theorem with the verifier *Isabelle HOL* has been presented in [14], whereas [12] provides a formalization of finite group theory including Sylow's theorems verified by the *Coq* proof assistant.

Such efforts are partly motivated by the need of producing proofware for a generic theory, such as group theory is, so as it can be reused in the context of more complex or specific theories. Additionally, group theory includes theorems with long and complex semi-formal proof (as for instance Feit-Thompson theorem, whose original proof is 274 pages long [8]), that constitute real challenges for proof verifiers.

¹ The interested reader can access the system by asking for an account to the administrator of *Referee*: <Eugenio Omodeo> eomodeo@units.it.

Thus, our contribution besides the primary goal of providing a formalization of group theory, and in particular of Lagrange's theorem, to be reused in other Referee proof scenarios, has the additional goal of comparing our approach with the ones pursued in other, more widespread, proof assistants.

2 **ÆtnaNova/Referee**

ÆtnaNova/Referee is an automatic tool that can be used online to check the correctness of mathematical proofs. Fragments of mathematics to be checked for correctness are formalized in a set theoretic language, very close to the everyday standard mathematical language, put in scripts, and submitted to the verifier. If the verification task succeeds, the formal scenario is certified as valid proofware, that may be reused in the context of other proof scenarios. Otherwise it is rejected and a suitable diagnostic file is returned to the user.

Currently, *Referee* is a working prototype written in SETL2, a high-level programming language based on set primitives [23]. *Referee* is being extensively tested through the development of several proof scenarios in mathematics, as the ones illustrated in [19, 6] concerning such foundational notions as pairs, maps, ordinals, inductive sets, the set of real numbers, and so on. We cite also some proof scenarios in computer science, as the ones described in [18, 20] concerning the correctness of a bisimulation algorithm and of the Davis-Putnam procedure, respectively. All proofware checked by *Referee* is stored in a common scenario ready to be reused when needed in the development of other proofs.

The specification language and deductive apparatus of the verifier are grounded on a version of Zermelo-Fraenkel set theory which employs, together with the usual set theoretic operators, constructs designed to assist in the formal development of proofs. The reason behind this choice is to take advantage of the great expressive power of set theory, that, taking sets as primitive objects is in fact a proper ground on which to construct the whole mathematics. Even notions such that of integer numbers, considered primitive by other proof assistants, are reduced to sets, following von Neumann's approach.

Set theory can also be used to represent, by means of suitable operations, various constructs of symbolic languages, such as logic and programming languages. Such characteristic has shown to be very useful in the development of proof scenarios in computer science. For instance, in [20], classical results on the representation of propositional formulae in CNF by means of sets of clauses, have been employed to check the correctness of the Davis-Putnam decision procedure.

In this section we illustrate a few features of the *Referee* verifier, specifically those which have been employed within the proof scenario of our interest. For a more detailed account on *Referee* and on its definitional and inferential mechanisms, the reader is referred to [18, 19] and to the online book draft [22].

2.1 **Formalization and definition tools**

The language used to formalize definitions and proofs to be given in input to the verifier contains the usual set theoretic operators. Among them we mention the

binary operations of union \cup , intersection \cap , and set difference \setminus . The singleton operator $X \mapsto \{X\}$ is also available to define nested sets. Unordered lists can be defined as $\{X_1, \dots, X_n\} =_{Def} \{X_1\} \cup \dots \cup \{X_n\}$. Ordered pairs are definable in several ways. The definition used in our scenario, due to J.T. Schwartz, is illustrated in Fig. 1 (cf. Def 1). Ordered tuples are plain extensions of ordered pairs. Some useful operations derived from the basic set theoretic operations are: X with $Y := X \cup \{Y\}$ (yields a set whose elements are those of X plus the element Y), X less $Y := X \setminus \{Y\}$ (removing Y from X), and $\text{next}(X) := X$ with X . By repeated applications of the latter operation, one can build the natural numbers *à la* von Neumann, starting from the empty set \emptyset (regarded as the natural number 0).

The selection of an element from a set X is executed by the global choice function arb , which satisfies the properties $\text{arb}(\emptyset) = \emptyset$, $\text{arb}(X) \in \text{next}(X)$ and $X \cap \text{arb}(X) = \emptyset$, for all X .

As illustrated in Fig. 1, the above mentioned operators suffice to define the notions of ordered pairs and of the extractors of their first and second components.

```

Def 1: [Ordered pair] Def([X,Y]) := {{X},{{X},{Y},Y}}
Def 2: [First component of ordered pair] car(X) := arb(arb(X))
Def 3: [Second component of ordered pair] cdr(X) := arb(arb(arb(X - {arb(X)}) -
{arb(X)}))

```

Fig. 1. Definition of ordered pairs and of their extractor functions.

Set formers are defined in the general form

$$\{e : x_0 C_0 t_0, x_1 C_1 t_1, \dots, x_n C_n t_n \mid \varphi\},$$

where each C_i is either the \in relation or the \subseteq relation, e and φ are respectively a set-term and a condition in which the x_i s may occur free.² Set formers allow one to define operations such as the general union, the powerset and the Cartesian product.

Formal definitions of maps, of single-valued maps, and of one-to-one maps can plainly be built up from the definition of ordered pairs and from the set former construct. Their definition, as formalized in the common scenario, is reported in Fig. 2 (notice that the symbol $\bullet\text{imp}$ stands for the classical logical implication \rightarrow).

² Given a denumerable collection of variables, the constant symbol \emptyset , the binary set theoretic operations \cup , \cap , \setminus , the singleton operation $\{\cdot\}$, and the relation symbols \in , $=$, and \subseteq , set-terms can be defined inductively as follows: Each variable is a set-term, and the constant \emptyset is a set-term; $e_1 \cup e_2$, $e_1 \cap e_2$, $e_1 \setminus e_2$, and $\{e_1\}$ are set-terms, for any two given set-terms e_1 and e_2 ; set formers are set-terms. Conditions are first-order set theoretic formulae that may contain set-terms and the predicates \in and \subseteq .

```

-- Next we define various familiar notions of set theory: (possibly multivalued) maps,
-- their ranges and domains, and the subclasses of single-valued and 1-1 maps.
-- After giving these definitions we build up a few small utility theories which
-- ease subsequent work with these predicates.

Def 4: [Is-a-map predicate] Is_map(F) := F = {[car(x), cdr(x)]: x in F}
Def 5: [Map domain] Domain(F) := {car(x): x in F}
Def 6: [Map range] Range(F) := {cdr(x): x in F}
Def 7: [Single-valued map predicate] Svm(F) := Is_map(F) & (FORALL x in F, y in F |
(car(x) = car(y)) •imp (x = y))
Def 8: [One-one map predicate] One_1_map(F) := Svm(F) & (FORALL x in F, y in F |
(cdr(x) = cdr(y)) •imp (x = y))

```

Fig. 2. Formal definitions related to maps.

The set theoretic axioms of extensionality, choice, infinity, and replacement are built-in in the version of set theory underlying the Referee system. This, as mentioned earlier, is also provided with a defining mechanism, called **THEORY**, which allows to define new symbols. The **THEORY** construct is also useful to modularize scenarios and for proofware reuse.

Theories defined by the **THEORY** construct can be seen as procedures of a programming language. In fact, they are provided with a list of formal parameters that are required to satisfy certain assumptions. Once a theory **T** is specified, it can be used in the scenario by the command **ENTER_THEORY T**. It is possible to use the assumptions of the current theory in the scenario by the primitive **Assump**. Once a theory has been defined, it can be applied using new parameters satisfying the assumptions of the theory. The primitive devoted to such a task is **APPLY**.

2.2 Deduction mechanisms

In Referee, proofs are developed in a “natural deduction” style. This involves the formulation of temporary assumptions that are successively discarded. In particular, theorems are proved by contradiction, namely by attempting at deriving absurd consequences from their negations.

The inferential apparatus of Referee comprises fifteen deduction rules. Among them we mention the primitives **ELEM**, **Suppose_not**, **EQUAL**, and **Use_def**. We will briefly discuss all of them later in this section.

A proof is a sequence of statements, also called inferential steps, each of which consists of three parts: a *hint* indicating which deduction rule to invoke, the sign **==>**, and the *assertion* of the statement, a well-formed formula in the set theoretic language of Referee which has to be deduced by the chosen deduction rule in its own *context*. If not otherwise specified, the context of an occurrence of a rule within a proof is constituted by all the assertions preceding it in the proof. It is possible (and sometimes needed for efficiency reasons) to reduce the context of an occurrence of a rule by labeling the assertions actually needed by the rule to carry out its derivation (see for example the reduction of context of

the occurrence of the **ELEM** primitive at line 28 of the formal proof of Theorem **trcb18** in Fig. 8, Section 4).

Among the deduction rules provided by **Referee**, **ELEM** is the most important one. It implements in an efficient way an optimized decision procedure (cf. [4, 7]) for the satisfiability problem for a variant of the multilevel syllogistic with singleton, called **MLSS**.

MLSS is an unquantified fragment of set theory consisting of a denumerable infinity of set variables, the null set constant \emptyset , the binary set operators \cup , \cap , and \setminus , the set predicates \in , $=$, and \subseteq , and the propositional connectives. The semantics of **MLSS** is based upon the von Neumann cumulative hierarchy of sets.

The satisfiability problem for **MLSS** consists in determining whether or not any given **MLSS**-formula φ is satisfiable. A first decision procedure was presented in [9]. Subsequently, it was shown that the satisfiability problem for conjunctions of ‘flat’ **MLSS**-literals of the forms $x = y$, $x \neq y$, $x \in y$, $x \notin y$, $x = y \cup z$, $x = y \setminus z$, $x = \{y\}$, to be called normalized **MLSS**-conjunctions, is NP-complete (cf. [5]); more recently, its decision procedure was optimized in [4, 7] by means of semantic tableaux.

The **Suppose_not** primitive occurs exclusively and always as a hint of the first statement of a proof. It usually has the form **Suppose_not**(**c1**, . . . , **cn**) \implies . . . where **c1**, . . . , **cn** are parameters (local to the proof), corresponding in number and position to the (universally quantified) variables of the statement of the theorem. **Suppose_not**(**c1**, . . . , **cn**) negates the statement of the theorem and instantiates its variables (now existentially quantified) with the parameters **c1**, . . . , **cn**. The assertion at the right of **Suppose_not** is allowed to be any well-formed formula logically equivalent to the negation of the statement of the theorem instantiated with the parameters **c1**, . . . , **cn**. At the end of the proof there must appear a statement of the form **Discharge** \implies **QED**.

Two further primitives are **EQUAL** and **Use_def**. The **EQUAL** deduction rule works as follows: it takes into account all the equalities of the form $s = t$, where s and t are terms which have been derived in the context K of the rule, and propagates them transitively within all terms and formulae occurring in K . **Use_def** substitutes each occurrence of a previously defined function or predicate symbol present in the assertions of its context with its actual definition. The resulting context is then used as the basis for an **ELEM** deduction.

3 A brief outline of group theory

Next we briefly outline some basic concepts of group theory used in the formalization of Lagrange’s theorem. For further details the reader is referred to textbooks as [21].

A *group* is an ordered pair (G, mult) where G is a nonempty set and **mult** is a binary function (called *multiplicative operation*) satisfying the following properties:

- (A1) *Closure*:
 $(\forall x \in G)(\forall y \in G)(\text{mult}(x, y) \in G)$

(A2) *Associativity:*

$$(\forall x \in G)(\forall y \in G)(\forall z \in G)(\text{mult}(x, \text{mult}(y, z)) = \text{mult}(\text{mult}(x, y), z))$$

(A3) *Existence of left identity and left inverse:*

$$(\exists e \in G)((\forall x \in G)(\text{mult}(e, x) = x) \wedge (\forall x \in G)(\exists y \in G)(\text{mult}(y, x) = e))$$

A *left identity* of a group (G, mult) is an element $e \in G$ such that $\text{mult}(e, x) = x$, for all $x \in G$. If e is a left identity of (G, mult) and $x, y \in G$, we say that y is a *left inverse* of x relative to e if $\text{mult}(y, x) = e$. By (A3), a left identity e exists, and every element $x \in G$ has a left inverse relative to e . It can easily be shown that the left identity of a group (G, mult) is uniquely determined. Similarly, for each element $x \in G$ there is exactly one $y \in G$ which is a left inverse of x relative to e . We call e *the identity* of (G, mult) and, for each $x \in G$, we call the (unique) left inverse of x relative to e *the inverse* of x . From now on, we denote the identity of a group (G, mult) with *identity* and the inverse of an element $x \in G$ with $\text{inverse}(x)$.

Let (G, mult) be a group and S a nonempty subset of G , then S is a *subgroup* of (G, mult) if S is closed under the operations mult and inverse .

If S is a subgroup of (G, mult) and $t \in G$, the *right coset* of S with representative t is the set $\text{RightCoset}(S, t) = \{\text{mult}(x, t) : x \in S\}$. The *set of all right cosets* of S is denoted with $\text{RightCosets}(S)$, i.e.,

$$\text{RightCosets}(S) = \{\text{RightCoset}(S, t) : t \in G\}.$$

A group (G, mult) is finite if G is a finite set. In such a case the number of elements of G is called the *order* of (G, mult) and denoted by $|G|$. (In general, given any finite set X we denote with $|X|$ the *cardinality* of X , i.e., the number of elements of X .) Trivially, if (G, mult) is finite and S is a subgroup of (G, mult) , then S is also finite. Moreover, S has also a finite number of right cosets, and $\text{RightCosets}(S) = \{\text{RightCoset}(S, t_1), \dots, \text{RightCoset}(S, t_n)\}$, where $G = \{t_1, \dots, t_n\}$. The number of right cosets of S is called the *index* of S (in G) and is denoted by $[G : S]$, i.e., $[G : S] = |\text{RightCosets}(S)|$ (see [21]).

Using the above definitions, we can formally state Lagrange's theorem as follows (see [21]):

Theorem 1 (Lagrange's theorem). *Let (G, mult) be a finite group and let S be a subgroup of (G, mult) . Then $|S| \cdot [G : S] = |G|$, thus the order of the subgroup (S, mult) divides the order of (G, mult) .*

4 A proof scenario on Lagrange's theorem

Our proof scenario of Lagrange's theorem consists of a theory, called **group**, whose input parameters, G and mult , have to satisfy the assumptions presented in Fig. 3. These are formalizations of the properties of groups (A1), (A2), and (A3), given in Section 3.

The complete proof scenario can be found at the URL:

<http://www.dmi.unict.it/~cantone/Referee/Lagrange.txt>.

It consists of 1120 lines of proofware (including comments), 76 theorems and 20 symbol definitions. Referee verifies the scenario in 6 seconds (on average).

```

THEORY group(G, mult(x,y))
-- Closure properties
(FORALL x, y | ((x in G) & (y in G)) •imp (mult(x,y) in G))
-- Associativity
(FORALL x, y, z | (((x in G) & (y in G)) & (z in G)) •imp (mult(x,mult(y,z)) =
mult(mult(x,y),z)))
-- Existence of left identity and left inverse
(EXISTS E | (E in G) & (FORALL X | (X in G) •imp (mult(E,X) = X)) & (FORALL X | (X in
G) •imp (EXISTS Y | (Y in G) & (mult(Y,X) = E))))
END group

```

Fig. 3. The assumptions of the theory `group`.

4.1 Preliminary definitions and properties

Initially, some basic notions of group theory are formally defined and introduced in the theory `group`. Among them, in particular, we mention the notions of the identity and of the inverse function of a group. Their formalizations, which make use of the set former and of the `arb` operator, are reported in Fig. 4. The formalization of the identity deserves a few comments. After introducing

```

-- An element E of the group G is an identity (of G) if mult(X,E) = X, for all X in G.
-- We define the predicate "is_identity(E)" to mean that E is an identity.
Def dg1: is_identity(E) := (FORALL X | (X in G) •imp (mult(E,X) = X))

-- The set of all identities is defined by:
Def dg2: identities := {E in G | is_identity(E)}

-- Using the choice operator "arb" we define the identity of the group G.
Def dg3: identity := arb(identities)

-- Given an element X of the groups G, an inverse of X is an element Y of G such that
-- mult(Y,X) is an identity.
-- We define the predicate "is_inverse(X,Y)" as follows:
Def dg4: is_inverse(X,Y) := is_identity(mult(Y,X))

-- The set of the inverses of X is denoted by "inverses(X)".
Def dg5: inverses(X) := {Y in G | is_inverse(X,Y)}

-- Using the choice operator "arb" we define the inverse of an element X of the group
-- G.
Def dg6: inverse(X) := arb(inverses(X))

```

Fig. 4. The basic definitions of identity and inverse.

the predicate $\text{is_identity}(e) \Leftrightarrow_{Def} (\forall x \in G)(\text{mult}(e,x) = x)$ (Def `dg1`), which provides the defining property of group identity in (G, mult) , we then define the set $\text{identities} =_{Def} \{e \in G \mid \text{is_identity}(e)\}$ (Def `dg2`), representing the collection of all the identities of (G, mult) . Finally, using the global choice operator `arb` (Def

dg3), we define the identity of a group (G, mult) , as an “arbitrary” element of the set identities (formally, $\text{identity} =_{\text{Def}} \text{arb}(\text{identities})$). The formalization of the inverse operation, described through Defs dg4-dg6 in Fig. 4, follows along similar lines.

It is then checked that the notions of identity and of inverse are correctly formalized, in the sense that identity and inverse, as defined above, satisfy some important basic properties such as the one stating that

$$\begin{aligned} \text{mult}(x, \text{identity}) &= \text{mult}(\text{identity}, x) = x \quad \text{and} \\ \text{mult}(\text{inverse}(x), x) &= \text{mult}(x, \text{inverse}(x)) = \text{identity}, \end{aligned}$$

for all $x \in G$. It is also certified that both identity and $\text{inverse}(x)$, for every $x \in G$, are uniquely determined.

Many other elementary algebraic properties of the identity and the inverse operation are formalized and proved as well. These include, for instance, the following ones:

Uniqueness of idempotent elements:

if $x \in G$ and $\text{mult}(x, x) = x$, then $x = \text{identity}$;

Double inverse:

if $x \in G$ then $\text{inverse}(\text{inverse}(x)) = x$;

Inverse of the multiplication:

if $x, y \in G$ then $\text{inverse}(\text{mult}(x, y)) = \text{mult}(\text{inverse}(y), \text{inverse}(x))$;

Cancellation law:

if $x, y, z \in G$ and $\text{mult}(x, z) = \text{mult}(y, z)$ then $x = y$.

The basic tools for the formalization of the proof of Lagrange’s theorem include, together with the notions and properties reported above, the definitions of subgroup and right coset, whose formalization is illustrated in Fig. 5 (the symbol $\bullet\text{incin}$ in Definition dg7 stands for the set inclusion \subseteq symbol).

```

-- A subgroup (of G) is a nonempty subset S of G which is closed under the operations
-- mult and inverse of G.
-- The predicate "subgroup(S)" denotes the fact that S is a subgroup (of G).
Def dg7: subgroup(S) := (S /= 0) & (S •incin G) & (FORALL x, y | ((x in S) & (y in S))
•imp (mult(x,y) in S)) & (FORALL x | ((x in S) •imp (inverse(x) in S)))

-- Given a subgroup S (of G) and an element T of G, the right coset of S with
-- representative T is the set of all elements of the form mult(X,T), where X belongs
-- to S.
Def dg8: RightCoset(S,T) := {mult(X,T) : X in S}

-- The set of all right cosets of a subgroup S.
Def dg9: RightCosets(S) := {RightCoset(S,T) : T in G}

```

Fig. 5. The definitions of subgroups and right cosets.

```

-- Now we turn to Lagrange's theorem.
-- We define explicitly a function "RCB(S)" such that, for any subgroup S, RCB(S) is a
-- map whose domain is the Cartesian product of S with the set of the right cosets
-- of S, and whose range is the group G.
-- The definition of RCB(S) follows:
Def dg10: RCB(S) := {[[X,R],mult(X,arb(R))] : X in S, R in RightCosets(S)}

```

Fig. 6. The formal definition of set $\text{RCB}(S)$.

4.2 The main result

Let us consider the following three facts:

- (a) for every subgroup S of (G, mult) , the set $\text{RightCosets}(S)$ of the right cosets of S is a partition of G ,
- (b) every right coset of a subgroup S of (G, mult) is equipollent to S itself,³ and
- (c) if \mathcal{P} is a partition of a set X and any set belonging to \mathcal{P} is equipollent to a (fixed) set Y , then the Cartesian product $Y \times \mathcal{P}$ is equipollent to X .

One could first show separately that the facts (a), (b), and (c) hold, and then prove Lagrange's theorem as a plain consequence of them. In fact, from (a), (b), and (c), it follows immediately that (i) $S \times \text{RightCosets}(S)$ is equipollent to G . Additionally, we have (ii) $|A \times B| = |A| \cdot |B|$. Thus, if (G, mult) is a finite group, the sets G , S , and $\text{RightCosets}(S)$ are all finite, and hence, by (i), $|S \times \text{RightCosets}(S)| = |G|$. Therefore, by (ii), we have that $|S| \cdot |\text{RightCosets}(S)| = |G|$, i.e., $|S| \cdot [G : S] = |G|$ which is just the conclusion of Lagrange's theorem.

The approach just outlined is basically the same adopted in most algebra textbooks (see for instance [21]). In our proof scenario we followed it as well, but with some slight differences which made the overall proof process a little bit simpler. Specifically, we showed that (i) can be proved without using (b) and (c). To do so, it is convenient to define the following set

$$\text{RCB}(S) =_{\text{Def}} \{[[x, R], \text{mult}(x, \text{arb}(R))] \mid x \in S, R \in \text{RightCosets}(S)\},$$

whose formal definition within Referee is reported in Fig. 6. It turns out that $\text{RCB}(S)$ is in fact a one-to-one map whose domain is the Cartesian product $S \times \text{RightCosets}(S)$ and whose range is G , thus implying immediately (i).

To begin with, one first derives that $\text{RCB}(S)$ is a single-valued map and that $\text{Domain}(\text{RCB}(S)) = S \times \text{RightCosets}(S)$, from the very definition of $\text{RCB}(S)$.

The proof that $\text{Range}(\text{RCB}(S)) \subseteq G$ holds can be formalized starting from the first assumption in the theory `group` (see Fig. 4) and from the following two facts:

- $S \subseteq G$, and

³ We recall that, by definition, a set A is equipollent to a set B if and only if there exists a one-to-one correspondence f from A onto B .

- $\emptyset \subset R \subseteq G$ and $\text{arb}(R) \in R$, for all $R \in \text{RightCosets}(S)$,

by introducing only a few simple intermediate lemmas.

The most complicated part consists in proving formally that $G \subseteq \text{Range}(\text{RCB}(S))$ (this, together with the fact that $\text{Range}(\text{RCB}(S)) \subseteq G$, implies that $\text{Range}(\text{RCB}(S)) = G$) and that $\text{RCB}(S)$ is one-to-one. These results have been established by developing the formal proof of a consistent number of properties of subgroups and of right cosets of a subgroup.

A complete description of a formal proof. We devote the last part of the present section to the description of a formal proof that $\text{RCB}(S)$ is a one-to-one map. In view of the definition of one-to-one maps given in Fig. 2, such proof consists in proving that

- $\text{RCB}(S)$ is a single-valued map, and that
- if $A, B \in \text{RCB}(S)$ and $\text{cdr}(A) = \text{cdr}(B)$, then $A = B$.

We focus on the second statement, which is formalized by **Theorem trcb18** reported in Fig. 8, whose semi-formal proof is given next.

Semi-formal proof of Theorem trcb18. We must show that if $A, B \in \text{RCB}(S)$ and $\text{cdr}(A) = \text{cdr}(B)$, then $A = B$. Assume, by way of contradiction, that this is not the case, i.e., that $A, B \in \text{RCB}(S)$, $\text{cdr}(A) = \text{cdr}(B)$, but $A \neq B$.

By the definition of $\text{RCB}(S)$ we have that

$$A = [[x_1, r_1], \text{mult}(x_1, \text{arb}(r_1))] \text{ and } B = [[x_2, r_2], \text{mult}(x_2, \text{arb}(r_2))],$$

where $x_1, x_2 \in S$ and $r_1, r_2 \in \text{RightCosets}(S)$. Therefore, since $\text{cdr}(A) = \text{cdr}(B)$, it follows that

$$\text{mult}(x_1, \text{arb}(r_1)) = \text{mult}(x_2, \text{arb}(r_2)). \quad (1)$$

At this point we use the following two properties, whose proofs have been formalized in the scenario as well (see Fig. 7):

Property 1: If $R \in \text{RightCosets}(S)$, $y \in R$, and $x \in S$, then $\text{mult}(x, y) \in R$.

Property 2: If $X, Y \in \text{RightCosets}(S)$ and $X \cap Y \neq \emptyset$, then $X = Y$.⁴

In fact, from Property 1 it follows that

$$\text{mult}(x_1, \text{arb}(r_1)) \in r_1 \quad \text{and} \quad \text{mult}(x_2, \text{arb}(r_2)) \in r_2$$

and hence, using (1) above, we get that $r_1 \cap r_2 \neq \emptyset$. From this, using Property 2, we obtain that $r_1 = r_2$ and therefore $\text{arb}(r_1) = \text{arb}(r_2)$. Thus, by putting $x = \text{arb}(r_1) = \text{arb}(r_2)$, by (1) again we have that $\text{mult}(x_1, x) = \text{mult}(x_2, x)$ and hence, by the cancellation law, we get $x_1 = x_2$. Summing up, we have shown that $x_1 = x_2$ and $r_1 = r_2$, which imply that $A = B$, contradicting the assumption that $A \neq B$. Hence $\text{RCB}(S)$ must be one-to-one.

⁴ The proof of these two properties can be found in most common textbooks of group theory. For instance, property (2) corresponds to Theorem 2.9 of [21].

```

Theorem trcb1: (subgroup(S) & (R in RightCosets(S)) & (X in S) & (Y in R)) •imp
(mult(X,Y) in R).

Theorem tp2: (subgroup(S) & (X in RightCosets(S)) & (Y in RightCosets(S)) & ((X * Y) /=
0)) •imp (X = Y).

```

Fig. 7. The formalized statements of Property 1 and Property 2, as Theorems trcb1 and tp2, respectively.

```

1. Theorem trcb18: (subgroup(S) & (A in RCB(S)) & (B in RCB(S)) & (cdr(A) = cdr(B))) •imp
(A = B). Proof:
2. Suppose_not(s,a,b) ==> subgroup(s) & (a in RCB(s)) & (b in RCB(s)) & (cdr(a) = cdr(b)) &
(a /= b)
3. Use_def(RCB) ==> Stat0: (a in {[[X,R],mult(X,arb(R))]: X in s, R in RightCosets(s)})
4. (x1,r1)-->Stat0 ==> (x1 in s) & (r1 in RightCosets(s)) & (a = [[x1,r1],mult(x1,arb(r1))])
5. Use_def(subgroup) ==> Staty1: (x1 in G)
6. ELEM ==> Statl1: a = [[x1,r1],mult(x1,arb(r1))]
7. Use_def(RCB) ==> Stat1: (b in {[[X,R],mult(X,arb(R))]: X in s, R in RightCosets(s)})
8. (x2,r2)-->Stat1 ==> (x2 in s) & (r2 in RightCosets(s)) & (b = [[x2,r2],mult(x2,arb(r2))])
9. Use_def(subgroup) ==> Staty2: (x2 in G)
10. ELEM ==> Statl2: b = [[x2,r2],mult(x2,arb(r2))]
11. EQUAL ==> cdr([[x1,r1],mult(x1,arb(r1))]) = cdr([[x2,r2],mult(x2,arb(r2))])
12. ([[x1,r1],mult(x1,arb(r1))])-->T8 ==> cdr([[x1,r1],mult(x1,arb(r1))]) =
mult(x1,arb(r1))
13. ([[x2,r2],mult(x2,arb(r2))])-->T8 ==> cdr([[x2,r2],mult(x2,arb(r2))]) =
mult(x2,arb(r2))
14. EQUAL ==> Statz1: mult(x1,arb(r1)) = mult(x2,arb(r2))
15. (s,r1)-->Ttp1 ==> (r1 /= 0)
16. ELEM ==> arb(r1) in r1
17. (s,r1,x1,arb(r1))-->Ttrcb1 ==> mult(x1,arb(r1)) in r1
18. (s,r2)-->Ttp1 ==> (r2 /= 0)
19. ELEM ==> arb(r2) in r2
20. (s,r2,x2,arb(r2))-->Ttrcb1 ==> mult(x2,arb(r2)) in r2
21. EQUAL ==> (mult(x2,arb(r2)) in r2) & (mult(x2,arb(r2)) in r1)
22. ELEM ==> (r1 * r2) /= 0
23. (s,r1,r2)-->Ttp2 ==> Statx1: r2 = r1
24. EQUAL(Statx1,Statz1) ==> Staty4: mult(x1,arb(r1)) = mult(x2,arb(r1))
25. (s,r1,arb(r1))-->Ttp3 ==> Staty3: arb(r1) in G
26. (x1,x2,arb(r1))-->Ttcnc ==> Staty5: ((x1 in G) & (x2 in G) & (arb(r1) in G) &
(mult(x1,arb(r1)) = mult(x2,arb(r1)))) •imp (x1 = x2)
28. (Staty1,Staty2,Staty3,Staty4,Staty5)ELEM ==> Statx2: x2 = x1
29. EQUAL(Statx1,Statx2) ==> Statl3: [[x2,r2],mult(x2,arb(r2))] = [[x1,r1],mult(x1,arb(r1))]
30. EQUAL(Statl1,Statl2,Statl3) ==> a = b
31. ELEM ==> false;
32. Discharge ==> QED

```

Fig. 8. The formalized version of the theorem that $RCB(S)$ is one-to-one.

Description of the formal proof (Fig. 8). Following the proof style of the verifier Referee, we proceed by contradiction, i.e., we negate the theorem, thus assuming that $A, B \in RCB(S)$, $cdr(A) = cdr(B)$, but $A \neq B$. This first step is formalized in line 2 of Fig. 8, through the `Suppose_not` primitive with the parameters s , a , and b . More specifically, the directive `Suppose_not(s,a,b)` negates the statement of the theorem and instantiates its variables S , A , and B with the

constants \mathbf{s} , \mathbf{a} , and \mathbf{b} , respectively. The resulting formula is the conjunction of the formulae $\text{subgroup}(\mathbf{s})$, $(\mathbf{a} \text{ in } \text{RCB}(\mathbf{s}))$, $(\mathbf{b} \text{ in } \text{RCB}(\mathbf{s}))$, $(\text{cdr}(\mathbf{a}) = \text{cdr}(\mathbf{b}))$, and $(\mathbf{a} \neq \mathbf{b})$, appearing immediately to the right of the \Rightarrow sign.

The informal proof continues as follows:

“By the definition of $\text{RCB}(S)$ we have that $A = [[x_1, r_1], \text{mult}(x_1, \text{arb}(r_1))]$ and $B = [[x_2, r_2], \text{mult}(x_2, \text{arb}(r_2))]$ hold, where $x_1, x_2 \in S$ and $r_1, r_2 \in \text{RightCosets}(S)$ ”.

These inferences have been formalized in lines 3–6 and in lines 7–10 of the proof, respectively.

In line 3, the `Use_def` primitive is used to replace the function symbol `RCB` with its formal definition in all the assertions of its context. In this case, the context of `Use_def(RCB)` consists in the formula $\text{subgroup}(\mathbf{s}) \ \& \ (\mathbf{a} \text{ in } \text{RCB}(\mathbf{s})) \ \& \ (\mathbf{b} \text{ in } \text{RCB}(\mathbf{s})) \ \& \ (\text{cdr}(\mathbf{a}) = \text{cdr}(\mathbf{b})) \ \& \ (\mathbf{a} \neq \mathbf{b})$, namely the only formula introduced in the formal proof before the occurrence of `Use_def(RCB)`. Plainly, `Use_def(RCB)` yields the formula

$$(\mathbf{a} \text{ in } \{[[X, R], \text{mult}(X, \text{arb}(R))]\} : X \text{ in } \mathbf{s}, R \text{ in } \text{RightCosets}(\mathbf{s})),$$

labeled `Stat0`. Such a label is used in the subsequent step of the proof to reference the same formula. In fact, in line 4, we Skolemize the formula `Stat0` by instantiating the existentially quantified variables `X` and `R` with the local constants `x1` and `r1`, respectively, obtaining the assertion immediately to the right of \Rightarrow . This type of inference is called *statement citation*.

The inference in line 5 is another application of the `Use_def` primitive, this time the argument being the predicate name `subgroup`. By substituting it with its definition in the preceding formulae of the proof, we infer $(\mathbf{x1} \text{ in } \mathbf{G})$. The derivation proceeds as follows. By substituting the symbol `subgroup` with its definition in the statement `subgroup(s)` (cf. line 2), we infer immediately $\mathbf{s} \subseteq \mathbf{G}$ and hence, from the assertion $(\mathbf{x1} \text{ in } \mathbf{s})$ (cf. line 4), we obtain $(\mathbf{x1} \text{ in } \mathbf{G})$. Such inferences are performed internally by the system through an implicit call to the `ELEM` primitive, from the directive `Use_def`. Notice that the resulting formula (i.e., $(\mathbf{x1} \text{ in } \mathbf{G})$) has been labeled by `Staty1`. As in line 3, the label has the purpose of referencing the formula in subsequent steps of the proof, this time in line 28. In this case, however, the usage will be different. In fact the label `Staty1` is used now for restricting the context of the primitive `ELEM`. More precisely, since the `ELEM` rule is supplied with the list of labels $(\text{Staty1}, \text{Staty2}, \text{Staty3}, \text{Staty4}, \text{Staty5})$, its context is automatically restricted to exactly those formulae of the proof to which the selected labels have been attached. The possibility of using labels in this way is a utility that `Referee` provides to speedup the overall process of proof checking.

In line 6 we find the first explicit use of the `ELEM` primitive. The inferred formula, namely the one to the right of \Rightarrow , is a consequence of the statement of line 4, belonging to the actual context of `ELEM`.

Lines 7–10 can be commented much as lines 3–6 in Fig. 8.

The semi-formal proof then continues as follows:

“Therefore, since $\text{cdr}(A) = \text{cdr}(B)$, it follows that

$$\text{mult}(x_1, \text{arb}(r_1)) = \text{mult}(x_2, \text{arb}(r_2)).”$$

Such inference is formalized in lines 11–14. In particular, in line 11 the primitive EQUAL is used to propagate transitively all equalities in its context. In this specific case, since the equalities $\mathbf{a} = [[x_1, r_1], \text{mult}(x_1, \text{arb}(r_1))]$ and $\mathbf{b} = [[x_2, r_2], \text{mult}(x_2, \text{arb}(r_2))]$ (cf. lines 6 and 10, respectively) have been already derived and since the formula $(\text{cdr}(\mathbf{a}) = \text{cdr}(\mathbf{b}))$ (line 2) occurs in the context of the rule EQUAL, its application yields $(\text{cdr}([[x_1, r_1], \text{mult}(x_1, \text{arb}(r_1))])) = \text{cdr}(\mathbf{b})$ and then $\text{cdr}([[x_1, r_1], \text{mult}(x_1, \text{arb}(r_1))]) = \text{cdr}([[x_2, r_2], \text{mult}(x_2, \text{arb}(r_2))])$, which is exactly the formula appearing to the right of EQUAL. Lines 12 and 13 make reference to a basic theorem on pairs (T8) through a *theorem citation*. This inference step consists in instantiating the free variables of such theorem, X and Y , with the constants $[x_1, r_1]$ and $\text{mult}(x_1, \text{arb}(r_1))$ at line 12 and $[x_2, r_2]$ and $\text{mult}(x_2, \text{arb}(r_2))$ at line 13.

The semi-formal proof terminates as follows:

“At this point we use the following two properties, whose proofs have been formalized in the scenario as well (see Fig. 7):

P 1: If $R \in \text{RightCosets}(S)$, $y \in R$, and $x \in S$, then $\text{mult}(x, y) \in R$.

P 2: If $X, Y \in \text{RightCosets}(S)$ and $X \cap Y \neq \emptyset$, then $X = Y$.

In fact, from Property 1 it follows that

$$\text{mult}(x_1, \text{arb}(r_1)) \in r_1 \quad \text{and} \quad \text{mult}(x_2, \text{arb}(r_2)) \in r_2$$

and hence, using (1) above, we get that $r_1 \cap r_2 \neq \emptyset$. From this, using Property 2, we obtain that $r_1 = r_2$ and therefore $\text{arb}(r_1) = \text{arb}(r_2)$. Thus, putting $x = \text{arb}(r_1) = \text{arb}(r_2)$, by (1) we have that $\text{mult}(x_1, x) = \text{mult}(x_2, x)$ and hence, by the cancellation law, we get $x_1 = x_2$. Summing up, we have shown that $x_1 = x_2$ and $r_1 = r_2$ which imply that $A = B$; this contradicts the assumption that $A \neq B$. Hence $\text{RCB}(S)$ must be one-to-one.”

Such inferences are formalized starting from line 15 in Fig. 8. The deduction steps are of the same types of the ones reviewed so far and therefore we omit any further comment. However, a particular care deserves the following derivation. At line 23, a theorem citation instantiates the free variables S , X , and Y of Theorem tp2 (see Fig. 7) with the constants \mathbf{s} , $\mathbf{r1}$, and $\mathbf{r2}$, respectively, thus yielding the formula

$$\begin{aligned} &(\text{subgroup}(\mathbf{s}) \ \& \ (\mathbf{r1} \ \text{in} \ \text{RightCosets}(\mathbf{s})) \ \& \ (\mathbf{r2} \ \text{in} \ \text{RightCosets}(\mathbf{s}))) \\ &\quad \& \ ((\mathbf{r1} \ * \ \mathbf{r2}) \ / = \ 0) \ \bullet \text{imp} \ (\mathbf{r1} = \mathbf{r2}). \end{aligned}$$

Since $\text{subgroup}(\mathbf{s})$, $(\mathbf{r1} \ \text{in} \ \text{RightCosets}(\mathbf{s}))$, and $(\mathbf{r2} \ \text{in} \ \text{RightCosets}(\mathbf{s}))$ are conjuncts of previously derived formulae (lines 2, 4, and 8, respectively), and since the formula $(\mathbf{r1} \ * \ \mathbf{r2}) \ / = \ 0$ has been derived in the preceding step (cf.

line 22), a further implicit application of `ELEM` allows one to infer the formula `(r1 = r2)`, which appears exactly to the right of the `==>` symbol.

Finally, concerning the inferences in lines 31 and 32, we notice that the former one, i.e., `ELEM ==> false;`, simply states that a contradiction has been reached because of the two complementary formulae `(a /= b)` and `(a = b)`, appearing in the context of `ELEM` (lines 2 and 30, respectively), whereas the latter one, namely `Discharge ==> QED`, just concludes the proof of the theorem: since a contradiction has been reached we can discharge the negated statement of the theorem (assumed by the `Suppose_not` assumption at line 2), thus concluding the proof of `Theorem trcb18`.

5 Related work

Group theory plays a central rôle in mathematics. It is therefore not surprising that its basic notions and results have been formalized and checked by several verifiers. Here we mention only a few such contributions.

In [14], the proof assistant Isabelle has been employed to certify the first Sylow’s theorem, which in some sense is the inverse of Lagrange’s theorem. In fact, while Lagrange’s theorem says that “*the order of a subgroup of a finite group divides the order of the group*”, the first Sylow’s theorem states that “*if p is a prime number and p^α divides the order of a group, then there exists a subgroup of order p^α* ”.

Isabelle is a generic interactive prover based on a minimal higher-order logic (called ‘pure’ [25]) that can be instantiated to more specific provers for arbitrary logics by means of theories definition. Isabelle’s theories contain type declarations, constants, definitions, and rules. Some of them are used as independent provers, such as Isabelle-ZF (Zermelo-Fraenkel set theory) and Isabelle-HOL (higher order logic). However, Isabelle’s notion of theory is different from Referee’s one, discussed in Section 2.1, where a theory is a mechanism to define new symbols and to modularize proofs. The formalization of the first Sylow’s theorem (a quite big experiment involving 278 theorems) has been carried out on the Isabelle-HOL prover requiring the development of two theories: Group and Sylow. The first one contains the basic declarations, definitions and theorems about groups, in particular it also contains Lagrange’s theorem. Groups are defined by means of the constant `Group` which is a typed set of quadruples `(G, f, inv, e)` satisfying the axioms of group theory. Our approach (see Fig. 3) is more concise. In fact, we derive the notions of identity and of inverse from the pair `(G, mult)` and from the axioms of group theory.

A wide collection of results in group theory, including Lagrange’s theorem as well, has been formally proved by the Mizar proof checker and stored in the Mizar Mathematical Library (MML) [16].

The Mizar proof assistant is based on first-order logic enriched with the axioms of the Tarski-Grothendieck set theory [24] (a slight extension of Zermelo-Fraenkel set theory). Like Referee, the Mizar system is based on untyped set theory. It allows, however, to define and use syntactical types such as, in particular,

the type `Group`. Groups, are defined starting from a radix type, called `LoopStr`, by describing the carrier, the operation `mult`, which is the binary operation of the structure, and the constant `zero`, the identity element, and by adding the axioms of group theory. A proof of Lagrange's theorem, of Sylow's theorem, and of many other important results in group theory are all documented in the MML library. The article of the MML library devoted to Lagrange's theorem is 3050 lines long and is processed by Mizar in 12 seconds.

Other interesting results concerning the formalization of group theory have been obtained by the proof assistant Coq, which is based on an intuitionistic type theory called Calculus of Inductive Constructs. In [12], a formalization of finite group theory is presented, including Sylow's theorems and the Cauchy-Frobenius lemma. Finite groups are defined in [12] as a structure where both the identity and the inverse function are explicitly given together with the carrier and the multiplication operation. In particular, they are not obtained as a subclass of generic groups (as one may expect) but as a subclass of finite types. The reason behind that is that the authors decided to reuse the libraries developed for the formal proof of the Four Colors theorem [11].

Currently there is an in-progress, long-term effort on the formalization of the Feit-Thompson theorem for finite groups [10]. Since its semi-formal proof is very long, its mechanization requires much care on the choice of the proof engineering techniques to be employed.

6 Conclusions and future work

We have presented a formalization of the proof of Lagrange's theorem for algebraic groups, which has been certified by the proof assistant `Referee`. The scenario, containing basic notions and properties of group theory too, has been carried out through the construction of a theory, called `Group`. The theory `Group` simply takes in input two sets `G` and `mult`, and contains, as assumptions, the axioms of group theory formalized along the lines of classical algebra textbooks such as [21]. This minimal set of premises allows one to derive the notions of identity and of inverse function of the group, together with several other basic properties. It turns out that our formalization of group theory is more concise than other formalizations present in literature and checked by other proof assistants [14, 12], in which the identity and the inverse functions of a group are given as primitive elements.

Our scenario includes further definitions, such as those of subgroup and right coset, together with the formalization and proof of their main properties. Finally, Lagrange's theorem is fully formalized, along with its proof.

The results reported in this paper gave us the opportunity of confronting the `Referee` system with other, more used and well-known proof assistants. We found out that formal proofs developed with `Referee` are, in average, more concise and closer to the common mathematical language used in semi-formal proofs. This is particularly evident in the case of the Mizar verifier (closer to `Referee` in that it

is based on classical logic and set theory), as shown by the experimental results reported in Sections 4 and 5.

On the other hand, the current version of Referee still needs to be refined in its defining and inference mechanisms. For instance, the THEORY construct could be improved by introducing suitable theory inheritance mechanisms. In addition, strategies of proof development other than that of reduction *ab absurdum* should be allowed (as, for instance, the proof assistant Mizar does), so to let users write their own proof scenarios in the manner they prefer.

Another important issue is the usability of the system. The current graphical interface of Referee is quite basic. We plan to make it more user-friendly, providing it with features helping the user in preparing the scenarios.

Concerning our scenario on algebraic groups, we plan to enrich it by verifying other fundamental results in group theory such as Sylow's theorems. Our interest in algebraic structures includes also Boolean algebras. In fact, we are currently working at a formalization of the correctness proof of the Boolean unification algorithm *à la* Büttner-Simonis (cf. [3]).

References

1. R.D. Arthan. Mathematical Case Studies: Some Group Theory. <http://www.lemma-one.com/ProofPower/examples/wrk068.pdf>
2. Y. Bertot, P. Casteran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, 2004, ISBN 3-540-20854-2.
3. W. Büttner, H. Simonis. Embedding Boolean expressions into logic programming, *Journal of Symbolic Computation*, vol. 4(2), pp. 191–205, 1987.
4. D. Cantone. A fast saturation strategy for set-theoretic tableaux. In D. Galmiche, ed., *Proc. International Conference on Theorem Proving with Analytic Tableaux and Related Methods - TABLEAUX '97*, vol. 1227 of LNAI, pp. 122–137, Berlin, May 13–16 1997. Springer-Verlag.
5. D. Cantone, E. G. Omodeo, A. Policriti. The automation of syllogistic. II. Optimization and complexity issues. *Journal of Automated Reasoning*, vol. 6(2), pp. 173–187, 1990.
6. D. Cantone, E. G. Omodeo, J. T. Schwartz, P. Ursino. Notes from the Logbook of a Proof-Checker's Project. In N. Dershowitz, editor, *Proc. International Symposium on Verification: Theory and Practice*, vol. 2772 of LNCS, pp. 182–207, Berlin, 2003. Springer-Verlag.
7. 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*, vol. 1761 of LNAI, pp. 127–137. Springer-Verlag, 2000.
8. W. Feit, J.G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics* vol. 13(3), pp. 775–1029 (1963).
9. A. Ferro, E. G. Omodeo, J. T. Schwartz. Decision procedures for elementary sublanguages of set theory. I. Multi-level syllogistic and some extensions. *Comm. Pure Applied Math.*, vol. 33(5), pp. 599–608, 1980.
10. F. Garillot. Mechanized foundations of finite group theory. <http://research.microsoft.com/en-us/um/redmond/about/collaboration/phd-summerschool/2008summerschool/francoisgarillot-handout.pdf>

11. G. Gonthier. A computer-checked proof of the Four Colour Theorem. <http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf>
12. G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, L. Thery. A modular formalization of Finite Group Theory. In K. Schneider and J. Brandt, editors, *Proc. of Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007*, vol. 4732 of LNCS, pp. 86–101. Springer-Verlag, 2007.
13. M. J. C. Gordon, T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
14. F. Kammüller, L. C. Paulson. A Formal Proof of Sylow’s Theorem. *J. Autom. Reason.*, vol. 23, n. 3, 1999, pp. 235–264, Kluwer Academic Publishers, Hingham, MA, USA.
15. <http://mizar.org/>
16. <http://mizar.org/library/>
17. T. Nipkow, L. C. Paulson, M. Wenzel. Isabelle/HOL, A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer Verlag, 2002.
18. E. G. Omodeo, D. Cantone, A. Policriti, J. T. Schwartz. A computerized Referee. In O. Stock and M. Schaerf (Eds.), *Aiello Festschrift, LNAI*, vol. 4155, Springer, pp. 114–136, 2006.
19. E. G. Omodeo, J. T. Schwartz. A ‘Theory’ Mechanism for a Proof-Verifier Based on First-Order Set Theory. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pp. 214–230, 2002, Springer-Verlag, London, UK.
20. E. G. Omodeo, A. I. Tomescu. Using *ÆtnaNova* to formally prove that the Davis-Putnam satisfiability test is correct. *Le Matematiche*, vol. LXIII (2008), Fasc. I, pp. 85–105.
21. J. J. Rotman. *An introduction to the theory of groups*. 4th ed, Springer-Verlag, Graduate Texts in Mathematics 148, 1995.
22. J. T. Schwartz, D. Cantone, E. G. Omodeo. *Computational logic and set theory*. <http://www.settheory.com/intro.html>.
23. J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, E. Schonberg. *Programming with Sets: An Introduction to SETL*. In *Texts and Monographs in Computer Science Series*, Springer-Verlag, New York.
24. <http://mizar.org/JFM/pdf/tarski.pdf>
25. F. Wiedijk (Ed.). *The Seventeen Provers of the World*. LNCS, vol. 3600, 2006.

Incremental Learning from Positive Examples

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

Università degli Studi di Bari, Dipartimento di Informatica, 70125 Bari, Italy
{gbombini,ndm,esposito,ferilli}@di.uniba.it

Abstract. Classical supervised learning techniques are generally based on an inductive mechanism able to generalise a model from a set of positive examples, assuring its consistency with respect to a set of negative examples. In case of learning from positive evidence only, the problem of over-generalisation comes into account. This paper proposes a general technique for incremental multi-class learning from positive examples only, which has been embedded in the learning system INTHELEX. The idea is to incrementally suppose the positive evidence for a class to be a negative evidence for all other classes until the environment explicitly declares the contrary.

An application of the proposed technique to the agent learning domain has been provided. The proposed framework has been used to simulate an agent learning and revising in an incremental way a logical model of a task by imitating skilled agents. In particular, demonstrations are incrementally received and used as training examples while the agent interacts in a stochastic environment. The experimental results prove the validity of the proposed approach on this application domain.

Key words: Inductive Logic Programming, Incremental Learning

1 Introduction

The goal of inductive Machine Learning systems is to discover a description of a target concept from a set of observations provided by the expert and possibly a background knowledge. Inductive learning can be cast as a search problem in the space of all possible hypotheses [1]. In the supervised setting a background knowledge and a set of positive and negative examples are exploited by the learning system in order to find a hypothesis explaining all positive examples and none of the negative ones. In particular, negative examples are used to bias the search in the space of all possible hypotheses.

However, in many cases it may be necessary to learn from positive examples only because no negative example is available, as in the case of a child learning to speak, or when an animal or an agent learns how to act by observing and/or by imitating other agents. In this scenario the learning system could produce an overly general hypothesis, such as “everything is true”, due to the absence of contradicting evidence.

This paper proposes a general technique for incremental multi-class learning from positive examples only. The idea is to incrementally suppose the positive

evidence for the class c_i to be a negative evidence for all other classes c_j ($i \neq j$), until the environment explicitly declares the opposite. In this paper, a first order logic description language to express possible hypotheses is used. First-order logic is a powerful symbolic representation language able to represent the intrinsic relations among the domain. It is also understandable by humans.

An application of the proposed technique will be provided, where an agent must learn and revise in an incremental way a logical representation of a task by imitating a skilled agent. In particular, the proposed technique has been used to learn a policy adopting a relational language to describe both the demonstrations and the policy. The relational description language makes the adoption of the background knowledge very simple. The agent actively interacts with the teacher by deciding which action to execute next and requesting demonstrations.

2 Related work

From an Inductive Logic Programming (ILP) [2] point of view, one of the most popular approaches to the problem of learning from positive examples only, is to exploit some form of clustering, adopting symbolic generalisations in the given representation language (as used by Kirsten and Wrobel [3]). Given some elements of the space of all possible observations, the learner tries to generalise over various subsets of the observed data yielding proper rules. Each subset, known as a “cluster”, is made up of examples that are deemed near to each other. However, this is an unsupervised batch approach that does not fit the requirements of learning by imitation.

Other approaches to positive only learning include systems that adopt the Bayes theorem [4], like Progol [5]. In [4] the problem has been studied in a Bayesian framework where the distribution of the examples is assumed to be known. The solution provided is to generate random examples of the target concept by sampling the type range of each attribute, and then using these examples as negative. After selecting the positive instance to be generalised, the most specific clause within the language constraints that entails it is constructed. In order to find a concept description, a search in the hypothesis space of clauses that are more general than such as most specific clause is performed. In the induction phase, a measure to evaluate how well a clause explains all the examples is employed.

MERLIN 2.0 [6] uses Hidden Markov Models to facilitate learning from positive examples in situations where predicate invention is required, such as learning predicates that classify or generate sequences. MERLIN induces Hidden Markov Models from positive sequences only to determine when to invent new predicates. Due to the use of this induction technique, incremental learning is possible.

The following systems apply a positive-only learning approach to unlabelled data exploiting positive observations. In [7], Yu et al. proposed a technique based on Support Vector Machines (SVM), named PEBL, to classify Web pages given positive and unlabelled pages. This approach uses heuristics to identify unlabelled examples that are likely to be negative and applies a standard two-class

(positive and negative) learning method to these examples and the positive examples. Essentially their method consists of two steps: the first identifies a set of reliable negative documents from the unlabelled set (strong negative documents) i.e. those documents that do not contain any feature of the positive data, and the second builds a classifier using SVMs. The number of positive examples affects the performances of PEBL. Indeed, when the positive data is small, the results are often very poor.

In [8] Lee and Liu defined the problem of learning from a set of positive and unlabelled examples as a two-class learning problem with noisy negative examples. They heuristically identify some reliable negative examples in the unlabelled set using logistic regression. They assign weights to unlabeled examples and use weighted logistic regression to learn the classes of “positive” and “negative”. The proportion of noise in the labeled examples can be approximated by the proportion of noise in a random sample from the overall set of examples. The unlabeled examples are interpreted as weighted negative examples.

3 Learning from positive examples only

The learning problem for ILP can be formally defined:

Given: A finite set of clauses \mathcal{B} (*background knowledge*) and sets of clauses E^+ and E^- (positive and negative *examples*).

Find: A theory Σ (a finite set of clauses), such that $\Sigma \cup \mathcal{B}$ is *correct* with respect to E^+ and E^- , i.e.: a) $\Sigma \cup \mathcal{B}$ is *complete* with respect to E^+ : $\Sigma \cup \mathcal{B} \models E^+$; and, b) $\Sigma \cup \mathcal{B}$ is *consistent* with respect to E^- : $\Sigma \cup \mathcal{B} \not\models E^-$.

Given the formula $\Sigma \cup \mathcal{B} \models E^+$, deriving E^+ from $\Sigma \cup \mathcal{B}$ is *deduction*, and deriving Σ from \mathcal{B} and E^+ is *induction*. In the simplest model, \mathcal{B} is supposed to be empty and the deductive inference rule \models corresponds to θ -*subsumption* between clauses.

Common ILP approaches produce general clauses from positive examples and restrict their coverage by the help of negative examples. In domains where only positive examples are available, the learning systems may not be able to learn the concepts correctly because of the lack of restrictions induced by negative examples. Indeed, learning from positive examples only may cause over-generalisation that leads to inconsistent resulting hypotheses. In a multiple concept learning setting, given a positive example for a class, it is possible to exploit the other classes to assume negative examples. In particular, the given example is regarded as negative for the other classes.

Incremental learning is necessary when incomplete information is available at the time of initial theory generation. The learned theory is checked to be valid on any new available example. In case of failure, a revision process is activated on it in order to restore the completeness and the consistency properties. When a positive example which is inconsistent with the hypothesis biased drawn from previous negative examples is processed, all such negative examples are excluded from the learning process. When a candidate negative example is considered, the

Algorithm 1 IIL

```

1: errors  $\leftarrow$  0
2:  $T \leftarrow \emptyset$ 
3:  $M \leftarrow \emptyset$ 
4: loop
5:    $(c, e) \leftarrow$  get a new positive example
6:    $N \leftarrow \{(\neg c_i, e) \mid c_i \in C \setminus \{c\}\}$ 
7:    $s \leftarrow$  classify( $e, T$ )
8:   if  $s \neq c$  then
9:     errors++
10:    for all  $m \in M$  do
11:      if  $m$  is negative for the class  $c$  with the same body as  $e$  then
12:         $M \leftarrow M \setminus \{m\}$ 
13:      generalize( $T, c \leftarrow e, M$ )
14:     $M \leftarrow M \cup \{c \leftarrow e\}$ 
15:    for all  $c_i \in N$  do
16:      if  $M$  does not contain the positive example  $c_i \leftarrow e$  then
17:        if  $s = c_i$  then
18:          errors++
19:          specialize( $T, \neg c_i \leftarrow e, M$ )
20:         $M \leftarrow M \cup \{c_i \leftarrow e\}$ 

```

system checks whether a previous positive example has already been processed for the same class. Hence, it is necessary that the system is capable of taking into account the examples in an incremental way and learn simultaneously several concepts, possibly related to each other.

Algorithm 1 reports the tuning procedure to refine the current theory. M represents the set of all positive and negative examples already processed, E is the current example to be examined, and T is the theory learned from the examples in M which are incrementally provided by an expert. It starts with an empty theory and an empty historical memory. Whenever a new example is taken into account, it is also stored in the historical memory. The training examples are exploited by the system to modify incorrect hypotheses according to a data-driven strategy.

Each evidence e belongs to a specific learning class c . When a positive example $E = \{c, e\}$ is provided to the learning system, it is considered as a negative example for all the other classes $c_i \in C$ with $c_i \neq c$ that must be learned. N represents the set of negative examples assumed. For each positive example, the system verifies the soundness of T trying to classify it using the learned model. If there is a misclassification, a theory revision is needed and all the negative examples for class c having the same description of e are excluded from historical memory.

A generalisation process on the current theory produces a revised theory obtained in one of the following ways:

1. generalise one of the definitions pertaining to the theory that relate to the concept of the example, by removing conditions. This must be done by ensuring that the revised theory covers the new example and is consistent with the all negative examples previous examined;
2. add a new definition to the theory to explain the example;
3. add the example as a positive exception.

All the candidate negative examples $(\neg c_j, e) \in N$ for which a positive example $c_j \leftarrow e$ has already been processed are ignored. Indeed, since that evidence was already taken into account as positive in the learning process, it can not be assumed as a negative example for c_j .

When a negative example is covered, a specialization process outputs a revised theory by performing one of the following actions:

1. add positive conditions to each definition that explains the example. These conditions must be able to characterize all positive examples already considered and exclude the current negative example;
2. add a negative condition to the faulty definitions to differentiate the current negative example from positive examples previously considered;
3. add the negative example as an negative exception.

In case of concepts that have many positive examples and few negative ones that distinguish them, the system learns different definitions. The negative examples are used for specialise the definitions that apply to specific cases. At the end of the learning process, T represents the learned model.

In order to explain the general procedure previously described, we present two simple examples belonging the blocks world domain. This domain consists of 4 blocks (a, b, c and d), where blocks can be on the floor (denoted by f) or can be stacked on each other. Predicate $on(X, Y)$ denotes that block X is on block Y , and that X and Y belong to the same stack. In this domain, the available actions are of the kind $move(X, Y)$, with $X \in a, b, c, d$ and $Y \in a, b, c, d, f$, $X \neq Y$, to be interpreted as the action to put X on Y . Predicate $block(X)$ means that X represents a block, $clear(X)$ means that X is clear, $floor(X)$ means that X represents the floor. Predicate $goal_on(X, Y)$ means that the goal is achieve when block X is on block Y .

The following set of literals represents a possible state in this domain:
 $\{goal_on(a, b), clear(c), on(c, a), on(a, f), on(d, b), clear(d), on(b, f), block(a), block(b), block(c), block(d), floor(f)\}$

In this state, it is correct to move both block c and block d on the floor to achieve the goal. If the positive example $move(c, f)$ with the previous observation is presented to the system, it adds the following definition to the theory to explain the example: $move(A, B) :- clear(A), block(A), floor(B), on(D, B), block(D), goal_on(D, E), block(E), on(F, E), clear(F), block(F), on(E, B), on(A, D)$.

Then the system assumes $not(move(c, d))$, $not(move(d, c))$, $not(move(d, f))$ as negative examples with the same evidence as the positive example. These

examples are not explained by the currently learned theory and thus is not necessary to specialise it.

Further examples concerning the same state may be presented to the system. In particular, a possible positive example is $move(d, f)$ with the evidence previously considered. In this case, the system assumes $not(move(c, d))$, $not(move(d, c))$, $not(move(c, f))$ as negative examples.

When the system takes into account a positive example, is necessary to check whether a previous negative example (in this case $not(move(d, f))$) was previously processed and stored in the historical memory, in which case it must be removed.

Then the positive example is processed, and in this case a generalisation process is necessary on the learned clause. The negative examples $not(move(c, d))$, $not(move(d, c))$ are not covered by the theory previously generalised, and therefore there is no need of further processing to revise the learned theory.

When the system takes into account the negative example $not(move(c, f))$, in the historical memory the corresponding positive example with the same evidence is found, and then the negative example is not processed.

The only clause learned from these few examples results: $move(A, B) :- clear(A), block(A), floor(B), on(D, B), block(D), on(E, B), block(E), on(F, E), clear(F), block(F), on(A, D)$.

4 INTHELEX

The proposed algorithm, was integrated into INTHELEX (INcremental THEory Learner from EXample) [9], an incremental learning system for the induction of first-order logic theories from positive and negative examples. It can learn simultaneously several concepts, possibly related to each other. The approach used by this system may be defined as “multistrategic” since it combines different forms of reasoning in the same symbolic paradigm. It learns theories in form of sets of Datalog clauses, interpreted according to the Object Identity (OI for short) [10] assumption. Under the OI assumption, within a clause, terms (even variables) denoted with different symbols must be distinct.

INTHELEX is a fully and inherently incremental learning system. The learning phase can start by taking as input a previously generated version of the theory or an empty theory (from the first available example). When the theory is incorrect wrt an example, this is rejected and a process of theory revision starts. Algorithm 2 reports the general tuning procedure used in INTHELEX for refining the theory to be learned. INTHELEX uses two inductive refinement operators, one for generalising definitions that reject positive examples (completeness), and the other for specialising definitions that explain negative examples (consistency). In order to perform its task, the system exploits a previous theory (if any) and a *historical memory* of all the past (positive and negative) examples that led to the current theory.

The system can be provided with a background knowledge (i.e. some partial concept definitions known to be correct about the domain and hence not modifiable) in the same format as a theory rules.

Algorithm 2 tuning(E, T, M)**Input:** E : example; T : theory; M : historical memory;

```

1: Add  $E$  to  $M$ 
2: if  $E$  is a positive example not covered from  $T$  then
3:   generalize( $T, E, M$ )
4: else
5:   if  $E$  is a negative example covered by  $T$  then
6:     specialize( $T, E, M$ )

```

5 Sample application: Learning from demonstration

Learning from demonstration or *learning by imitation* [11, 12] represents a promising approach in the area of human-robot interaction. Recent research in other fields considers the imitative learning as an essential part of human development [13]. Indeed, humans and animals use imitation as a mechanism for acquiring knowledge. Imitation-based learning [14] and learning by demonstration [15] are exploited to enable an unskilled agent, the *observer*, to learn tasks by simply observing performances of a skilled agent, the *teacher* [16, 17]. This approach can be viewed as a collaborative learning based on the interaction between the human and the agent. The agent gathers information about the task in the form of perceptual inputs and action results, and estimates the latent control policy of the demonstrator. The estimated policy can then be used to control the agent's autonomous behaviour. As reported in [18], this method can reduce learning time when compared to classical exploration-based methods such as reinforcement learning (RL) [17].

Learning from demonstration is strongly related to supervised learning where the goal is to learn a policy given a fixed set of labelled data [19]. From this point of view it is possible to collect the interactions between the teacher and the observer and to use them as learning examples. Furthermore, data should be gathered in an incremental way by minimising the number of required labelled data useful to learn the given policy.

In the scenario of an agent acting in a stochastic world, the *correct actions* of the agent may be considered as positive examples in a supervised learning task.

Here, we assume that an agent A aims at learning how to act in an environment by imitating an *expert agent* E . The environment in which the agent acts is defined by a finite set of states S . For each state $s \in S$, the agent has available a finite set of actions $A(s) \subseteq \mathcal{A}$ which cause stochastic state transition. In particular, an action $a \in A(s)$ causes a transition to state s'_a when executed in state s , where \mathcal{A} is the set of all primitive actions. Given a goal, each training example $e = \{a, o\}$ belongs to a specific learning class based on the action a . For each class a theory must be learned in order to be able to predict the corresponding action on unseen observations.

Hence, each action taken from E in a given state may be considered as a positive example. Given an action-state pair (a, s) and a set $A(s)$ of possible actions that can be taken in state s , all other actions in $A(s) \setminus a$ are assumed to

be negative examples until the expert agent actually takes any of them in the same state s . Actions represent the target concepts, and the state represent the evidence. At the end of the learning process, the learned theory T represents the optimal policy.

The agent is assumed to observe a demonstrator that performs the correct sequence of actions useful to reach a given goal by starting from an initial state of the environment. During each training sequence, the agent records the observation about the environment and the corresponding action performed by the demonstrator. An observation $o \in S$ is represented as a set of ground Datalog literals. Each training example, $e = \{a, o\}$, consists of an action $a \in \mathcal{A}$ selected by the demonstrator and an observation $o \in S$. Obviously, we assume that the demonstrator uses a good policy π to achieve the goal. Hence, the aim of the agent is to learn such as hidden policy $\pi : S \rightarrow \mathcal{A}$ mapping states (observations) to actions.

In case of imitative learning, given a state s , if the teacher takes the action $a_i \in A(s)$, then the observer can assume that a_i is a positive example and all other actions $a_j \in A(s) \ 1 \leq j \neq i \leq |A|$ are negative ones. A negative example a_j for the state s is considered reliable until the demonstrator performs a_j in s . Furthermore, the process of imitative learning is naturally modeled by an agent able to modify its theory in an incremental way, where each new incoming example may give rise a theory revision process. We represent the policy as a set of logical clauses where the *head* of the clause represents the action while the *body* represents the state. In particular, a clause represents an action that may be performed in a given state.

The proposed learning framework has been applied to a pursuit domain, generally used in the field of agent learning. The experiment regards the problem of learning a policy in a domain where a predator should capture a prey. This stochastic environment consists of a 4x4 grid surrounded by a wall, with a predator and a prey inside. We assume that the predator has caught the prey if the prey lies on the same square as the predator at the end of its move. The prey moves at random, while the predator follows a *good user-defined* strategy in order to capture the prey. Both the predator and the prey can move in four directions (north, east, south and west). The action of an agent consists in moving to the square immediately adjacent in the selected direction. In case the target square is a wall, the agent stays in the same square. The two agents move alternate turns.

An observation is made up of the agent's perception about the squares surrounding it in the four directions and the square under it. The state of each square may be *empty*, *wall* or *agent*. Starting from an initial state, once captured the prey the sequence of observations does not restart by placing agents in random positions, but it continues from the positions of catch. For example, a predator agent having a wall to the west and the prey to the east, the observation is represented by the following set of literals: $\{observation(a, d1, e), observation(a, d2, e), observation(a, d3, p), observation(a, d4, w), under(a, e), direction(d1, named1), direction(d2, named2), direction(d3, named3), direction(d4, named4),$

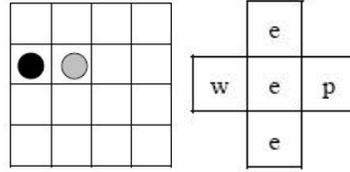


Fig. 1. A sample predator-prey domain. The black circle represents the predator and the gray circle represents the prey. The figure on the right represents the predator agent percept.

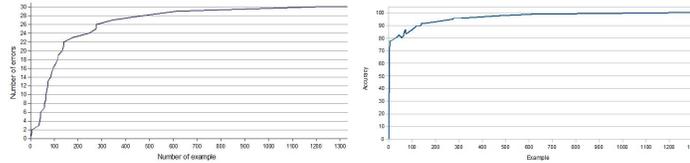


Fig. 2. Errors during the learning phase. The figure on the right represents the prediction accuracy (%) over the entire historical memory for each theory revision.

$north(named1)$, $south(named2)$, $east(named3)$, $west(named4)$, $prey(p)$, $predator(a)$, $wall(w)$, $empty(e)$ where a stays for predator agent, p for prey agent, e for empty, and w for wall.

After fixing the strategy to capture the prey, we simulated a scenario in which a predator instructs another agent to capture the prey with a minimum number of steps. Hence, given an observation, the action taken from the predator represents a positive training example, while all other possible actions are supposed to be negative examples.

We have generated 10 sequences of observations. Each sequence, containing traces of prey's captures, is made up of 322,5 positive and 967,5 negative observation-action pairs on average.

A first experiment has been performed on the whole dataset without providing the system with any background knowledge. Over all the 10 sequences the system learned a theory (policy) made up of 9,4 clauses, obtained by 14,8 generalisations and 2,2 specialisations (26,4 errors) on average. Each clause is composed of 13.5 literals on average. The system needs 0.325 seconds per example to learn the correct policy.

In order to evaluate the behaviour of the learning process, we generated a sequence made up of 1332 observation-action pairs.

Figure 2, reports on the left, the number of errors (a generalisation or a specialisation request) the agent made during the learning phase. As we can see, the number of errors grows until the system learns the correct policy (i.e., the learned classification theory). Figure 2, on the right, reports the evolution

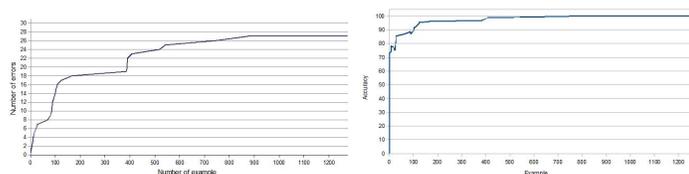


Fig. 3. On the left: errors during the learning phase. On the right: prediction accuracy (%) over the entire historical memory at each revision theory with the use background knowledge.

of prediction accuracy of the policy, learnt by the agent during the imitation process.

An example of learned rule is $\{move(A, B) :- predator(A), direction(B, D), direction(F, H), observation(A, I, J), observation(A, B, E), observation(A, F, G), south(H), north(D), wall(E), prey(G), empty(J), under(A, J).\}$

where a predator A moves toward north when at south there is the prey and at north there is a wall.

Another experiment has been performed with the system provided with a background knowledge containing several rules defining the concept of “next direction”. An example is $\{next(A, B) :- north(A), east(B).\}$. This rule denotes that north direction followed by the east direction. The learned theory abstracts the concept of direction. An example of learned rule is

$\{move(A, B) :- predator(A), direction(B, D), direction(G, F), next(D, H), next(H, F), next(F, E), next(E, D), observation(A, B, I), observation(A, G, K), wall(I), under(A, J), empty(J), prey(K).\}$

where predator A moves in direction D when it observes that in D there is a wall and opposite there is the prey K at any edge of the grid (D may be any of the four direction). This action applies to four different states.

On the same 10 sequences of the previous experiment, the system learned a theory made up of 6,4 clauses, obtained by 12,1 generalisations and 3,5 specialisations (22 errors) on average. Each clause is composed of 15.3 literals on average. The system needs 0.337 seconds per example to learn the correct policy. In order to evaluate the behavior of the learning process with the use of a background knowledge, we generated a sequence made up of 1276 observation-action pairs.

Figure 3 reports the number of errors (a generalization or a specialization request) the agent made during the learning phase and prediction accuracy of the policy.

It is interesting to note that the theory learned with the help of a background knowledge does not exploit specific directions, but rather the relationships between them. This allows an abstraction on states. Indeed, the policy learned turns out to be more compact (a lower number of clauses on average compared with the first experiment), where each clause is applicable to a subset of states.

6 Conclusion

In domains where only positive examples are available, a classical supervised learning system may not be able to learn the concepts correctly because of the lack of restrictions induced by negative examples. Indeed, negative examples are used to bias the search in the space of all possible hypotheses. In this case, the problem of over-generalisation comes into account.

A general technique for incremental multi-class learning from positive examples only has been presented. The problem of over-generalisation has been avoided assuming as negative examples the positive example for the other classes, until the environment does not explicitly declare the contrary.

The proposed technique has been implemented and tested on a dataset belonging to the field of agent learning from demonstration. The proposed framework allows to quickly, accurately and incrementally train an unskilled agent to imitate a (human or artificial) demonstrator. The results confirm the validity of the technique.

Acknowledgements

This work is partially funded by Italian Ministry of University and Scientific Research FAR Project MBLab “The Molecular Biodiversity Laboratory”.

References

1. Mitchell, T.M.: Generalization as search. *Artif. Intell.* **18**(2) (1982) 203–226
2. Lavrac, N., Dzeroski, S.: *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York (1994)
3. Kirsten, M., Wrobel, S.: Relational distance-based clustering. In: *ILP*. (1998) 261–270
4. Muggleton, S.: Learning from positive data. In: *Inductive Logic Programming Workshop*. (1996) 358–376
5. Muggleton, S.: Inverse entailment and progol. *New Generation Comput.* **13**(3&4) (1995) 245–286
6. Boström, H.: Predicate invention and learning from positive examples only. In: *ECML*. (1998) 226–237
7. Yu, H., Han, J., Chang, K.C.C.: Pebl: positive example based learning for web page classification using svm. In: *KDD*. (2002) 239–248
8. Lee, W.S., Liu, B.: Learning with positive and unlabeled examples using weighted logistic regression. In: *Proceedings of the Twentieth International Conference on Machine Learning (ICML)*. (2003)
9. Esposito, F., Ferilli, S., Fanizzi, N., Basile, T., Di Mauro, N.: Incremental learning and concept drift in inthelex. *Intelligent Data Analysis Journal, Special Issue on Incremental Learning Systems Capable of Dealing with Concept Drift* **8**(3) (2004) 213–237
10. Semeraro, G., Esposito, F., Malerba, D.: Ideal refinement of datalog programs. In: Proietti, M., ed.: *Logic Program Synthesis and Transformation*. Volume 1048 of LNCS., Springer (1996) 120–136

11. Billard, A., Siegwart, R.: Robot learning from demonstration. *Robotics and Autonomous Systems* **47**(2-3) (2004) 65–67
12. Schaal, S., Ijspeert, A., Billard, A.: Computational approaches to motor learning by imitation. *Philosophical Transactions: Biological Sciences* **358**(1431) (2003) 537–547
13. Meltzoff, A.: The "like me" framework for recognizing and becoming an intentional agent. *Acta Psychologica* **124**(1) (2007) 26–43
14. Schaal, S.: Is imitation learning the route to humanoid robots? *Trends in cognitive sciences* **3**(6) (1999) 233–242
15. Niolescu, M., Mataric, M.: Natural methods for robot task learning: instructive demonstrations, generalization and practice. In: *Proceedings of the second international joint conference on Autonomous agents and multiagent systems (AAMAS03)*, ACM (2003) 241–248
16. Atkeson, C., Schaal, S.: Robot learning from demonstration. In Fisher, D., ed.: *Proceedings of the 14th International Conference on Machine Learning (ICML)*. (1997) 12–20
17. Smart, W., Kaelbling, L.: Effective reinforcement learning for mobile robots. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Volume 4. (2002) 3404–3410
18. Chernova, S., Veloso, M.: Confidence-based policy learning from demonstration using gaussian mixture models. In: *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, New York, NY, USA, ACM (2007) 1–8
19. Bentivegna, D., Atkeson, C., Cheng, G.: Learning from observation and practice using primitives. In: *AAAI Fall Symposium Series, 'Symposium on Real-life Reinforcement Learning'*. (2004)

Constraint based implementation of a PDDL-like language with static causal laws and time fluents

Agostino Dovier¹ and Jacopo Mauro²

¹ Dipartimento di Matematica e Informatica, Università di Udine
dovier@dimi.uniud.it

² Dipartimento di Scienze dell'Informazione, Università di Bologna
jmauro@cs.unibo.it

Abstract. Planning Domain Definition Language (PDDL) is the most used language to encode and solve planning problems. In this paper we propose two PDDL-like languages that extend PDDL with new constructs such as static causal laws and time fluents with the aim of improving the expressivity of PDDL language. We study the complexity of the main computational problems related to the planning problem in the new languages. Finally, we implement a planning solver using constraint programming in GECODE that outperforms the existing solvers for similar languages.

1 Introduction

In the context of knowledge representation and reasoning, a very important application of logic programming within artificial intelligence is that of developing languages and tools for reasoning about actions and change and, more specifically, for the problem of planning [2]. The proposals on representing and reasoning about actions and change have relied on the use of concise and high-level languages, commonly referred to as *action description languages*. Some well-known examples include the languages \mathcal{A} and \mathcal{B} [11] and extensions like \mathcal{K} [7]. Action languages allow one to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system. An *action description* is a specification of a planning problem using the action language.

Since 1998 a declarative language for planning has been defined either for establishing a common syntax or for allowing different research groups to participate to the planning competitions. This language is known as PDDL and its last release is 3.1 (see [16, 10, 12] for information on planning competitions and PDDL).

The goal of this work is to build two languages on the top of PDDL, called *APDDL* and *BPDDL*, allowing new constructs and then explore the relevance of constraint solving for handling them. The main ideas of the constraint encoding comes from [6]. However, here we employed the C++ constraint solver platform GECODE [1] which is faster than the constraint solver of SICStus

Prolog used in [6], and we solve more precisely the frame problem. Moreover, we define a front-end from the PDDL-like languages to GECODE. We always outperform the running time of [6]; in some cases the improvements are really sensible.

The presentation is organized as follows. In Section 2 we introduce the language APDDL and in Section 3 we formally define its semantics. In Section 4 we define the language BPDDL. In Section 5 we report the complexities of some interesting problems related to planning in this language. The solver implementation and the tests are then discussed in Sections 6 and 7. Some proofs are reported in Appendix.

2 The language APDDL

APDDL is an extension of the well-known language PDDL and every APDDL program consists of two parts: the *domain definition* used to model the planning problem, and the *instance definition* used to define the instance of the problem to solve. We need to define a set \mathcal{F} of fluent names. Each $f \in \mathcal{F}$ is assigned to a domain $\text{dom}(f)$. We also need to define a set \mathcal{A} of action names. Each action a is associated to a precondition $\text{pre}(a)$ and an effect $\text{eff}(a)$ all expressed as a Boolean combinations of arithmetic constraints on fluents (see Table 1 for a simplified syntax³).

C	::=	0 1 (not C) (and C ⁺) (or C ⁺) (AOP AC AC) (eqv C C) (imp C C) (xor C C) TIME_FLUENT
AC	::=	n TIME_FLUENT (OP AC AC)
AOP	::=	> ≥ < ≤ = ≠
OP	::=	+ - * / mod rem
TIME_FLUENT	::=	f (at n f)

Table 1. Abstract syntax of constraints (C)— $n \in \mathbb{Z}, f \in \mathcal{F}$

The concrete syntax of the language is described by a EBNF grammar available at [15]. We just give here a taste of the syntax using a simple example: the famous Sam Lloyd’s n -puzzle that consists of a frame of numbered square tiles in random order with one tile missing. The tiles should be arranged in increasing order, with the hole in the bottom right corner. Types can be used for differentiating objects. In this example we can define a type for the position of a tile, one for the direction of the move to do and one for the numbers on the tile. This can be done in the following way:

```
(:types positions directions tiles_numbers)
```

³ where `eqv`, `imp`, `xor`, `mod`, `rem` are respectively the equivalence, implication, exclusive or, module and remainder operators

We can associate names (also known, with abuse of terminology, as constants) to constant and function symbols (with arity 0 and greater than 0, respectively). We can define the function symbol `near` with arity 2 used to determine what position should occupy a tile moved from a position `pos` according to a particular direction `dir`.

```
(near ?pos - positions ?dir - directions) - positions
```

In the example we can consider a missing tile as a tile numbered 0. We thus define the constant `empty_tile_number` for representing it.

```
empty_tile_number - tiles_numbers
```

We use these preliminary definitions to encode the relevant proprieties of the objects that we want to consider. These properties are the *fluents* and they are represented by a (multivalued) function. Boolean functions are called predicates. In our example we are interested in knowing the number in a particular position. This multi-valued fluent can be defined in the following way:

```
(has ?position - positions) - tiles_numbers
```

The last ingredient for the domain definitions are the definitions of the possible effects of the actions given their preconditions. In this example there is only an action: moving a tile. This action can be defined as follows⁴:

```
(:action move
  :parameters (?from ?to - positions)
  :precondition (and
    (exists ?direction - directions
      (== (near ?from ?direction) ?to)
    )
    (== (has ?to) empty_tile_number) )
  :effect (and
    (== (has ?from) empty_tile_number)
    (== (has ?to) (at -1 (has ?from))))
)
```

Let us suppose that we want to solve this problem in a 3×3 board where the missing tile is at the bottom right corner. This can be encoded into the problem definition. First, all the objects involved are listed. In this case we have the following three types of objects⁵:

```
(:objects
  (set 1 9) - positions
  Left Right Up Down - directions
  (set 0 8) - tiles_numbers
)
```

⁴ The term `(at -1 (has ?from))` is a time fluent and it will be described later

⁵ `set` is an APDDL operator for defining set of integers in a concise way

Second, all the constants should be instantiated.

```
(:constants
  (== empty_tile_number 0)
  (== (near 1 Right) 2) (== (near 1 Down) 4)
  (== (near 2 Right) 3) ...
)
```

The problem definition includes also the definition of the values of the fluents in the initial state and the goal. In the problem we are considering this can be done in the following way:

```
(:init (== (has 1) 2) (== (has 2) 5) ... )
(:goal (== (has 1) 1) (== (has 2) 2) ... )
```

We impose that the fluents in the initial state are completely defined.

If the goal of the problem is to obtain an optimal plan it is possible to define a cost function. This function referred as *metric* has the following syntax:

```
M ::= (:metric MOP MC)
MOP ::= minimize | maximize
MC ::= AC | (is_violated C) | (OP MC MC)
```

where AC and C are defined as in Table 1. For example, suppose that a state has cost 0 if the number 1 is in the first row and 1 otherwise. If we want to minimize the cost, the metric can be defined in the following way:

```
(:metric minimize
  (is_violated (or (== (has 1) 1) (== (has 2) 1) (== (has 3) 1))))
```

Finally, at the end of the problem definition, it is necessary to specify the length of the plan we want to obtain. This can be done using the length primitive:

```
(:length 18)
```

The language APDDL has few more features like the possibility to introduce a metric function to maximize or minimize and additional constraints called *plan constraints*. For example in the *n*-puzzle problem it is possible to state that the tile with the number 1 should be in the last line at least once:

```
(:constraints (sometimes (or (== (has 7) 1)
  (== (has 8) 1)
  (== (has 9) 1))))
```

The main difference between the PDDL language and the APDDL is the possibility to use more operators in the action precondition and effects (division, remainder, exclusive or, ...) and the notion of time fluent.

A *time fluent* is an expression of the form $(\text{at } i \ f)$ where $i \in \mathbb{Z}$ is an integer and f is a fluent. This construct is used in actions for referring to the value of a fluent f in time instant i . If $i = 0$ then $(\text{at } 0 \ f)$ (or, in short, f) is called present

fluent because it refers to the value of the fluent f in the current state. If $i < 0$ (resp. $i > 0$) the term $(\text{at } i \ f)$ is called instead past fluent (resp. future fluent). Let us observe that if the time fluent is used in an action precondition it refers to the state at which the action is executed. If it is used in the effect it refers to the state produced by the execution of the action.

An example of the use of time fluents is the following action that decreases the number of objects in a barrel in the next two states if during the last two state transitions at least one object is added into the barrel.

```
(:action empty
  :parameters (?barrel - barrel)
  :precondition (and
    (> (contains ?barrel) (at -1 (contains ?barrel)))
    (> (at -1 (contains ?barrel)) (at -2 (contains ?barrel))))
  :effect (and
    (== (contains ?barrel) (- (at -1 (contains ?barrel)) 1))
    (== (at 1 (contains ?barrel)) (- (contains ?barrel) 1)))
)
```

In goal constraints, plan constraints and metrics it is not possible to use past or present fluents while in init constraint it is not possible to use past fluents.

3 APDDL Semantics

Given an APDDL program P it is possible to obtain an equivalent ground instance $ground(P)$ by grounding all variables with all constants satisfying the types. In $ground(P)$ actions preconditions and effects, goal conditions and all the information on the initial state are (Boolean combinations of) Finite Domain constraints on time fluents.

A state is characterized by the values of all the fluents involved. We will use the term $val(s, f)$ to denote the value of the fluent f in the state s . Any action a is characterized by its preconditions $pre(a)$ and its effects $eff(a)$. We allow parallel executions of different actions a_1 and a_2 provided their effects are independent. We impose a strong syntactic requirement: the sets of time fluents occurring in $eff(a_1)$ and $eff(a_2)$ must be disjoint.

Let us consider a sequence of states s_0, s_1, \dots, s_n , and a constraint c . With $shift_i(c)$ we denote the constraint obtained replacing each time fluent $(\text{at } t \ f)$ with the value $val(s_{t+i}, f)$. If $t + i < 0$ or $t + i > n$ then the value is \perp (undefined). Let us observe that $c' = shift_i(c)$ is a Boolean combination of ground arithmetic constraints (or \perp). If \perp occurs in it, then its value is **false**. Otherwise, its value is determined by the usual semantics of arithmetic and Boolean operators on ground formulas. If the value of c' is **true**, we say that $s_0, s_1, \dots, s_n \models c'$, otherwise $s_0, s_1, \dots, s_n \not\models c'$.

Let G be the set of *goal constraints* and I be the set of *initial constraints*. Then a plan of length n ($n \geq 0$) is a sequence $s_0, A_1, s_1, \dots, A_n, s_n$ where

1. s_0, \dots, s_n are states

2. A_1, \dots, A_n are (possibly empty) sets of actions
3. $\forall i \in \{1, \dots, n\} \forall a \in A_i . s_0 \dots s_n \models \text{shift}_{i-1}(\text{pre}(a))$
4. $\forall i \in \{1, \dots, n\} \forall a \in A_i . s_0 \dots s_n \models \text{shift}_i(\text{eff}(a))$
5. $\forall c \in G . s_0 \dots s_n \models \text{shift}_n(c)$
6. $\forall c \in I . s_0 \dots s_n \models \text{shift}_0(c)$
7. $\forall a_1 \in A_i \forall a_2 \in A_i . a_1 \neq a_2$ $\text{eff}(a_1)$ and $\text{eff}(a_2)$ do not share future or present fluents.
8. if no action executed refers to a fluent f in s_i ($i > 0$) then $\text{val}(s_i, f) = \text{val}(s_{i-1}, f)$ (inertia condition)

When further *plan constraints* are used, the plan definition must entail more constraints. For instance, if the constraint (`sometimes c`) is added, then there must be i such that $s_0 \dots s_n \models \text{shift}_i(c)$.

4 BPDDL

Starting from the language APDDL we add the possibility to use a construct like the static causal laws (briefly denoted here as *rules*) introduced in the language \mathcal{B} [11], obtaining the new language BPDDL. A rule has a precondition and an effect similar to the action preconditions and effects, but without future fluents. Informally, the semantics of a rule is that at every state of the plan if the precondition is true then also the effect must be true.

Rules are more powerful than PDDL axioms [18]. As a matter of fact, differently from axioms, rules do not require to define predicates using only stratified programs (and this is a strong constraint for knowledge representation); moreover, they are allowed to change values of fluents that can be also used in action effects.

The possibility of using rules increases the expressiveness of the language. For instance it is possible to change a fluent value after a transition even if no action has been executed. A simple example is the implementation of a clock simply using a fluent and a rule

```
(:rule
  :parameters ( ?time - time)
  :effect (== ?time (+ (at -1 time) + 1))
)
```

Another interesting use of rules is the propagation of an action effect. Let us consider for example a colored directed graph where we want that all the nodes connected with edges in a set E_1 have the same color. This propriety can be encoded in the following way:

```
(:rule
  :parameters ( ?edge - edge)
  :precondition (is_edge_in_E_1 ?edge)
  :effect (== (node_colour (head ?edge))
             (node_colour (tail ?edge)))
)
```

When clear from the context, we will use the abstract notation $c_1 \rightarrow c_2$ for rules.

4.1 BPDDL Semantics and Inertia

Dealing with inertia in presence of rules is obviously more difficult than in a language that does not allow them. This is particularly true if the implementation of a language is based on the notion of constraint. Two rules stating the implications $p \rightarrow q$ and $q \rightarrow p$ are satisfied either by $p = q = 0$ or by $p = q = 1$. However, an arbitrary change of the values from 0 to 1 or vice versa cannot be simply justified by these rules.

A simple attempt of solution considered is that of choosing the states with a minimum change of fluents. Unfortunately, this definition cuts off a lot of solutions, as already pointed out in [3].

We instead used a solution based on the following principle: “*Given some action effects if something can be left unchanged then it must be unchanged*”

Let us define with $\text{Act}(s_0, A_1, s_1, \dots, A_n, s_n)$ the fluents of s_n that can be modified as a direct effect of an action in $\bigcup A_i$. Suppose that $\Delta F(s, s')$ is the set of fluents that have different values between the state s and s' . Now, in a plan $s_0, A_1, s_1, \dots, A_n, s_n$ we say that there is a *critical situation* between s_{i-1} and s_i if there is a sequence $s_0, A_1, s_1, \dots, s_{i-1}, A_i, s'$ where the state s' :

- entails all the rules
- $\Delta F(s_{i-1}, s') \subset \Delta F(s_{i-1}, s_i)$
- $\forall f \in \text{Act}(s_0, A_1, s_1, \dots, A_i, s_i). \text{val}(s_i, f) = \text{val}(s', f)$

Intuitively when there is a critical situations there is at least a fluent that can remain the same but instead has been changed by a rule. Therefore when there is a critical situation in a plan the over mentioned principle is violated.

Just a comment on the past references in rules. If a rule refers to a value of a fluent prior to the initial state s_0 of a plan the rule is trivially satisfied.

The semantics of the language BPDDL is similar to the semantics of APDDL with only two further requirements:

- the inertia condition is now the absence of critical conditions between two consecutive states in the plan
- the states must entail the applicable rules

5 Complexity

In this section we study the complexity of the main computational problems related to planning expressed within the APDDL and BPDDL languages. In particular, we focus our attention to ground APDDL/BPDDL programs. For APDDL programs some of the problems are equivalent, other are simpler or not meaningful. We assume moreover that no *plan constraints* is used in the program. Their inclusion would make the proofs more complicated but they would not affect the results. We studied the complexity of the following decision problems:

1. *has_critical_situation* (APDDL: not meaningful. BPDDL: NP-complete)
input: a program, a sequence of states and actions $s_0, A_1, s_1, \dots, A_n, s_n$ and two consecutive states s_i, s_{i+1} that entail all the conditions in the plan definition except the inertia
output: 1 iff there is a critical situation between s_i, s_{i+1} , otherwise 0
2. *validity* (APDDL: P; BPDDL: co-NP-complete)
input: a program and a sequence of states and actions $s_0, A_1, s_1, \dots, A_n, s_n$
output: 1 iff $s_0, A_1, s_1, \dots, A_n, s_n$ is a plan, otherwise 0
3. *k-plan* (APDDL: NP-complete; BPDDL: Σ_2^P -complete)
input: a program
output: 1 iff there is a plan of length k that solves the problem encoded into the BPDDL program, otherwise 0
4. *plan:* (APDDL and BPDDL: PSPACE complete)⁶
input: a program
output: 1 iff there is a plan that solves the problem encoded into the BPDDL program, otherwise 0

We give here only the main ideas used. The complete proofs of the results are reported in Appendix.

The proof of NP-hardness of *has_critical_situation* is based on a reduction from a variant of SAT in which all false and all true assignments are forbidden. Let us consider the Boolean formula $\varphi = (X \vee Z) \wedge (\neg X \vee \neg Y \vee \neg Z)$ and consider the following program based on three rules with fluents f_X, f_Y, f_Z (\oplus stands for exclusive or while f_W^{-1} for the past fluent (**at** $-1 f_W$)):

$$\begin{aligned}
\text{true} &\rightarrow f_X \vee f_Z \vee (f_X \wedge f_Y \wedge f_Z) \vee (\neg f_X \wedge \neg f_Y \wedge \neg f_Z) \\
\text{true} &\rightarrow \neg f_X \vee \neg f_Y \vee \neg f_Z \vee (f_X \wedge f_Y \wedge f_Z) \vee (\neg f_X \wedge \neg f_Y \wedge \neg f_Z) \\
\text{true} &\rightarrow (f_X^{-1} \oplus f_X) \vee (f_Y^{-1} \oplus f_Y) \vee (f_Z^{-1} \oplus f_Z)
\end{aligned}$$

Let us consider now two states s_0 and s_1 , where for every fluent f in $\{f_X, f_Y, f_Z\}$ it holds that $\text{val}(s_0, f) = 0$ and $\text{val}(s_1, f) = 1$. And, let us analyze the problem: is there a critical situation between s_0, s_1 ? The first two rules are satisfied if all the fluents are true or all are false or if there is an assignment that satisfies the formula φ . The last rule, instead, forces at least one fluent to change.

Let us observe that φ is not satisfiable by a trivial assignment⁷. One of the possible non trivial assignments that satisfy φ is instead $\{X/\text{true}, Y/\text{false}, Z/\text{false}\}$. Using this assignment we can define a state s' such that $\text{val}(s', f_X) = 1, \text{val}(s', f_Y) = 0, \text{val}(s', f_Z) = 0$ that satisfies all the rules and with fluent variations included w.r.t. those between s_0 and s_1 . Therefore there is a critical situation.

⁶ APDDL and BPDDL are PSPACE complete if the maximum temporal reference used is polynomially bounded on the length of the program encoding. See details in proofs.

⁷ An assignment is trivial if all the variables are assigned to true or all the variables are assigned to false

As far as the validity problem is concerned, checking if all the plan conditions but the inertia holds can be done in polynomial time (and this is what is needed in APDDL). Verifying if there are no critical situations in BPDDL can be done in polynomial time using a co-NP oracle machine that solves the complement of `has_critical_situation`.

k -plan problem membership to Σ_2^P derives directly by the NP-completeness of `has_critical_situation`. To prove the Σ_2^P hardness of k -plan we reduce it to the problem of finding an answer set for the extended disjunctive logic program (EDLP programs) [4]. An EDLP program is a set of rules of the form

$$l_1 | \dots | l_p \leftarrow l_{p+1}, \dots, l_m, \text{not } l_{m+1}, \dots, \text{not } l_n$$

where $n \geq m \geq p \geq 0$ and each l_i is a literal, i.e. an atom a or the classical negation $\neg a$ of an atom in a first-order language, and `not` is a negation-as-failure operator. The symbol $|$ is used to distinguish disjunction in the head of a rule from disjunction \vee used in classical logic.

The problem of deciding if a propositional (i.e. ground) EDLP has an answer set is Σ_2^P complete [8].

We introduce the reduction with an example. Consider the following propositional EDLP from [17] which states that everyone is pronounced not guilty unless proven otherwise:

$$\begin{aligned} \text{innocent} | \text{guilty} &\leftarrow \text{charged} \\ \neg \text{guilty} &\leftarrow \text{not proven} \\ \text{charged} &\leftarrow \end{aligned}$$

From this program we can generate a program based on the following rules with fluents `innocent`, `guilty`, `charged`, `proven`, w .

$$\begin{aligned} (w^{-1} = 2 \wedge \text{charged} = 1) &\longrightarrow (\text{innocent} = 1 \vee \text{guilty} = 1) \\ (w^{-1} = 2 \wedge \neg(\text{proven} = 1)) &\longrightarrow \text{guilty} = 0 \\ w^{-1} = 2 &\longrightarrow \text{charged} = 1 \end{aligned}$$

If the fluents `innocent`, `guilty`, `charged`, `proven`, w can have values in $\{0, 1, 2\}$ and at the initial state all fluents have value 2 then there is a plan of length 1 iff the EDLP program has a stable model. Intuitively the answer set has an atom a (resp. $\neg a$) if in the final state $a = 1$ (resp. $a = 0$). If $a = 2$ then it is not in the answer set. In the previous case the single answer set is $\{\neg \text{guilty}, \text{innocent}, \text{charged}\}$ and the plan that solves the problem is

$$\begin{aligned} &\{\text{guilty}/2, \text{innocent}/2, \text{charged}/2, \text{proven}/2, w/2\}, \\ &\quad \emptyset, \\ &\{\text{guilty}/0, \text{innocent}/1, \text{charged}/1, \text{proven}/2, w/2\} \end{aligned}$$

PSPACE membership can be proven viewing the planning problem as a reachability problem on a graph where the nodes are states and the arcs are set of actions. Encoding a state and checking if there is an arc between two states is feasible in polynomial space and therefore reachability can be decided in PSPACE.

The plan problem is PSPACE complete because APDDL/BPDDL is more expressive than STRIPS [9]. A STRIPS program can be mapped into a APDDL or BPDDL program straightforwardly and thus since plan in STRIPS is PSPACE complete [5] the plan problem is PSPACE complete also in APDDL or BPDDL.

6 Solver

The positive results of the approach [6] encouraged us to write a constraint-based solver for the languages APDDL and BPDDL. The implementation of BPDDL subsumes that of APDDL. We decided to exploit the constraint solver GECODE, implemented in C++, that offers competitive performance w.r.t. both runtime and memory usage. Starting from the context-free grammar of BPDDL we have defined a lexical analyzer and a parser using the standard tools flex (Fast Lexical Analyser) and Bison. The developed solver solves the k -plan problem.

The overall structure of the solver is similar to that developed in [6] and it deals with the following variables:

1. for every fluent in every state one FD variable (a Boolean variable if the fluent is a predicate) represents the value of the fluent in that state
2. for every action in every transition one boolean variable represents if the action is executed

Constraints for checking action preconditions and imposing action effects are then added. In the case of the BPDDL-solver we also introduced a set of constraints to verify the closure of a state w.r.t. the rules and to solve the frame problem.

Verifying that there are no critical situations can lead to a definition of an exponential number of constraints. As pointed out in [13] and [14] for Answer Set Programming, it is not possible to solve the frame problem adding only a polynomial number of formulas of polynomial length unless $P = NP$.

Let now define a function $\text{shiftRule}(F, r)$ that taking a set of fluents F and a rule r , decreases by one the time reference of all the time fluents ($\text{at } 0 \text{ } f$) in r if $f \in F$.

Let $\text{ruleModified}(f, s_0, A_1, s_1, \dots, A_n, s_n)$ be the constraint that is true iff $f \notin \text{Act}(s_0, A_1, s_1, \dots, A_n, s_n)$ and $\text{val}(s_n, f) \neq \text{val}(s_{n-1}, f)$.

There is no critical situation between two states s_{i-1}, s_i in a plan $s_0, A_1, s_1, \dots, s_n$ if for all non empty subset of fluents F

$$s_0, \dots, s_n \models \text{shift}_i \left(\bigwedge_{r \text{ rule}} \text{shiftRule}(F, r) \rightarrow \neg \bigwedge_{f \in F} \text{ruleModified}(f, s_0, A_1, s_1, \dots, A_i, s_i) \right)$$

Intuitively, we check if the rules are fulfilled even if the fluents in F are left unchanged. When this happens we must assure that in this case there is at least a fluent in F that is not modified only by rules.

In the BPDDL solver we tried to minimize the number of constraints added to state the inertia. Suppose P is a partition such that for every $(\text{at } 0 \text{ } f_1), (\text{at}$

0 f_2) defined in a rule one of its elements contains f_1, f_2 . To avoid critical situations it is possible to add one of the above-mentioned constraints for all the non empty subsets of the elements in P .

If a metric is used we employ the branch and bound algorithm for finding the optimum solution. Otherwise, we use the default algorithm provided by GECODE for exploring the search space (depth first search).

We also developed two optional heuristics for reducing the search space. The first one, called `no_state_repetition` avoids the possibility of returning to an already visited state (drawback: if a k -plan exists only with multiple visits of a node, we don't find it).

The second heuristics is called `confluent_actions` that imposes a partial order on actions. We say that $a_1 < a_2$ when for every plan of length 2 the execution of a_1 and a_2 has the same effect of the execution a_2 and a_1 . We notice that $a_1 < a_2$ is always true if the set of fluents in a_1 effect is disjoint from the set of fluents in a_2 precondition and vice versa. When this happens we impose that in a plan the action a_1 should be executed before the action a_2 . This heuristic can reduce the plan symmetries.

Since sometimes we are interested in finding a sequential plan we allow the programmer to choose at most or exactly one action per transition.

7 Tests

For the scope of this paper, we compared the performances of the GECODE based APDDL solver with the performances of the Solver for the language \mathcal{B}^{MV} [6]. For the tests we used an AMD Opteron 2.2 GHz Linux Machine. The APDDL solver uses GECODE 2.1.1 and was compiled with the 4.1.2 version of g++. The \mathcal{B}^{MV} solver instead is written and executed in SICStus Prolog 4.0.4. As benchmarks we chose some of the domains studied and presented in [6]. We are planning some other tests also with other systems (e.g., MIPS-XXL, SG-Plan5, SatPlan) when they are applicable (e.g. for domains without time fluents and rules). All the solver codes and the examples of the program used for the testing are available at [15].

As explained in [6], for implementation choice, treatment of inertia in \mathcal{B}^{MV} can be incorrect for some programs where rules introduce loops. The BPDDL solver, instead, works correctly on those examples. Anyway, in our tests we choose to compare \mathcal{B}^{MV} with APDDL on domains without rules. BPDDL has basically the same running time as APDDL (with a 5% of overhead) on the tested domains.

For every instance of the problems we considered both the time needed by the solver to post the constraints and the time needed to find the first solution (if any). Timings are expressed in ms and are given as a sum of the post time (first term) and the search time (last term in the addition). Even if the two languages used are different (PDDL like vs Prolog like) we encoded the domains basically in the same way (same actions, same preconditions, etc) and we have used both

the solvers with the default parameters (A/BPDDL solver chooses the variable with the smallest domain size and the smallest value during the search process).

Since the \mathcal{B}^{MV} solver is studied for sequential plans we impose the same constraint for the A/BPDDL solver. Table 2 contains the execution times for the n -puzzle problem, the solitaire peg game⁸ (a plan with 31 moves), the problem of finding a knight walk on a 4×4 chessboard, and the well-known three barrels problem with barrels of 20-11-9 liters.

Knight and peg has been launched with and without the `no_state_repetition` heuristic.

Table 2. Experimental results for the 3x3 puzzle, knight walk, peg solitaire and barrels (the * means that the APDDL solver was used without the `no_state_repetition` heuristics)

Prob.	Instance	Len.	Sol.	APDDL	\mathcal{B}^{MV}	$\frac{\mathcal{B}^{MV}}{\text{APDDL}}$
puzzle	I_1	19	No	10 + 3660	150 + 12580	3,5
puzzle	I_1	20	Yes	0 + 70	140 + 4210	62,1
puzzle	I_2	24	No	10 + 93970	150 + 270080	2,9
puzzle	I_2	25	Yes	10 + 38010	180 + 314930	8,3
puzzle	I_3	20	No	10 + 8910	90 + 31140	3,5
puzzle	I_3	25	No	10 + 129760	170 + 463500	3,6
knight		24	Yes	40 + 98610	670 + 2743660	27,8
knight *		24	Yes	30 + 68120	1550 + 2620060	38,5
peg		11	No	10 + 13790	620 + 841340	61,0
peg *		11	No	10 + 9510	640 + 849610	89,3
peg		31	Yes	50 + 41390	1850 + 47910	1,2
peg *		31	Yes	30 + 16690	1780 + 46360	2,88
barrels	20-11-9	18	No	10 + 350	60 + 560	1,7
barrels	20-11-9	19	Yes	10 + 150	60 + 240	1,9

In Table 3 we compare the times for two multivalued problems. The gas diffusion problem where Diabolik wishes to fill a room with sufficient amount of gas in order to generate an explosion below the central bank⁹ and a community problem where, according to some rules, rich people wish to give money to poor people in order to reach an equilibrium.

The tests reveal that the APDDL solver is always the fastest one. In particular for the toughest instances the times can be decreased by an order of magnitude or more (see for example the results obtained in the gas diffusion and community problem).

⁸ This problem is one of the benchmarks of the 2008 planning competition.

⁹ This is a variant of the *pipesworld* domain of the 2006 planning competition.

Table 3. Experimental results for the gas diffusion problem and the community problem

Prob.	Instance	Len.	Sol.	APDDL	\mathcal{B}^{MV}	
					APDDL	APDDL
gas	A_1	6	Yes	0 + 220	70 + 1348	6
gas	A_1	7	Yes	0 + 10	100 + 5350	545
gas	B_1	10	No	10 + 8500	170 + 3846200	452
gas	B_1	11	Yes	0 + 10	140 + 1802760	180290
gas	B_1	12	Yes	10 + 20	150 + 933350	31117
gas	B_1	13	Yes	10 + 70	160 + 302340	3781
gas	B_1	14	Yes	10 + 170	140 + 4600	26
community	A_5	5	No	0 + 9500	50 + 264760	28
community	A_5	6	Yes	0 + 100	30 + 200	2
community	A_5	7	Yes	0 + 12610	40 + 930080	74
community	B_5	5	No	0 + 5080	30 + 131630	26
community	B_5	6	Yes	10 + 0	30 + 110	14
community	B_5	7	Yes	0 + 10	40 + 170	21

8 Conclusion and Future Work

In this work we presented two extensions to PDDL-like languages. The first extension introduces new operators and the notion of time fluents that allow to express plan problems in a more concise way. For example suppose that we have a board of lights and when one of the lights is off or on also the status of the neighbor lights change. In BPDDL this situation can be easily modeled in the following way:

```
(:action press
:parameters (?light - lights)
:effect (and
(xor (at -1 (is_on ?cell)) (is_on ?cell))
(forall ?neighbor - lights
(imp
(== (neighbor ?cell) ?neighbor)
(xor (at -1 (is_on ?neighbor)) (is_on ?neighbor))
))))
```

Then we introduced static causal laws and we provide a solver based on GECODE which has been proved to be effective and it also represents a solid starting point for future extensions. We then characterized the complexities of the major problems relating to planning within the proposed languages. With static causal laws the plan problem became harder than in the case of absence.

Some of the possible next steps are:

- extend the language BPDDL with some construct to query and constrain the occurrences of the action directly in the language
- extend the language to allow more expressive metrics (e.g. it is possible to define metrics giving for every action a certain cost)

- supporting in both languages other features like preferences or hierarchical types
- support multi-agent planning and forms of concurrent actions
- create a compiler from the language PDDL to APDDL
- port the code of the solver using the new GECODE 3.0 environment
- compare the solver with the state-of-the-art PDDL planning solvers

Acknowledgments

We thank Andrea Formisano and Enrico Pontelli for the several useful discussions and technical help. The research is partially supported by PRIN and FIRB RBNE03B8KK projects.

References

1. Gecode: Generic constraint development environment, 2008. Gecode is currently developed by Christian Schulte, Mikael Lagerkvist, Guido Tack and it is available from <http://www.gecode.org>.
2. C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
3. C. Bell, A. Nerode, R. T. Ng, and V. S. Subrahmanian. Mixed integer programming methods for computing nonmonotonic deductive databases. *J. ACM*, 41(6):1178–1215, 1994.
4. R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994.
5. T. Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994.
6. A. Dovier, A. Formisano, and E. Pontelli. Multivalued Action Languages with Constraints in CLP(FD). In V. Dahl and I. Niemelä, editors, *ICLP 2007 Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2007. Extended version in <http://www.dimi.uniud.it/dovier/PAPERS/rrUD.01-09.pdf>.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Log.*, 5(2):206–263, 2004.
8. T. Eiter and G. Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *ILPS*, pages 266–278, 1993.
9. R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.
10. M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.
11. M. Gelfond and V. Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
12. A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.
13. V. Lifschitz and A. A. Razborov. Why are there so many loop formulas? *ACM Trans. Comput. Log.*, 7(2):261–268, 2006.

14. F. Lin and Y. Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
15. J. Mauro and A. Dovier. APDDL and BPDDL codes and examples. Available from <http://www.dimi.uniud.it/dovier/MISIGE/>.
16. D. V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.
17. T. C. Przymusiński. Stable semantics for disjunctive programs. *New Generation Comput.*, 9(3/4):401–424, 1991.
18. S. Thiébaux, J. Hoffmann, and B. Nebel. In defense of pddl axioms. *Artif. Intell.*, 168(1-2):38–69, 2005.

A Complexity results and proofs

Let us start with some basic observations. Given a ground BPDDL program of length n , the rules, the actions, the goal constraints, the initial constraints have all length bounded by n . Similarly, since all the fluents in the initial state must be uniquely determined by the initial constraints, the number of fluents is bounded by n . Therefore, checking if a rule, an action precondition or an action effect is entailed by a state can be done in polynomial time on n . In a similar way checking if two actions can not be done simultaneously is feasible in polynomial time.

Let us define an assignment $\{X_1/v_1, \dots, X_n/v_n\}$ non trivial if $\exists i, j \in \{1, \dots, n\}$ s.t. $v_i = \text{true}$ and $v_j = \text{false}$. Let non-trivial-SAT be the problem of deciding if a formula is satisfied only by a non trivial assignment.

Lemma 1. *non-trivial-SAT is NP-complete*

Proof. NP membership derives from the fact that checking if an assignment satisfies a boolean formula and it is non trivial are two polynomial problems.

NP-hardness can be proved via reduction from SAT. Let φ be a SAT formula and X and Y two fresh boolean variables. Let $\phi = \varphi \wedge (X \vee Y) \wedge (\neg X \vee \neg Y)$. We have that φ is satisfiable iff there is a non trivial assignment that satisfies ϕ . \square

Theorem 1. *has_critical_situation is NP-complete*

Proof. NP membership can be proved by noticing that a certificate of the problem is a state s' that entails the rules and $\Delta F(s_i, s_{i+1}) \supset \Delta F(s_i, s')$. Checking such a certificate can be done in polynomial time.

NP-hardness can be proved via reduction from non-trivial-SAT.

Let us consider a formula $\varphi = \psi_1 \wedge \dots \wedge \psi_k$ where ψ_i are clauses. Let \mathcal{F} be the set of variables in φ . We build a BPDDL program as follows (we use an abstract syntax for the sake of clarity):

- for all $i = 1..k$ ($\psi_i \vee \bigwedge_{f \in \mathcal{F}} f = 0 \vee \bigwedge_{f \in \mathcal{F}} f = 1$)
- $\bigvee_{f \in \mathcal{F}} (\text{at } -1 f) \neq f$

Let us consider the plan s_0, \emptyset, s_1 where $\text{val}(s_0, f) = 0$ and $\text{val}(s_1, f) = 1$ for all $f \in \mathcal{F}$.

It is possible to prove that φ is satisfiable only by a non trivial assignment iff there is a critical situation between s_0 and s_1 . \square

Theorem 2. *validity is co-NP-complete*

Proof. Given a plan, checking if all the plan conditions except the inertia are respected can be done in polynomial time. Verifying if there are no critical situations can be done using a co-NP machine that solves the complement of has_critical_situation. Validity is therefore in co-NP.

Let be no_validity the complement of validity. In Theorem 1 we proved that has_critical_situation is still NP hard even with some restrictions (the plan

has 2 states, there are no actions, ...). With these restrictions no_validity is has_critical_situation and thus it is NP hard.

Since no_validity is NP hard than validity is co-NP hard. \square

In APDDL instead the validity problem is only polynomial on the length of the program encoding. The difference between the two languages is made by the definition of inertia that in the case of the BPDDL leads to a potentially exponential search space of states on the number of the fluents.

Theorem 3. *k-plan is in Σ_2^P*

Proof. Let M be a non deterministic Turing machine that guesses a sequence of states and actions $s_0, A_1, s_1, \dots, A_k, s_k$ and checks if all the plan conditions except inertia are satisfied. Then for checking the inertia M calls k times an oracle machine that solves has_critical_situation.

If all the checks are positive M returns 1, otherwise 0.

M solves k-plan in polytime. Therefore k-plan is in $NP^{NP} = \Sigma_2^P$ \square

Given a EDLP program P we define as $T(P)$ the BPDDL program where:

- a multi-fluent a is defined for every atom a in P . These fluents can be 0, 1 or 2
- w is a new fluent
- for every rule $l_1 | \dots | l_j \leftarrow l_{j+1}, \dots, l_m, not l_{m+1}, \dots, not l_n$ in P there is a rule

$$\left(((at - 1 w) = 2) \wedge \bigwedge_{i=j+1}^m \sigma(l_i) \wedge \bigwedge_{i=m+1}^n \neg \sigma(l_i) \right) \rightarrow \bigvee_{i=1}^j \sigma(l_i)$$

where

$$\sigma(l_i) = \begin{cases} (a = 1) & \text{if } l_i = a \\ (a = 0) & \text{if } l_i = \neg a \end{cases}$$

- the initial constraint is $(w = 2) \wedge \bigwedge_{a \text{ atom}} (a = 2)$
- there are no actions and goal constraints

Given a set S of P literals we use $T(S)$ for the sequence of state and actions s_0, A_1, s_1 where

- $\text{val}(s_0, w) = \text{val}(s_1, w) = 2$
- for every atom in P $\text{val}(s_0, a) = 0$ and

$$\text{val}(s_1, a) = \begin{cases} 1 & \text{if } a \in S \\ 0 & \text{if } \neg a \in S \\ 2 & \text{otherwise} \end{cases}$$

- $A_1 = \emptyset$

Just a simple consideration; given a program P and its transformation $T(P)$ all the initial states satisfy the rules since no rule is applicable to the first state.

We extend the notion of the translation T to rules of EDLP. We use the term $T(r)$ for the BPDDL rule obtained from the EDLP rule r .

With P^S we refer to the Gelfond-Lifschitz transformation [4].

Lemma 2. *if r is a rule of a disjunctive logic program then $S \models r \leftrightarrow T(S) \models T(r)$*

Proof. For definition of $T(S)$ for every literal l $S \models l \leftrightarrow T(S) \models \sigma(l)$ and $T(S) \models ((\text{at} - 1 \ w) = 2)$. For definition of \wedge, \vee, \leftarrow we have the thesis. \square

Lemma 3. *$S \models P^S$ iff $T(S) \models T(P)$*

Proof. Given S the rules in P can be divided in the following tree disjoint sets.

1. rules that have a (*not l*) term where $l \in S$
2. rules that do not have (*not l*) terms
3. the remaining rules

The lemma can be proven by induction on the number of rules in the EDLP program.

If $P = \emptyset$ then $S \models P = P^S$ and $T(S) \models T(P)$

Suppose that $P = P_1 \cup \{r\}$.

- if r is in the first set $r \notin P^S$. Then $S \models P^S$ iff $S \models P_1^S$. If S is not a model of P_1^S then for inductive hypothesis $T(S) \not\models T(P_1)$ and thus $T(S) \not\models T(P) = T(P_1) \wedge T(r)$. Conversely if $T(S) \models T(P_1)$ then $T(S) \not\models (\text{not } \sigma(l))$ if $l \in S$. Since in $T(r)$ precondition there is at least one of these literals then $T(S) \models T(r)$ and thus $T(S) \models T(P_1) \wedge T(r) = T(P)$.
- if r is in the second set then for lemma 2 $S \models \{r\} \leftrightarrow T(S) \models T(r)$. For inductive hypothesis $S \models P_1^S \leftrightarrow T(S) \models T(P_1)$ and thus $S \models P^S = P_1^S \cup \{r\} \leftrightarrow S \models P_1^S \wedge S \models \{r\} \leftrightarrow T(S) \models T(P_1) \wedge T(S) \models T(r) \leftrightarrow T(S) \models T(P_1) \wedge T(r) = T(P)$
- if r is in the third set then for all the terms (*not l*) in r $l \notin S$. Therefore $T(S) \models (\text{not } \sigma(l))$ and thus $T(S) \models r \leftrightarrow T(S) \models r'$ where r' is the rule r without the (*not l*) terms. Now since $P^S = P_1^S \cup \{r'\}$ we can derive that $S \models P^S \leftrightarrow T(S) \models T(P)$

\square

Theorem 4. *k -plan is Σ_2^P complete*

Proof. We reduce the existence of the answer set in propositional EDLP program to 1-plan.

Suppose that P has answer set S . Let us assume that $T(S) = s_0, \emptyset, s_1$ is not a plan. For lemma 3 $T(S) \models T(P)$. Since $T(S)$ is not a plan there is a critical situation between s_0, s_1 and therefore there exist s' s.t. $s_0, \emptyset, s' \models T(P)$ and $\Delta F(s_0, s') \subset \Delta F(s_0, s_1)$. If S' is a set of literals s.t. $T(S') = s_0, \emptyset, s'$ we have for

lemma 3 that S' is model of $P^{S'}$ but this is a contradiction since $S' \subset S$ and we supposed S answer set.

Suppose that there exist a plan s_0, \emptyset, s_1 for $T(P)$. Then there exist S such that $T(S) = s_0, \emptyset, s_1$. For lemma 3 we have that S is a model of P and therefore P has an answer set \square

Theorem 5. *if t_{max}, t_{min} are the maximum and minimum time reference in fluents and $t_{max}, |t_{min}|$ are polynomially bounded on the length of the encoding then plan is PSPACE complete.*

Proof. Every fluent can assume $O(2^n)$ values and thus the number of possible states is $O(2^n)$. Given two states s, s' if $t_{max}, |t_{min}|$ are polynomially bounded it is possible to compute in polynomial space if there exist A s.t. s, A, s' is a subsequence of a plan. This can be done generating non deterministically a polynomial number of states and actions and then solving the validity problem without checking the entailment of goal and initial constraints.

The plan problem can therefore be seen as the reachability problem. Since it is possible to define a state and check if two states are connected in polynomial space using a non-deterministic Turing machine then the entire plan problem can be solved in polynomial space using a non-deterministic Turing machine. Since $\text{NPSPACE} = \text{PSPACE}$, plan is in PSPACE.

The plan problem is PSPACE complete because A/BPDDL is more expressive than STRIPS [9]. A STRIPS program can be mapped into a A/BPDDL program straightforwardly and thus since plan in STRIPS is PSPACE complete the plan problem is PSPACE complete also in A/BPDDL. \square

Even if metric functions are used the optimization problem derived is in PSPACE. This is due to the fact that the metric function depends on the value of fluents in the final state and thus the metric value can be encoded in $O(n^2)$ space.

Towards the use of Simplification Rules in Intuitionistic Tableaux

Mauro Ferrari¹, Camillo Fiorentini² and Guido Fiorino³

¹ Dipartimento di Informatica e Comunicazione, Università degli Studi dell'Insubria,
`mauro.ferrari@uninsubria.it`

² Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
`fiorenti@dsi.unimi.it`

³ Dipartimento di Metodi Quantitativi per le Scienze Economiche ed Aziendali,
Università degli Studi di Piazza dell'Ateneo Nuovo 1, 20126 Milano, Italy
`guido.fiorino@unimib.it`

Abstract. By replacement it is meant the substitution of one or more occurrences of a formula with an equivalent one. In automated deduction this can be useful to reduce the search space. In tableau calculi for classical and modal logics this technique is known as *simplification* and consists in replacing a formula with a logical constant (\top or \perp). Recently, this idea has been applied to Intuitionistic Logic. This work in progress investigates further conditions on the applicability of Simplification in Intuitionistic Logic.

1 Introduction

It is well-known that the problem of deciding propositional Intuitionistic Logic is PSPACE-complete [7]. As a consequence, to perform automated theorem proving strategies reducing the search space are needed. A technique is *replacement*, that is the substitution of one or more occurrences of a formula with an equivalent one [3].

In the framework of tableau calculi for classical and modal logics, it has been described a technique, known as *simplification* [4], consisting in replacing every occurrence of a formula proved to be true with the logical constant \top and replacing a formula proved to be false with \perp . As an example, if A can be replaced with \top , we can simplify the formula $A \vee B$ in $\top \vee B$, which turns out to be equivalent to \top . In the tableau systems for classical logic the notions of provable and unprovable are codified by means of the *signs* \mathbf{T} and \mathbf{F} [6]. It is well-known that the sign (polarity) of a formula determines also the sign of every occurrence of its subformulas. Moreover, if the sign of a propositional variable occurring in a set of signed formulas is always \mathbf{T} (respectively \mathbf{F}), then such a variable is equivalent to \top (respectively \perp).

Also in the intuitionistic setting the sign of a formula determines the sign of every subformula. Differently from classical logic, the signs \mathbf{T} and \mathbf{F} are not dual, in particular $\mathbf{F}A$ does not imply that A is equivalent to \perp . Thus, simplifications can be performed only if further conditions are satisfied. In this

paper we provide some simplification rules. In Sections 4 and 5 we introduce the rules **T**-permanence, **T** \neg -permanence and **F**-permanence that allows to replace, under suitable conditions, propositional variables with \top and \perp . After the substitutions, we can apply the known boolean simplification rules and reduce the size of the set of formulas to be decided. The simplification rules we identify in this paper derive from a semantical analysis of validity of formulas in Kripke models.

We remark that our simplification rules are essentially independent from the tableau calculus at hand. Moreover, these rules are invertible. This means that we can apply them at any point of a proof search strategy without affecting its completeness. Finally, via the usual translation, these rules can also be applied in implementations based on sequent calculi.

This is a work in progress. We have implemented a Prolog prototype and we have compared its performances with PITP [1], which is, by now, the fastest prover for Intuitionistic Propositional Logic on the formulas of the benchmark ILTP Library [5]. As discussed in Section 6, the results are encouraging. We plan to continue the investigation by studying further criteria for variable replacement.

2 Notation and Preliminaries

We consider the propositional language \mathcal{L} based on a denumerable set of propositional variables \mathcal{PV} , the logical connectives $\neg, \wedge, \vee, \rightarrow$, the logical constants \top and \perp . We recall the main definitions about Kripke semantics (see e.g. [2] for more 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 poset with minimum ρ and the *forcing* relation \Vdash is a binary relation on $P \times \mathcal{PV}$ such that $\alpha \Vdash p$ and $\alpha \leq \beta$ imply $\beta \Vdash p$ (monotonicity property). The forcing relation extends to arbitrary formulas of \mathcal{L} as follows:

- $\alpha \Vdash \top$;
- $\alpha \not\Vdash \perp$;
- $\alpha \Vdash A \wedge B$ iff $\alpha \Vdash A$ and $\alpha \Vdash B$;
- $\alpha \Vdash A \vee B$ iff $\alpha \Vdash A$ or $\alpha \Vdash B$;
- $\alpha \Vdash A \rightarrow B$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash A$ implies $\beta \Vdash B$;
- $\alpha \Vdash \neg A$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \not\Vdash A$.

It is easy to prove that the monotonicity property holds for arbitrary formulas, i.e., $\alpha \Vdash A$ and $\alpha \leq \beta$ imply $\beta \Vdash A$. A formula A is *valid in a Kripke model* $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ iff $\rho \Vdash A$. It is well-known that Intuitionistic Propositional Logic **Int** coincides with the set of formulas valid in all Kripke models [2].

A tableau calculus \mathcal{T} works on *signed formulas*, namely formulas of \mathcal{L} prefixed with one of the signs **T** or **F**. The semantics of formulas extends to signed formulas. Given a Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$, $\alpha \in P$ and a signed formula H , α *realizes* H in \underline{K} ($\underline{K}, \alpha \triangleright H$) iff:

- $H \equiv \mathbf{TA}$ and $\alpha \Vdash A$;

- $H \equiv \mathbf{F}A$ and $\alpha \not\# A$.

\underline{K} realizes H ($\underline{K} \triangleright H$) iff $\underline{K}, \alpha \triangleright H$ for some $\alpha \in P$. H is *realizable* iff $\underline{K} \triangleright H$, for some Kripke model \underline{K} . The above definitions extend in the obvious way to sets Δ of signed formulas; for instance, $\underline{K}, \alpha \triangleright \Delta$ means that $\underline{K}, \alpha \triangleright H$, for every $H \in \Delta$. By definition, $A \in \mathbf{Int}$ iff $\mathbf{F}A$ is not realizable.

We remark that, by the monotonicity property the \mathbf{T} -signed formulas are *persistent*, namely: $\underline{K}, \alpha \triangleright \mathbf{T}A$ and $\alpha \leq \beta$ imply $\underline{K}, \beta \triangleright \mathbf{T}A$. On the other hand, \mathbf{F} -signed formulas are not persistent.

In general, a tableau calculus \mathcal{T} consists of a set of rules \mathcal{R} of the form:

$$\frac{\Delta}{\Delta_1 \mid \cdots \mid \Delta_n} r$$

where Δ (the *premise* of r) and $\Delta_1, \dots, \Delta_n$ (the *consequences* of r) are sets of signed formulas of \mathcal{L} . A *proof table* for Δ is a tree τ such that:

- the root of τ is Δ ;
- given a node Δ' of τ , the successors $\Delta_1, \dots, \Delta_n$ of Δ in τ are the consequences of an instance of a rule of \mathcal{R} having Δ' as premise.

A set Δ of signed formulas is *contradictory* if either $\mathbf{T}\perp \in \Delta$ or $\mathbf{F}\top \in \Delta$. When all the leaves of a proof table τ are contradictory, we say that τ is *closed*. A finite set of signed formulas Δ is *provable* in \mathcal{T} iff there exists a closed proof table for Δ .

Let r be a rule with premise Δ and consequences $\Delta_1, \dots, \Delta_n$. r is *sound* iff Δ realizable implies that there exists $k \in \{1, \dots, n\}$ such that Δ_k is realizable. r is *invertible* iff r is sound and, for every $1 \leq k \leq n$, Δ_k realizable implies Δ realizable.

In this paper we refer to the calculus **Tab** of Figure 1, but one can consider *any* complete calculus for **Int**. In the formulation of the rules, we use the notation Δ, H as a shorthand for $\Delta \cup \{H\}$. Writing Δ, H in the premise of a rule we assume that $H \notin \Delta$. **Tab** is inspired to the calculus in [1] which uses the sign \mathbf{F}_c besides the usual signs \mathbf{T} and \mathbf{F} . In **Tab** the rules for \mathbf{F}_c are translated by substituting $\mathbf{F}_c A$ with the equivalent signed formula $\mathbf{T}\neg A$. **Tab** turns out to be complete for **Int**, that is $A \in \mathbf{Int}$ iff $\{\mathbf{F}A\}$ is provable in **Tab**. More than this, the decision procedure discussed in [1] can be easily adapted to **Tab** preserving the time and space performances.

In this paper we provide invertible rules that can reduce the search space of the formula to be proved. This means that the decision procedure does not require to backtrack in the points where these rules are applied.

3 Replacement and simplification rules

First of all we recall the invertible rules introduced in [1]. Such rules allow us to simplify the signed formulas occurring in a node by replacing some of their

$$\begin{array}{c}
\frac{\Delta, \mathbf{T}(A \wedge B)}{\Delta, \mathbf{TA}, \mathbf{TB}}^{\mathbf{T}\wedge} \quad \frac{\Delta, \mathbf{F}(A \wedge B)}{\Delta, \mathbf{FA} \mid \Delta, \mathbf{FB}}^{\mathbf{F}\wedge} \quad \frac{\Delta, \mathbf{T}\neg(A \wedge B)}{\Delta_{\mathbf{T}}, \mathbf{T}\neg A \mid \Delta_{\mathbf{T}}, \mathbf{T}\neg B}^{\mathbf{T}\neg\wedge} \\
\frac{\Delta, \mathbf{T}(A \vee B)}{\Delta, \mathbf{TA} \mid \Delta, \mathbf{TB}}^{\mathbf{T}\vee} \quad \frac{\Delta, \mathbf{F}(A \vee B)}{\Delta, \mathbf{FA}, \mathbf{FB}}^{\mathbf{F}\vee} \quad \frac{\Delta, \mathbf{T}\neg(A \vee B)}{\Delta, \mathbf{T}\neg A, \mathbf{T}\neg B}^{\mathbf{T}\neg\vee} \\
\frac{\Delta, \mathbf{TA}, \mathbf{T}(A \rightarrow B)}{\Delta, \mathbf{TA}, \mathbf{TB}}^{\mathbf{T}\rightarrow Atom} \quad \text{with } A \text{ an atom} \\
\frac{\Delta, \mathbf{F}(A \rightarrow B)}{\Delta_{\mathbf{T}}, \mathbf{TA}, \mathbf{FB}}^{\mathbf{F}\rightarrow} \quad \frac{\Delta, \mathbf{T}\neg(A \rightarrow B)}{\Delta_{\mathbf{T}}, \mathbf{TA}, \mathbf{T}\neg B}^{\mathbf{T}\neg\rightarrow} \quad \frac{\Delta_{\mathbf{T}}, \mathbf{T}(A \rightarrow B)}{\Delta_{\mathbf{T}}, \mathbf{T}\neg A \mid \Delta_{\mathbf{T}}, \mathbf{TB}}^{\mathbf{T}\rightarrow\text{-special}} \\
\frac{\Delta, \mathbf{F}\neg A}{\Delta_{\mathbf{T}}, \mathbf{TA}}^{\mathbf{F}\neg} \quad \frac{\Delta, \mathbf{T}\neg\neg A}{\Delta_{\mathbf{T}}, \mathbf{TA}}^{\mathbf{T}\neg\neg} \\
\frac{\Delta, \mathbf{T}((A \wedge B) \rightarrow C)}{\Delta, \mathbf{T}(A \rightarrow (B \rightarrow C))}^{\mathbf{T}\rightarrow\wedge} \quad \frac{\Delta, \mathbf{T}(\neg A \rightarrow B)}{\Delta_{\mathbf{T}}, \mathbf{TA} \mid \Delta, \mathbf{TB}}^{\mathbf{T}\rightarrow\neg} \\
\frac{\Delta, \mathbf{T}((A \vee B) \rightarrow C)}{\Delta, \mathbf{T}(A \rightarrow p), \mathbf{T}(B \rightarrow p), \mathbf{T}(p \rightarrow C)}^{\mathbf{T}\rightarrow\vee} \quad \text{with } p \text{ a new atom} \\
\frac{\Delta, \mathbf{T}((A \rightarrow B) \rightarrow C)}{\Delta_{\mathbf{T}}, \mathbf{TA}, \mathbf{F}p, \mathbf{T}(p \rightarrow C), \mathbf{T}(B \rightarrow p) \mid \Delta, \mathbf{TC}}^{\mathbf{T}\rightarrow\rightarrow} \quad \text{with } p \text{ a new atom} \\
\frac{\Delta, \mathbf{TA}, \mathbf{FA}}{\Delta, \mathbf{T}\perp}^{contr_1} \quad \frac{\Delta, \mathbf{TA}, \mathbf{T}\neg A}{\Delta, \mathbf{T}\perp}^{contr_2} \\
\text{where } \Delta_{\mathbf{T}} = \{\mathbf{TA} \mid \mathbf{TA} \in \Delta\}
\end{array}$$

Fig. 1. The **Tab** calculus

subformulas either with \perp or \top . Given a signed formula H and two formulas A and B , we denote with $H[B/A]$ the signed formula obtained by replacing every occurrence of A in H with B . If Δ is a set of signed formulas, $\Delta[B/A]$ is the set of signed formula $H[B/A]$ such that $H \in \Delta$.

It is easy to prove the following facts:

Lemma 1. *Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model, let H be a signed formula, A a formula and $\alpha \in P$.*

- (i) *If $\underline{K}, \alpha \triangleright \mathbf{TA}$, then $\underline{K}, \alpha \triangleright H$ iff $\underline{K}, \alpha \triangleright H[\top/A]$.*
- (ii) *If $\underline{K}, \alpha \triangleright \mathbf{T}\neg A$, then $\underline{K}, \alpha \triangleright H$ iff $\underline{K}, \alpha \triangleright H[\perp/A]$.*

Let us consider the following rules:

$$\frac{\Delta, \mathbf{TA}}{\Delta[\top/A], \mathbf{TA}} \text{Replace-}\mathbf{T} \quad \frac{\Delta, \mathbf{T}\neg A}{\Delta[\perp/A], \mathbf{T}\neg A} \text{Replace-}\mathbf{T}\neg$$

By Lemma 1 it immediately follows that:

Theorem 1. *The rules Replace- \mathbf{T} and Replace- $\mathbf{T}\neg$ are invertible.*

The above rules are the intuitionistic version of the analogous rules for classical tableaux discussed in [4]. After having applied a replacement rule, we can simplify the formulas in the consequence of the rule by means of the invertible rules in Figure 2.

$$\begin{array}{cccc}
\frac{\Delta}{\Delta[\perp/A \wedge \perp]}^{S \wedge \perp} & \frac{\Delta}{\Delta[\perp/\perp \wedge A]}^{S \perp \wedge} & \frac{\Delta}{\Delta[A/A \wedge \top]}^{S \wedge \top} & \frac{\Delta}{\Delta[A/\top \wedge A]}^{S \top \wedge} \\
\frac{\Delta}{\Delta[A/A \vee \perp]}^{S \vee \perp} & \frac{\Delta}{\Delta[A/\perp \vee A]}^{S \perp \vee} & \frac{\Delta}{\Delta[\top/A \vee \top]}^{S \vee \top} & \frac{\Delta}{\Delta[\top/\top \vee A]}^{S \top \vee} \\
\frac{\Delta}{\Delta[\top/\perp \rightarrow A]}^{S \perp \rightarrow} & \frac{\Delta}{\Delta[\neg A/A \rightarrow \perp]}^{S \rightarrow \perp} & \frac{\Delta}{\Delta[A/\top \rightarrow A]}^{S \top \rightarrow} & \frac{\Delta}{\Delta[\top/A \rightarrow \top]}^{S \rightarrow \top} \\
\frac{\Delta}{\Delta[\perp/\neg \top]}^{S \neg \top} & \frac{\Delta}{\Delta[\top/\neg \perp]}^{S \neg \perp} & &
\end{array}$$

Fig. 2. Simplification rules

Now, we present the replacement rule for \mathbf{F} -signed formulas [1]. We remark that, differently from classical logic, where the meaning of the signs \mathbf{F} and \mathbf{T} are opposite, in Intuitionistic Logic \mathbf{F} and \mathbf{T} -signed formulas have an asymmetric behavior. In particular, as noted in Section 2, \mathbf{T} -signed formulas are persistent while \mathbf{F} -signed formulas are not. Due to this asymmetry the replacement rule for \mathbf{F} -signed formulas involves a notion of *partial substitution* which is weaker than the “full” substitution since the substitution does not act on propositional variables under the scope of implication or negation. Formally, given the formulas Z , A and B , we denote with $Z\{B/A\}$ the *partial substitution of A with B in Z* defined as follows:

- if $Z = A$, then $Z\{B/A\} = B$;
- if $Z = (X \odot Y)$, then $Z\{B/A\} = X\{B/A\} \odot Y\{B/A\}$, where $\odot \in \{\wedge, \vee\}$;
- if $Z = X \rightarrow Y$ or $Z = \neg X$ or Z is a propositional variable different from A , then $Z\{B/A\} = Z$.

We remark that differently from the “full” substitution rule denoted by square brackets, partial substitutions do not apply to subformulas with main connective \rightarrow or \neg . For instance, while $((X \rightarrow Y) \vee Y)[\perp/Y]$ produces $(X \rightarrow \perp) \vee \perp$, the partial substitution $((X \rightarrow Y) \vee Y)\{Y/\perp\}$ yields $(X \rightarrow Y) \vee \perp$. Given a signed formula $\mathcal{S}Z$ with $\mathcal{S} \in \{\mathbf{T}, \mathbf{F}\}$, we denote with $\mathcal{S}Z\{B/A\}$ the signed formula $\mathcal{S}(Z\{B/A\})$. Given a set of signed formulas Δ , $\Delta\{B/A\}$ is the set containing $K\{B/A\}$ for every $K \in \Delta$.

It is easy to prove the following result [1]:

Lemma 2. *Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model, let $\alpha \in P$ and let H and $\mathbf{F}A$ be two signed formulas. If $\underline{K}, \alpha \triangleright \mathbf{F}A$, then $\underline{K}, \alpha \triangleright H$ iff $\underline{K}, \alpha \triangleright H\{\perp/A\}$.*

Now, let us consider the rule:

$$\frac{\Delta, \mathbf{F}A}{\Delta\{\perp/A\}, \mathbf{F}A} \text{Replace-}\mathbf{F}$$

By the above Lemma 2 it immediately follows that:

Theorem 2. *The rule Replace- \mathbf{F} is invertible.*

4 Propositional variables with constant sign

The replacement rules of Section 3 can be applied whenever a signed formula $\mathbf{T}A$, $\mathbf{T}\neg A$ or $\mathbf{F}A$ occurs in Δ . These rules together with the simplification rules in Figure 2 can considerably reduce the search space, as witnessed by the performances of PITP [1]. In this section, we exploit some conditions under which we can apply the replacement rules of Section 3 also to sets of signed formulas not explicitly containing $\mathbf{T}A$, $\mathbf{T}\neg A$ or $\mathbf{F}A$. The applicability of replacement rules can be foreseen evaluating the polarity of the propositional variables occurring in a signed formula.

Given a signed formula H and a propositional variable p , we introduce the notions $p \preceq^+ H$ (p *positively* occurs in H) and $p \preceq^- H$ (p *negatively* occurs in H). Hereafter we use \mathcal{S} to denote either \mathbf{T} or \mathbf{F} . The definition of $p \preceq^l H$, with $l \in \{+, -\}$ is by induction on the structure of H :

- $p \preceq^- \mathbf{F}p$ and $p \preceq^+ \mathbf{T}p$
- $p \preceq^l \mathcal{S}\top$ and $p \preceq^l \mathcal{S}\perp$
- $p \preceq^l \mathcal{S}q$, where q is any propositional variable such that $q \neq p$
- $p \preceq^l \mathcal{S}(A \odot B)$ iff $p \preceq^l \mathcal{S}A$ and $p \preceq^l \mathcal{S}B$, where $\odot \in \{\wedge, \vee\}$
- $p \preceq^l \mathbf{F}(A \rightarrow B)$ iff $p \preceq^l \mathbf{T}A$ and $p \preceq^l \mathbf{F}B$
- $p \preceq^l \mathbf{T}(A \rightarrow B)$ iff $p \preceq^l \mathbf{F}A$ and $p \preceq^l \mathbf{T}B$
- $p \preceq^l \mathbf{F}\neg A$ iff $p \preceq^l \mathbf{T}A$
- $p \preceq^l \mathbf{T}\neg A$ iff $p \preceq^l \mathbf{F}A$.

Given a set of signed formulas Δ , $p \preceq^l \Delta$ iff, for every $H \in \Delta$, $p \preceq^l H$.

Let us consider the following constructions over Kripke models. Given $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ and a propositional variable p :

- $\underline{K}_p = \langle P, \leq, \rho, \Vdash' \rangle$, where $\Vdash' = \Vdash \cup \{(\alpha, p) \mid \alpha \in P\}$;
- $\underline{K}_{\neg p} = \langle P, \leq, \rho, \Vdash' \rangle$, where $\Vdash' = \Vdash \setminus \{(\alpha, p) \mid \alpha \in P\}$.

Note that, for every $\alpha \in P$, $\underline{K}_p, \alpha \triangleright \mathbf{T}p$ and $\underline{K}_{\neg p}, \alpha \triangleright \mathbf{T}\neg p$. It is easy to prove the following facts:

Lemma 3. *Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model, let H be a signed formula and let p be a propositional variable.*

- (1) If $p \preceq^+ H$ then, for every $\alpha \in P$, $\underline{K}, \alpha \triangleright H$ implies $\underline{K}_p, \alpha \triangleright H$.
(2) If $p \preceq^- H$ then, for every $\alpha \in P$, $\underline{K}, \alpha \triangleright H$ implies $\underline{K}_{-p}, \alpha \triangleright H$.

Proof. The proof easily goes by structural induction on H . As an example, we prove Point (1) for $H = \mathbf{T}(A \rightarrow B)$. Let us assume that $\underline{K}, \alpha \triangleright \mathbf{T}(A \rightarrow B)$ and let β be any element of P such that $\alpha \leq \beta$ and $\underline{K}_p, \beta \triangleright \mathbf{T}A$. To prove $\underline{K}_p, \alpha \triangleright \mathbf{T}(A \rightarrow B)$ we have to show that $\underline{K}_p, \beta \triangleright \mathbf{T}B$. Since $p \preceq^+ \mathbf{F}A$ we have $\underline{K}, \beta \not\triangleright \mathbf{F}A$, otherwise, by the induction hypothesis, $\underline{K}_p, \beta \triangleright \mathbf{F}A$, in contradiction with the above assumption. Thus $\underline{K}, \beta \triangleright \mathbf{T}A$ and, since $\underline{K}, \alpha \triangleright \mathbf{T}(A \rightarrow B)$ and $\alpha \leq \beta$, it follows that $\underline{K}, \beta \triangleright \mathbf{T}B$. Since $p \preceq^+ \mathbf{T}B$, by the induction hypothesis $\underline{K}_p, \beta \triangleright \mathbf{T}B$. \square

Now, let us consider the following rules:

$$\frac{\Delta}{\Delta[\top/p]} \mathbf{T}\text{-permanence} \quad \text{where } p \preceq^+ \Delta$$

$$\frac{\Delta}{\Delta[\perp/p]} \mathbf{T}\neg\text{-permanence} \quad \text{where } p \preceq^- \Delta$$

Essentially these rules state that, if $p \preceq^+ \Delta$ ($p \preceq^- \Delta$), we can consistently replace every occurrence of p in Δ with \top (\perp , respectively). From the previous lemma it follows that:

Theorem 3. *The rules \mathbf{T} -permanence and $\mathbf{T}\neg$ -permanence are invertible.*

Proof. Let us consider the case of the rule \mathbf{T} -permanence. We have to show that Δ is realizable iff $\Delta[\top/p]$ is realizable. Let us assume that Δ is realizable. Then, there exists a Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ and $\alpha \in P$ such that $\underline{K}, \alpha \triangleright \Delta$. Since $p \preceq^+ \Delta$, by Point (1) of Lemma 3, $\underline{K}_p, \alpha \triangleright \Delta$ and, by definition of its forcing relation, $\underline{K}_p, \alpha \triangleright \mathbf{T}p$. It follows that $\Delta, \mathbf{T}p$ is realizable and, by the soundness of the rule Replace- \mathbf{T} (Lemma 1(i)), we get that $\Delta[\top/p]$ is realizable. Conversely, let us suppose that $\Delta[\top/p]$ is realizable and let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model and $\alpha \in P$ such that $\underline{K}, \alpha \triangleright \Delta[\top/p]$. Since p does not occur in $\Delta[\top/p]$, it holds that $p \preceq^+ \Delta[\top/p]$. By Point (1) of Lemma 3, $\underline{K}_p, \alpha \triangleright \Delta[\top/p]$. Since $\underline{K}_p, \alpha \triangleright \mathbf{T}p$, by Lemma 1(i) $\underline{K}_p, \alpha \triangleright \Delta$, hence Δ is realizable. The case of the rule $\mathbf{T}\neg$ -permanence is similar. \square

We show an example of derivation where \mathbf{T} -permanence works. Let

$$A = ((p \rightarrow q) \wedge ((\neg\neg r \rightarrow s) \rightarrow t) \wedge ((\neg\neg s \rightarrow t) \rightarrow p)) \rightarrow q$$

The formula A is classically valid but not intuitionistically valid¹. To decide A , we have to search for a proof of $\mathbf{F}A$. Since $r \preceq^+ \mathbf{F}A$, we can apply the rule \mathbf{T} -permanence to get the set

$$\Delta_1 = \{ \mathbf{F}((p \rightarrow q) \wedge ((\neg\neg\top \rightarrow s) \rightarrow t) \wedge ((\neg\neg s \rightarrow t) \rightarrow p)) \rightarrow q \}$$

¹ A is the formula SYJ211+1.001 of ILTP Library [5].

and, simplifying $\neg\neg\top \rightarrow s$ to s with the rules of Figure 2, we get:

$$\Delta_2 = \{ \mathbf{F}((p \rightarrow q) \wedge (s \rightarrow t) \wedge ((\neg\neg s \rightarrow t) \rightarrow p)) \rightarrow q \}$$

Now, we can only proceed applying the rules $\mathbf{F} \rightarrow$ and $\mathbf{T} \wedge$ and we get:

$$\Delta_3 = \{ \mathbf{T}(p \rightarrow q), \mathbf{T}(s \rightarrow t), \mathbf{T}((\neg\neg s \rightarrow t) \rightarrow p), \mathbf{F}q \}$$

The only rule applicable to Δ_3 is the branching rule $\mathbf{T} \rightarrow \rightarrow$ and we obtain the nodes

$$\begin{aligned} \Delta_4 &= \{ \mathbf{T}(p \rightarrow q), \mathbf{T}(s \rightarrow t), \mathbf{T}\neg\neg s, \mathbf{F}a, \mathbf{T}(a \rightarrow p), \mathbf{T}(t \rightarrow a) \} \\ \Delta_5 &= \{ \mathbf{T}(p \rightarrow q), \mathbf{T}(s \rightarrow t), \mathbf{T}p, \mathbf{F}q \} \end{aligned}$$

where a is a new propositional variable. Applying rules $\mathbf{T} \rightarrow$ *Atom* and *contr*₁ to Δ_5 we get a contradictory set. As for Δ_4 , we have that $q \preceq^+ \Delta_4$, hence, applying \mathbf{T} -permanence we get

$$\Delta_6 = \{ \mathbf{T}(p \rightarrow \top), \mathbf{T}(s \rightarrow t), \mathbf{T}\neg\neg s, \mathbf{F}a, \mathbf{T}(a \rightarrow p), \mathbf{T}(t \rightarrow a) \}$$

Simplifying we obtain

$$\Delta_7 = \{ \mathbf{T}\top, \mathbf{T}(s \rightarrow t), \mathbf{T}\neg\neg s, \mathbf{F}a, \mathbf{T}(a \rightarrow p), \mathbf{T}(t \rightarrow a) \}$$

Now, $p \preceq^+ \Delta_7$, hence by \mathbf{T} -permanence and simplification, $\mathbf{T}(a \rightarrow p)$ reduces to $\mathbf{T}\top$ and we get

$$\Delta_8 = \{ \mathbf{T}\top, \mathbf{T}(s \rightarrow t), \mathbf{T}\neg\neg s, \mathbf{F}a, \mathbf{T}(t \rightarrow a) \}$$

Now, we can we can only apply the $\mathbf{T}\neg\neg$ rule and we obtain the set

$$\Delta_9 = \{ \mathbf{T}\top, \mathbf{T}(s \rightarrow t), \mathbf{T}s, \mathbf{T}(t \rightarrow a) \}$$

which is clearly not contradictory. Since in our derivation there is no backtrack point in the proof table, we conclude that $\mathbf{F}A$ is not provable.

If we disregard the rule \mathbf{T} -permanence, we have to begin the proof of $\mathbf{F}A$ by applying the rules $\mathbf{F} \rightarrow$ and $\mathbf{T} \wedge$ obtaining the set

$$\{ \mathbf{T}(p \rightarrow q), \mathbf{T}((\neg\neg r \rightarrow s) \rightarrow t), \mathbf{T}((\neg\neg s \rightarrow t) \rightarrow p), \mathbf{F}q \}$$

At this point we have a backtracking point since the rule $\mathbf{T} \rightarrow \rightarrow$ can be applied to $\mathbf{T}((\neg\neg r \rightarrow s) \rightarrow t)$ or to $\mathbf{T}((\neg\neg s \rightarrow t) \rightarrow p)$.

We discuss in Section 6 the impact of the permanence rules on performances of PITP.

5 The rule \mathbf{F} -permanence

In the previous section we have seen how the polarity of a propositional variable p can be used to predict if $\mathbf{T}p$ or $\mathbf{T}\neg p$ can be added to a deduction so to activate

replacement and simplification rules. In this section we introduce a similar rule allowing us to predict when a **F**-signed propositional variable can be added to a deduction. Also in this case, such a prediction can be used to activate simplifications by applying the rule Replace-**F**. In this case the applicability of replacement rules can be foreseen evaluating if a propositional variable *weakly negatively occurs* in a set of signed formulas.

Given a propositional variable p and a signed formula H , let us define the relation $p \preceq_w^- H$ (p weakly negatively occurs in H) by induction on the structure of H :

- $p \preceq_w^- \mathcal{S}\top$ and $p \preceq_w^- \mathcal{S}\perp$
- $p \preceq_w^- \mathbf{F}A$ and $p \preceq_w^- \mathbf{T}\neg A$ for every A
- $p \preceq_w^- \mathbf{T}q$ if $q \neq p$
- $p \preceq_w^- \mathbf{T}(A \odot B)$ iff $p \preceq_w^- \mathbf{T}A$ and $p \preceq_w^- \mathbf{T}B$, where $\odot \in \{\wedge, \vee\}$
- $p \preceq_w^- \mathbf{T}(A \rightarrow B)$ iff $p \preceq_w^- \mathbf{T}B$.

We remark that $p \preceq^- H$ implies $p \preceq_w^- H$; on the other hand, the \preceq_w^- relation permits weaker simplifications. Given a set Δ of signed formulas, we say that $p \preceq_w^- \Delta$ iff, for every $H \in \Delta$, $p \preceq_w^- H$.

Now let us consider the following construction over Kripke models. Given $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ and a propositional variable p , let $\rho' \notin P$. By \underline{K}_p^w we denote the Kripke model $\langle P', \leq', \rho', \Vdash' \rangle$ such that:

$$\begin{aligned} P' &= P \cup \{\rho'\} & \leq' &= \leq \cup \{(\rho', \alpha) \mid \alpha \in P'\} \\ \Vdash' &= \Vdash \cup \{(\rho', q) \mid \rho \Vdash q \text{ and } q \neq p\} \end{aligned}$$

Note that, for every signed formula H , $\underline{K}, \rho \triangleright H$ iff $\underline{K}_p^w, \rho \triangleright H$.

Lemma 4. *Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model, let H be a signed formula and let p be a propositional variable such that $p \preceq_w^- H$. Then, $\underline{K}, \rho \triangleright H$ implies $\underline{K}_p^w, \rho' \triangleright H$.*

Proof. Let us assume $\underline{K}, \rho \triangleright H$. We prove $\underline{K}_p^w, \rho' \triangleright H$ by induction on H . If $H = \mathbf{F}A$ then $\underline{K}_p^w, \rho \triangleright \mathbf{F}A$, hence $\underline{K}_p^w, \rho' \triangleright \mathbf{F}A$. If $H = \mathbf{T}q$ then $q \neq p$ (indeed $p \preceq_w^- \mathbf{T}p$ does not hold) and hence, by definition of \Vdash' , $\underline{K}_p^w, \rho' \triangleright H$. The cases $H = \mathbf{T}(A \wedge B)$ and $H = \mathbf{T}(A \vee B)$ easily follow by the induction hypothesis. Let $H = \mathbf{T}(A \rightarrow B)$; since $\underline{K}, \rho \triangleright H$, we have $\underline{K}_p^w, \rho \triangleright \mathbf{T}(A \rightarrow B)$. To prove that $\underline{K}_p^w, \rho' \triangleright \mathbf{T}(A \rightarrow B)$, it only remains to show that $\underline{K}_p^w, \rho' \triangleright \mathbf{T}A$ implies $\underline{K}_p^w, \rho' \triangleright \mathbf{T}B$. If $\underline{K}_p^w, \rho' \triangleright \mathbf{T}A$, then $\underline{K}_p^w, \rho \triangleright \mathbf{T}A$, and this implies $\underline{K}, \rho \triangleright \mathbf{T}A$. Since $\underline{K}, \rho \triangleright \mathbf{T}(A \rightarrow B)$, we get $\underline{K}, \rho \triangleright \mathbf{T}B$. Since $p \preceq_w^- \mathbf{T}B$, by induction hypothesis we conclude $\underline{K}_p^w, \rho' \triangleright \mathbf{T}B$. The case $H = \mathbf{T}\neg A$ is similar. \square

Moreover, it is easy to prove:

Lemma 5. *Let H be a signed formula and p a propositional variable. If $p \preceq_w^- H$ then $p \preceq_w^- H\{\perp/p\}$.*

Proof. The proof is by induction on the structure of H . If $H = \mathbf{F}A$ or $H = \mathbf{T}\neg A$ or $H = \mathbf{T}q$ with q a propositional variable, the assertion immediately follows. If $H = \mathbf{T}(A \wedge B)$, then $p \preceq_w^- \mathbf{T}A$ and $p \preceq_w^- \mathbf{T}B$. By induction hypothesis, $p \preceq_w^- \mathbf{T}A\{\perp/p\}$ and $p \preceq_w^- \mathbf{T}B\{\perp/p\}$. Since $\mathbf{T}(A \wedge B)\{\perp/p\} = \mathbf{T}(A\{\perp/p\} \wedge B\{\perp/p\})$, it follows that $p \preceq_w^- H\{\perp/p\}$. The other cases are similar.

Now, let us consider the rule:

$$\frac{\Delta}{\Delta\{\perp/p\}} \mathbf{F}\text{-permanence} \quad \text{where } p \preceq_w^- \Delta$$

Along the lines of the proof of Theorem 3, by lemmas 4 and 5 we get:

Theorem 4. *The rule \mathbf{F} -permanence is invertible.*

As an application of the above rule, let us consider the set

$$\Delta_1 = \{ \mathbf{T}(p \vee q), \mathbf{F}(q \wedge r), \mathbf{F}(p \wedge r), \mathbf{F}(r \rightarrow q) \}$$

First of all, we notice that the propositional variables p , q and r do not occur in Δ_1 with constant sign, that is $x \not\preceq^+ \Delta_1$ and $x \not\preceq^- \Delta_1$ for every $x \in \{p, q, r\}$. Thus, the replacement rules discussed in the previous sections cannot be applied to Δ_1 . On the other hand $r \preceq_w^- \Delta_1$, hence we can apply \mathbf{F} -permanence and we get the set

$$\Delta_2 = \{ \mathbf{T}(p \vee q), \mathbf{F}(q \wedge \perp), \mathbf{F}(p \wedge \perp), \mathbf{F}(r \rightarrow q) \}$$

Applying the boolean simplifications to Δ_2 we get

$$\Delta_3 = \{ \mathbf{T}(p \vee q), \mathbf{F}\perp, \mathbf{F}(r \rightarrow q) \}$$

Now, since $p \preceq^+ \Delta_3$, by applying the rules \mathbf{T} -permanence and the simplifications rules we obtain the set

$$\Delta_4 = \{ \mathbf{T}\top, \mathbf{F}\perp, \mathbf{F}(r \rightarrow q) \}$$

which is non contradictory. Since the proof does not contain any branch we conclude that Δ_1 is not provable.

6 Timings

We devote this section to discuss the impact of the rules presented above. To this aim we have developed a Prolog implementation of the calculus of PITP [1] and we have tested how the performances are affected by the above rules. In particular, we compare the following theorem provers:

- BPPI (Basic Prolog Prover for Intuitionism) is the implementation of the calculus **Tab** extended with the rules Replace- \mathbf{T} , Replace- $\mathbf{T}\neg$ and the rules of Figure 2.

Prover	0-1s	1-10s	10-100s	100-600s	>600s
BPPI	1025(101.4)	51(158.9)	11(281.8)	4(972.3)	9(n.a.)
IPPI	856(165.4)	227(721.0)	12(416.2)	4(1745.5)	1(953.97)
EPPI	859(161.9)	226(710.7)	11(392.7)	4(1607.9)	0(0.0)
PITP	1085(11.3)	7(19.9)	3(165.3)	2(409.2)	3(20900.4)

Fig. 3. Timings

- IPPI (Intermediate Prolog Prover for Intuitionism) extends BPPI with the rules \mathbf{T} -permanence and $\mathbf{T}\neg$ -permanence.
- EPPI (Efficient Prolog Prover for Intuitionism) extends IPPI with the rule \mathbf{F} -permanence.

Experiments² have been carried out along the lines of [5] and their results are summarized in Figure 3. The experiments have been performed on random generated formulas with 1024 connectives and a number of variables ranging from 1 to 1024. In every entry we indicate the number of formulas decided in the specified time range (expressed in seconds) and between brackets we put the total time required to decide them. The last row of the table refers to PITP [1] which is written in C++.

The results emphasize that for formulas decidable in few steps, the overhead of computing the variables with constant signs slows-down the prover, but when the formula to be decided requires a lot of computation, then the optimization is effective. As a matter of fact EPPI decides all the formulas within 10 minutes. The worth of our optimizations is confirmed when compared with the performances of PITP.

To conclude, in this paper we have presented a preliminary study about simplification rules in tableau calculi for Intuitionistic Logic. First of all, we remark that this topic has been scarcely studied in the literature, while it is central in classical theorem proving from its very beginning. Indeed, as far as we know, the rules presented in [4] are the only simplification rules for non classical logics described in the literature. These are the rules implemented in the calculus of PITP [1] and in our basic Prolog implementation BPPI. Now, considering the impact that the simplification rules have in implementation, we think that this topic deserves a deeper investigation.

References

1. A. Avellone, G. Fiorino, and U. Moscato. Optimization techniques for propositional intuitionistic logic and their implementation. *Theoretical Computer Science*, 409(1):41–58, 2008.

² We used a 3.00GHz Intel Xeon CPU computer with 2MB cache size and 2GB RAM.

2. A. Chagrov and M. Zakharyashev. *Modal Logic*. Oxford University Press, Oxford, 1997.
3. S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.
4. F. Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In Harrie C. M. de Swart, editor, *TABLEAUX*, volume 1397 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1998.
5. T. Rath, J. Otten, and C. Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 31:261–271, 2007.
6. R.M. Smullyan. *First-Order Logic*. Springer, Berlin, 1968.
7. R. Statman. Intuitionistic logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

Exploiting Semantic Technology in Computational Logic-based Service Contracting

Marco Alberti¹, Massimiliano Cattafi², Marco Gavanelli², and Evelina Lamma²

¹ CENTRIA, Universidade Nova de Lisboa
Quinta da Torre - 2825-144 Caparica (Portugal)

m.alberti@fct.unl.pt

² ENDIF, Università di Ferrara
Via Saragat, 1 - 44100 Ferrara (Italy)
massimiliano.cattafi@unife.it
marco.gavanelli@unife.it
evelina.lamma@unife.it

Abstract. Dynamic composition of web services requires an automated step of contracting, i.e., the computation of a possibly fruitful interaction between two (or more) services, based on their policies and goals. In previous work, the *SCIFF* abductive logic language was used to represent the services' policies, and the associated proof procedure to perform the contracting.

In this paper, we build on that work in order to exploit the results of the Description Logics research area to represent domain specific knowledge, either by importing the knowledge encoded in an ontology into a *SCIFF* knowledge base, or by interfacing the *SCIFF* proof procedure to an existing ontological reasoner.

1 Introduction

Service Oriented Architectures, exploiting the Internet as a means of communication, are emerging as a simple and effective paradigm for distributed application development. Heterogeneous entities, in terms of hardware and software settings, can interoperate effectively following well established communication standards.

Service providers can expose their policies, in order for potential customers to evaluate the feasibility of a fruitful interaction. Such an evaluation can be performed manually, if the service's interface is bound not to change over time.

However, a more promising approach is for customers to choose the appropriate service provider at run-time, based on the service provider's exposed policies.

This approach requires a reasoning engine that reasons on the provider and customer's goals and policies, in order to devise a sequence of actions that lets both achieve their goals, while respecting their policies.

In [1], Alberti et al. proposed a contracting architecture based on the *SCIFF* abductive logic framework [2]. In that work, the policies were expressed by means of integrity constraints, and the domain knowledge was expressed in *SCIFF*'s logic clauses. The sound and complete *SCIFF* proof procedure was employed to find, if possible, a course of action that was satisfactory for both.

However, in a practical perspective, we envisage a possible improvement in representing the domain specific knowledge exploiting the vast body of results from the Knowledge Representation field and, in particular, Description Logics. In this way, we can address the Ontology layer of the Semantic Web stack.

In this paper, we propose two different approaches to let *SCIFF* access existing knowledge, expressed by means of an ontology:

- by translating (part of) an ontology into *SCIFF* clauses, which can subsequently be handled by the *SCIFF* proof procedure in the usual way;
- by interfacing *SCIFF* with ontological reasoners, distinguishing syntactically the ontology-related predicates and delegating their computation to the external reasoners.

This approach bears two main practical advantages: first, it makes the knowledge encoded in ontologies available to *SCIFF*, and second (with the second approach) it takes advantage of the formal properties enjoyed by the existing ontological reasoners for the associated computational tasks, in particular concerning decidability and efficiency.

The paper is structured as follows: in Sect. 2 we briefly recall the *SCIFF* approach to contracting, in Sect. 3 we show how the relevant domain knowledge can be expressed by means of an ontology, and in Sect. 4 we describe the two different approaches to access ontologies from *SCIFF*: importing an ontology into a *SCIFF* knowledge base (Sect. 4.1), and interfacing the *SCIFF* proof procedure with the Pellet [3] ontological reasoner (Sect. 4.2). A brief discussion of related work and conclusions follow.

2 Contracting with *SCIFF*

In this section, we briefly recall the *SCIFF*-based contracting framework [1].

2.1 The *SCIFF* framework

A web service's policy is defined, in the *SCIFF* language, as an abductive logic program (ALP). An ALP is defined as the triplet $\langle KB_S, \mathcal{A}, IC \rangle$, where KB_S is a logic program in which the clauses can contain special atoms, that belong to the set \mathcal{A} and are called *abducibles*. Such atoms are not defined by means of clauses in the KB_S , and, as such, they cannot be proved: their truth value can be only hypothesized. In order to avoid unconstrained hypotheses, a set of *integrity constraints* (IC) must always be satisfied. Integrity constraints, in our language, are in the form of implications, and can relate abducible literals, defined literals, as well as constraints with Constraint Logic Programming [4] semantics.

In particular, the integrity constraints can relate the web services' information exchanges with the expected input from peer web services. The possible relations can be of various types, and include temporal relations, such as deadlines, linear constraints, inequalities and inequalities, all defined by means of constraints. The definitions stated in this way are then used to make assumptions on the possible evolutions of the interaction.

A web service can reason about *happened* events, denoted with $\mathbf{H}(S, R, M, T)$, meaning that a Sender S sends message M to a Receiver R at some time T . All the parameters can be logical terms, or variables, possibly subject to constraints. Also, the policy might describe that a peer should provide some input, by sending a message. Such *expectation* of the incoming message is represented as $\mathbf{E}(S, R, M, T)$, and it will be satisfied in case the message, in the form of \mathbf{H} atom, actually arrives. In some cases, the *expectation* could be *negative*, as in $\mathbf{EN}(S, R, M, T)$: the policy says in this case that a matching \mathbf{H} event would be unwelcome, as it would contradict other assumptions.

For example, an integrity constraint can state that, if a peer web service sends me a request, then I will accept:

$$\mathbf{H}(S, me, request(X), T) \rightarrow \mathbf{H}(me, S, accept(X), T_a).$$

Another rule can say that, if I send a request, I expect the peer either to accept or to refuse, within 5 time units:

$$\begin{aligned} \mathbf{H}(me, S, request(X), T) \rightarrow & \mathbf{E}(S, me, accept(X), T_a), T_a < T + 5 \\ & \mathbf{E}(S, me, refuse(X), T_r), T_r < T + 5 \end{aligned}$$

Once the specification of a policy is defined by means of integrity constraints and expectations, they are processed by the *SCIFF* proof procedure, the operational counterpart of the language. The proof-procedure performs abductive reasoning, i.e., it can make assumptions and generate hypotheses. In particular, it can hypothesize that some message will be sent, or that some expectation will be raised. In case the expectations are matched by corresponding events, the abductive process succeeds, otherwise the current branch will fail, and another alternative will be selected (if there exists one). In this way, the *SCIFF* proof-procedure is able to find if there exists at least a set of events that satisfies a given ALP, and provides in output both the abduced events and expectations.

In contracting applications, the potential interacting services expose their policies (in a format resulting from the extension of RuleML), and *SCIFF* operates on the merging of the policies. A successful computation yields a sequence of actions that satisfies all the parties' policies.

2.2 A contracting scenario

In [1] the *SCIFF* framework is applied to contracting in an e-commerce scenario, which we extend in this paper in order to demonstrate our approach to integration.

The two actors are *eShop* (which wants to sell a device) and *alice* (a potential customer for that device), each with policies expressed as *SCIFF* ICs. *SCIFF* is used as a reasoner in a component that acts as a mediator, considering the actors' policies and trying to devise a course of action that will let both reach their goals. The two actors' policies, expressed in *SCIFF* integrity constraints, are as follows.

alice's policy. "If the shop asks me to pay cash, I will, but if the shop asks me to pay by credit card, I will require evidence of the shop's affiliation to the Better Business Bureau."

$$\begin{aligned}
& \mathbf{H}(\text{tell}(\text{Shop}, \text{alice}, \text{ask}(\text{pay}(\text{Item}, \text{cc}))), \text{Ta}) \rightarrow \\
& \mathbf{H}(\text{tell}(\text{alice}, \text{Shop}, \text{request_guar}(\text{BBB})), \text{Trg}) \wedge \\
& \mathbf{E}(\text{tell}(\text{Shop}, \text{alice}, \text{give_guar}(\text{BBB})), \text{Tg}) \wedge \text{Tg} > \text{Trg} \wedge \text{Trg} > \text{Ta}. \\
\\
& \mathbf{H}(\text{tell}(\text{Shop}, \text{alice}, \text{ask}(\text{pay}(\text{Item}, \text{cc}))), \text{Ta}) \wedge \\
& \mathbf{H}(\text{tell}(\text{Shop}, \text{alice}, \text{give_guar}(\text{BBB})), \text{Tg}) \rightarrow \\
& \mathbf{H}(\text{tell}(\text{alice}, \text{Shop}, \text{pay}(\text{Item}, \text{cc})), \text{Tp}) \wedge \text{Tp} > \text{Ta} \wedge \text{Tp} > \text{Tg}. \\
\\
& \mathbf{H}(\text{tell}(\text{Shop}, \text{alice}, \text{ask}(\text{pay}(\text{Item}, \text{cash}))), \text{Ta}) \rightarrow \\
& \mathbf{H}(\text{tell}(\text{alice}, \text{eShop}, \text{pay}(\text{Item}, \text{cash})), \text{Tr}) \wedge \text{Ta} < \text{Tr}.
\end{aligned} \tag{1}$$

eShop's policy. “If an acceptable customer requests an item from me, then I expect the customer to pay for the item with an acceptable means of payment. If the customer is not acceptable, I will inform him/her of the failure. If an acceptable customer pays with an acceptable means of payment, I will deliver the item. If a customer requests evidence of my affiliation to the Better Business Bureau (BBB), I will provide it.”

$$\begin{aligned}
& \mathbf{H}(\text{tell}(\text{Customer}, \text{eShop}, \text{request}(\text{Item})), \text{Tr}) \wedge \text{accepted_payment}(\text{How}) \rightarrow \\
& \text{accepted_customer}(\text{Customer}) \wedge \mathbf{H}(\text{tell}(\text{eShop}, \text{Customer}, \text{ask}(\text{pay}(\text{Item}, \text{How}))), \text{Ta}) \wedge \\
& \mathbf{E}(\text{tell}(\text{Customer}, \text{eShop}, \text{pay}(\text{Item}, \text{How})), \text{Tp}) \wedge \text{Tp} > \text{Ta} \wedge \text{Ta} > \text{Tr} \\
& \vee \text{rejected_customer}(\text{Customer}) \wedge \mathbf{H}(\text{tell}(\text{eShop}, \text{Customer}, \text{inform}(\text{fail})), \text{Ti}) \wedge \text{Ti} > \text{Tr}. \\
\\
& \mathbf{H}(\text{tell}(\text{Customer}, \text{eShop}, \text{pay}(\text{Item}, \text{How})), \text{Tp}) \\
& \wedge \text{accepted_customer}(\text{Customer}) \wedge \text{accepted_payment}(\text{How}) \rightarrow \\
& \mathbf{H}(\text{tell}(\text{eShop}, \text{Customer}, \text{deliver}(\text{Item})), \text{Td}) \wedge \text{Td} > \text{Tp}. \\
\\
& \mathbf{H}(\text{tell}(\text{Customer}, \text{eShop}, \text{request_guar}(\text{BBB})), \text{Trg}) \rightarrow \\
& \mathbf{H}(\text{tell}(\text{eShop}, \text{Customer}, \text{give_guar}(\text{BBB})), \text{Tg}) \wedge \text{Tg} > \text{Trg}.
\end{aligned} \tag{2}$$

The notion of acceptability for customers and payment methods from *eShop's* viewpoint, defined by the *accepted_customer/I* and *accepted_payment/I* predicates, is defined in *eShop's* knowledge base. In [1], only EU residents are accepted customers, as defined by the following clauses:

```

accepted_customer (Customer) :-
    resident_in (Customer, Location),
    accepted_destination (Location).
rejected_customer (Customer) :-
    resident_in (Customer, Location),

```

```
not accepted_destination(Location) .
accepted_destination(european_union) .
accepted_payment(cc) .
accepted_payment(cash) .
```

This knowledge is merged with that provided by the customer, for example

```
resident_in(alice, european_union) .
```

so that in the resulting knowledge base, on which SCIFF operates, `accepted_customer(alice)` is true.

However, this method would not work in case the customer declared `resident_in(alice, italy)`, as `italy` does not unify with `european_union`. One could, of course, add to the knowledge base `accepted_destination/1` facts for all the current EU members, but such knowledge should be updated when new countries join the EU. In other cases, acceptability could be defined by a transitive, symmetric relation, which could introduce cycles in the knowledge base, possibly leading to loops.

3 Representing domain knowledge with ontologies

An alternative way to represent (part of) the domain specific knowledge is to use technology and concepts developed, with focus on this very purpose, in the Knowledge Representation field, and to rely, in particular, on ontologies. The W3C recommendation for ontology representation on the Web is the Web Ontology Language (OWL) [5] based on the well established semantics of Description Logics [6] and on XML and RDF syntax. Using OWL for domain knowledge representation improves expressiveness (with such features as stating sub-class relations, constructing classes on property restrictions or by set operators, defining transitive properties and so on) yet keeping decidability (if using OWL Lite or OWL DL) in a straight-forward and domain modeling-oriented notation. Moreover, since OWL is tailored for the Web, it provides support for expressing knowledge in distributed contexts (identified by URIs) and its recognized standard status is a warranty on interoperability and reuse issues. As a plus, it can be mentioned that community driven development of Semantic Web tools already provides good support for OWL ontology management tasks such as editing [7] also for not KR-skilled users.

For instance, in Fig. 1 we show a possible ontological representation of *eShop*'s policies concerning acceptable customers and means of payments, merged with *alice*'s own knowledge.

The following OWL axiom says that the `acceptedCustomer` class is a subclass of the `potentialCustomer` class, and that it is disjoint from the `rejectedCustomer` class:

```
<owl:Class rdf:about="#acceptedCustomer">
  <rdfs:subClassOf rdf:resource="#potentialCustomer" />
  <owl:disjointWith rdf:resource="#rejectedCustomer" />
</owl:Class>
```

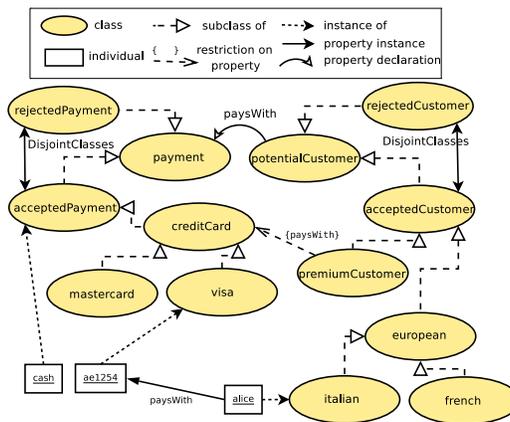


Fig. 1. A graphical representation of the ontology

The following fact states that `cash` is an instance of the `acceptedPayment` class:

```
<owl:Thing rdf:about="#cash">
  <rdf:type rdf:resource="#acceptedPayment" />
</owl:Thing>
```

The following axiom declares the `paysWith` property:

```
<owl:ObjectProperty rdf:ID="paysWith">
  <rdfs:domain rdf:resource="#potentialCustomer" />
  <rdfs:range rdf:resource="#payment" />
</owl:ObjectProperty>
```

The following fact states that `alice` is an instance of `italian`, with value `ae1254` for the `paysWith` property:

```
<owl:Thing rdf:about="#alice">
  <rdf:type rdf:resource="#italian" />
  <paysWith rdf:resource="#ae1254" />
</owl:Thing>
```

It can be noticed that the ontological notation for the KB makes it possible to infer that `alice` is `European` (and therefore an `accepted customer`) even if she just states being `Italian`, while if we had put `resident_in(alice, italy)` instead of `resident_in(alice, europe)` in the KB at the end of Sect. 2.2 she would not have been recognized as such.

Moreover, we defined a class `premiumCustomer` representing the `accepted customers` who pay with a credit card, and which we could use to add refinement to policies (for instance providing a faster delivery). Since `alice` is an `accepted customer`

and pays with her credit card, the ontological reasoning allows to recognize her as a `premiumCustomer`.

4 Handling semantic knowledge with *SCIFF*

We implemented the *SCIFF* access to OWL ontologies in two different ways, that we describe in this section. In both cases it should be noticed that even if OWL relies on the Open World Assumption, when *SCIFF* comes to reasoning involving it, the logic programming peculiar Closed World Assumption is reintroduced, thus assuming to have all (relevant) information on the domain available at that time and providing usual features such as negation as failure. In this section we present the two approaches, and their experimental evaluation.

4.1 Importing ontologies into the *SCIFF* KB_S

Considering that Logic Programming (LP) and Description Logics (DL) have a common root in First Order Logic, a first approach is to find their intersection and to translate ontologies to LP clauses. These two problems have already been addressed by Grosz et al. [8] who named this intersection DLP (Description Logics Program) and by Hustadt et al. [9] who proposed a method for translation. On these basis, the `dlpconvert` [10] tool was developed and made available by the Karlsruhe University, which converts (the DLP fragment of) OWL ontologies to datalog clauses. We used `dlpconvert` to translate domain knowledge described in OWL documents to *SCIFF* clauses. Reasoning is then performed by *SCIFF* in the usual way. However, this solution limits ontological expressivity. First of all, since the DLP fragment is a proper subset of DL, some OWL axioms are not included. For instance, out of the features mentioned as available in [11] by Vrandečić et al., `DisjointClasses` and the important DL (and OWL) feature of class definition by restriction on properties are missing. Moreover, some axioms' translation is not actually suitable for reasoning with goal-driven operational semantics, such as resolution or unfolding, employed in *SCIFF*, because it leads to loops. As an example, consider

```
<owl:Class rdf:about="#danaan">
  <owl:equivalentClass rdf:resource="#achaeaan"/>
</owl:Class>
```

whose translation is

```
danaan(X) :- achaeaan(X).
achaeaan(X) :- danaan(X).
```

and

```
<owl:ObjectProperty rdf:ID="ancestorOf">
  <rdf:type rdf:resource="&owl;TransitiveProperty" />
  <rdfs:domain rdf:resource="#person" />
  <rdfs:range rdf:resource="#person" />
</owl:ObjectProperty>
```

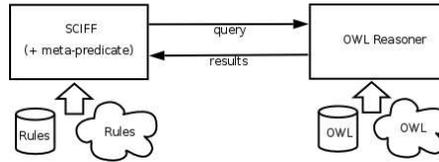


Fig. 2. Integration architecture

whose translation is

```
ancestorOf(X, Z) :- ancestorOf(X, Y), ancestorOf(Y, Z).
```

For all these reasons the e-commerce ontology must be modified, in particular removing the premiumCustomer class, and the DisjointClasses axioms.

4.2 Interfacing SCIFF and ontological reasoners

An approach that aims to overcome the limitations described in Sect. 4.1 consists of interfacing SCIFF with an external specific ontology-focused component which can be, when necessary, queried by SCIFF and which performs the actual ontological reasoning and gives back results. As represented in Fig. 2, the architecture for this solution involves a Prolog meta-predicate which invokes the ontological reasoning on desired goals, an intercommunication interface from SCIFF to the external component (which incorporates a query and results translation schema) and the actual reasoning module. Both modules can access both local and networked knowledge.

Goals given to the meta-predicate are handled like suggested in [9] and [11] considering single arity predicates as “belongs to class (with same name of predicate)” queries and double arity ones as “are related by property (with same name of predicate)” queries. To reduce the overhead caused by external communication, our implementation of the meta-predicate provides a caching mechanism: it is first checked if a similar query (i.e., involving the same predicate) has been performed before and, only if not, the external reasoner is invoked and answers are cached by storing them as Prolog facts (by means of `asserta/1`). The OWL reasoning module uses the Pellet [3] API, while the communication interface uses the Jasper Prolog-Java library [12]. This solution enables access to the full OWL(-DL) expressivity, including features such as equivalence of classes and properties, transitive properties, declaration of classes on property restriction and property-based individual classification.

4.3 Experimental evaluation

Both approaches were tested successfully in simple reasoning scenarios, such as the one depicted in Fig. 1 (as far as allowed by the expressiveness limitations of the ontology import approach discussed in Sect. 4.1).

Performance comparisons of the two approaches are hardly significant, as they mainly differ in expressive power. However, to assess their scalability, we experimented

N	Ontology import			Interface with Pellet		
	Load time	Query time	Total	Load time	Query time	Total
100	3.4	~ 0	3.4	~ 0	~ 0	~ 0
500	5.8	~ 0	5.8	1.0	~ 0	1.0
1000	8.2	~ 0	8.2	1.0	~ 0	1.0
5000	14.9	~ 0	14.9	2.0	1.2	3.2
10000	26.6	~ 0	26.6	4.0	2.8	6.8

Table 1. Performance comparison (all times are in seconds, average over 50 runs)

with randomly generated ontologies. Each ontology, composed of N classes, was built starting from its root node, and recursively attempting, for each node, five attempts of child generation, each with probability $1/3$.

The reasoners were queried about the belonging of an entity to the hierarchy root class. For both approaches, we report in Tab. 1 the time spent for loading the ontology into the reasoner³ and for the actual query. Both approaches appear to scale reasonably (on PC equipped with an Intel Celeron 2.4 GHz CPU). In both cases, the loading time is higher than the query time; for the importing approach, this is particularly noticeable, and encouraging, as in real applications the ontology would be imported only once.

5 Related work

One of the first works about the ontological representation of knowledge in a logic programming context is F-Logic [13] (implemented in Flora-2 [14]). F-Logic ontologies are similar to the formalism of frames and provide an attribute-value and taxonomical (with inheritance) structure for objects and classes. However they lack the feature of class definition based on restriction on attributes (or properties), which is a peculiar characteristic of DL (and OWL). Even if the Semantic Web community adopted the DL paradigm for the ontology layer of the Semantic Web cake, the urge for introducing rules for improving expressivity, for instance to model Semantic Web Services, partially made F-Logic come back as a suitable candidate, like in the case of WSMO (Web Services Modeling Ontology [15]) framework. An extensive study of how rules and ontologies can be integrated, with a specific focus on Semantic Web, can be found in [16] where a language, WSML, is proposed to be used as a Web Service Modeling Language in WSMO. Kifer et al. in [17] propose a comprehensive solution based on Flora-2, in which F-logic is used to express ontologies and Transaction Logic is used to model declaratively how services behave.

On the other hand, work is being done for the foundations of a tight integration of Description Logic and Logic Programming and one of the most important issues is how to deal with the different assumptions (Closed World for LP, Open World for DL) made by these two families of languages. In [18] decidability and complexity results are

³ For the importing approach, the load time is the time spent for translating the ontology and parsing the resulting clauses and ICs, while for the Pellet-based approach it is the time spent for parsing ICs and loading the ontology into a persistent OWLOntology object.

presented for an integration of DL and Disjunctive Datalog, while in [19] an integration of DL and LP is based on the Minimal Knowledge and Negation as Failure (MKNF) logic [20]. In [21] this proposal is extended providing a three-valued semantics. In our work, we apply the closed world assumption to ontological predicates, when computing their truth value in a *SCIFF* derivation; this simplification has proved acceptable for the applications that we have considered.

6 Conclusions

In this paper, we proposed an integration of the previously developed *SCIFF* reasoning framework for service contracting with Description Logics standards. We proposed two approaches: first, to translate ontological knowledge into the *SCIFF* formalism; second, to interface the *SCIFF* reasoner to ontological reasoners, delegating the ontological reasoning to them. In this way, we make the results (in terms of representation capability and computational efficiency) obtained in the Description Logics area available to the *SCIFF*-based contracting framework.

In future work, we intend to experiment with the integration of ontological reasoning to other applications of the *SCIFF* framework, such as agent programming [22], following the approach proposed by Moreira et al. [23].

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: Web service contracting: specification and reasoning with *SCIFF*. In Franconi, E., Kifer, M., May, W., eds.: *ESWC*. Volume 4519 of *LNAI*. (2007)
2. 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 Logics* **9** (2008)
3. Parsia, B., Sirin, E.: Pellet: An OWL DL reasoner. In van Harmelen, F., ed.: *ISWC 2004*. (2004)
4. Jaffar, J., Maher, M.: Constraint logic programming: a survey. *J. of Logic Programming* **19-20** (1994) 503–582
5. : W3C recommendation: OWL, web ontology language. <http://www.w3.org/TR/owl-guide/> (2004)
6. Lutz, C.: Description logic resources. dl.kr.org (2008)
7. Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W., Musen, M.A.: Creating semantic web contents with Protégé-2000. *IEEE Int. Systems* **16** (2001) 60–71
8. Grosz, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: *WWW '03*, *ACM* (2003) 48–57
9. Hustadt, U., Motik, B., Sattler, U.: Reducing *SHIQ*⁻ description logic to disjunctive datalog programs. (In Dubois, D., Welty, C., Williams, M.A., eds.: *KR2004*)
10. Motik, B., Vrandečić, D., Hitzler, P., Sure, Y., Studer, R.: Dlpconvert - Converting OWL DLP statements to logic programs. System Demo at the 2nd *ESWC* (2005)
11. Vrandečić, D., Haase, P., Hitzler, P., Sure, Y., Studer, R.: DLP-an introduction. Tech.Rep., Univ. Karlsruhe (2006)
12. : (SICStus Prolog) <http://www.sics.se/sicstus>.

13. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object oriented and frame based languages. *Journal of the Association for Computing Machinery* **42** (1995) 741–843
14. : Flora-2: An object-oriented knowledge base language. (<http://flora.sourceforge.net/>)
15. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., M.Stollberg, A.Polleres, C.Feier, C.Bussler, Fensel, D.: Web service modeling ontology. *Appl. Ontology* **1(1)** (2005)
16. de Bruijn, J.: Semantic Web Language Layering with Ontologies, Rules, and Meta-Modeling. PhD thesis, Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck (2008)
17. Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., Fensel, D.: A logical framework for web service discovery. In Martin, D., Lara, R., Yamaguchi, T., eds.: SWS. Volume 119 of CEUR Workshop Proc. (2004)
18. Rosati, R.: DI+log: Tight integration of description logics and disjunctive datalog. In Doherty, P., Mylopoulos, J., Welty, C.A., eds.: Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006, AAAI Press (2006) 68–78
19. Motik, B., Rosati, R.: A faithful integration of description logics with logic programming. In Veloso, M.M., ed.: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. (2007) 477–482
20. Lifschitz, V.: Nonmonotonic databases and epistemic queries. In Mylopoulos, J., Reiter, R., eds.: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI), Sidney, Australia, Morgan Kaufmann (1991) 381–386
21. Knorr, M., Alferes, J.J., Hitzler, P.: Towards tractable local closed world reasoning for the semantic web. In Neves, J., Santos, M.F., Machado, J., eds.: Progress in Artificial Intelligence, 13th Portuguese Conference on Artificial Intelligence, EPIA 2007, Workshops: GAIW, AIASTS, ALEA, AMITA, BAOSW, BI, CMBSB, IROBOT, MASTA, STCS, and TEMA, Guimarães, Portugal, December 3-7, 2007, Proceedings. Volume 4874 of Lecture Notes in Computer Science., Springer (2007) 3–14
22. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P.: A verifiable logic-based agent architecture. In Esposito, F., Raś, Z.W., Malerba, D., Semeraro, G., eds.: Foundations of Intelligent Systems - 16th International Symposium, ISMIS 2006 Bari, Italy, September 27-29, 2006 Proceedings. Volume 4203 of Lecture Notes in Artificial Intelligence., Berlin Heidelberg, Springer-Verlag (2006) 188–197
23. Moreira, Á.F., Vieira, R., Bordini, R.H., Hübner, J.F.: Agent-oriented programming with underlying ontological reasoning. In Baldoni, M., Endriss, U., Omicini, A., Torroni, P., eds.: Declarative Agent Languages and Technologies III, Third International Workshop, DALT 2005, Utrecht, The Netherlands, July 25, 2005, Selected and Revised Papers. Volume 3904 of Lecture Notes in Computer Science., Springer (2005) 155–170

Multi-Agent Planning in CLP

Agostino Dovier¹, Andrea Formisano², and Enrico Pontelli³

¹ Università di Udine, dovier@dimi.uniud.it

² Università di Perugia, formis@dipmat.unipg.it

³ New Mexico State University, epontell@cs.nmsu.edu

Abstract. This paper explores the use of Constraint Logic Programming (CLP) as a platform for experimenting with planning problems in presence of multiple interacting agents. The paper develops a novel *constraint-based* action language, \mathcal{B}^{MAP} , that enables the declarative description of large classes of multi-agent and multi-valued domains. \mathcal{B}^{MAP} supports several complex features, including combined effects, concurrency control, interacting actions, and delayed effects. The paper presents a mapping of \mathcal{B}^{MAP} theories to CLP and it demonstrates the effectiveness of an implementation in SICStus Prolog on several benchmark problems.

1 Introduction

Representing and programming intelligent and cooperating agents that are able to *acquire*, *represent*, and *reason* with knowledge is a challenging problem in Artificial Intelligence. In the context of single-agent domains, an extensive literature exists, presenting different languages for the description of planning domains (see, e.g., [9, 8, 2, 6]).

It is well-known that logic programming languages offer many properties that make them very suitable as knowledge representation languages, especially to encode features like defaults and non-monotonic reasoning. Indeed, logic programming has been extensively used to encode domain specification languages and to implement reasoning tasks associated to planning. In particular, *Answer Set Programming (ASP)* [1] has been one of the paradigms of choice—where distinct answer sets represent different trajectories leading to the desired goal.

Recently, an alternative line of research has started looking at *Constraint Programming* and *Constraint Logic Programming over Finite Domains* as another viable paradigm for reasoning about actions and change (e.g., [13, 14, 17, 5]). In particular [5] made a strong case for the use of constraint programming, demonstrating in particular the flexibility of constraints in modeling several extensions of action languages, necessary to address real-world planning domains.

The purpose of this paper is to build on the work in [5] and address planning problems arising in the domains that include multiple agents. Each agent can have different capabilities (i.e., they can perform different types of actions); the actions of the agents can also be *cooperative*—i.e., their cumulative effects are required to apply a change to the world—or *conflicting*—i.e., some actions may exclude other actions from being executed. Each agent maintains its own view

of the world, but groups of agents may share knowledge of certain features of the world (in the form of shared fluents).

The starting point of our proposal is represented by the design of a novel action language for encoding multi-agent planning domains. The action language, named \mathcal{B}^{MAP} builds on a similar spirit as the single-agent language of [5], where constraints are employed to describe properties of the world (e.g., properties the state of the world should satisfy after the execution of an action). \mathcal{B}^{MAP} adopts the perspective, shared by many other researchers (e.g., [3, 12, 16]), of viewing a multi-agent system from a *centralized* perspective, where a centralized description defines the modifications to the world derived from the agents' action executions (even though the individual agents may not be aware of that). This is different from the *distributed* perspective, where there is no centralized knowledge of how actions performed by different agents may interact and lead to changes of the state of the world.

We demonstrate how \mathcal{B}^{MAP} can be correctly mapped to a constraint satisfaction problem, and how Constraint Logic Programming (over finite domains) can be employed to support the process of computing plans, in a centralized fashion.

Observe that the use of logic programming for reasoning in multi-agent domains is not new; various authors have explored the use of other flavors of logic programming, such as normal logic programs and abductive logic programs, to address cooperation between agents (e.g., [11, 15, 7]). Some aspects of concurrency have been formalized and addressed also in the context of existing action languages (e.g., \mathcal{C} [2]) and in the area of multi-agent planning (e.g., [4, 3]).

The presentation is organized as follows. In Sect. 2 we illustrate the syntax of the action description language \mathcal{B}^{MAP} while in Sect. 3 we provide its semantics. The guidelines of the constraint-based implementation of its semantics are reported in Sect. 4. Some experimental results are reported in Sect. 5. Finally, some conclusions are drawn in Sect. 6.

2 Syntax of the language \mathcal{B}^{MAP}

In this section, we introduce the syntax of the action description language \mathcal{B}^{MAP} that captures (through constraints) the capabilities of a collection of agents that can perform interacting actions. We will emphasize the novelties and difference with respect to traditional single-agent languages.

The signature of the \mathcal{B}^{MAP} language consists of the following sets:

- \mathcal{G} : *agent* names, used to identify the agents participating in the domain.
- \mathcal{F} : *fluent* names; we assume that $\mathcal{F} = \bigcup_{a \in \mathcal{G}} \mathcal{F}_a$, where \mathcal{F}_a are the fluents used to describe the knowledge of agent a . Without loss of generality, we assume that for each $a, a' \in \mathcal{G}$, $a \neq a'$, we have that $\mathcal{F}_a \cap \mathcal{F}_{a'} = \emptyset$.
- \mathcal{A} : *action* names.
- \mathcal{V} : values for the fluents in \mathcal{F} . We assume the use of multi-valued fluents. In the following, we assume $\mathcal{V} = \mathbb{Z}$.

We will use a, b for agent names, f, g for fluent names, and x, y for action names.

2.1 \mathcal{B}^{MAP} Axioms

A theory in \mathcal{B}^{MAP} is composed of a collection of axioms. The axioms describe the different agents, their possible states, and their capabilities to change the world.

Agents and Fluents. The agent axiom is used to identify the agents present in the system. An assertion (*agent declaration*) of the type $\mathbf{agent}(a)$, where $a \in \mathcal{G}$, states the existence of the agent named a . The fluents that can be used by agent a to describe its own knowledge, are described by axioms of the form:

$$\mathbf{fluent}(a, f, \{v_1, \dots, v_k\})$$

with $a \in \mathcal{G}$ and $f \in \mathcal{F}_a$. The statement above also fixes the set of admissible values for f to $\{v_1, \dots, v_k\} \subseteq \mathcal{V}$. We also admit the alternative notation $\mathbf{fluent}(a, f, v_1, v_2)$ to specify all the values in the interval $[v_1, v_2]$ as admissible.

Fluents can be used in *Fluent Expressions* (FE), which are defined inductively as follows (where $n \in \mathcal{V}$, $t \in \mathbb{Z}$, $\oplus \in \{+, -, *, /, \mathbf{mod}\}$, $f \in \mathcal{F}$, and $r \in \mathbb{N}$):

$$\mathbf{FE} ::= n \mid f^t \mid f \ @ \ r \mid \mathbf{FE}_1 \oplus \mathbf{FE}_2 \mid \neg(\mathbf{FE}) \mid \mathbf{abs}(\mathbf{FE}) \mid \mathbf{rei}(C)$$

The meaning of a fluent expression will depend on the specific point in time during the evolution of the world. Given an integer number t , an expression of the form \mathbf{FE}^t is an *annotated* fluent expression. Intuitively, for $t > 0$ ($t < 0$), the expression refers to the value \mathbf{FE} will have t steps in the future (had $-t$ steps in the past). Hence, annotated expressions refer to points in time, relatively to the current state. The ability to create formulae that refer to different time points along the evolution of the world enables the encoding of non-Markovian processes. f is a shorthand for f^0 .

An expression of the form $\mathbf{FE} @ r$ denotes the value \mathbf{FE} has at the r^{th} step in the evolution of the world (i.e., it refers to an *absolutely* specified point in time).

The last alternative (reified expression) requires the notion of fluent constraint C (defined next). The intuitive semantics is that an expression $\mathbf{rei}(C)$ assumes a Boolean value (0 or 1) depending on the truth of C .

A *primitive fluent constraints* (PC) is a formula $\mathbf{FE}_1 \ \mathbf{op} \ \mathbf{FE}_2$, where \mathbf{FE}_1 and \mathbf{FE}_2 are fluent expressions, and $\mathbf{op} \in \{=, \neq, \geq, \leq, >, <\}$ is a relational operator.

Fluent constraints are propositional combinations of primitive fluent constraints:

$$\mathbf{PC} ::= \mathbf{FE}_1 \ \mathbf{op} \ \mathbf{FE}_2 \qquad \mathbf{C} ::= \mathbf{PC} \mid \neg \mathbf{C} \mid \mathbf{C}_1 \wedge \mathbf{C}_2 \mid \mathbf{C}_1 \vee \mathbf{C}_2 \mid \mathbf{C}_1 \rightarrow \mathbf{C}_2$$

\mathbf{true} and \mathbf{false} can be used as shorthands for true constraints (e.g., $0 = 0$) and unsatisfiable constraints (e.g., $0 \neq 0$).

Remark 1 (Language Extensions). Throughout the paper we will introduce forms of syntactic sugar. One of them is to accept expressions of the form $f^{[t_1, t_2]}$, which corresponds to $f^{t_1} \wedge \dots \wedge f^{t_2}$. Similarly, $f@[t_1, t_2]$ corresponds to $f@t_1 \wedge \dots \wedge f@t_2$.

We will assume the existence of an equivalence relation $\equiv_{\mathcal{F}} \subseteq \mathcal{F} \times \mathcal{F}$. Intuitively, if $\mathcal{F}_a \ni f \equiv_{\mathcal{F}} f' \in \mathcal{F}_b$, then the two fluents f and f' represent knowledge that is in common between two agents a and b —i.e., the two agents can view the same property of the world.

Actions Description. The following classes of axioms are used to describe actions and their behavior:

1. An axiom of the form: $\text{action}(Ag, x)$

where $Ag \subseteq \mathcal{G}$ and $x \in \mathcal{A}$, declares that x is meant to be executed collectively by the set of agents Ag . In particular,

- If $|Ag| = 1$, then the action is called an *individual action*.
- If $|Ag| > 1$, then the action is called a *collective action*.
- If $|Ag| = 0$, then a represents an *exogenous action*.

For each axiom $\text{action}(Ag, x)$, we introduce the expression $\text{actocc}(Ag, x)$, called *action flag* to denote the execution of that action. Action flags are intended to be Boolean valued expressions (when evaluated w.r.t. a state transition) and can be used to extend the notion of constraint. They can be used either as Boolean predicates or as Boolean functions.

Action-fluent expressions (AFE) extend the structure of fluent expressions by allowing propositions related to action occurrences:

$$\begin{aligned} \text{AFE} ::= n \mid f^t \mid f @ r \mid \text{actocc}(Ag, x)^t \mid \text{actocc}(Ag, x) @ r \mid \\ \text{AFE}_1 \oplus \text{AFE}_2 \mid \neg(\text{AFE}) \mid \text{abs}(\text{AFE}) \mid \text{rei}(C) \end{aligned}$$

where $n \in \mathcal{V}$, $t \in \mathbb{Z}$, $r \in \mathbb{N}$, $f \in \mathcal{F}$, $x \in \mathcal{A}$, $Ag \subseteq \mathcal{G}$, and $\oplus \in \{+, -, *, /, \text{mod}\}$. Time-annotated action-fluent expressions allow us to refer to the occurrences of actions at any point during the evolution of the world. Action-fluent-expressions can be used to form *action-fluent constraints*, as done for fluent constraints.

2. An axiom of the form: $\text{executable}(Ag, x, C)$

where $Ag \subseteq \mathcal{G}$, $x \in \mathcal{A}$, and C is an action-fluent constraint states that C has to be entailed by the current state for the action x to be executable by the agents in Ag . We assume that an executability axiom is present for each pair Ag, x with $Ag \subseteq \mathcal{G}$ and $x \in \mathcal{A}$ such that the axiom $\text{action}(Ag, x)$ is defined.

3. Axioms of the form $\text{causes}(\text{Eff}, \text{Prec})$

encode the effects of dynamic causal laws. Prec is an action-fluent constraint, called the *precondition constraint*. Eff is a fluent constraint, called the *effect constraint*. The axiom asserts that if Prec is **true** with respect to the current state, then Eff must hold in the next state. If Prec contains the conjunction of two (or more) action occurrences, then the effects refer to a *compound action*. Basically, a compound action could be seen as a collective action without name.

Example 1. Let us consider a domain with two agents, a and b , each capable of performing two actions, `push_door` and `pull_door`. If we want to capture the fact that the door can be opened only if both agents apply the same action to it, we can use the dynamic causal laws:⁴

$$\begin{aligned} \text{causes}(\text{opendoor} = 1, \text{actocc}(\{a\}, \text{push_door}) \wedge \text{actocc}(\{b\}, \text{push_door})) \\ \text{causes}(\text{opendoor} = 1, \text{actocc}(\{a\}, \text{pull_door}) \wedge \text{actocc}(\{b\}, \text{pull_door})) \end{aligned}$$

⁴ For simplicity, in what follows we often implicitly assume the relation $\equiv_{\mathcal{F}}$ defined so that all agents share all fluents. Moreover, we denote by an unique representative all $\equiv_{\mathcal{F}}$ -equivalent fluents (e.g., the fluent `opendoor` in this example).

Hence, the door can be opened only by the combined activity of both agents.

Consider a situation where agent a can receive a message only if it has been sent by agent b ; the executability condition of the receive action is expressed as

$$\text{executable}(\{a\}, \text{receive}, \text{actocc}(\{b\}, \text{send}))$$

We can also impose constraints on effect duration, e.g.,

$$\text{causes}(\text{stay_in_class}^{[0,60]} = 1, \text{actocc}(\{\text{Pontelli}\}, \text{start_lesson}))$$

where $\text{stay_in_class}^{[0,60]} = 1$ can be written in concrete syntax as follows:

$$\text{stay_in_class} = 1 \wedge \text{stay_in_class}^1 = 1 \wedge \dots \wedge \text{stay_in_class}^{60} = 1 \quad \square$$

Static causal laws and other State Constraints. Static causal laws can be added using axioms of the form: $\text{caused}(C_1, C_2)$ stating that the action-fluent constraint $C_1 \rightarrow C_2$ must be entailed in any state encountered.

We introduce some syntactic sugar used to represent certain classes of static causal laws that are frequently encountered. The syntax of action-fluent expressions and action-fluent constraints allows the use of two forms of fluent and action time annotations: *relative time* annotations (AFE^t with $t \in \mathbb{Z}$) and *absolute time* annotations ($\text{AFE}@t$ for $t \in \mathbb{N}$). Intuitively, relative expressions are meant to be evaluated with respect to a certain point in the evolution history of the world, while absolute expressions are evaluated with respect to the starting point of the evolution. We classify an action-fluent constraint C as follows:

- if C does not contain any relative or absolute annotation, then it will be referred to as a *timeless constraint*
- if C contains only relative (absolute) annotations, then it will be referred to as a *relative (absolute) time constraint*.

The following specialized versions of static causal laws are then introduced:

– If C is a timeless constraint, then $\text{always}(C)$ denotes the collection of static causal laws

$$\text{caused}(\text{true}, C@t) \quad \forall t \in \mathbb{N}$$

(which is in turn equivalent to the static law $\text{caused}(\text{true}, C)$).

– In \mathcal{B}^{MAP} we will focus on domains where each agent can perform at most one action per time step. The specification of dynamic causal laws enables to accommodate for effects derived from the concurrent execution of actions. Similarly, we may encounter situations where certain actions cannot be executed concurrently by different agents. This constraint can be enforced using the notions of action-fluent expressions and constraints. The axiom:

$$\text{concurrency_control}(C)$$

states that the action-fluent constraint must hold. A concurrency control axiom represents a syntactic sugar for a static causal law of the form $\text{caused}(\text{true}, C)$.

Example 2. Two agents can walk through a revolving door only one at the time. This can be captured by

$$\text{concurrency_control}(\text{actocc}(\{a\}, \text{walk_through}) + \text{actocc}(\{b\}, \text{walk_through}) \leq 1)$$

Similarly, the fact that the action `switch_on` can not be repeated consecutively by agent a can be expressed as

$$\text{concurrency_control}(\text{actocc}(\{a\}, \text{switch_on}) + \text{actocc}(\{a\}, \text{switch_on})^{-1} \leq 1) \quad \square$$

2.2 Costs

In \mathcal{B}^{MAP} it is possible to specify information about the *cost* of each action and about the global cost of a plan. In particular:

- `action_cost(Ag, x, Val)` where $Ag \subseteq \mathcal{G}$, $x \in \mathcal{A}$, and Val specifies the cost of executing the action described by the axiom `action(Ag, x)` (otherwise, a default cost of 1 is assigned).

Example 3. If we would like to express that the cost of one agent constructing a fence is double the cost of two agents performing the same job, then we can provide the axioms

$$\begin{aligned} &\text{action_cost}(\{a\}, \text{build_fence}, 100). && \text{action_cost}(\{b\}, \text{build_fence}, 100). \\ &\text{action_cost}(\{a, b\}, \text{build_fence}, 50) && \square \end{aligned}$$

- `state_cost(FE)` specifies the cost of a generic state as the result of the evaluation of the fluent expression FE , built using the fluents present in the state (otherwise, a default cost of 1 is assumed).

Example 4. Let us consider a two-agent domain where each agent may have stocks of a company that they are trying to sell. Let us assume that the following two fluents are defined:

$$\text{fluent}(a, \text{has_stock}(a), 0, 10). \quad \text{fluent}(b, \text{has_stock}(b), 0, 10).$$

Agent a has greater experience and can sell his stocks at twice the price as agent b ; thus the value of a state is

$$\text{state_cost}(\text{has_stock}(a) * 2 + \text{has_stock}(b)) \quad \square$$

2.3 Action Domains

An *action domain description* \mathcal{D} is a collection of axioms of the formats described earlier. In particular, we denote the following subsets of \mathcal{D} : $\mathcal{EL}_{\mathcal{D}}$ is the set of executability conditions, $\mathcal{DL}_{\mathcal{D}}$ is the set of dynamic causal laws, and $\mathcal{SL}_{\mathcal{D}}$ is the set of static causal laws (and all the derived axioms, such as concurrency control, cost axioms, and temporal constraints).

A specific instance of a planning problem is a tuple $\langle \mathcal{D}, \mathcal{I}, \mathcal{O} \rangle$, where \mathcal{D} is an action domain description, \mathcal{I} is a collection of axioms of the form `initially(C)` (describing the initial state of the world), and \mathcal{O} is a collection of axioms of the form `goal(C)`, where C is a fluent constraint.

3 Semantics of \mathcal{B}^{MAP}

Let \mathcal{D} be a planning domain description and $\langle \mathcal{D}, \mathcal{I}, \mathcal{O} \rangle$ be a planning problem. As a language design choice, we will explore multi-agent problems where each agent can perform at most one action at each time step (semantics can be developed similarly without this constraint). The set of actions involving the agent $a \in \mathcal{G}$ are defined as follows: $\mathcal{A}_a = \{x \in \mathcal{A} \mid \text{action}(Ag, x) \in \mathcal{D}, a \in Ag\}$.

The starting point for the definition of a state is the notion of interpretation. Given the collection of fluents \mathcal{F} , an interpretation I is a mapping $I : \mathcal{F} \rightarrow \mathbb{Z}$ that satisfies the following properties:

- if $\text{fluent}(a, f, \text{Set})$ is an axiom in \mathcal{D} , then $I(f) \in \text{Set}$.
- Given $\equiv_{\mathcal{F}}$ (c.f. Sect. 2.1), if $f \equiv_{\mathcal{F}} g$, then $I(f) = I(g)$.

Given two interpretations I, I' , and a set of fluents $S \subseteq \mathcal{F}$, we define

$$\Delta(I, I', S) = \begin{cases} I'(f) & \text{if } f \in S \\ I(f) & \text{otherwise} \end{cases}$$

A *state-transition* sequence is a tuple $\nu = \langle I_0, A_1, I_1, A_2, I_2, \dots, A_N, I_N \rangle$ where I_0, \dots, I_N are interpretations and for $i \in \{1, \dots, N\}$, A_i is a function $A_i : \mathcal{G} \rightarrow \mathcal{A} \cup \{\emptyset\}$ such that $A_i(a) \in \mathcal{A}_a \cup \{\emptyset\}$. We denote with $\nu|_j$ the sequence $\langle I_0, A_1, I_1, \dots, A_j, I_j \rangle$.

We provide an interpretation based on a state-transition sequence for the various types of formulae. In particular, we give the definition for the action-fluent expressions and constraints whose forms subsume the other classes of formulae. Given a state-transition sequence ν and an AFE φ , the value of φ w.r.t. ν and $0 \leq i \leq N$ (denoted by $\nu_i(\varphi)$) is an element of the set of fluents values \mathcal{V} computed as follows:

- $\nu_i(n) = n$ and $\nu_i(f) = I_i(f)$
- $\nu_i(\text{AFE}@t) = \nu_i(\text{AFE})$
- if $f \in \mathcal{F}$: $\nu_i(f^t) = \nu_{i+t}(f)$ if $0 \leq i+t \leq N$; $\nu_i(f^t) = \nu_i(f@0)$ if $i+t < 0$; $\nu_i(f^t) = \nu_i(f@N)$ otherwise.
- $\nu_i(\text{actocc}(Ag, x)) = 1$ if $i > 0$ and for each $a \in Ag$ we have that $A_i(a) = x$; $\nu_i(\text{actocc}(Ag, x)) = 0$ otherwise.
- if $x \in \mathcal{A}$ and $Ag \subseteq \mathcal{G}$: $\nu_i(\text{actocc}(Ag, x)^t) = \nu_{i+t}(\text{actocc}(Ag, x))$ if $1 < i+t \leq N$; $\nu_i(\text{actocc}(Ag, x)^t) = 0$ otherwise.
- $\nu_i(\text{AFE}_1 \oplus \text{AFE}_2) = \nu_i(\text{AFE}_1) \oplus \nu_i(\text{AFE}_2)$
- $\nu_i(-\text{AFE}) = -\nu_i(\text{AFE})$
- $\nu_i(\text{abs}(\text{AFE})) = |\nu_i(\text{AFE})|$

An AFE constraint φ is entailed by ν at time i ($\nu \models_i \varphi$) as follows:

- $\nu \models_i FE_1 \text{ op } FE_2$ iff $\models \nu_i(FE_1) \text{ op } \nu_i(FE_2)$
- $\nu \models_i \neg C$ iff $\nu \not\models_i C$
- $\nu \models_i C_1 \wedge C_2$ iff $\nu \models_i C_1$ and $\nu \models_i C_2$
- $\nu \models_i C_1 \vee C_2$ iff $\nu \models_i C_1$ or $\nu \models_i C_2$
- Moreover, $\nu_i(\text{rei}(C)) = 1$ iff $\nu \models_i C$.

The following properties characterize a state-transition sequence ν :

- ν is *initialized* w.r.t. a planning problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{O} \rangle$, if $\nu \models_0 C$ for each $\text{initially}(C) \in \mathcal{I}$.
- ν is *correct* if, for each axiom $\text{action}(Ag, x)$ in \mathcal{D} and for each $i \in \{1, \dots, N\}$, if $x \in A_i(\mathcal{G})$, then $\{a \in \mathcal{G} \mid A_i(a) = x\} = Ag$.
- ν is *closed* if the following property is met: for each static law of the form $\text{caused}(C_1, C_2)$ in \mathcal{D} and for each $0 \leq j \leq N$, we have that $\nu \models_j C_1 \rightarrow C_2$.

In order to model the notion of *trajectory*—intended to represent a correct evolution of the world that leads to a solution of a planning problem—we will consider the collection of fluent and action-fluent constraints accumulated during an execution. Let us denote with $\text{Shift}_j^F(C)$ the constraint obtained from C by replacing each occurrence of f^t with $f^{\mathbb{Q}(t+j)}$ and with $\text{Shift}_j^A(C)$ the constraint obtained from C by replacing each occurrence of $\text{actocc}(Ag, x)^t$ with $\text{actocc}(Ag, x)^{\mathbb{Q}(t+j)}$. We denote with $\text{Shift}_j = \text{Shift}_j^F \circ \text{Shift}_j^A$.

Let \mathbf{A} denote the tuple $\langle A_1, \dots, A_N \rangle$.

- For $i \in \{1, \dots, N\}$, let

$$\text{Concr}_i(\mathbf{A}) = \bigwedge_{x \in A_i(\mathcal{G}), Ag = \{a \in \mathcal{G} \mid A_i(a) = x\}} (\text{actocc}(Ag, x) @i) = 1.$$

These are action-fluent constraints describing one step in an action sequence.

- $C_0 = \text{Shift}_0 \left(\bigwedge_{\text{initially}(C) \in \mathcal{I}} C \right)$
- Intuitively, the following constraint summarizes, as implications, all possible effects from execution of actions:

$$AC_i = \bigwedge_{\text{causes}(EC, PC) \in \mathcal{D}} \text{Shift}_i^A(\text{Shift}_{i-1}^F(PC)) \rightarrow \text{Shift}_i^F(EC)$$

- For $i \in \{1, \dots, N\}$, let

$$\text{Exec}_i(\mathbf{A}) = \text{Shift}_{i-1} \left(\bigwedge_{\substack{x \in A_i(\mathcal{G}), Ag = \{a \in \mathcal{G} \mid A_i(a) = x\} \\ \text{executable}(Ag, x, C) \in \mathcal{D}}} C \right)$$

The action sequence \mathbf{A} represents a skeleton of a trajectory w.r.t. the problem specification which is stated by the action-fluent constraint:

$$\text{Skel}(\mathbf{A}) \equiv C_0 \wedge \bigwedge_{i=1}^N AC_i \wedge \bigwedge_{i=1}^N \text{Concr}_i(\mathbf{A}) \wedge \bigwedge_{i=1}^N \text{Exec}_i(\mathbf{A})$$

The next step is to “fill” the skeleton of an action sequence with intermediate states. The selection of the states should guarantee closure, satisfaction of action effects, and avoidance of unnecessary changes. The state-transition sequence $\nu = \langle I_0, A_1, I_1, A_2, \dots, A_N, I_N \rangle$ is a *trajectory* if it satisfies the following conditions:

- ν is closed
- $\langle A_1, \dots, A_N \rangle$ is correct
- $\nu \models_0 \text{Skel}(\mathbf{A})$
- $\nu \models_N \bigwedge_{\text{goal}(C) \in \mathcal{O}} C$

- v. for any $\emptyset \neq S \subseteq \mathcal{F}$ and for each $1 \leq i \leq \mathbf{N}$ there are no interpretations $I'_{i+1}, \dots, I'_\mathbf{N}$ such that

$$\nu' = \langle I_0, A_1, I_1, \dots, A_i, \Delta(I_i, I_{i-1}, S), A_{i+1}, I'_{i+1}, \dots, A_\mathbf{N}, I'_\mathbf{N} \rangle$$
and ν' satisfies the conditions (i)–(iv).

If ν is a trajectory, then we will refer to \mathbf{A} as a *plan*.

In presence of cost declarations, it becomes possible to compare trajectories according to their costs. Given a plan \mathbf{A} , the cost of the plan is so defined: let $\mu(A_i) = \{(x, Ag) \mid x \in A_i(\mathcal{G}), Ag = \{a \in \mathcal{G} \mid A_i(a) = x\}\}$ and $pcost(A_i) = \sum_{(x, Ag) \in \mu(A_i)} val(x, Ag)$, where

$$val(x, Ag) = \begin{cases} Val & \text{action_cost}(Ag, x, Val) \in \mathcal{D} \\ 1 & \text{otherwise.} \end{cases}$$

Then, we define the cost of a plan as $pcost(\mathbf{A}) = \sum_{i=1}^{\mathbf{N}} pcost(A_i)$. Trajectories can be selected based on their plan cost, either by requiring bounds on the plan cost or requesting optimal plan cost. A plan β is *optimal* if there is no other plan β' for the same problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{O} \rangle$ such that $pcost(\beta') < pcost(\beta)$.

We also admit constraints aimed at bounding the cost of a plan; these are denoted by axioms of the form **plan_cost**(**op n**), where n is a number and **op** is a relational operator. A plan $\langle A_1, \dots, A_\mathbf{N} \rangle$ is plan-cost-admissible if $pcost(\langle A_1, \dots, A_\mathbf{N} \rangle) \mathbf{op} n$.

Similar considerations can be done for the case of state costs. Let us assume that we have an axiom of the form **state_cost**(FE). For any trajectory ν we define $scost(\nu) = \nu_\mathbf{N}(FE)$. We can define a trajectory to be optimal if there is no other trajectory ν' such that $scost(\nu') < scost(\nu)$. We allow in the domain specification axioms of the form **goal_cost**(**op n**); a trajectory ν is state-cost-admissible if $scost(\nu) \mathbf{op} n$.

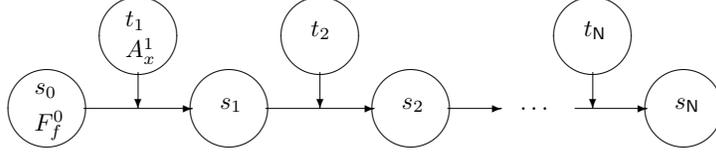
4 Implementation

Let us describe how \mathcal{B}^{MAP} action descriptions are mapped to finite domain constraints, and how the implementation is realized in a concrete constraint logic programming system, specifically SICStus Prolog. The implementation is based on the implementation of the (single agent and non-concurrent) language \mathcal{B}^{MV} [5].

We assume the following concrete syntax for dynamic and static causal laws and executability conditions:

$$\begin{aligned} &\text{causes}(FC, [PC_1, \dots, PC_m]) \\ &\text{executable}(Ag, x, [PC_1, \dots, PC_m]) \\ &\text{caused}([PC_1, \dots, PC_m], PC) \end{aligned}$$

where PC, PC_1, \dots, PC_m are primitive action-fluent constraints (and the list notation denotes conjunction), FC is a primitive fluent constraint, x is an action name and Ag a set (represented by a list) of agent names. Let us focus on the problem of finding a valid trajectory containing \mathbf{N} transitions $\langle t_1, \dots, t_\mathbf{N} \rangle$, involving a sequence of $\mathbf{N} + 1$ interpretations $\langle s_0, \dots, s_\mathbf{N} \rangle$. We depict such a plan as follows:



Let f be a fluent, declared by means of the axiom $\mathbf{fluent}(a, f, \text{dom}(f))$. We represent its value in the state s_j through a constrained variable F_f^j , with finite-domain $\text{dom}(f)$.⁵ Consequently, a state is represented by a list of Prolog terms of the form $\mathbf{fluent}(a, f, F_f^j)$.

A transition t_i from the state s_{i-1} to s_i is represented by a list of Prolog terms $\mathbf{action}(Ag, x, A_x^i)$, where $A_{Ag,x}^i$ is a Boolean variable representing the occurrence of action x executed by agents Ag during the transition. Therefore, the variable $A_{Ag,x}^i$ represents the value of the action flag $\mathbf{actocc}(Ag, x)$ at the i^{th} time step. Since Ag is in general a set of more than one agent, for each time step i we introduce a Boolean variable $G_{Ag,x,a}^i$ for each agent $a \in \mathcal{A}$.

Let us introduce some useful notations. Given a state transition t_i and a primitive constraint $c \equiv E_1 \mathbf{op} E_2$ we denote by c^j (resp., $c *^j$) the constraint obtained from c by replacing each fluent f with F_f^j and each action flag $\mathbf{actocc}(Ag, x)$ with $A_{Ag,x}^j$ (resp., $A_{Ag,x}^{j+1}$). This notation is generalized to any list/conjunction of action-fluent constraints $\alpha \equiv [c_1, \dots, c_m]$ by denoting with α^j (resp., $\alpha *^j$) the expression $(c_1)^j \wedge \dots \wedge (c_m)^j$ (resp., $(c_1) *^j \wedge \dots \wedge (c_m) *^j$).⁶ By relying on this representation, the implementation asserts suitable constraints to relate the values the fluents assume along the trajectory and the action occurrences.

For simplicity, let us focus on a single state transition, say, between the states s_i and s_{i+1} . Domains for the constrained variables are set by imposing that, for each $f \in \mathcal{F}$, $F_f^i, F_f^{i+1} \in \text{dom}(f)$; for each defined $\mathbf{action}(Ag, x)$, $A_{Ag,x}^{i+1} \in \{0, 1\}$ and $G_{Ag,x,a}^{i+1} \in \{0, 1\}$ for all $a \in \mathcal{G}$. Executability conditions are rendered by imposing the constraint $A_{Ag,x}^{i+1} \rightarrow \bigvee_{j=1}^{p_x} (C_j) *^i$, for each defined $\mathbf{action}(Ag, x)$, where $\mathbf{executable}(Ag, x, C_1), \dots, \mathbf{executable}(Ag, x, C_{p_x})$ are all the associated executability laws. The effects of action occurrences are imposed through the constraints $\bigwedge_{j=1}^{m_f} ((PC_j) *^i \rightarrow (EC_j)^{i+1})$ for each fluent $f \in \mathcal{F}$, where $\mathbf{causes}(EC_1, PC_1), \dots, \mathbf{causes}(EC_{m_f}, PC_{m_f})$

are all the dynamic causal laws involving f . Correctness of the action-sequence is rendered by imposing

- $G_{Ag,x,a}^{i+1} = 0$ for all defined $\mathbf{action}(Ag, x)$, and agent $a \in \mathcal{G} \setminus Ag$
- $\bigwedge_{a \in Ag} G_{Ag,x,a}^{i+1} = A_{Ag,x}^{i+1}$, for all defined $\mathbf{action}(Ag, x)$ and $a \in Ag$

⁵ Notice that, the current implementation does not deal directly with locality of fluents. At this level all fluents are known by all agents. Restrictions on accessibility of fluents by specific agents can be dealt with while parsing the action theory. For the sake of simplicity, we do not discuss this point here.

⁶ This notation is used to reflect at the concrete level, the notions $\mathbf{Shift}_j^F(C)$ and $\mathbf{Shift}_j^A(C)$ introduced at the abstract semantic level.

Moreover, the condition that each agent can execute at most one action per time, is encoded by means of the constraints $\sum_{(Ag,x)} G_{Ag,x,a}^{i+1} \leq 1$ (for each $a \in \mathcal{G}$). Initial and goal requirements are forced by simply imposing the constraints specified by the corresponding **initially** and **goal** axioms.

Intuitively, the conjunction of all these constraints, for all the transitions in the trajectory, enforces the conditions (ii)-(iv) listed in page 8. Closure (i.e., condition (i)) is rendered by imposing a constraint $C_j^i \rightarrow SC_j^i$ for each j such that **caused**(C_j, SC_j) is a static causal law, and for each i^{th} step of the trajectory.

We are left with condition (v), expressing the minimality of changes (in the value of fluents) in the trajectory sought for.

As usual in CLP, once all constraints on variables have been stated, the inference engine searches for a solution by performing a labeling phase. In our case, this phase assigns suitable values (if any) to all action flags. Consequently, it determines, for each i , which actions are performed by the agents during the i^{th} transition. Notice that, once the first i transitions have been determined, a solution of those constraints characterizing the states s_0, \dots, s_i is also obtained in form of a substitution (i.e., an assignment of values) σ_i for all fluent variables in s_0, \dots, s_i . In particular, each σ_i can be seen as an extension of σ_{i-1} , namely, σ_i agrees with σ_{i-1} on $\text{dom}(\sigma_{i-1})$ (i.e., on the fluents of s_0, \dots, s_{i-1}). Hence, in order to enforce inertia, we adopt a suitable order of variables in the labeling phase, while imposing further constraints on how σ_{i-1} is extended to obtain σ_i .

As regards the labeling, variables are labeled so that each σ_i (and, consequently, s_i) is determined only when σ_{i-1} (i.e., all the preceding states s_0, \dots, s_{i-1}) has been calculated. Then, condition (v) can be incrementally imposed as follows. Let s_0, \dots, s_{i-1} be the states of the partial trajectory described by the (already calculated) substitution σ_{i-1} . The labeling proceeds by assigning values to the action flags of the i^{th} transition. This, in turn, determines the admissible assignments for the variables in $\text{dom}(\sigma_i) \setminus \text{dom}(\sigma_{i-1})$. The partial trajectory determined by such a σ_i is acceptable, only if there is no other extension ρ_i of σ_{i-1} , determining a state $r \neq s_i$, such that the following constraint is satisfied:

$$\bigwedge_{f \in \mathcal{F}} \left(\bigvee_{j=1}^{m_f} \sigma_i((PC_j)^*{}^i) \rightarrow \sigma_i(F_f) = \rho_i(F_f) \right) \wedge \bigwedge_{f \in \mathcal{F}} \left(\sigma_i(F_f) \neq \rho_i(F_f) \rightarrow \rho_i(F_f) = \sigma_{i-1}(F_f) \right)$$

Intuitively, the satisfaction of such a formula witnesses the existence of a counterexample for the minimality of s_0, \dots, s_i . If for all i no such ρ_i exists, then $\sigma_{\mathbf{N}}$ is a solution of the entire constraint system and determines a trajectory satisfying the conditions (i)-(v).

The above outlined approach completely enforces inertia. However, it involves the introduction of rather complex constraints. In order to achieve a compromise w.r.t. the efficiency, a different encoding has been realized. Such an encoding is based on the notion of cluster of fluents w.r.t. static causal laws (cf., [5]).

Given a set $L \subseteq \mathcal{F}$ of fluents, let $\mathcal{SL}_L \subseteq \mathcal{SL}$ be the collection of all static causal laws in which at least one fluent in L occurs. Moreover, for simplicity, let \mathcal{SL}_f denote $\mathcal{SL}_{\{f\}}$ (i.e., the set of all static causal laws involving the fluent f).

Let us define a relation $R \subseteq \mathcal{F} \times \mathcal{F}$ so that $f_1 R f_2$ iff $\mathcal{SL}_{f_1} \cap \mathcal{SL}_{f_2} \neq \emptyset$. R is an equivalence relation and it partitions \mathcal{F} . Each element (i.e., equivalence class) of the quotient \mathcal{F}/R is said to be a *cluster* (w.r.t. \mathcal{SL}). Notice that a cluster can be a singleton $\{f\}$. Let f be a fluent, we denote with L_f its cluster w.r.t. \mathcal{SL} .

Clusters are exploited in defining the following constraints:

$$Stat_f^i \leftrightarrow \bigwedge_{g \in L_f} \left(\bigvee_{j=1}^{m_f} ((PC_j)^i \wedge g \in \text{fluents}(PC_j)) \rightarrow F_g^i = F_g^{i+1} \right) \quad (1)$$

$$Stat_f^i \rightarrow \bigwedge_{g \in L_f} F_g^i = F_g^{i+1} \quad (2)$$

where $\text{fluents}(C)$ denotes the set of fluents occurring in C . Intuitively, enforcing (1) and (2) imposes that if all the fluents that belong to the cluster L_f are left unchanged in the transition, then all the fluents of L_f should not change their value. In fact, a cluster is a set of fluents whose values have been declared to be mutually dependent through a set of static causal laws. In a state transition, changes in the fluents of a cluster might occur because of their mutual influence, not being (indirectly) caused by dynamic laws. Constraints (1) and (2) impose inertia on all the fluents of a cluster whenever none of them is influenced by dynamic laws. Notice that imposing (1)–(2) does not completely guarantee condition (v). This is because state transitions violating the inertia are admitted. In fact, (1)–(2) do not impose inertia on the fluents of a cluster when at least one of them is changed by the dynamic laws. This might lead to invalid transitions in which a change in the value of a fluent of cluster happens even if this is not necessary in order to satisfy all static causal laws.

This means that the concrete implementation may produce solutions (i.e., plans) that the semantics of \mathcal{B}^{MAP} would forbid because of non-minimal effect of (clusters of) static causal laws.

5 Experiments

The interpreter of the language \mathcal{B}^{MAP} is available at www.dimi.uniud.it/dovier/CLPASP/MAP along with some planning domains. As explained in Sect. 4, at the first level the labeling strategy is mainly a “leftmost” strategy that follows the temporal evolution of a plan. We first consider the transition $\langle s_0, t_1, s_1 \rangle$, then the transition $\langle s_1, t_2, s_2 \rangle$, and so on. However, we leave the programmer the ability of choosing the strategy within each of these sets (of course, there exists no best strategy for all problems—see, e.g., [18]). One can choose leftmost, **ff** (first-fail), or **ffc** (first-fail with a choice on the most constrained variable) and try the best for her/his domain. We also noticed that on our tests **ff** and **ffc** have, in most of the cases, similar performances. A further labeling strategy, **ffcd**, combines **ffc** with a downward selection of values for constrained variables. In most cases this strategy gave the best performances.

We run the system on some classical single-agent domains, such as the three barrels (12-7-5), a *Sam Lloyd's* puzzle, the goat-cabbage-wolf problem, the peg-solitaire (csplib 037, also in the 2008 planning competition IPC08—in [10] the authors solve it in 388s after a difficult encoding using operations research techniques. We solve it in less than 45 seconds with a simple \mathcal{B}^{MAP} encoding), and *the gas diffusion problem* [5]:

A building contains a number of rooms in the first floor. Each room is connected to (some) other rooms via gates. Initially, all gates are closed and some of the rooms contain certain amounts of gas (the other rooms are assumed to be empty). Each gate can be opened or closed. When a gate between two rooms is opened the gas contained in these rooms flows through the gate. The gas diffusion continues until the pressure reaches an equilibrium. The only condition to be always satisfied is that a gate in a room can be opened only if all other gates are closed. The goal for Diabolik is to move a desired quantity of gas in the specified room which is below the central bank (in order to be able to generate an explosion).

We tested a 11-room building (see also [5]).

We also tested the \mathcal{B}^{MAP} implementation on the suite of Peg Solitaire instances used in IPC08 for the “sequential satisficing track”. The competition imposed these restrictions: the plan has to be produced within 30 minutes, by using at most 2GB of memory. The suite is composed of 30 problems. The \mathcal{B}^{MAP} planner found the optimal plan for 24 problems.

Then we have tested the interpreter on some inherently concurrent domains, such as the dining philosophers (usual rules, when one eat he'll be alive for 10 seconds. We ask for a plan that ensures all philosophers to be alive at a certain time), a problem of cars and fuels (EATCS bulletin N. 89, page 183—by Laurent Rosaz—tested with four cars), a *social-game* invented by us (that required 6 actions if solved by one agent), and two problems described in previous works on concurrency and knowledge representation. The first problem is adapted from the working example of [4]:

Bob is in the park. Mary is at home. Bob wishes to call Mary to join him. Between the park and Mary's house there is a narrow road. The door is old and heavy. First Bob needs to ring the bell. Then they can open the old door in cooperation. Mary cannot leave the house if the door is closed.

We have modeled it either using collective actions or using compound actions (for opening the door). We report the latter domain in Figure 1.

The second problem is instead adapted from the working example of [3] and is related to two agents, some blocks, two rooms, and a table that could help the agents to carry the blocks all together from a room to another. We experimented with a basic \mathcal{B}^{MAP} encoding of this problem, combined with different static and `concurrency_control` laws, which impose commonsense conditions (e.g., the same block cannot be simultaneously grabbed by two agents; agents must avoid

undoing the effects of previously performed actions or moving between the rooms without carrying objects; and so on), as well as forms of symmetry-breaking rules (e.g., blocks have to be grabbed in order). The goal asks that in the final state all the objects should be placed on the ground of the second room. A short and smart plan has been found: the two agents move all the blocks on the table and move the table to the second room, then one of the agent releases the table, so, in a single move, all the blocks fall on the ground.

Table 1. Some experiments. 1–5 are single agent. 6–10 are multi-agent. Plan length is the length of the plan required (e.g., in example 1, the goal is `:- bmap(11).`). Vars denotes the number of still unknown variables for actions after all constraints are added. We report the running time for leftmost, `ffc`, and `ffcd` labeling strategies (cf., Sect. 4). The symbol ‘-’ denotes no answer within 1 hour.

		Plan length	Vars	leftmost (s)	ffc (s)	ffcd (s)
1.	Three Barrels	11	62	0.12	0.11	0.07
2.	Goat-Wolf etc.	23	219	0.14	0.04	0.28
3.	Gas diffusion	6	132	34.9	34.9	9.65
4.	Puzzle	15	915	62.0	64.7	4.55
5.	Peg Solitaire	31	1999	–	–	44.7
6.	Bob and Mary	5	25	0.01	0.01	0.01
7.	Social Game	2	40	0.04	0.04	0.06
8.	Dining philosophers	9	210	339	439	–
9.	Fuel and Cars	10	90	736	743	0.48
10.	Robots and Table	7	184	316	461	118

6 Conclusions

We presented a constraint-based action description language, \mathcal{B}^{MAP} , that extends the previously proposed language \mathcal{B}^{MV} [5]. Such a new language retains all the valuable features of \mathcal{B}^{MV} , namely the availability of multi-valued fluents and the possibility of referring to fluents in any different state of the trajectory, to describing preconditions and effects of actions.

The major novelty of \mathcal{B}^{MAP} consists of allowing declarative formalization of planning problems in presence of multiple interacting agents. Each agent can have a different (partial) view of the world and a different collection of executable actions. Moreover, preconditions, as well as effects, of the actions it performs, might interact with those performed by other agents. Concurrency and cooperation are easily modeled by means of static and dynamic causal laws, that might involve constraints referring to action occurrences (even performed by different agents in different points in time). The specification of cost-based policies is also supported in order to better guide the search for a plan.

We provided a semantics for \mathcal{B}^{MAP} based on the notion of transition system, very much in the spirit of [8]. A concrete implementation has been realized by mapping the action language onto a concrete CLP system, such as SICStus Prolog. The implementation has been tested on a number of paradigmatic

multi-agent planning problems drawn from literature on multi-agent systems (cf., [3, 4]). The reader is referred to the web site www.dimi.uniud.it/dovier/CLPASP/MAP where the source code of the planner, together with some \mathcal{B}^{MAP} action descriptions are available.

We are currently expanding this line of research to experiment with alternative constraint-based platforms, e.g., Gecode, and to explore more complex forms of cooperation between agents.

References

- [1] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
- [2] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *J. of Logic Programming*, 31(1–3):85–117, 1997.
- [3] C. Boutilier and R. Brafman. Partial order planning with concurrent interacting actions. *JAIR*, 14:105–136, 2001.
- [4] M. Brenner. From individual perceptions to coordinated execution. In B. J. Clement, editor, *Proc. of Workshop on Multiagent Planning and Scheduling*, pages 80–88, 2005. Associated to ICAPS’05.
- [5] A. Dovier, A. Formisano, and E. Pontelli. Multivalued action languages with constraints in CLP(FD). In *ICLP2007*, volume 4670 of *LNCS*, pages 255–270, 2007. Extended version in http://www.dimi.uniud.it/dovier/PAPERS/rrUD_01-09.pdf.
- [6] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer Set Planning Under Action Costs. *Journal of Artificial Intelligence Research*, 19:25–71, 2003.
- [7] G. Gelfond and R. Watson. Modeling Cooperative Multi-Agent Systems. In *Proceedings of ASP Workshop*, 2007.
- [8] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.
- [9] A. Gerevini and D. Long. Plan Constraints and Preferences in PDDL3. Technical Report RT 2005-08-47, University of Brescia, 2005.
- [10] C. Jefferson, A. Miguel, I. Miguel, and S. A. Tarim. Modelling and solving English Peg Solitaire. *Comput. Oper. Res.*, 33(10):2935–2959, 2006.
- [11] A. Kakas, P. Torroni, and N. Demetriou. Agent Planning, negotiation and control of operation. In *ECAI*, 2004.
- [12] C. Knoblock. Generating Parallel Execution Plans with a Partial-Order Planner. In *Int. Conf. on AI Planning Systems*, pages 98–103, 1994.
- [13] A. Lopez and F. Bacchus. Generalizing graphplan by formulating planning as a CSP. In *Proc. of IJCAI-03*, pages 954–960. Morgan Kaufmann, 2003.
- [14] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Bradford Books, Cambridge, MA, 2001.
- [15] F. Sadri and F. Toni. Abductive Logic Programming for Communication and Negotiation Amongst Agents. In *ALP Newsletter*, 2003.
- [16] L. Sauro, J. Gerbrandy, W. van der Hoek, and M. Wooldridge. Reasoning about Action and Cooperation. In *AAMAS*, 2006.
- [17] M. Thielscher. Reasoning about actions with CHRs and finite domain constraints. *Lecture Notes in Computer Science*, 2401:70–84, 2002.
- [18] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

```

%%% Three places: Park -> 0 narrow road -> 1 Mary's house -> 2
place(0). place(1). place(2).

%%% Agents
agent(bob). agent(mary).

%%% Fluents
fluent(stay(X),0,2) :- agent(X).
fluent(door,0,1).
fluent(bell,0,1).

%%% Actions
action([X],move(A,B)) :- place(A), place(B), agent(X), 1 is abs(A-B).
action([X],ring) :- agent(X).
action([X],push) :- agent(X).
action([X],pull) :- agent(X).

%%% Executability
executable([X],move(0,1),[stay(X) eq 0]) :- agent(X).
executable([X],move(1,0),[stay(X) eq 1]) :- agent(X).
executable([X],move(1,2),[stay(X) eq 1, door eq 1]) :- agent(X).
executable([X],move(2,1),[stay(X) eq 2, door eq 1]) :- agent(X).
executable([X],ring,[stay(X) eq 1]) :- agent(X).
executable([X],push,[stay(X) eq 1, bell eq 1]) :- action([X],push).
executable([X],pull,[stay(X) eq 2, bell eq 1]) :- action([X],pull).

%%% EFFECTS
causes(stay(X) eq B, [actocc([X],move(A,B))]) :-
    action([X],move(A,B)).
causes(bell eq 1, [actocc([X],ring),bell eq 0]) :-
    action([X],ring).
causes(bell eq 0, [actocc([X],ring),bell eq 1]) :-
    action([X],ring).
causes(door eq 1,[actocc([X],push),actocc([Y],pull)]) :-
    action([X],push), action([Y],pull).

%%%%%% Initial and final conditions
initially(stay(bob) eq 0). initially(stay(mary) eq 2).
initially(bell eq 0). initially(door eq 0).

goal(stay(bob) eq 0). goal(stay(mary) eq 0).

```

Fig. 1. The “Bob and Mary” domain

Extending and implementing RASP

Andrea Formisano and Davide Petturiti

Università di Perugia, Dipartimento di Matematica ed Informatica
formis@dipmat.unipg.it, davidepetturiti@gmail.com

Abstract. In previous work an extension of ASP, called RASP (standing for ASP with Resources), has been proposed. RASP supports declarative reasoning on production and consumption of (amounts of) resources. The approach combines stable model semantics with quantitative reasoning and relies on an algebraic structure to support computations and comparisons of amounts. The resulting framework also offered some form of preference reasoning on resources usage. In this paper we go further in this direction by introducing more expressive constructs to support complex preferences specification. The complexity of establishing the existence of an answer set, in such an enriched framework, is then shown to be NP-complete. A prototypical implementation of RASP has been realized. The tool, named *raspberry*, consists in a compiler that, given a ground RASP program, produces a pure ASP encoding suitable to be processed by commonly available ASP-solvers.

Key words: Answer set programming, quantitative reasoning, preferences, language extensions.

Introduction

Previous work [4] proposed an extension of the Answer Set Programming (ASP) framework by explicitly introducing the notion of *resource*. Such an extension, named RASP (standing for ASP with Resources), supports both formalization and quantitative reasoning on consumption and production of amounts of resources. These are modeled by *amount-atoms* of the form $q\#a$, where q represents a specific type of resource and a denotes the corresponding amount. Resources can be produced or consumed (or declared available from the beginning). The processes that transform some amounts of resources into other resources are specified by *r-rules*, for instance, as in this simple example:

```
computer#1 ← cpu#1, harddisk#2, motherboard#1, ram_module#2.
```

where we model the fact that an instance of the resource `computer` can be obtained by “consuming” some other resources, in the indicated amounts.

In their most general form, *r-rules* might involve regular ASP literals together with amount-atoms. Semantics for RASP programs is given by combining stable model semantics with a notion of *allocation*. While stable models are used to deal with usual ASP literals, allocations are exploited to take care of amounts

and resources. Intuitively, an allocation assigns to each amount-atom a (possibly null) quantity. Quantities are interpreted in an auxiliary algebraic structure that supports comparisons and operations on amounts. Admissible allocations are those satisfying, for all resources, the requirement that one can consume only what has been produced. Clearly, alternative allocations might be possible, corresponding to different ways of using the same resources.

The RASP framework supports some limited form of preference specification on resource usage. The reader is referred to [5, 4] for a description on this and for a detailed comparison with previous approaches to preference reasoning, as well as for a discussion on the related works involving the notion of resource in logic programming frameworks.

The following simple example illustrates the way in which preferences are exploitable in RASP to specify different uses of the same resources.

Example 1. Assembling different PCs requires different sets of components (motherboard, processor(s), ram modules, etc.), depending on the kind of PC. In case of servers one might prefer SCSI disks rather than EIDE disks and vice versa for normal PCs:

```
cpu#5.    scsihd#5.    eidehd#9.    motherboard#7.    ram_module#20.
pc(server)#1 ← cpu#2, (scsihd#2>eidehd#2), motherboard#1, ram_module#4.
pc(desktop)#1 ← cpu#1, (eidehd#2>scsihd#2), motherboard#1, ram_module#2.
```

Notice that some resources might be declared as available from the beginning. This is done by means of *r-facts*. As in the first line of the above program.

In this paper we further enrich the RASP language by introducing more expressive constructs to support complex preferences specification (Sect. 2). For all the proposed extensions we provide an encoding into ASP. As a consequence, we show that the enriched framework retains the same computational complexity. We also briefly report (Sect. 3) on a prototypical implementation of RASP. Sect. 1 recalls syntax and semantics of RASP (more details can be found in the appendix).

1 A glimpse of RASP

In this section we briefly present the basic notions on RASP. For lack of space, here we have to summarize many aspects of RASP's semantics (refer to the appendix for a much complete presentation).

The language of RASP. 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* (*a-symbols*, for short), while the elements of Π_R are said *resource-predicates* (*r-predicates*).

A *program-term* (*p-term*) is either a variable or a constant symbol. An *a-term* is either a variable or an a-symbol.

Let $\mathcal{A}(X, Y)$ denote the collection of all atoms $p(t_1, \dots, t_n)$, with $p \in X$ and $\{t_1, \dots, t_n\} \subseteq Y$. Then, a *p-atom* is an element of $\mathcal{A}(\Pi_P, \mathcal{C} \cup \mathcal{V})$. An *r-term* is an element of $\Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C} \cup \mathcal{V})$. An *a-atom* is a writing of the form $q\#a$ where q is an r-term and a is an a-term. We call *resource-symbols* (*r-symbols*) the ground r-terms, i.e. the elements of $\tau_R = \Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C})$.

Some examples: in the two expressions $p\#3$ and $q(2)\#b$, p and $q(2)$ are r-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 $3, b \in \mathcal{C}_R$ a-symbols). Because the set of variables is not partitioned, the same variable may occur both as a p-term and as an a-term. Hence, we admit expressions such as $p(X)\#V$ where V, X are variables. In such cases, quantities are derived through instantiation.

Ground a-atoms contain no variables. A *program-literal* (*p-literal*, for short) L is a p-atom A or the negation *not* A of a p-atom (intended as negation-as-failure).¹ If $L = A$ (resp., $L = \text{not } A$) then \bar{L} denotes *not* A (resp., A).

We generalize the notion of a-atom so to permit the description of preferences. A preference in resources usage is specified by means of a *preference-list* of a-atoms (*p-list*, for short). It is a writing of the form $q_1\#a_1 > \dots > q_k\#a_k$, with $k \geq 1$. We say that $q_i\#a_i$ has *degree of preference* i in the p-list. Plainly, a-atoms are particular p-lists made of a single element (i.e., with $k = 1$).

An *r-literal* is either a p-literal or a p-list.

Notice that, we do not allow negation of a-atoms (cf., [4] for a discussion on this point). Finally, we distinguish between *p-rules* (plain ASP program rules, including the case of ASP *constraints*, i.e., rules with empty head) and *r-rules* which differ from p-rules in that they may contain p-lists (and hence, a-atoms).

An *r-rule* γ has the form

$$Idx : H \leftarrow B_1, \dots, B_m.$$

where B_1, \dots, B_m ($m > 0$) are r-literals, H is either a p-atom or a p-list, and at least one a-atom occurs in γ . *Idx* is of the form $[N_{1,1}-N_{1,2}, \dots, N_{h,1}-N_{h,2}]$, with $h \geq 1$, and each $N_{j,\ell}$ is a variable or a positive integer number.

Intuitively, when all the $N_{j,\ell}$ s are integers, *Idx* denotes the union of h (possibly void) intervals in $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. It is intended to restrain the number of times the rule can be used, i.e., *fired*. Such a number must belong to *Idx* or the rule cannot be fired at all. We admit that each $N_{j,\ell}$ is a variable. Then, after grounding (see below), each $N_{j,\ell}$ has to be instantiated to a positive integer.

Without loss of generality, in what follows we often assume $h = 1$ in r-rules.

An *r-fact* has the form $q\#a \leftarrow$, where $q\#a$ is a ground a-atom. It models a fixed amount of resource q that is available “from the beginning”.

A *rule* is either a p-rule or an r-rule. An *r-program* is a finite set of rules.

¹ We will only deal with negation-as-failure. Though, classical negation of program literals could be used in RASP programs and treated as usually done in ASP.

The grounding of an r-program P is the set of all ground instances of rules (and facts) of P , obtained through ground substitutions over the constants occurring in P .²

Notice that in any r-program only a finite number of a-symbols of \mathcal{C}_R occurs, also because all r-facts must be ground. Hence, as far as a-atoms are concerned, a finite number of ground instances can be generated by the grounding process. This is because all instances of a-terms are among the instances of the terms occurring in p-atoms. A “smart” grounder for RASP would avoid generating instances of r-rule where variables occurring as a-terms are instantiated to constants of \mathcal{C}_P instead of constants of \mathcal{C}_R . Such “wrong” instances are however both semantically and practically irrelevant (apart from the waste of space).

Semantics of RASP. Semantics of a (ground) r-program is determined by interpreting p-literals as usually done in ASP (namely, by exploiting stable model semantics) and a-atoms in an auxiliary algebraic structure Q . In principle, such a structure can be of full generality, provided that it supports operations and comparisons among amounts. For simplicity, we fix $Q = \mathbb{Z}$ as collection of *quantities* and consider given a mapping $\kappa : \mathcal{C}_R \rightarrow \mathbb{Z}$ that associates integers to a-symbols. Positive and negative integers will be used to model produced and consumed amounts, respectively. Moreover, to further simplify the treatment, we identify \mathcal{C}_R with \mathbb{Z} (and κ being the identity).

The semantics of RASP is described through a translation of r-rules into fragments of a plain ASP program. As regards a-atoms, the notion of *allocation* is introduced to model the amounts of resources that are consumed/produced when an r-rule is *fired*. Intuitively, given a ground r-program P , an allocation for P is a mapping that assigns elements of Q to the a-symbols in P in such a way that for each r-symbol q , the overall sum of quantities allocated to (produced and consumed) a-atoms of the form $q\#a$ is not negative. A last component of an interpretation copes with the repeated firing of r-rules. (Clearly, multiple firings must be taken into account by the allocation.)

We have the following definition (see also Def. 2 in App. A):

Definition 1. *An r-interpretation for a (ground) r-program P is a triple $\mathcal{I} = \langle I, \mu, \xi \rangle$, where $I \subseteq \mathcal{A}(\Pi_P, \mathcal{C})$, μ is an allocation for P , and ξ is a mapping $\xi : P \rightarrow \mathbb{N}^+$.*

In Def. 1, I plays the role of a usual answer set assigning truth values to p-literals and ξ associates to each rule the number of times the r-rule is used.

An interpretation \mathcal{I} for a ground r-program P determines which r-rules are fired. In particular, \mathcal{I} is an *answer set* of P if:

- it satisfies all the p-rules in P and all the fired r-rules (in the usual way) as concerns their p-literals;

² As it is well-known, at present, almost all ASP solvers perform a preliminary grounding step as they are able to find the answer sets of ground programs only. Work is under way to overcome at least partially this limitation (cf., [6], for instance).

- for each p-list (and hence, a-atom) occurring in a fired r-rule, one of its a-atoms is selected;
- the correct amounts are allocated for all the selected a-atoms while null amounts are allocated for all other a-atoms;
- the global balance of each r-symbol is not negative (i.e., all consumed amounts have been also produced by rule firings or available from r-facts).

App. A provides a formal definition of answer set for RASP (Def. 3).

Finally, we say that \mathcal{I} is an answer set of an r-program P if it is an answer set for the grounding of P .

Clearly, different selections of a-atoms in p-lists originate different answer sets. To impose a preference order on such answer sets, any *preference criterion* can be used. Such a criterion should order the collection of answer sets by reflecting the (preference degrees in the) p-lists. Any criterion has to take into account that each rule determines a (partial) preference ordering on answer sets, and it should aggregate/combine all such “local” partial orders to obtain a global one. This yields the notion of *most preferred answer set* of an r-program. Simple criteria to rank the collection of answer sets are described in [5].

Complexity. Computational complexity of (ground) RASP has been assessed in [4] for the simplified case in which all p-lists are singletons (namely, a-atoms might occur in a program but no preferences are specified). In that case, NP-completeness of deciding whether an r-program has an answer set is shown by providing a polynomial translation of ground programs from RASP into ASP. A correspondence is then exhibited between the answer sets of the r-program and the answer sets of its ASP encoding. We will see in the sequel that the same result holds also when preferences are involved.

As regards the complexity of the problem of determining if a given literal is true in a most preferred answer set, different results are obtained depending on the specific preference criterion which is adopted. In [5] it is shown that complexity of credulous reasoning for RASP with preferences are in line with that of LPOD [3]. For instance, if a *Pareto*-like criterion is adopted, Σ_P^2 -completeness of both RASP and LPOD is obtained.

2 Dealing with complex preferences

In this section we present a number of extensions of RASP that permit the specification of more complex forms of preference on resources usage. A first step in this direction has been made with the introduction of conditional p-lists in [5]. In what follows, we provide a generalization of such a proposal and establish a complexity result, thus settling a problem left open in previous works.

As a preliminary step, we introduce two notions of “*compound*” resource. Namely, given the a-atoms $q_1\#a_1, \dots, q_k\#a_k$, the occurrence of the writing $\{q_1\#a_1, \dots, q_k\#a_k\}$ in an r-rule, simply denotes the set of all these a-atoms. This notation has to be intended conjunctively, i.e., all the a-atoms are simultaneously consumed/produced when the r-rule is fired.

Similarly, $\{q_1\#a_1; \dots; q_k\#a_k\}$ denotes a collection of a-atoms whose production/consumption has to be intended disjunctively, i.e., when the r-rule is fired, exactly one of the listed a-atoms is non-deterministically chosen for consumption/production. Notice that, while the first notation is syntactic sugar, the second one introduces a novelty w.r.t. the previously proposed language of RASP. An ASP encoding of r-rules involving disjunctive compound resources can be designed. We do not describe here such an encoding, since it will be a particular cases of cp-lists (to be seen in Sect. 2.2).

2.1 Preferences between sets of a-atoms.

Let us slightly generalize the notion of p-list introduced in Sect. 1. Namely, we consider p-lists of the form $s_1 > \dots > s_k$, where each s_i is a compound resource, i.e., a set of a-atoms $\{q_{i,1}\#a_{i,1}, \dots, q_{i,k_i}\#a_{i,k_i}\}$. This form of p-list allows one to express preferences on set of a-atoms, instead of a single a-atoms. Hence, the intended meaning is that, for all j, ℓ , $j < \ell$, it is preferred to produce/consume all the resources in $\{q_{j,1}\#a_{j,1}, \dots, q_{j,k_j}\#a_{j,k_j}\}$ than all those in $\{q_{\ell,1}\#a_{\ell,1}, \dots, q_{\ell,k_\ell}\#a_{\ell,k_\ell}\}$. (Note that, by imposing $k_i = 1$, for all $i \in \{1, \dots, k\}$, we obtain p-lists as defined in Sect. 1.)

A second form of p-list involves compound resources of the kind $s_i = \{q_{i,1}\#a_{i,1}; \dots; q_{i,k_i}\#a_{i,k_i}\}$. In such cases, for all j, ℓ , $j < \ell$, it is preferred to produce/consume one of the resources in s_j , than one of those in s_ℓ . Situations in which both forms occur in the same p-list are handled coherently.

Let us now describe how an r-program P involving such kind of p-lists can be polynomially encoded into ASP. This encoding will show that adding preferences on resource usage in RASP does not affect its complexity. Let us focus on the case of a single p-list occurring in the body of a ground r-rule γ of P :

$$Idx : H \leftarrow B_1, \dots, B_m, s_1 > \dots > s_k$$

where, for each $i \in \{1, \dots, k\}$, $s_i = \{q_{i,1}\#a_{i,1}, \dots, q_{i,k_i}\#a_{i,k_i}\}$ and Idx is $[N_{1,1}-N_{1,2}, \dots, N_{h,1}-N_{h,2}]$, denoting a collection of disjoint integer intervals—i.e., $h \geq 1$, each $N_{j,b} \in \mathbb{N}^+$, and $N_{1,1} \leq N_{1,2} < N_{2,1} \leq N_{2,2} < \dots < N_{h,1} \leq N_{h,2}$. (The cases of p-lists occurring in the head and/or involving disjunctive compound resources can be treated similarly.)

For each $i \in \{1, \dots, k\}$ let aux_i be a fresh r-symbol not occurring elsewhere in the program. Then, we introduce the following r-rules:

$$(\gamma') \quad Idx : H \leftarrow B_1, \dots, B_m, pl\#1, z\#-1.$$

$$(\gamma'') \quad pl\#N_{h,2}.$$

$$(\gamma_i) \quad [1-N_{h,2}] : aux_i\#1 \leftarrow q_{i,1}\#a_{i,1}, \dots, q_{i,k_i}\#a_{i,k_i}, z\#1. \quad \text{for each } i \in \{1, \dots, k\}$$

where pl and z are fresh r-symbols (note that an a-atom with negative amount in the body of an r-rule, actually denotes resource production). The rationale is as follows. The auxiliary resource z acts as a counter of the number of firings of γ' : each time γ' is fired, an instance of z is produced and one instance of pl is consumed. Conversely, each resource aux_i is produced only if γ_i is fired and this can happen only by consuming one instance of z . Hence, each firing of γ'

corresponds to one firing of one of the γ_i s. The r-fact γ'' ensures that γ' cannot be fired more than $N_{h,2}$ times.

Notice that γ'' , γ' , and all of the γ_i do not involve preferences. Hence, their ASP encoding can be obtained through the translation introduced in [4]. We list here the relevant fragment of such encoding (the lack of space prevents us to provide the complete translation, see also App. B):

- (1) $r_rule(n_{\gamma'})$.
- (2) $firings(n_{\gamma'}, N_{i,1}..N_{i,2})$ for $i \in \{1, \dots, h\}$
- (3) $a_atom(n_{\gamma'}, 0, pl, -1)$.
- (4) $a_atom(n_{\gamma'}, 1, z, 1)$.
- (5) $\leftarrow \overline{B}_i, fired(n_{\gamma'})$ for $i \in \{1, \dots, m\}$
- (6) $H \leftarrow B_1, \dots, B_m, fired(n_{\gamma'})$.
- (7) $r_rule(n_{\gamma''})$.
- (8) $a_atom(n_{\gamma''}, 0, pl, N_{h,2})$.
- (9) $firings(n_{\gamma''}, 1) \quad fired(n_{\gamma''})$.
- (10) $r_rule(n_{\gamma_i})$ for $i \in \{1, \dots, k\}$
- (11) $firings(n_{\gamma_i}, 1..N_{h,2})$ for $i \in \{1, \dots, k\}$
- (12) $a_atom(n_{\gamma_i}, 0, aux_i, 1)$ for $i \in \{1, \dots, k\}$
- (13) $a_atom(n_{\gamma_i}, j, q_j, a_j)$ for $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, k_i\}$
- (14) $a_atom(n_{\gamma_i}, k_i + 1, z, -1)$ for $i \in \{1, \dots, k\}$

By means of facts (1), (7), and (10), unique identifiers for r-rules are introduced. Facts a_atom lists all the resources involved in r-rules. The predicate $firings$ is used to declare the admissible number of firings for each r-rule. Lines (5) and (6) relate the firing of a rule γ' to the satisfiability of its p-literals.

To complete the translation we impose a direct dependency between the uses of the resource pl and the resources aux_1, \dots, aux_k . This is done by means of this fragment of ASP program (where val and $iter$ act as domain predicates):

$$\begin{aligned}
&auxres(n_{\gamma'}, pl). \\
&res_pl(n_{\gamma'}, pl, aux_i, i) \quad \text{for } i \in \{1, \dots, k\} \\
&1\{use_pl(n_{\gamma'}, aux_1, I, 1, pl), \dots, use_pl(n_{\gamma'}, aux_k, I, 1, pl)\}1 \leftarrow \\
&\quad use(n_{\gamma'}, 0, pl, -1 * NumFirings), \quad iter(I), \\
&\quad I <= NumFirings, count(n_{\gamma'}, NumFirings). \\
&res_symb(Pl) \leftarrow auxres(G, Pl), r_rule(G). \\
&use(G, Pos, R, N) \leftarrow sum_use_pl(G, R, N), res_pl(G, Pl, R, Grade), \\
&\quad r_rule(G), val(N), a_atom(G, Pos, Pl, 1), N! = 0. \\
&sum_use_pl(G, R, N) \leftarrow N = sum\{A : use_pl(G, R, Iter, Q), val(Q), iter(Iter)\} \\
&\quad res_pl(G, Pl, R, Grade), r_rule(G), val(N).
\end{aligned}$$

Briefly, the facts in the first two lines associate each aux_i with the resource pl . The rule in the third line implements a correspondence between firings of γ' (i.e., uses of pl) and firings of γ_i (i.e., uses of aux_i). Being $NumFirings$ the number of times γ' is fired, this rule imposes that each time one instance of pl is used, (namely, once for each I , $1 \leq I \leq NumFirings$), exactly one instance of one resource among the aux_i s is used (i.e., one of the facts $use_pl(n_{\gamma'}, aux_i, I, 1, pl)$ is true). In turn, this forces one firing for the r-rule γ_i . (Notice the use of a

cardinality constraint to force the truth of exactly one atom *use_pl*. The use of such a constraint could be avoided, but it allows a more succinct encoding. See, for instance, [2] for details on cardinality constraints and on how to surrogate them through usual ASP rules.) The r-rules in the last three lines extend the inference engine introduced in [4]. They are used to evaluate the balance of each resource occurring in the initial p-list. This is achieved by means of an aggregate literal (cf., [7, 9, 10]) that sums up the amounts of real resources (i.e., the q_i s) corresponding to instances of the auxiliary resources (i.e., the aux_i s).

Observe that the above sketched translation can be applied to treat each p-list in a program, independently from the treatment of the other p-lists. The resulting encoding involves the introduction of a number of ASP rules and literals which is polynomially bounded w.r.t. the length of the given r-program.

Consequently, by exploiting the completeness and soundness results holding for RASP in case of absence of p-lists [4], we can establish a one-to-one correspondence between the answer set of an r-program and those of its ASP encoding. Hence, the complexity results mentioned in Sect. 1 are not affected by the presence of compound resources in p-lists.

2.2 Conditional p-lists

Conditional p-lists, called cp-lists, have been introduced in [5], where preferences between single a-atoms were considered. By proceeding in analogy to what done in Sect. 2.1, we introduce a generalization of cp-lists that admits preferences on compound resources. In this context, a cp-list is a writing of the form (r **pref_when** L_1, \dots, L_n), where r is a p-list, and L_1, \dots, L_n are p-literals.

The intended meaning of a cp-list occurring in the body of an r-rule ρ (the case of the head is analogous) is that, whenever ρ is fired, one of the (compound) resources s_i in $r = s_1 > \dots > s_k$ has to be consumed. If the firing occurs in correspondence of an answer set that satisfies L_1, \dots, L_n , then the choice is ruled by the preference expressed through r . Otherwise, if any of the L_i is not satisfied, a non-deterministic choice among the s_i s is made. (Hence the conjunction L_1, \dots, L_n does not need to be satisfied in order to fire ρ .) More precisely, if L_1, \dots, L_n does not hold, the r-rule containing the cp-list becomes equivalent to k r-rules, each containing exactly one of the s_i s, in place of the cp-list.

As before, let us focus on the case of a single cp-list occurring in the body of a ground r-rule ρ of P of the form:

$$Idx : H \leftarrow B_1, \dots, B_m, (s_1 > \dots > s_k \text{ **pref_when** } L_1, \dots, L_n)$$

where each s_i and Idx are as before. As done in Sect. 2.1, ρ can be replaced by these r-rules (where p and np are fresh p-atoms, and pl is a fresh r-symbol):

$$\begin{array}{ll} (\rho') & Idx : H \leftarrow B_1, \dots, B_m, pl\#1, z\#-1. \\ (\rho'') & Idx : pl\#1 \leftarrow p, s_1 > \dots > s_k, z\#1 \\ (\rho_i) & [1-N_{h,2}] : pl\#1 \leftarrow np, s_i, z\#1. \quad \text{for each } i \in \{1, \dots, k\} \\ & p \leftarrow \text{not } np. \quad np \leftarrow \text{not } p. \\ & \leftarrow np, L_1, \dots, L_n. \\ & \leftarrow p, \overline{L_j}. \quad \text{for each } j \in \{1, \dots, n\} \end{array}$$

Atoms p and np characterize the situations in which L_1, \dots, L_n are all satisfied or not, respectively. As before, z acts as a counter of the number of times ρ' is fired. Depending on the truth of p (resp., np), each firing of ρ' is forced to correspond to one firing of ρ'' (resp., of one among the ρ_i s).

All the above rules, except ρ'' , do not involve p-lists. Hence, we are left with the translation of ρ'' , which can be completed as outlined in Sect. 2.1. As before, the ASP encoding can be generalized to treat r-rules involving more than one cp-list (even with different kinds of compound resources), always introducing a polynomially bounded number of (occurrences of) atoms.

Notice that, the effect of using a compound resource $\{q_1\#a_1; \dots; q_k\#a_k\}$ can be rendered by using the cp-list $(q_1\#a_1 > \dots > q_k\#a_k \text{ *pref_when* } p, \text{ not } p)$, where p is any p-atom. Hence, its ASP encoding could be drawn from the one described so far. (A similar, approach is also viable when such a compound resource occurs within the scope of a cp-list.)

A second form of cp-list available in RASP is ($r \text{ *only_when* } L_1, \dots, L_n$). In this case, the p-list is effective only when the condition holds. More specifically, if the r-rule is fired in correspondence of an answer set that satisfies all of the literals L_1, \dots, L_m , the firing of the rule has to consume one of the resources in r . Which one is determined by the expressed preference. In case some L_i does not hold, the firing can still be performed but this does not require any consumption of resources in r .

Also in this case a generalization can be proposed to admit a p-list r involving compound resources. The ASP encoding of these enriched cp-lists largely coincides with the one described above. It is made of the r-rules ρ, ρ' , together with the following one (in place of all ρ_i s):

$$(\rho''') \quad [1-N_{h,2}] : \text{ pl}\#1 \leftarrow np, z\#1.$$

We conclude by observing that also in this case it is immediate to verify that the introduction of cp-lists does not affect the complexity of RASP. Completeness and soundness of the encoding also follow from the analogous results holding for RASP.

2.3 Expressing arbitrary preferences.

In general, there might be cases in which useful (conditional) preferences are not expressible as a linear order on a set of alternatives. Moreover, preferences might depend on specific contextual conditions that are not foreseeable in advance.

A simple example: next summer Jake would like to visit a foreign country. It might be that his sister Jill joins, but only if she does not get a job for the summer. Jack would like to go either to Brazil, France, Spain, Norway, or Iceland. He prefers Brazil the most. In case going to Brazil is not possible, he considers equally interesting visiting either France or Spain. The least preferred options are Norway and Iceland, but no preference is expressed between them, except in August, when Norway is preferred. This simple case of preference order, being not linear, cannot be modeled by p-lists.

P-sets are a generalization of p-lists that allows one to use any binary relation (not necessarily a partial order) in expressing (collections of alternative) p-lists. A p-set may occur in any place where a p-list does and is a writing of the form:

$$\{q_1\#a_1, \dots, q_k\#a_k \mid \text{pred}\}$$

where *pred* is a binary program predicate (defined elsewhere in the program).

Considering a specific answer set M , a particular extension is defined for the predicate *pred* (namely, the set of pairs $\langle a, b \rangle$ such that *pred*(a, b) is true in M). Let X be the set of r-symbols $\{q_1, \dots, q_k\}$. We consider the binary relation $R \subseteq X^2$ obtained by restricting to X the extension of *pred* in M . R is interpreted as a preference relation over X : namely, for any $q_i, q_j \in X$ the fact that $\langle q_i, q_j \rangle \in R$ models a preference of q_i on q_j . The case of p-lists is a particular case of p-sets, obtained when R describes a total order.³

As mentioned, R does not need to be a partial order, e.g., for instance, it may involve cycles. In such cases, those resources that belong to the same cycle in R are considered equally preferable (e.g., France and Spain, in the above example). On the other hand, R might be a partial relation. So, there might exist elements on X that are incomparable (e.g., Norway and Iceland in July).

Because of the presence of incomparable resources and equivalent resources, R can be seen as a representation of a collection of p-lists, one for each possible total order on X compatible with R . In particular, in case of equally preferable options, a non-deterministic choice is made. Whereas, in case of incomparable options, one among the possible total ordering of these options is arbitrarily selected. In the above example, the extension of *pred* should include the pairs $\langle \text{Brazil}, \text{France} \rangle$, $\langle \text{France}, \text{Spain} \rangle$, $\langle \text{France}, \text{Iceland} \rangle$, $\langle \text{France}, \text{Norway} \rangle$, and $\langle \text{Spain}, \text{France} \rangle$ (plus $\langle \text{Norway}, \text{Iceland} \rangle$, if it is August). Consequently, the admissible linear orders, if it is not August, are:

$$\begin{array}{ll} \text{Brazil} > \text{Spain} > \text{Norway} > \text{Iceland}, & \text{Brazil} > \text{France} > \text{Norway} > \text{Iceland}, \\ \text{Brazil} > \text{Spain} > \text{Iceland} > \text{Norway}, & \text{Brazil} > \text{France} > \text{Iceland} > \text{Norway}, \end{array}$$

while only the first two should be considered in August.

Let us consider the following r-rule η (where *Idx* is as before):

$$\text{Idx} : H \leftarrow B_1, \dots, B_m, \{q_1\#a_1, \dots, q_k\#a_k \mid \text{pred}\}.$$

By introducing a fresh symbol *ps* (that is univocally associated with the p-set at hand), the r-rule η can be replaced by these r-rules:

$$\begin{array}{ll} (\eta') & \text{Idx} : H \leftarrow B_1, \dots, B_m, \text{ps}\#1. \\ (\eta'') & \text{ps}\#N_{h,2}. \end{array}$$

These r-rules do not involve p-lists, hence they are treated as described in Sect. 2.1. To complete the translation we have to take into account each possible total order implicitly represented the extension of *pred* (i.e., its interpretation in the answer set at hand). To this aim the following ASP fragment is introduced to handle the specific p-set:

³ Notice that, here, we are introducing a change in the syntax of RASP. Namely we are admitting r-symbols as arguments of p-literals.

- (20) $dom(ps, q_i). \quad number(ps, i). \quad \text{for } i \in \{1, \dots, k\}$
- (21) $pset_name(ps) \leftarrow fired(n_\eta).$
- (22) $trcl(ps, X, Y) \leftarrow pred(X, Y), X \neq Y, fired(n_\eta).$
- (23) $1\{use_pl(n_\eta, q_1, I, a_1, ps), \dots, use_pl(n_\eta, q_k, I, a_k, ps)\}1 \leftarrow$
 $use(n_\eta, 0, ps, -1 * NumFirings), iter(I),$
 $I \leq NumFirings, count(n_\eta, NumFirings).$
- (24) $res_pl(n_\eta, ps, T, N) \leftarrow order(ps, T, N).$

In the above code, line (20) defines two domain predicates that enumerate the (relevant) arguments of $pred$ (the elements in X) and a collection of possible indices (as we will see, different indices represent different preference degrees), respectively. The following rules extend the inference engine and are independent from the specific p-sets:

- (30) $trcl(PS, X, Y) \leftarrow dom(PS, X), dom(PS, Y), dom(PS, Z),$
 $trcl(PS, X, Z), trcl(PS, Z, Y), X \neq Y, pset_name(PS).$
- (31) $equiv(PS, T_1, T_2) \leftarrow trcl(PS, T_1, T_2), trcl(PS, T_2, T_1),$
 $dom(PS, T_1), dom(PS, T_2), pset_name(PS).$
- (32) $1\{ord_idx(PS, T, N) : number(PS, N)\}1 \leftarrow dom(PS, T), pset_name(PS).$
- (33) $used_idx(PS, N) \leftarrow dom(PS, T), ord_idx(PS, T, N),$
 $number(PS, N), pset_name(PS).$
- (34) $\leftarrow dom(PS, T), ord_idx(PS, T, N), number(PS, N),$
 $N > 1, N_1 = N - 1, not\ used_idx(PS, N_1), pset_name(PS).$
- (35) $\leftarrow dom(PS, T_1), dom(PS, T_2), ord_idx(PS, T_1, N_1), ord_idx(PS, T_2, N_2),$
 $number(PS, N_1), number(PS, N_2), N_2 \leq N_1, T_1 \neq T_2,$
 $trcl(PS, T_1, T_2), not\ trcl(PS, T_2, T_1), pset_name(PS).$
- (36) $\leftarrow not\ equiv(PS, T_1, T_2), dom(PS, T_1), dom(PS, T_2), ord_idx(PS, T_1, N),$
 $ord_idx(PS, T_2, N), number(PS, N), T_1 \neq T_2, pset_name(PS).$
- (37) $ord_idx(PS, T_2, N) \leftarrow equiv(PS, T_1, T_2), dom(PS, T_1), dom(PS, T_2),$
 $ord_idx(PS, T_1, N), number(PS, N), pset_name(PS).$
- (38) $order(PS, T, N) \leftarrow ord_idx(PS, T, N), not\ other(PS, T, N), dom(PS, T),$
 $number(PS, N), pset_name(PS).$
- (39) $other(PS, T, N) \leftarrow order(PS, T_2, N), T \neq T_2, dom(PS, T), dom(PS, T_2),$
 $number(PS, N), pset_name(PS).$

The rule at line (30), together with the one in line (22) (which appears in one instance for each specific p-set), evaluates the transitive closure of the relation R defined by $pred$. Line (31) determines the equivalences between r-symbols. Rules at lines (32)–(34) index the elements of X with consecutive (possibly repeated) integers. Lines (35)–(37) restrict the possible indexing to those that do not violate the (closure of) the relation R : elements are assigned equal indices if and only if they are equally preferred (i.e., equivalent in R). Higher indices are assigned to less preferred r-symbols. Finally, rules (38)–(39) generate (compatibly with the extension of $pred$) all admissible orders for X . Each of these orders admits a corresponding p-list. Such indexing of r-symbols is used (in the context of a specific answer set) through the rules (23)–(24), seen before, analogously to what done for the translation of plain p-lists (cf., Sect. 2.1).

Example 2. Let us consider the example mentioned above. This is a fragment of r-program describing Jack’s preferences on buying plane tickets:

```

jack_happy ← {ticket(b)#N, ticket(f)#N, ticket(s)#N, ticket(i)#N, ticket(n)#N
              | jack_pref}, is_summer, num_of_tickets(N).
jack_pref(ticket(b), ticket(f)).  jack_pref(ticket(f), ticket(i)).
jack_pref(ticket(f), ticket(s)).  jack_pref(ticket(s), ticket(f)).
jack_pref(ticket(f), ticket(n)).  jack_pref(ticket(n), ticket(i)) ← is_august.
ticket(P)#N ← money#C, cost(P, C1), C = C1 * N, num_of_tickets(N).
cost(b, 9).  cost(f, 6).  cost(s, 5).  cost(i, 5).  cost(n, 6).
num_of_tickets(1) ← jill_works.  num_of_tickets(2) ← not_jill_works.

```

Here is another example involving different forms of preference specification:

Example 3. Assume that, in making a cake or some cookies, you might choose among different ingredients, and you have to consider some constraints due to possible allergy or diet. Note that the preference among *chocolate*, *nuts* and *coconut*, i.e., a particular p-list, is determined depending on the extension of the predicate *less_caloric*, which might be different for different answer sets and has to be established dynamically (through the predicate *calory*, defined elsewhere in the program).

```

cake#1>cookie#15 ← egg#2, flour#2, raisin#4,
                  ({aspartame#1, skim_milk#6}>{sugar#4, whole_milk#6} pref_when diet),
                  ({vanilla#1; lemon#2}>cinnamon#1 only_when not allergy),
                  {chocolate#1, nuts#1, coconut#1 | less_caloric}.
less_caloric(X, Y) ← calory(X, A), calory(Y, B), A < B.
calory(X, Y) ← ...

```

3 Raspberry: a concrete implementation

The above-outlined translation from (ground) RASP programs into ASP has been implemented in a stand alone tool, named *raspberry*.⁴ Raspberry, essentially consists in a compiler that, given an r-program, produces its pure ASP encoding. In turn, this encoding is joined to an ASP specification of an inference engine which performs the real reasoning on resources allocation (and that remains independent from the particular program at hand). The resulting ASP program is suitable to be processed by commonly available ASP-solvers (in particular, the input syntax for gringo and lparse are supported [1]).

The answer sets found by the ASP solver encode the solutions of the RASP problem.

As regards preference criteria (see [5]), the search for most preferred solutions is implemented by exploiting the optimization features offered, for example, by smodels and clasp.

⁴ Actually, non-ground programs are also correctly translated, provided that all atoms are ground (i.e., variables occur only in p-literals). The treatment of generic non-ground r-rules is subject for future work.

The prototypical release of raspberry has been implemented in C++ and is available in <http://www.dipmat.unipg.it/~formis/raspberry>. Such a prototype is still under development, but covers all of the features described in this paper. As done in this paper, also in the implementation, we fix the algebraic structure Q (modeling amounts) to be the set \mathbb{Z} of integer numbers. This is because commonly available ASP solvers offer operations on integers as built-in features. Refinements of the tool able to deal with other groups are a theme for future work. In the mentioned web page, together with the tool, some examples and a minimal documentation can be retrieved.

We conclude this section by noting that, since the concrete implementation strictly reflects the described translation from RASP to ASP, its correctness and completeness directly follow.

Concluding remarks

In this work, we started from RASP, an extension of ASP that supports reasoning about resources and their amounts as well as the specification, in form of rules, of those processes that produce and consume resources. We proposed several extensions of the RASP language in order to allow the specification of complex forms of preferences on resource allocation. We also shown that such new language constructs, far from being, in many cases, trivial syntactic sugar, do not imply increase in the computational complexity of the framework. In particular, the problem establishing the existence of an answer set for an r-program is still NP-complete. Consequently, the complexity of credulous reasoning for RASP, when such form of complex preferences are involved, is in line with that of LPOD [3].

The extension of (R)ASP we described is, to the best of our knowledge, an original proposal. This is so for, at least, two aspects. First, the kind of preference admitted in RASP have a *local scope*: each p-list (cp-list, or p-set) is seen in the context of a particular rule (which models a specific process in manipulating some resources). Consider, for instance, the r-program of Example 1, where completely antithetic orders are expressed in the two r-rules. Notice that both r-rules might be fired at the same time, since enough resources are available. Clearly, such a local aspect is strictly related to the constraints on global resource balance and resource availability. Consequently, preferences locally stated for different rules might/should be expected to interact “over distance” with those expressed in other rules.

A second novelty is represented by the possibility of dealing with non-linear preference orders in resource usage (cf., Sect. 2.3). In this sense, DLPOD [8] is a similar proposal, recently appeared in literature. In this case, pure ASP is considered and preferences are imposed on the truth of program literals occurring in heads of rules (in analogy to what done in LPOD [3]). (So, also in this proposal preferences have a global flavor.) These preferences, in turn, determine an ordering of the answer sets of a DLPOD program. In [8] non-linear preferences

are introduced by combining ordered and unordered disjunction in the same framework.

Clearly, RASP and DLPOD have different purposes and tend to solve different problems and consequently have different expressive power. In particular, notice that DLPOD has LPOD as special case. Hence, it is reasonable that RASP and DLPOD also differs as regards the computational complexity, for instance, of credulous reasoning. We shown that RASP is in line with LPOD while [8] conjectures Σ_P^3 -completeness for DLPOD.

Acknowledgements The authors would like to thank Torsten Schaub and Sven Thiele for the support and the advices about the grounder gringo. The parser of rasperry plainly extends the one included in the distribution of gringo. This research has been partially supported by GNCS.

References

- [1] Web references for some ASP solvers. ASSAT: assat.cs.ust.hk; Ccalc: www.cs.utexas.edu/users/tag/ccalc; Clasp: potassco.sourceforge.net; Cmodels: www.cs.utexas.edu/users/tag/cmodels; DeReS and aspps: www.cs.uky.edu/ai; DLV: www.dbai.tuwien.ac.at/proj/dlv; Smodels: www.tcs.hut.fi/Software/smodels.
- [2] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
- [3] G. Brewka, I. Niemelä, and T. Syrjänen. Logic programs with ordered disjunction. *Computational Intelligence*, 20(2):335–357, 2004.
- [4] S. Costantini and A. Formisano. Answer set programming with resources. *Journal of Logic and Computation*, 2009. To appear. Draft available as Report-16/2008 of Dip. di Matematica e Informatica, Univ. di Perugia: www.dipmat.unipg.it/~formis/papers/report2008_16.ps.gz.
- [5] S. Costantini and A. Formisano. Modeling preferences and conditional preferences on resource consumption and production in ASP. *Journal of of Algorithms in Cognition, Informatics and Logic*, 64(1), 2009.
- [6] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: ASP with lazy grounding. In *Proc. of LaSh08*, 2008.
- [7] W. Faber, G. Pfeifer, N. Leone, T. Dell’Armi, and G. Ielpa. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming*, To appear.
- [8] P. Kärgler, N. Lopes, A. Polleres, and D. Olmedilla. Towards logic programs with ordered and unordered disjunction. In *Proc. of ASPOCP’08 Workshop of ICLP08*, 2008.
- [9] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *Prog. of the 1991 International Logic Programming Symposium*, pages 387–401, 1991.
- [10] T. C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3), 2007.

The material of the following sections is essentially drawn from [4, 5]. It has been recalled here for ease of the reader.

A A quick rush through the semantics of RASP

Semantics of a (ground) r-program is determined by interpreting p-literals as in ASP and a-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. On the other hand, an interpretation involves the allocation of actual quantities to a-atoms. In fact, this allocation is one of the components of an interpretation: an answer set of an r-program will model an r-rule only if it is satisfied (in the usual way, relying on stable model semantics) as concerns its p-literals, and the correct amounts are allocated for the a-atoms. A last component of an interpretation copes with the repeated firing of a rule: in case of several firings, the resource allocation must be iterated accordingly.

In order to define semantics of r-programs, we have to fix an interpretation for a-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 a-symbols. Positive (resp. negative) integers will be used to model produced (resp. consumed) amounts of resources.

For the sake of simplicity, in what follows we will identify \mathcal{C}_R with \mathbb{Z} (and κ being the identity). This will not cause loss in the generality of the treatment.

Notation. Before going on, we introduce some useful notation. Given two sets X, Y , let $\mathcal{FM}(X)$ denote the collection of all finite multisets 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.

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 .

To deal with the disjunctive aspect of p-lists and to model the degrees of preference, we mark each a-atom with an integer index. For each p-list its composing a-atoms are associated, from left to right, with successive indices starting from 1. For single a-atoms, the index will always be 0. So, any a-atom will be represented as a pair in $\mathbb{N} \times Q$ that we call an *amount couple*. For example: an interpretation for *skim_milk#2 > whole_milk#2*, occurring in the head of an r-rule, will involve one of the couples $\langle 1, 2 \rangle$ and $\langle 2, 2 \rangle$, where the first components of the couples reflect the degree of preference and the second elements are the quantities. For single a-atoms (in a head of an r-rule), such as *egg#2*, no preference is involved and a potential interpretation is $\langle 0, 2 \rangle$.

Given an amount couple $r = \langle n, x \rangle$, let $degree(r) = n$ 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, as one expects: namely, if X is a multiset then $degree(X)$ is defined as the multiset $\{\{n \mid \langle n, x \rangle \text{ is in } X\}\}$, and similarly for $amount(X)$. E.g., if $X = \{\{ \langle 1, 2 \rangle, \langle 3, 1 \rangle, \langle 1, 2 \rangle \}\}$ then $degree(X)$ is $\{\{1, 3, 1\}\}$ and $amount(X)$ is $\{\{2, 1, 2\}\}$.

Interpretation of RASP Programs. In what follows, we will apply a syntactical restriction on the form of the r-rules. Namely, we impose that each a-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 case.

An interpretation for an r-program P must determine an allocation of amounts for all occurrences of such amount symbols in P . We represent produced quantities (i.e., a-atoms in heads) by positive values, while negative values model consumed amounts (i.e., a-atoms in bodies). For each r-symbol q , the overall sum of quantities allocated to (produced and consumed) a-atoms of the form $q\#a$ must not be negative. The collection \mathbb{S}_P of all potential allocations (i.e., those having a non-negative global balance)—for any single r-symbol occurring in P (considered as a set of rules)—is the following collection of mappings:

$$\mathbb{S}_P = \left\{ F \in (\mathcal{FM}(\mathbb{N} \times Q))^P \mid 0 \leq \sum \left(\bigcup_{\gamma \in P} \text{amount}(F(\gamma)) \right) \right\}$$

The rationale behind the definition of \mathbb{S}_P is as follows. Let q be a fixed r-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 quantities to each occurrence of a-atoms of the form $q\#a$ in γ . All such F s satisfy (by definition of \mathbb{S}_P) the 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.

An r-interpretation of the amount symbols in a ground r-program P is defined by providing a mapping $\mu : \tau_R \rightarrow \mathbb{S}_P$. Such a function determines, for each r-symbol $q \in \tau_R$, a mapping $\mu(q) \in \mathbb{S}_P$. In turn, this mapping $\mu(q)$ assigns to each rule $\gamma \in P$ a multiset $\mu(q)(\gamma)$ of quantities, as explained above. The use of multisets allows us to handle multiple copies of the same a-atom: each of them corresponds to a different amount of resource to be taken into account.

The following is a more precise definition of *r-interpretation* (cf., Def. 1).

Definition 2. An r-interpretation for a (ground) r-program P is a triple $\mathcal{I} = \langle I, \mu, \xi \rangle$, with $I \subseteq \mathcal{A}(\Pi_P, \mathcal{C})$, $\mu : \tau_R \rightarrow \mathbb{S}_P$, and ξ a mapping $\xi : P \rightarrow \mathbb{N}^+$.

Intuitively: I plays the role of a usual answer set assigning truth values to p-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 p-rules and r-facts. For this kind of rules we assume the interval $[N_1-N_2] = [1-1]$ as implicitly specified in the rule definition, as a constraint on the number of firings.

The firing of an r-rule (which may involve consumption/production of resources) can happen only if the truth values of the p-literals satisfy the rule. We reflect the fact that the satisfaction of an r-rule γ depends on the truth of its p-literals by introducing a suitable fragment of ASP program $\hat{\gamma}$. Let the r-rule γ have L_1, \dots, L_k as p-literals and R_1, \dots, R_h as a-atoms (or p-lists). The ASP-program $\hat{\gamma}$ is so defined:

$$\hat{\gamma} = \begin{cases} \{ \leftarrow \overline{L_1}, \dots, \leftarrow \overline{L_k} \} & \text{if the head of } \gamma \text{ is an a-atom or a p-list} \\ \{ \leftarrow \overline{L_1}, \dots, \leftarrow \overline{L_k}, \\ H \leftarrow L_1, \dots, L_k \} & \text{if } \gamma \text{ has the p-atom } H \text{ as head} \\ & \text{and } h > 0 \\ \{ \gamma \} & \text{otherwise (e.g., } \gamma \text{ is a p-rule).} \end{cases}$$

Def. 3, to be seen, states that in order to be a model, an r-interpretation that allocates non-void amounts to the r-symbols of γ , has to model the ASP-rules in $\hat{\gamma}$. Some preliminary notion is in order.

So far we have developed a semantic structure in which r-rules are interpretable by singling-out suitable collections of amount couples. Different ways of allocating amount of resources to an r-program are possible. To be acceptable, an allocation has to reflect, for each p-list r in P , one of the admissible choices that r represents. In order to denote such admissible choices we need some further notation. Let ℓ be either a p-list (or, an a-atom) in r-rule γ . Let

$$\text{setify}(\ell) = \begin{cases} \langle 0, q, a \rangle & \text{if } \ell \text{ is } q\#a \\ \langle 1, q_1, a_1 \rangle, \dots, \langle h, q_h, a_h \rangle & \text{if } \ell \text{ is } q_1\#a_1 > \dots > q_h\#a_h \text{ for } h > 1 \end{cases}$$

We will use *setify* to represent the a-atoms of rules as triples denoting: the position in each preference list where they occur; the r-symbol they contain; the amount that is required for this r-symbol in that preference list. We generalize the notion to any multiset X : $\text{setify}(X) = \{\{\text{setify}(\ell) \mid \ell \text{ in } X\}\}$.

Let $r\text{-head}(\gamma)$ and $r\text{-body}(\gamma)$ denote the multiset of p-lists occurring in the head and in the body of γ , respectively. To distinguish, in the representation, between a-atoms occurring in heads and in bodies, we define $\text{setify}_b(\gamma)$ and $\text{setify}_h(\gamma)$ as the multisets $\{\{\text{setify}(x) \mid x \in r\text{-body}(\gamma)\}\}$ and $\{\{\text{setify}(x) \mid x \in r\text{-head}(\gamma)\}\}$, respectively.

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 a-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 a-atoms occurring in the body of γ are negative, as these resources are *consumed*. Vice versa, the quantities associated to a-atoms of the head are positive, as these resources are *produced*.

$$\mathcal{R}(\gamma) = \left\{ \begin{aligned} & \{\{\langle i, q, \kappa(a) \rangle \mid \langle i, q, a \rangle = c_1(S_1) \text{ and } S_1 \text{ in } \text{setify}_h(\gamma)\}\} \\ & \cup \{\{\langle i, q, -\kappa(a) \rangle \mid \langle i, q, a \rangle = c_2(S_2) \text{ and } S_2 \text{ in } \text{setify}_b(\gamma)\}\} \\ & \mid \text{for } c_1 \text{ and } c_2 \text{ choice functions for } \text{setify}_h(\gamma) \text{ and } \text{setify}_b(\gamma), \text{ resp.} \end{aligned} \right\}$$

where c_1 (resp. c_2) ranges on all possible choice functions for $\text{setify}_h(\gamma)$ (resp. for $\text{setify}_b(\gamma)$).

To account for multiple firings, we need to be able 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

$$\mathcal{R}^n(\gamma) = \left\{ \bigcup \{\{X_1, \dots, X_n\} \mid \{X_1, \dots, X_n\} \in \mathcal{FM}(\mathcal{R}(\gamma))\} \right\}$$

and

$$\mathcal{R}^n(q, \gamma) = \left\{ \{\{\langle i, v \rangle \mid \langle i, q, v \rangle \text{ is in } X\} \mid X \in \mathcal{R}^n(\gamma)\} \right\}.$$

While $\mathcal{R}(\gamma)$ represents all the different ways of choosing one a-atom from each p-list of γ , the collection $\mathcal{R}^n(\gamma)$ represents all the possible ways of making n times this choice (possibly, in different manners). Fixed an r-symbol q , the set $\mathcal{R}^n(q, \gamma)$ extracts from each alternative in $\mathcal{R}^n(\gamma)$ the multiset of amount couples relative to q . Def. 3 exploits the set $\mathcal{R}^n(q, \gamma)$ to impose restrictions on the global resource balance, for each q .

Definition 3. 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 $\gamma \in P$

$$\left(\forall q \in \tau_R (\mu(q)(\gamma) = \emptyset) \right) \vee \left(\forall q \in \tau_R (\mu(q)(\gamma) \in \mathcal{R}^{\xi(\gamma)}(q, \gamma)) \wedge (N_{1,1} \leq \xi(\gamma) \leq N_{1,2}) \right)$$

- I is a stable model for the ASP-program \widehat{P} , so defined

$$\widehat{P} = \bigcup \left\{ \widehat{\gamma} \mid \begin{array}{l} \gamma \text{ is a p-rule in } P, \text{ or} \\ \gamma \text{ is a r-rule in } P \text{ and } \exists q \in \tau_R (\mu(q)(\gamma) \neq \emptyset) \end{array} \right\}$$

The two disjuncts in Def. 3 correspond to the two cases: a) the rule γ is not fired, so null amounts are allocated to all its a-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). Note 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 $\mathcal{R}^{\xi(\gamma)}(q, \gamma)$.

B An ASP encoding of RASP

Let $\Pi_{\mathcal{T}}$ be a set of fresh p-symbols with $\{\text{notfired}, \text{fired}, \text{use}, \text{r_rule}, \text{a_atom}\} \subseteq \Pi_{\mathcal{T}}$. Given a ground r-program S , let $S_{\mathcal{R}} \subseteq S$ be the set of r-rules in S and let $S_{\mathcal{P}} = S \setminus S_{\mathcal{R}}$ (i.e., the p-rules). For any set of ASP rules X , let $\text{atoms}(X)$ denote the set of all atoms occurring in X . For any ASP rule γ , let $\text{lits}^+(\gamma)$ (resp., $\text{lits}^-(\gamma)$) be the set of atoms occurring positively (resp., negatively) in the body of γ . Moreover, let $\text{head}(\gamma)$ be the set of atoms occurring in the head of γ .

A translation \mathcal{T} from RASP into ASP is defined as follows. We start by univocally naming each r-rule γ in $S_{\mathcal{R}}$. This is done by introducing a fresh constant symbol r_{γ} (i.e., a constant not appearing elsewhere) and the fact:

$$\text{r_rule}(r_{\gamma}). \quad (1)$$

Let the r-rule γ be of the form

$$q_0 \# a_0, \dots, q_h \# a_h \leftarrow q_{h+1} \# a_{h+1}, \dots, q_k \# a_k, L_1, \dots, L_n.$$

for some $0 \leq h \leq k$, $n \geq 0$, and $(k - h) + n > 0$, with L_1, \dots, L_n p-literals. For each a-atom $q_i \# a_i$ in γ we introduce the fact:

$$\text{a_atom}(r_{\gamma}, i, q_i, \hat{a}_i). \quad (2)$$

where $\hat{a}_i = a_i$ if $0 \leq i \leq h$ and $\hat{a}_i = -a_i$ if $h < i \leq k$. These facts represent in the ASP translation the a-atoms occurring in $S_{\mathcal{R}}$. The second argument of a_atom is needed in the ASP translation to distinguish among different occurrences of identical a-atoms of the r-rule. Recall, in fact, that multiple copies of the same a-atom must not be identified, since they corresponds to a different amount of resource. Keeping track of multiple copies of a-atoms reflects, in the translation into ASP code, the use of multisets in defining the semantics of r-programs.

As mentioned, the two disjoints of the formula in Def. 3 discriminate the two situations in which an r-rule γ is fired or not. These two situations are modeled in ASP through the two rules

$$\text{fired}(r_{\gamma}) \leftarrow \text{not notfired}(r_{\gamma}). \quad \text{notfired}(r_{\gamma}) \leftarrow \text{not fired}(r_{\gamma}). \quad (3)$$

Whenever an r-rule γ is fired, all its resources are consumed/produced. We represent the fact that a certain amount a_i of a resource q_i (due to the a-atom $q_i \# a_i$ in γ), is actually used if and only if γ is fired, through the ASP rules (for each $i \in \{0, \dots, k\}$):

$$\begin{aligned} \text{use}(r_{\gamma}, i, q_i, \hat{a}_i) &\leftarrow \text{fired}(r_{\gamma}), \text{a_atom}(r_{\gamma}, i, q_i, \hat{a}_i). \\ \text{fired}(r_{\gamma}) &\leftarrow \text{use}(r_{\gamma}, i, q_i, \hat{a}_i). \\ \text{notfired}(r_{\gamma}) &\leftarrow \text{not use}(r_{\gamma}, i, q_i, \hat{a}_i). \end{aligned} \quad (4)$$

Finally, we impose that the firing of γ has to be enabled by the truth of the literals L_1, \dots, L_n , through the ASP rules (for each $j \in \{1, \dots, n\}$):

$$\text{aux}_{L_i, \gamma} \leftarrow \text{not aux}_{L_i, \gamma}, \overline{L_i}, \text{fired}(r_{\gamma}). \quad (5)$$

The translation $\mathcal{T}(\gamma)$ of γ is made of the rules (1) and (2)-(5):

$$\begin{array}{ll}
r_rule(r_\gamma). & a_atom(r_\gamma, i, q_i, \hat{a}_i). \\
fired(r_\gamma) \leftarrow not\ notfired(r_\gamma). & notfired(r_\gamma) \leftarrow not\ fired(r_\gamma). \\
use(r_\gamma, i, q_i, \hat{a}_i) \leftarrow fired(r_\gamma), a_atom(r_\gamma, i, q_i, \hat{a}_i). & fired(r_\gamma) \leftarrow use(r_\gamma, i, q_i, \hat{a}_i). \\
notfired(r_\gamma) \leftarrow not\ use(r_\gamma, i, q_i, \hat{a}_i). & aux_{L_i, \gamma} \leftarrow not\ aux_{L_i, \gamma}, \overline{L}_i, fired(r_\gamma).
\end{array}$$

where i and j range over $\{0, \dots, k\}$ and $\{1, \dots, n\}$, respectively.

The above described translation can to be slightly modified to treat r-rules having a p-atom as head (see [4] for the details). Facts are treated as r-rules that are supposed to be always fired. Hence, if γ is the r-factor $q\#a$. then $\mathcal{T}(\gamma)$ is as follows:

$$\begin{array}{ll}
r_rule(r_\gamma). & a_atom(r_\gamma, 0, q, a). \\
fired(r_\gamma). & use(r_\gamma, 0, q, a).
\end{array} \tag{6}$$

By defining $\mathcal{T}(\gamma) = \{\gamma\}$ for all p-rules, the translation of an r-program S is defined as the ASP program $\mathcal{T}(S) = \bigcup_{\gamma \in S} \mathcal{T}(\gamma)$.

Let M be a model of the program $\mathcal{T}(S)$. For some r-symbols q , some of the atoms $use(r_\gamma, i, q, a)$, occurring in $\mathcal{T}(S)$, are true in M . These atoms are intended to represent the amounts of resources involved in fired r-rules. To take into account the constraints on global balance of the allocated amounts, we introduce the condition $pos(M)$:

$$pos(M) = \forall q \in \tau_{\mathcal{R}} \left(\sum \{a \mid use(r_\gamma, i, q, a) \in M\} > 0 \right) \tag{7}$$

Such a condition can be imposed in the ASP encoding by means of a constraint involving an aggregate *sum*, as follows (cf., among others [9, 10, 7]):

$$\leftarrow sum\{A : use(Rule, I, Q, A)\} < 0, res_symb(Q).$$

Moreover, for each r-symbol q occurring in $\mathcal{T}(P)$ a fact $res_symb(q)$ is added to the encoding. Observe that we are introducing an aggregate literal in a constraint. Hence, no literal in $\mathcal{T}(S)$ is defined depending on such aggregate. This ensures that the resulting program is aggregate stratified. Stable model semantics can be smoothly extended to such class of programs by generalizing the notion of reduct of a program (cf., [9, 10, 7]).

An inference engine for RASP. The above outlined translation can be ameliorated, since the rules (3) and (4) can be factorized by exploiting variables. This allows us to design the core of an inference engine for reasoning on resource allocations and imposing correct usage of resources:

$$\begin{array}{l}
fired(Rule) \leftarrow not\ notfired(Rule), r_rule(Rule). \\
notfired(Rule) \leftarrow not\ fired(Rule), r_rule(Rule). \\
use(Rule, I, Res, Amount) \leftarrow fired(Rule), a_atom(Rule, I, Res, Amount). \\
fired(Rule) \leftarrow use(Rule, I, Res, Amount). \\
notfired(Rule) \leftarrow not\ use(Rule, I, Res, Amount).
\end{array}$$

The balance for each resource is evaluated by the following fragment of code. Observe the use of an ASP constraint involving an aggregate function to evaluate sums and to impose the condition (7):

$$\begin{array}{l}
res_symb(Res) \leftarrow a_atom(Rule, I, Res, Amount). \\
\leftarrow sum\{A : use(Rule, I, Q, A)\} < 0, res_symb(Q).
\end{array}$$

An ASP-based solver for RASP would act as follows: first each r-rule of the RASP program is translated as previously explained (recall that p-rules are left unchanged); then, the rendering of all r-rules must be joined with the above ASP program that acts as an inference engine and performs the concrete reasoning activity on resource allocations; finally, the answer sets (if any) of the obtained ASP program are calculated by means of a standard ASP solver [1]. From each answer set M so computed, an answer set \mathcal{I}_M of the original r-program can be extracted.

Transformational Verification of Linear Temporal Logic

Alberto Pettorossi¹, Maurizio Proietti², and Valerio Senni¹

¹ DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{pettorossi,senni}@disp.uniroma2.it

² IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
proietti@iasi.cnr.it

Abstract. We present a new method for verifying Linear Temporal Logic (LTL) properties of finite state reactive systems based on logic programming and program transformation. We encode a finite state system and an LTL property which we want to verify as a logic program on infinite lists. Then we apply a verification method consisting of two steps. In the first step we transform the logic program that encodes the given system and the given property into a new program belonging to the class of the so-called *linear monadic ω -programs* (which are stratified, linear recursive programs defining nullary predicates or unary predicates on infinite lists). This transformation is performed by applying rules that preserve correctness. In the second step we verify the property of interest by using suitable proof rules for linear monadic ω -programs. These proof rules can be encoded as a logic program which always terminates, if evaluated by using tabled resolution. Although our method uses standard program transformation techniques, the computational complexity of the derived verification algorithm is essentially the same as the one of the Lichtenstein-Pnueli algorithm [9], which uses sophisticated *ad-hoc* techniques.

1 Introduction

Model checking is a very successful technique for verifying finite state reactive systems, such as protocols, concurrent systems, and digital circuits [5]. In model checking the reactive system is formally specified as a Kripke structure and the properties to be verified are specified as formulas of a suitable temporal logic. Among the most popular logics that have been proposed are the temporal logic CTL* and its two fragments: (i) the Computational Tree Logic CTL, and (ii) the Linear-time Temporal Logic LTL (see [5] for a comprehensive account). In this paper we will focus our attention on the logic LTL.

The logic LTL has been shown to be decidable for finite state systems and several algorithms for LTL model checking have been proposed. These algorithms use sophisticated *ad-hoc* techniques based on tableaux [9], symbolic representations using BDDs [2,4], translations to Büchi automata [21], and translations to alternating automata [11].

In this paper we propose a method which is based on very general techniques developed in the field of logic programming. We encode the satisfaction relation of an LTL formula φ with respect to a Kripke structure \mathcal{K} by means of

a stratified logic program $P_{\mathcal{K},\varphi}$. Program $P_{\mathcal{K},\varphi}$ belongs to a class of programs, called ω -programs, which define predicates on infinite lists. Such predicates are needed because the definition of the satisfaction relation of φ is based on the computation paths of \mathcal{K} , which are infinite lists of states. The semantics of $P_{\mathcal{K},\varphi}$ is the *perfect model* $M(P_{\mathcal{K},\varphi})$ [15], which is defined in terms of a non-Herbrand interpretation for infinite lists.

Our verification method consists of two steps. In the first step we transform the program $P_{\mathcal{K},\varphi}$ into a *linear monadic ω -program*, that is, a stratified, linear recursive program which defines nullary or unary predicates on infinite lists. This transformation is performed by applying unfold/fold transformation rules [8,20] according to a strategy which is a variant of the strategy for the elimination of multiple occurrences of variables described in [13]. The use of those unfold/fold rules according to the given strategy guarantees the preservation of the perfect model of $P_{\mathcal{K},\varphi}$ [8,17,18].

In the second step of our verification method we use suitable proof rules for linear monadic ω -programs. Those proof rules are sound and complete, that is, any given quantified literal L is true in the perfect model of a linear monadic ω -program P iff we can prove that $M(P) \models L$ holds by using those proof rules. Moreover, those rules can be encoded in a straightforward way as a logic program which always terminates if we evaluate it by using tabled resolution [3,19]. As a consequence of this termination property, we get the decidability of the problem of verifying whether or not a quantified literal is true in the perfect model of a linear monadic ω -program.

We will prove that our verification method based on very general transformation techniques, has essentially the same time complexity of the algorithms based on *ad-hoc* techniques presented in [2,4,9,21].

The paper is structured as follows. In Section 2 we introduce the class of ω -programs and we present the encoding as an ω -program of the satisfaction relation for any given Kripke structure and LTL formula. In Section 3 we present our verification method. In particular, in Sections 3.1 and 3.2 we present the transformation rules and the strategy which allow us to transform an ω -program $P_{\mathcal{K},\varphi}$ into a linear monadic ω -program, in Section 3.3 we present the proof rules for linear monadic ω -programs and the encoding of those proof rules as logic programs, and in Section 3.4 we discuss the computational complexity of the verification technique. Finally, in Section 4 we discuss related work in the area of model checking and logic programming.

2 Encoding LTL Model Checking as a Logic Program

In this section we describe a method which, given a Kripke structure \mathcal{K} and an LTL formula φ , allows us to construct a logic program $P_{\mathcal{K},\varphi}$ that defines a predicate *prop* such that φ is true in \mathcal{K} , written $\mathcal{K} \models \varphi$, iff *prop* is true in the perfect model of $P_{\mathcal{K},\varphi}$, written $M(P_{\mathcal{K},\varphi}) \models \text{prop}$. Thus, the problem of checking whether or not $\mathcal{K} \models \varphi$, also called the problem of model checking φ with respect to \mathcal{K} , is reduced to the problem of checking whether or not $M(P_{\mathcal{K},\varphi}) \models \text{prop}$.

For a detailed definition of the logic LTL we refer to [5] (see also Appendix A). Throughout the paper we will consider a Kripke structure \mathcal{K} defined as a 4-tuple $\langle \Sigma, I, \rho, \lambda \rangle$, where: (i) $\Sigma = \{s_1, \dots, s_n\}$ is a finite set of *states*, (ii) $I \subseteq \Sigma$ is a set of *initial states*, (iii) $\rho \subseteq \Sigma \times \Sigma$ is a total *transition relation*, and (iv) $\lambda: \Sigma \rightarrow \mathcal{P}(Elem)$ is a total function that assigns to a state $s \in \Sigma$ a subset $\lambda(s)$ of the set *Elem* of *elementary properties*. A *computation path* of \mathcal{K} is an *infinite list* $[a_0 a_1 \dots]$ of states such that $a_0 \in I$ and, for every $i \geq 0$, $(a_i, a_{i+1}) \in \rho$. LTL formulas are constructed by using the elementary properties in *Elem*, the logical connectives \neg and \wedge , and the temporal operators X (*next time*) and U (*until*).

Since the definition of the satisfaction relation $\mathcal{K} \models \varphi$ refers to the computation paths of the Kripke structure \mathcal{K} and these paths are infinite lists, in order to encode that relation as a predicate defined by the program $P_{\mathcal{K}, \varphi}$ we need an extended form of logic programs where predicates have arguments that denote infinite lists. To this purpose in the following section we introduce the class of ω -*programs*.

2.1 Syntax and Semantics of ω -Programs

Let us consider a first order language \mathcal{L}_ω given by a set *Var* of variables, a set *Fun* of function symbols, and a set *Pred* of predicate symbols. We assume that *Fun* includes: (i) the set Σ of states of the Kripke structure, each state being a constant of \mathcal{L}_ω , (ii) the set *Elem* of the elementary properties of the Kripke structure, each elementary property being a constant of \mathcal{L}_ω , and (iii) the binary function symbol $[-, -]$, denoting the constructor of infinite lists. Thus, $[H|T]$ is an infinite list whose head is H and whose tail is the infinite list T .

We assume that \mathcal{L}_ω is a typed language [10] with three basic types: (i) **fterm**, which is the type of finite terms, (ii) **state**, which is the type of states, and (iii) **ilist**, which is the type of infinite lists of states. Every function symbol in *Fun* — $(\Sigma \cup \{[-, -]\})$, with arity $n (\geq 0)$, has type **fterm** $\times \dots \times$ **fterm** \rightarrow **fterm**, where **fterm** occurs n times to the left of \rightarrow . Every function symbol in Σ has type **state**. The function symbol $[-, -]$ has type **state** \times **ilist** \rightarrow **ilist**. A predicate symbol of arity $n (\geq 0)$ in *Pred* has type of the form $\tau_1 \times \dots \times \tau_n$, where $\tau_1, \dots, \tau_n \in \{\mathbf{fterm}, \mathbf{state}, \mathbf{ilist}\}$. ω -programs are logic programs constructed as usual from symbols in the typed language \mathcal{L}_ω [10]. In what follows, for reasons of simplicity, we will feel free to say ‘programs’, instead of ‘ ω -programs’.

The *definition* of a predicate p in a program P is the set of all clauses of P whose head predicate is p . If f is a term or a formula, then by $vars(f)$ we denote the set of variables occurring in f . By $\forall(\varphi)$ and $\exists(\varphi)$ we denote, respectively, the *universal closure* and the *existential closure* of the formula φ .

An interpretation for our typed language \mathcal{L}_ω , called ω -interpretation, is given as follows. Let HU be the Herbrand universe constructed from the set *Fun* — $(\Sigma \cup \{[-, -]\})$ of function symbols and let Σ^ω be the set of infinite lists of states. An ω -interpretation I is an interpretation such that: (i) to the types **fterm**, **state**, and **ilist**, I assigns the sets HU , Σ , and Σ^ω , respectively, (ii) to the function symbol $[-, -]$, I assigns the function $[-, -]_I$ such that, for any state $s \in \Sigma$ and infinite list $[s_1, s_2, \dots] \in \Sigma^\omega$, $[s|s_1, s_2, \dots]_I$ is the infinite list $[s, s_1, s_2, \dots]$,

(iii) I is an Herbrand interpretation for all function symbols in $Fun - (\Sigma \cup \{[-, -]\})$, and (iv) to every n -ary predicate $p \in Pred$ of type $\tau_1 \times \dots \times \tau_n$, I assigns a relation on $D_1 \times \dots \times D_n$, where, for $i = 1, \dots, n$, D_i is either HU or Σ or Σ^ω , if τ_i is either **fterm** or **state** or **ilist**, respectively. We say that an ω -interpretation I is an ω -model of a program P iff for every clause $\gamma \in P$ we have that $I \models \forall(\gamma)$. Similarly to the case of logic programs, we can define the (locally) stratified ω -programs and we have that every (locally) stratified ω -program P has a unique perfect ω -model (or perfect model, for short) denoted by $M(P)$ [1,15].

Definition 1 (Linear Monadic ω -Programs). A linear monadic ω -clause is a clause of one of the following forms:

$$\begin{array}{lll} p_0 \leftarrow & p_0 \leftarrow q_0 & p_0 \leftarrow \neg q_0 \\ p_0 \leftarrow q_1(L) & p_0 \leftarrow \neg q_1(L) & \\ p_1([s|L]) \leftarrow & p_1([s|L]) \leftarrow q_1(L) & p_1([s|L]) \leftarrow \neg q_1(L) \end{array}$$

where: (i) s is a constant of type **state** and (ii) L a variable of type **ilist**. A linear monadic ω -program is a stratified, finite set of linear monadic ω -clauses.

Example 1. The following set of clauses is an example of ω -program:

$$\begin{array}{lll} p([a|L]) \leftarrow p(L) & p([a|L]) \leftarrow \neg q(L) & q([a|L]) \leftarrow q(L) \\ p([b|L]) \leftarrow p(L) & p([b|L]) \leftarrow \neg q(L) & q([b|L]) \leftarrow \end{array}$$

Every infinite list in $\{a, b\}^\omega$ that satisfies predicate p is a list in $(a + b)^+ a^\omega$. Indeed, $q(L)$ holds for every infinite list L which has an occurrence of b .

2.2 Encoding the LTL Satisfaction Relation as an ω -Program

Given a Kripke structure \mathcal{K} and an LTL formula φ , we introduce a locally stratified ω -program $P_{\mathcal{K}, \varphi}$ which defines, among others, the following three predicates: (i) the unary predicate $path$, such that $path(\pi)$ holds iff π is an infinite list representing a computation path of \mathcal{K} , (ii) the binary predicate sat , which encodes the satisfaction relation for LTL formulas in the sense that for all paths π and LTL formulas ψ , we have that $\mathcal{K}, \pi \models \psi$ iff $M(P_{\mathcal{K}, \varphi}) \models sat(\pi, \psi)$, and (iii) the nullary predicate $prop$, which holds iff φ holds on all computation paths of \mathcal{K} , that is, $M(P_{\mathcal{K}, \varphi}) \models prop$ iff $\mathcal{K} \models \varphi$.

In the terms that encode LTL formulas, such as the second argument of the predicate sat , we will use the function symbols x and u standing for the operator symbols X and U , respectively.

Definition 2 (Encoding Program). Given a Kripke structure $\mathcal{K} = \langle \Sigma, I, \rho, \lambda \rangle$ and an LTL formula φ , the encoding program $P_{\mathcal{K}, \varphi}$ is the following ω -program:

$$\begin{array}{l} 1. \quad prop \leftarrow \neg p_1 \\ 2. \quad p_1 \leftarrow p_2(P) \\ 3. \quad p_2(P) \leftarrow path(P) \wedge sat(P, \neg \varphi) \\ 4. \quad path([X|P]) \leftarrow initial(X) \wedge \neg nopath([X|P]) \\ 5.1 \quad nopath([s_1|P]) \leftarrow q_1(P) \quad q_1([s_{11}|P]) \leftarrow \dots \quad q_1([s_{1 k_1}|P]) \leftarrow \\ \quad \vdots \\ 5.m \quad nopath([s_m|P]) \leftarrow q_m(P) \quad q_m([s_{m1}|P]) \leftarrow \dots \quad q_m([s_{m k_m}|P]) \leftarrow \end{array}$$

6. $nopath([X|P]) \leftarrow nopath(P)$
7. $sat([X|P], E) \leftarrow elem(E, X)$
8. $sat(P, \neg F) \leftarrow \neg sat(P, F)$
9. $sat(P, F_1 \wedge F_2) \leftarrow sat(P, F_1) \wedge sat(P, F_2)$
10. $sat([X|P], x(F)) \leftarrow sat(P, F)$
11. $sat(P, u(F_1, F_2)) \leftarrow sat(P, F_2)$
12. $sat([X|P], u(F_1, F_2)) \leftarrow sat([X|P], F_1) \wedge sat(P, u(F_1, F_2))$

together with the clauses defining the predicates *initial* and *elem*, where:

- (1) *initial*(s) holds iff $s \in I$, for every state $s \in \Sigma$;
- (2) *elem*(e, s) holds iff $e \in \lambda(s)$, for every property $e \in Elem$ and state $s \in \Sigma$;
- (3) *nopath*(P) holds if P is an infinite list $[a_0, a_1, \dots]$ of states which is *not* a computation path in \mathcal{K} , that is, for some $i \geq 0$, we have that $(a_i, a_{i+1}) \notin \rho$; and
- (4) clauses 5.1–5. m are obtained as follows. Let $\{s_1, \dots, s_m\}$ be the subset of Σ such that, for $i = 1, \dots, m$, there exists $s \in \Sigma$ for which $(s_i, s) \notin \rho$. For $i = 1, \dots, m$, the clauses 5. i are:

$$5.i \text{ } nopath([s_i|P]) \leftarrow q_i(P) \text{ and for all } s \in \Sigma \text{ such that } (s_i, s) \notin \rho, q_i([s|P]) \leftarrow .$$

Clauses 1, 2, and 3 of the above Definition 2 stipulate that *prop* holds iff the formula $\forall P (path(P) \rightarrow sat(P, \varphi))$ holds. Since the universal quantifier and the implication connective cannot appear in the body of a clause, we have defined the predicate *prop* starting from the equivalent formula $\neg \exists P (path(P) \wedge sat(P, \neg \varphi))$. Indeed, (i) $p_2(P)$ holds iff $path(P) \wedge sat(P, \neg \varphi)$, (ii) p_1 holds iff $\exists P p_2(P)$ holds, and (iii) *prop* holds iff $\neg p_1$ holds.

Clauses 4–6 stipulate that *path*(P) holds iff for every pair (a_i, a_{i+1}) of consecutive elements on the infinite list P we have that $(a_i, a_{i+1}) \in \rho$. Similarly to the encoding of the predicate *prop* above, we have defined the predicate *path* by using existential quantification and negation, instead of universal quantification and implication. Indeed, clauses 5.1–5. m and 6 stipulate that *nopath*(P) holds iff there exist two consecutive elements a_i and a_{i+1} of the list P such that $(a_i, a_{i+1}) \notin \rho$. Clause 6 is required for allowing the two consecutive elements a_i and a_{i+1} to occur at any position in P .

Clauses 7–12 define the satisfaction relation $sat(P, \varphi)$ by cases, according to the structure of the formula φ .

The program $P_{\mathcal{K}, \varphi}$ is locally stratified w.r.t. the stratification function σ from ground literals to natural numbers defined as follows (where, for an LTL formula ψ , we denote by $|\psi|$ the number of occurrences of function symbols in ψ): σ always returns 0, except that, for all infinite lists $\pi \in \Sigma^\omega$ and LTL formulas ψ , (i) $\sigma(prop) = |\varphi| + 2$, (ii) $\sigma(p_1) = \sigma(p_2(\pi)) = |\varphi| + 1$, (iii) $\sigma(path(\pi)) = 1$, (iv) $\sigma(sat(\pi, \psi)) = |\psi| + 1$, and (v) for every ground atom A , $\sigma(\neg A) = \sigma(A) + 1$.

Example 2. Let us consider the set $Elem = \{true, a, b\}$ of elementary properties and the Kripke structure $\mathcal{K} = \langle \Sigma, I, \rho, \lambda \rangle$, where: (i) Σ is $\{s_1, s_2\}$, (ii) $I = \Sigma$, (iii) ρ is the transition relation $\{(s_1, s_2), (s_2, s_1), (s_2, s_2)\}$, and (iv) λ is the function such that $\lambda(s_1) = \{a\}$ and $\lambda(s_2) = \{b\}$. Let us also consider the formula $\varphi = \neg (true \cup (a \wedge \neg (true \cup b)))$ (that is, $\varphi = G(a \rightarrow Fb)$). The encoding program $P_{\mathcal{K}, \varphi}$, where we wrote $u(true, a \wedge \neg u(true, b))$, instead of $\neg \varphi$, is as follows:

1. $prop \leftarrow \neg p_1$
2. $p_1 \leftarrow p_2(P)$
3. $p_2(P) \leftarrow path(P) \wedge sat(P, u(true, (a \wedge \neg u(true, b))))$
4. $path([X|P]) \leftarrow initial(X) \wedge \neg nopath([X|P])$
5. $nopath([s_1|P]) \leftarrow q_1(P) \qquad q_1([s_1|P]) \leftarrow$
6. $nopath([X|P]) \leftarrow nopath(P)$

together with clauses 7–12 of Definition 2 defining the predicate sat , and the following clauses defining the predicates $initial$ and $elem$:

$$initial(s_1) \leftarrow initial(s_2) \leftarrow elem(a, s_1) \leftarrow elem(b, s_2) \leftarrow elem(true, X) \leftarrow$$

Theorem 1 (Correctness of the Encoding Program). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure \mathcal{K} and an LTL formula φ . Then, $\mathcal{K} \models \varphi$ iff $M(P_{\mathcal{K},\varphi}) \models prop$.*

3 Transformational LTL Model Checking

In this section we present a technique based on program transformation for checking whether or not, for any given structure \mathcal{K} and LTL formula φ , $M(P_{\mathcal{K},\varphi}) \models prop$ holds, where $P_{\mathcal{K},\varphi}$ is constructed as indicated in Definition 2 above. Our technique consists of two steps. In the first step we transform the ω -program $P_{\mathcal{K},\varphi}$ into a *linear monadic* ω -program T such that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$. In the second step we check whether or not $M(T) \models prop$ holds by using a proof system for linear monadic ω -programs.

3.1 Unfold/Fold Transformation Rules

Now we introduce the transformation rules that will be used for transforming ω -programs into linear monadic ω -programs. These rules are specialized versions of the familiar unfold/fold rules (see, for instance, [8,20]).

A *transformation sequence* is a sequence P_0, \dots, P_n of ω -programs, where for $0 \leq k \leq n-1$, program P_{k+1} is derived from program P_k by the application of a transformation rule as indicated below. In what follows we assume that $\Sigma = \{s_1, \dots, s_n\}$ is the set of states of the given Kripke structure \mathcal{K} .

R1. Definition Introduction. Let us consider the following m (≥ 1) clauses:

$$\delta_1: newp(X_1, \dots, X_r) \leftarrow B_1, \quad \dots, \quad \delta_m: newp(X_1, \dots, X_r) \leftarrow B_m,$$

such that: (i) $newp$ is a predicate symbol that does not occur in $\{P_0, \dots, P_k\}$, (ii) X_1, \dots, X_r are distinct variables, (iii) for $i = 1, \dots, m$, $vars(B_i) = \{X_1, \dots, X_r\}$, and (iv) every predicate symbol occurring in $\{B_1, \dots, B_m\}$ also occurs in P_0 . By *definition introduction* (or *definition*, for short) from program P_k we derive the program $P_{k+1} = P_k \cup \{\delta_1, \dots, \delta_m\}$.

For $0 \leq k \leq n$, we denote by $Defs_k$ the set of clauses introduced by using rule R1 during the transformation sequence P_0, \dots, P_n . In particular, $Defs_0 = \emptyset$.

R2. Definition Elimination. By *definition elimination* w.r.t. a predicate symbol p , from program P_k we derive the new program $P_{k+1} = \{\gamma \in P_k \mid \text{the head}$

predicate of γ is either p or a predicate on which p depends} (see [1] for the definition of the dependency relation).

R3. Instantiation. Let $\gamma: H \leftarrow B$ be a clause in program P_k , L be a variable of type `ilist` in γ , and M be a variable of type `ilist` not occurring in γ . By *instantiation* of L in clause γ we derive the clauses:

$$\gamma_1: (H \leftarrow B)\{L/[s_1|M]\}, \quad \dots, \quad \gamma_h: (H \leftarrow B)\{L/[s_h|M]\}$$

(recall that $\{s_1, \dots, s_h\}$ is the set Σ of states of the given Kripke structure \mathcal{K}) and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\gamma_1, \dots, \gamma_h\}$.

R4. Positive Unfolding. Let $\gamma: H \leftarrow G_L \wedge A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ . Let

$$\gamma_1: K_1 \leftarrow B_1, \quad \dots, \quad \gamma_m: K_m \leftarrow B_m, \text{ for } m \geq 0,$$

be all clauses of program P'_k such that A is unifiable with K_1, \dots, K_m with most general unifiers $\vartheta_1, \dots, \vartheta_m$, respectively. By *unfolding clause γ w.r.t. the positive literal A* we derive the clauses

$$\eta_1: (H \leftarrow G_L \wedge B_1 \wedge G_R)\vartheta_1, \quad \dots, \quad \eta_m: (H \leftarrow G_L \wedge B_m \wedge G_R)\vartheta_m$$

and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \dots, \eta_m\}$.

R5. Negative Unfolding. Let $\gamma: H \leftarrow G_L \wedge \neg A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ . Let

$$\gamma_1: K_1 \leftarrow B_1, \quad \dots, \quad \gamma_m: K_m \leftarrow B_m, \text{ for } m \geq 0,$$

be all clauses of program P'_k such that A is unifiable with K_1, \dots, K_m with most general unifiers $\vartheta_1, \dots, \vartheta_m$, respectively. Assume that: (i) for $j = 1, \dots, m$, $A = K_j\vartheta_j$, that is, A is an instance of K_j , (ii) for $j = 1, \dots, m$, $\text{vars}(K_j) = \text{vars}(B_j)$, and (iii) from $G_L \wedge \neg(B_1\vartheta_1 \vee \dots \vee B_m\vartheta_m) \wedge G_R$ we get an equivalent disjunction $Q_1 \vee \dots \vee Q_r$ of conjunctions of literals, with $r \geq 0$, by first pushing \neg inside and then pushing \vee outside. By *unfolding clause γ w.r.t. the negative literal $\neg A$* we derive the clauses

$$\eta_1: H \leftarrow Q_1, \quad \dots, \quad \eta_r: H \leftarrow Q_r,$$

and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \dots, \eta_r\}$.

R6. Positive Folding. Let γ be a clause in P_k and let Defs'_k be a variant of Defs_k without variables in common with γ . Suppose that there exists a predicate in Defs'_k whose definition consists of the clause

$$\delta: K \leftarrow B, \text{ where } \text{vars}(K) = \text{vars}(B).$$

Suppose that there exists a substitution ϑ such that γ is of the form $H \leftarrow B\vartheta$. By *folding clause γ using clause δ* we derive the clause $\eta: H \leftarrow K\vartheta$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

R7. Negative Folding. Let γ be a clause in P_k and let Defs'_k be a variant of Defs_k without variables in common with γ . Suppose that there exists a predicate in Defs'_k whose definition consists of the clauses

$$\delta_1: K \leftarrow A_1, \quad \dots, \quad \delta_m: K \leftarrow A_m$$

where, for $j = 1, \dots, m$, A_j is an atom and $\text{vars}(K) = \text{vars}(A_j)$. Suppose that there exists a substitution ϑ such that γ is of the form $H \leftarrow \neg A_1\vartheta \wedge \dots \wedge \neg A_m\vartheta$. By *folding clause γ using clauses $\delta_1, \dots, \delta_m$* we derive the clause $\eta: H \leftarrow \neg K\vartheta$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

3.2 The Transformation Strategy

Now we present a transformation strategy which terminates for every input program $P_{\mathcal{K},\varphi}$ and produces a linear monadic ω -program T such that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$.

Our strategy makes use of the transformation rules presented in Section 3.1 and it is a variant of the transformation strategy for eliminating the so-called *unnecessary variables* presented in [13]. The strategy starts off from the clause $p_2(P) \leftarrow path(P) \wedge sat(P, \neg\varphi)$ (see Definition 2) and iterates a sequence of applications of the three procedures: *instantiate*, *unfold*, and *define-fold*. At each iteration, the set $InDefs$ of input clauses is transformed into a set Es of linear monadic ω -clauses, by possibly introducing some auxiliary (not linear monadic) clauses $NewDefs$. These auxiliary clauses are given in input to a subsequent iteration of the strategy until no more auxiliary clauses are introduced. Thus, our strategy terminates when all clauses are transformed into linear monadic ω -clauses without the need for new auxiliary clauses. As a final step of our strategy, we apply the definition elimination rule and we keep only the clauses for *prop* and for the predicates on which *prop* depends.

The Transformation Strategy.

Input: An ω -program $P_{\mathcal{K},\varphi}$, for a Kripke structure \mathcal{K} and an LTL formula φ .

Output: A linear monadic ω -program T such that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$.

```

 $T := P_{\mathcal{K},\varphi}; \quad Defs := \{p_2(P) \leftarrow path(P) \wedge sat(P, \neg\varphi)\}; \quad InDefs := Defs;$ 
while  $InDefs \neq \emptyset$  do
   $instantiate(InDefs, Cs); \quad T := (T - InDefs) \cup Cs;$ 
   $unfold(Cs, Ds); \quad T := (T - Cs) \cup Ds;$ 
   $define-fold(Ds, Defs, NewDefs, Es); \quad T := (T - Ds) \cup NewDefs \cup Es;$ 
   $Defs := Defs \cup NewDefs; \quad InDefs := NewDefs$ 
od;
 $T := \{\gamma \in T \mid \text{the head predicate of } \gamma \text{ is either } prop \text{ or a predicate on which } prop \text{ depends}\}.$ 

```

Now we describe the three procedures *instantiate*, *unfold*, and *define-fold* that are used in the transformation strategy. When describing these procedures we will rely on the fact that during the application of the transformation strategy we derive clauses with occurrences of a single variable of type **ilist**.

The *instantiate* procedure consists in applying rule R3 to each clause B in the set $InDefs$.

The *instantiate* Procedure.

Input: A set $InDefs$ of clauses. *Output:* A set Cs of clauses.

$Cs := \bigcup_{B \in InDefs} \{C \mid C \text{ is derived by instantiation of a variable } L \text{ of type } \mathbf{ilist} \text{ occurring in } B\}.$

The *unfold* procedure applies rules R4 and R5 to a set of clauses of the form $H \leftarrow G_1 \wedge L \wedge G_2$. The procedure applies positive unfolding if L is a positive literal, and negative unfolding if L is a negative literal.

The *unfold* Procedure.

Input: A set Cs of clauses in program T . *Output:* A set Ds of clauses.

$Ds := Cs$;

while there exists a clause D in Ds of the form $H \leftarrow G_1 \wedge L \wedge G_2$, where L is either an atom A or a negated atom $\neg A$ and for all clauses $K \leftarrow B$ in T either A and K are not unifiable or A is an instance of K **do**

$Ds := (Ds - \{D\}) \cup \{U \mid U \text{ is a clause derived by unfolding } D \text{ w.r.t. } L\}$

od

The *define-unfold* procedure applies the positive and negative folding rules R6 and R7. The procedure transforms every clause in Ds , which has been obtained by unfolding, into a linear monadic ω -clause. Each clause D in Ds is of the form $p([s|X]) \leftarrow L_1 \wedge \dots \wedge L_m$. If among L_1, \dots, L_m there is at least one positive literal, then the procedure introduces a new clause N of the form $newp(X) \leftarrow L_1 \wedge \dots \wedge L_m$, unless (a variant of) such a clause N was added to $Defs$ in a previous transformation step. Then D is folded using N (by applying rule R6), thereby deriving a linear monadic ω -clause of the form $p([s|X]) \leftarrow newp(X)$. The new clause N is added to the set $Defs$ of clauses that can be used for subsequent folding steps and to the set $InDefs$ of clauses to be processed in a subsequent iteration of the strategy.

If L_1, \dots, L_m are all negative literals of the form $\neg A_1, \dots, \neg A_m$, respectively, then the procedure introduces m new clauses $N_1: newp(X) \leftarrow A_1, \dots, N_m: newp(X) \leftarrow A_m$, unless (variants of) such clauses were added to $Defs$ in a previous transformation step. Then D is folded using N_1, \dots, N_m (by applying rule R7), thereby deriving a linear monadic ω -clause of the form $p([s|X]) \leftarrow \neg newp(X)$. The new clauses N_1, \dots, N_m are added to the sets $Defs$ and $InDefs$.

The *define-fold* Procedure.

Input: (i) A set Ds of clauses of the form $p([s|X]) \leftarrow G_1$, where $vars(G_1) \subseteq \{X\}$, and (ii) a set $Defs$ of clauses;

Output: (i) A set $NewDefs$ of clauses of the form $newp(X) \leftarrow G_3$, where $\{X\} = vars(G_3)$, and (ii) a set Es of linear monadic ω -clauses.

$NewDefs := \emptyset$; $Es := \emptyset$;

for each clause $D \in Ds$ of the form $p([s|X]) \leftarrow L_1 \wedge \dots \wedge L_m$ **do**

if D is a linear monadic ω -clause **then** $Es := Es \cup \{D\}$ **else**

(*Case 1. Positive Define-Fold*)

if for some $i \in \{1, \dots, m\}$, L_i is a positive literal

then **if** there exists no clause N of the form $newp(X) \leftarrow L_1 \wedge \dots \wedge L_m$ such that: (i) (a variant of) N belongs to $Defs \cup NewDefs$ and (ii) $newp$ does not occur in $(Defs \cup NewDefs) - \{N\}$

then $NewDefs := NewDefs \cup \{newp(X) \leftarrow L_1 \wedge \dots \wedge L_m\}$, where

$newp$ is a predicate not occurring in $Defs \cup NewDefs$ **fi**;
 $Es := Es - \{D\} \cup \{p([s|X]) \leftarrow newp(X)\}$ **fi**;

(Case 2. *Negative Define-Fold*)

if for all $i \in \{1, \dots, m\}$, L_i is a negative literal $\neg A_i$

then **if** there exists no set $Ns: \{newp(X) \leftarrow A_1, \dots, newp(X) \leftarrow A_m\}$ of clauses such that: (i) $Ns \subseteq Defs \cup NewDefs$ (modulo variants) and (ii) $newp$ does not occur in $(Defs \cup NewDefs) - Ns$

then $NewDefs := NewDefs \cup \{newp(X) \leftarrow A_1, \dots, newp(X) \leftarrow A_m\}$, where $newp$ is a predicate not occurring in $Defs \cup NewDefs$ **fi**;

$Es := Es - \{D\} \cup \{p([s|X]) \leftarrow \neg newp(X)\}$ **fi**

od

The correctness of our transformation strategy can be proved by showing that when the transformation rules R1–R7 are applied according to the strategy, the perfect model of $P_{\mathcal{K},\varphi}$ is preserved (see [8,17,18] for analogous proofs).

The termination of the transformation strategy follows from the termination of the procedures *instantiate*, *unfold*, and *define-fold*, and from the fact that the while loop can be executed only a finite number of times. Indeed, (i) the strategy terminates when no new clauses are introduced by the *define-fold* procedure and, thus, $InDefs = \emptyset$, (ii) only a finite number of new clauses can be introduced by the *define-fold* procedure, because every new clause is of the form $newp(X) \leftarrow L_1 \wedge \dots \wedge L_m$, where for $i = 1, \dots, m$, L_i is either an atom A_i or a negated atom $\neg A_i$ and A_i belongs to the set $\{path(X), initial(X), nopath(X), q_1(X), \dots, q_m(X)\} \cup \{sat(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$.

Theorem 2 (Correctness and Termination of the Transformation Strategy). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure \mathcal{K} and an LTL formula φ . The transformation strategy terminates for the input program $P_{\mathcal{K},\varphi}$ and returns an output program T such that: (i) T is a linear monadic ω -program and (ii) $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$.*

Example 3. Let us consider program $P_{\mathcal{K},\varphi}$ of Example 2. Our transformation strategy starts off from the sets of clauses $T = P_{\mathcal{K},\varphi}$ and $Defs = InDefs = \{3\}$, where:

$$3. p_2(P) \leftarrow path(P) \wedge sat(P, u(true, (a \wedge \neg u(true, b))))$$

The first execution of the loop body of our strategy applies the *instantiate* procedure to the set $InDefs$. We get the set of clauses $Cs = \{3.1, 3.2\}$, where:

$$3.1. p_2([s_1|P]) \leftarrow path([s_1|P]) \wedge sat([s_1|P], u(true, (a \wedge \neg u(true, b))))$$

$$3.2. p_2([s_2|P]) \leftarrow path([s_2|P]) \wedge sat([s_2|P], u(true, (a \wedge \neg u(true, b))))$$

Then, by applying the *unfold* procedure to the set Cs we get the set $Ds = \{3.3, 3.4, 3.5\}$, where:

$$3.3. p_2([s_1|P]) \leftarrow \neg q_1(P) \wedge \neg nopath(P) \wedge \neg sat(P, u(true, b))$$

$$3.4. p_2([s_1|P]) \leftarrow \neg q_1(P) \wedge \neg nopath(P) \wedge sat(P, u(true, (a \wedge \neg u(true, b))))$$

$$3.5. p_2([s_2|P]) \leftarrow \neg nopath(P) \wedge sat(P, u(true, (a \wedge \neg u(true, b))))$$

Finally, by applying the *define-fold* procedure, we get the sets $NewDefs = \{13, 14, 15, 16, 17\}$ and $Es = \{3.6, 3.7, 3.8\}$, where:

- 13. $p_3(P) \leftarrow q_1(P)$
- 14. $p_3(P) \leftarrow nopath(P)$
- 15. $p_3(P) \leftarrow sat(P, u(true, b))$
- 16. $p_4(P) \leftarrow \neg q_1(P) \wedge \neg nopath(P) \wedge sat(P, u(true, (a \wedge \neg u(true, b))))$
- 17. $p_5(P) \leftarrow \neg nopath(P) \wedge sat(P, u(true, (a \wedge \neg u(true, b))))$
- 3.6. $p_2([s_1|P]) \leftarrow \neg p_3(P)$
- 3.7. $p_2([s_1|P]) \leftarrow p_4(P)$
- 3.8. $p_2([s_2|P]) \leftarrow p_5(P)$

Thus, at the end of the first iteration of our strategy we get:

$$\begin{aligned} T &= (P_{\mathcal{K}, \varphi} - \{3\}) \cup \{13, 14, 15, 16, 17\} \cup \{3.6, 3.7, 3.8\} \\ Defs &= \{3\} \cup \{13, 14, 15, 16, 17\} \\ InDefs &= \{13, 14, 15, 16, 17\} \end{aligned}$$

Since $InDefs \neq \emptyset$ we perform a second execution of the loop body of our strategy. After a few more executions, and a final application of the definition elimination rule, we get the following linear monadic ω -program T :

$$\begin{array}{lll} prop \leftarrow \neg p_1 & p_3([s_2|P]) \leftarrow p_8(P) & p_6([s_1|P]) \leftarrow p_6(P) \\ p_1 \leftarrow p_2(P) & p_3([s_1|P]) \leftarrow p_6(P) & p_6([s_2|P]) \leftarrow \\ p_2([s_1|P]) \leftarrow \neg p_3(P) & p_3([s_2|P]) \leftarrow & p_6([s_2|P]) \leftarrow p_6(P) \\ p_2([s_1|P]) \leftarrow p_4(P) & p_3([s_2|P]) \leftarrow p_6(P) & p_7([s_1|P]) \leftarrow \\ p_2([s_2|P]) \leftarrow p_5(P) & p_4([s_2|P]) \leftarrow p_5(P) & p_8([s_1|P]) \leftarrow p_7(P) \\ p_3([s_1|P]) \leftarrow & p_5([s_1|P]) \leftarrow \neg p_3(P) & p_8([s_1|P]) \leftarrow p_8(P) \\ p_3([s_1|P]) \leftarrow p_7(P) & p_5([s_1|P]) \leftarrow p_4(P) & p_8([s_2|P]) \leftarrow p_8(P) \\ p_3([s_1|P]) \leftarrow p_8(P) & p_5([s_2|P]) \leftarrow p_5(P) & \end{array}$$

3.3 A Proof System for Linear Monadic ω -Programs.

Now we present a proof system for checking whether a quantified literal is true or not in the perfect model of a linear monadic ω -program.

A *closed literal* is a statement of one of the following forms: p , $\neg p$, $\forall(L)$, $\exists(L)$, where p is a nullary predicate and L is either an atom $q(X)$ or a negated atom $\neg q(X)$, and $\forall(L)$ and $\exists(L)$ denote the universal and existential closure of L , respectively. The proof rules in Figure 1 define a provability relation \vdash , such that for every linear monadic ω -program P and closed literal L , $P \vdash L$ iff $M(P) \models L$ (see Theorem 3 below). When applying these proof rules we assume that: (i) the set Σ of states is $\{s_1, \dots, s_h\}$, (ii) each clause in P of the form $H \leftarrow$ is written as $H \leftarrow true$ (so that no clause in P has an empty body) and (iii) the formulas $\exists(true)$ and $\forall(true)$, which may appear in the premise of rules S5 and S6, are identified with $true$.

Note that the proof rules S7, S8, and S9 have negative premises. The negated judgement ' $P \not\vdash L$ ' should be interpreted as ' $P \vdash L$ cannot be proved by using the proof rules S1-S9' and in this case we say that ' $P \vdash L$ has a disproof'.

$$\begin{array}{l}
\text{S1. } \frac{}{P \vdash \text{true}} \qquad \qquad \qquad \text{S2. } \frac{}{P \vdash p} \text{ if } p \leftarrow \text{true} \in P \\
\text{S3. } \frac{P \vdash L}{P \vdash p} \text{ if } p \leftarrow L \in P \text{ and } \text{vars}(L) = \emptyset \quad \text{S4. } \frac{P \vdash \exists(L)}{P \vdash p} \text{ if } p \leftarrow L \in P \text{ and } \text{vars}(L) \neq \emptyset \\
\text{S5. } \frac{P \vdash \exists(L)}{P \vdash \exists(p(Z))} \text{ if } p([s|Y]) \leftarrow L \in P \text{ for some } s \in \{s_1, \dots, s_h\} \\
\text{S6. } \frac{P \vdash \forall(L_1) \dots P \vdash \forall(L_h)}{P \vdash \forall(p(Z))} \text{ if } \{p([s_1|Y]) \leftarrow L_1, \dots, p([s_h|Y]) \leftarrow L_h\} \subseteq P \\
\text{S7. } \frac{P \not\vdash p}{P \vdash \neg p} \qquad \qquad \text{S8. } \frac{P \not\vdash \forall(p(Z))}{P \vdash \exists(\neg p(Z))} \qquad \qquad \text{S9. } \frac{P \not\vdash \exists(p(Z))}{P \vdash \forall(\neg p(Z))}
\end{array}$$

Fig. 1. Proof system for linear monadic ω -programs. $\Sigma = \{s_1, \dots, s_h\}$ is the set of states of the Kripke structure \mathcal{K} .

The interpretation of $P \not\vdash L$ as (finite or infinite) failure of $P \vdash L$ is meaningful because the program P is stratified and, thus, also the instances of the proof rules can be stratified. The stratification of these instances is induced by a well-founded ordering on closed literals such that, for every rule instance with conclusion $P \vdash L_1$, (i) if $P \vdash L_2$ occurs as a premise, then L_2 is not larger than L_1 , and (ii) if $P \not\vdash L_2$ occurs as a premise, then L_2 is strictly smaller than L_1 . Thus, in order to construct a proof for $P \vdash L$, we are never required to show that $P \not\vdash L$, that is, we are never required to show that no proof for $P \vdash L$ itself can be constructed.

The following theorem shows that the proof rules of Figure 1 are a sound and complete proof system for proving that a closed literal is true in the perfect model of a linear monadic ω -program.

Theorem 3. *For every linear monadic ω -program P and closed literal L , $P \vdash L$ iff $M(P) \models L$.*

The proof system for linear monadic ω -programs can be encoded as a logic program, which we call *Demo*. In *Demo* a closed literal L is represented by a ground term $[L]$ constructed as follows. Let v be a new constant symbol. (i) For any variable Z , $[Z]$ is v , (ii) for any list $[s|Z]$, where s is a state and Z is a variable, $[s|Z]$ is (s, v) , (iii) for any nullary predicate p , $[p]$ is p , (iv) for any unary predicate q and term t , $[q(t)]$ is $(q, [t])$, (v) for any atom A , $[\neg A]$ is $\text{not}([A])$, (vi) for any literal L , $[\exists(L)]$ is $e([L])$ and $[\forall(L)]$ is $a([L])$. An ω -program P is represented by the set $[P]$ of ground unit clauses of the form $\text{clause}([H], [B]) \leftarrow$ such that $H \leftarrow B$ is a clause of P .

In the clauses for *Demo*, which we list below, we also use the predicate $\text{nullary}(R)$ which holds iff R is a nullary predicate symbol, and $\text{emptyvars}(L)$ which holds iff L has no occurrences of the symbol v . The three clauses 1.1, 1.2, and 1.3 correspond to rule S1 (recall that in the proof system of Figure 1 we identify the three expressions true , $\exists(\text{true})$, and $\forall(\text{true})$).

- 1.1 $demo(true) \leftarrow$
- 1.2 $demo(e(true)) \leftarrow$
- 1.3 $demo(a(true)) \leftarrow$
2. $demo(R) \leftarrow nullary(R) \wedge clause(R, true)$
3. $demo(R) \leftarrow nullary(R) \wedge clause(R, L) \wedge emptyvars(L) \wedge demo(L)$
4. $demo(R) \leftarrow nullary(R) \wedge clause(R, L) \wedge \neg emptyvars(L) \wedge demo(e(L))$
5. $demo(e((R, v))) \leftarrow clause((R, (S, v)), L) \wedge demo(e(L))$
6. $demo(a((R, v))) \leftarrow clause((R, (s_1, v)), L_1) \wedge demo(a(L_1)) \wedge \dots \wedge$
 $clause((R, (s_h, v)), L_h) \wedge demo(a(L_h))$
7. $demo(not(R)) \leftarrow nullary(R) \wedge \neg demo(R)$
8. $demo(e(not((R, v)))) \leftarrow \neg demo(a((R, v)))$
9. $demo(a(not((R, v)))) \leftarrow \neg demo(e((R, v)))$

Since P is a stratified program, $Demo \cup [P]$ is a *weakly stratified* program [14] and, hence, it has a unique perfect model $M(Demo \cup [P])$.

Theorem 4. *For every linear monadic ω -program P and closed literal L , $P \vdash L$ iff $M(Demo \cup [P]) \models demo([L])$.*

Thus, by Theorems 3 and 4 for any linear monadic ω -program P , we can check whether or not $M(P) \models L$ holds by using any logic programming system which computes the perfect model of $Demo \cup [P]$. One can use, for instance, a system based on *tabled resolution* [3,19] which guarantees the termination of any query of the form $demo([L])$ and returns ‘yes’ iff $demo([L])$ belongs to $M(Demo \cup [P])$. Indeed, starting from $demo([L])$, we can only derive a finite set of queries of the form $demo([M])$, and the tabling mechanism ensures that each query is evaluated at most once.

Example 4. Let T be the linear monadic ω -program that is the output of our transformation strategy, as illustrated in Example 3. By using our proof system we obtain the proof in Figure 2 which shows that $T \vdash prop$. Thus, $M(P_{\mathcal{K}, \varphi}) \models prop$ and, therefore, $G(a \rightarrow Fa)$ holds in \mathcal{K} (see Example 2).

3.4 Complexity of the Verification Technique

The complexity of our verification technique will be measured in terms of: (i) the number of applications of transformation rules for generating the linear monadic ω -program T from $P_{\mathcal{K}, \varphi}$ (Step 1), and (ii) the number of closed literals that are checked by the proof system during the execution of the $Demo$ program (Step 2).

Let us consider Step 1. In the body of the clauses defining each new predicate introduced by the *define-fold* procedure, there is at most one occurrence of an atom in $\{q_1(X), \dots, q_m(X)\}$ and all other occurrences of atoms are taken from the set $\{path(X), nopath(X)\} \cup \{sat(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$. Thus, the number of new predicate symbols that can be introduced is $\mathcal{O}((m+1) \cdot 2^{|\varphi|})$. The number of clauses introduced for each new predicate symbol is $\mathcal{O}(|\varphi|)$.

For each new clause, our transformation strategy performs one execution of the loop body, which starts off by applying the *instantiate* procedure and

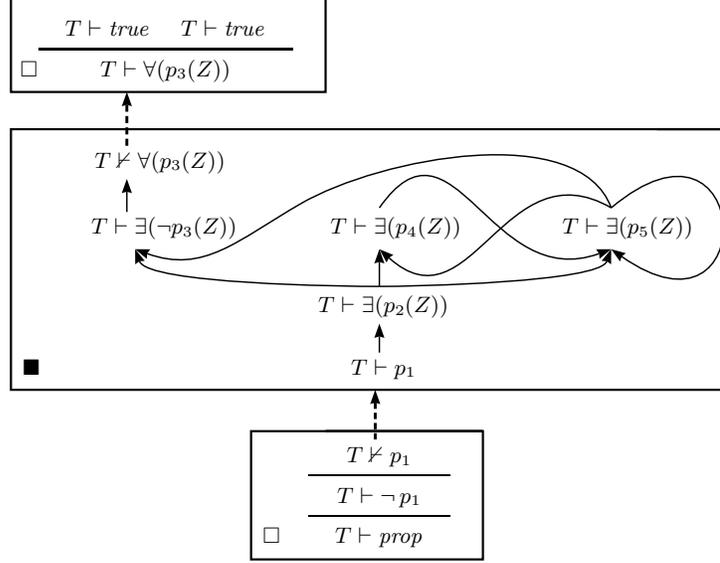


Fig. 2. Proof of $T \vdash prop$ (see Example 4). A rectangle marked by \square (or \blacksquare) shows the proof (or a disproof, respectively) of its root judgement at the bottom of the rectangle. In the rectangle marked by \blacksquare there is a solid arrow from judgement A to judgement B iff there exists an instance of a proof rule with conclusion A and premise B . A dashed uparrow from a negated judgement $T \not\vdash \varphi$ to a judgement $T \vdash \varphi$ indicates that in order to show that $T \not\vdash \varphi$ holds (or does not hold), we provide a disproof (or a proof, respectively) of $T \vdash \varphi$.

generates $\mathcal{O}(|\Sigma|)$ clauses. Then, it continues by applying the *unfold* procedure. The number of possible unfolding steps for each instantiated clause is $\mathcal{O}(|\varphi|)$. Thus, the total number of unfolding steps for each new clause is $\mathcal{O}(|\Sigma| \cdot |\varphi|)$, which is also the number of clauses generated by the instantiation and unfolding procedures. Finally, the *define-fold* procedure performs at most one folding step and definition step per clause. Considering all possible predicate symbols and recalling that the number of clauses introduced for each new predicate is $\mathcal{O}(|\varphi|)$, we get that the total number of transformation rule applications during Step 1 is $\mathcal{O}(|\Sigma| \cdot (m + 1) \cdot 2^{|\varphi| + 2 \log_2 |\varphi|})$.

In Step 2, by using tabled resolution, the proof system checks every closed literal at most once. The proof of a closed literal requires $\mathcal{O}((m + 1) \cdot 2^{|\varphi|})$ applications of proof rules. Therefore, we may conclude that the complexity of our algorithm is $\mathcal{O}(|\Sigma| \cdot (m + 1) \cdot 2^{|\varphi| + 2 \log_2 |\varphi|})$.

The complexity of the Lichtenstein-Pnueli algorithm is $\mathcal{O}((|\Sigma| + |\rho|) \cdot 2^{5|\varphi|})$ [9]. Since ρ is a total binary relation on Σ , we have that $|\Sigma| \leq |\rho| \leq |\Sigma|^2$. In the case where $|\rho| = |\Sigma|^2$ and, thus, $m = 0$, the complexity of the Lichtenstein-Pnueli algorithm is $\mathcal{O}(|\Sigma|^2 \cdot 2^{5|\varphi|})$ and the complexity of our algorithm is $\mathcal{O}(|\Sigma| \cdot 2^{|\varphi| + 2 \log_2 |\varphi|})$. In the case where ρ is a function, we have that $|\rho| = m = |\Sigma|$. Thus, the complexity of the Lichtenstein-Pnueli algorithm is $\mathcal{O}(|\Sigma| \cdot 2^{5|\varphi|})$ and the complexity of our algorithm is $\mathcal{O}(|\Sigma|^2 \cdot 2^{|\varphi| + 2 \log_2 |\varphi|})$. When $m = |\Sigma|$, it

may still be the case that $|\rho|$ is proportional to $|\Sigma|^2$ and, in this case, the Lichtenstein-Pnueli algorithm and our algorithm are both quadratic in the size of Σ . In the literature, $\mathcal{O}(2^{5|\varphi|})$ is often overestimated to $2^{\mathcal{O}(|\varphi|)}$ and, therefore, the Lichtenstein-Pnueli algorithm and our algorithm have essentially the same time complexity, that is, $\mathcal{O}(|\Sigma|^2) \cdot 2^{\mathcal{O}(|\varphi|)}$.

4 Related Work and Concluding Remarks

Various logic programming techniques and tools have been developed for model checking. For instance, tabled resolution has been shown to be quite effective for implementing a modal μ -calculus model checker for a CCS value passing language [16]. Techniques based on constraint logic programming, abstract interpretation, and program transformation have been proposed for performing CTL model checking of finite and infinite state systems (see, for instance, [6,7,12]).

The main novelties of this paper are the following: (i) we have proposed a method for specifying LTL properties of reactive systems based on ω -programs, that is, logic programs acting on infinite lists, (ii) we have also introduced the subclass of linear monadic ω -programs for which the truth in the perfect model is decidable and, finally, (iii) we have shown that we can transform, by applying semantics preserving unfold/fold rules, the logic programming specification of an LTL property into a linear monadic ω -program.

Our two step verification approach bears some similarity to the automata-theoretic approaches for LTL model checking, where the specification of a finite state system and an LTL formula are translated into nondeterministic Büchi automata [21] or alternating automata [11].

The automata-theoretic approach has the advantage that automata theory is very well studied and many results are available. However, we believe that also our approach has its advantages because of the following features. (1) The specification of the properties of the reactive systems, the transformation of the specification into a linear monadic ω -program, and the proof of the properties of a linear monadic ω -program can all be done within the single framework of logic programming, while in the automata-theoretic approach one has to use both the temporal logic formalism and the automata-theoretic formalism. (2) The translation of the specification into an ω -program can be performed by using semantics preserving transformation rules, thereby avoiding the burden of proving the correctness of the translation via *ad-hoc* methods.

Issues which can be investigated in future research include: (i) the relationships between linear monadic ω -programs, Büchi automata, and alternating automata, (ii) the strength of our transformational approach and its applicability to other logics, such as CTL* and the Monadic Second Order logic of successors, and (iii) the comparison of the efficiency of our approach w.r.t. that of other model checking techniques via experiments using practical examples.

References

1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.

2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
4. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01*, Technical Report DSSE-TR-2001-3, pages 85–96. Univ. Southampton, UK, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*, LNCS 3049, pp. 292–340. Springer-Verlag, 2004.
9. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *Proc. of POPL'85*, pp.97–107. ACM Press, 1985.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
11. D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of LICS '88*, pages 422–427. IEEE Press, 1988.
12. U. Nilsson and J. Lübbke. Constraint logic programming for local and symbolic model-checking. *Proc. of CL 2000*, LNAI 1861, pp. 384–398. Springer, 2000.
13. 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.
14. H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.
15. T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.
16. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. *Proceedings of CAV '97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
17. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I.V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science*, 13(3):387–403, 2002.
18. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
19. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB System, Version 2.2., 2000.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of ICLP'84*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
21. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). *Proc. LICS '86*, pp. 332–344. IEEE Press, 1986.

A Linear Temporal Logic

In this section we briefly recall the definition of the Linear Temporal Logic. We first introduce the notion of a Kripke structure, which models state transition systems, and then we define the semantics of LTL formulas with respect to Kripke structures. We refer the reader to [5] for further details.

Definition 3 (Kripke Structure). Let $Elem$ be a set of symbols denoting *elementary properties*. A *Kripke Structure* \mathcal{K} is a 4-tuple $\langle \Sigma, I, \rho, \lambda \rangle$ where:

1. Σ is a finite set of *states*;
2. $I \subseteq \Sigma$ is a set of *initial states*;
3. $\rho \subseteq \Sigma \times \Sigma$ is a total *transition relation*, that is, for every state $s \in \Sigma$ there exists a state $s' \in \Sigma$ such that $(s, s') \in \rho$;
4. $\lambda : \Sigma \rightarrow \mathcal{P}(Elem)$ is a total function that assigns to a state $s \in \Sigma$ a subset $\lambda(s)$ of $Elem$ which is the subset of the elementary properties that hold in s .

A *computation path* π in \mathcal{K} is an *infinite* list $[s_0, s_1, \dots]$ of states such that $s_0 \in I$ and, for every $i \geq 0$, $(s_i, s_{i+1}) \in \rho$. We use π_i to denote the sequence obtained by taking the suffix of π which starts at state s_i .

LTL is a propositional temporal logic for expressing properties of the computation paths of a Kripke structure. LTL makes use of the temporal operators X (*next time*) and U (*until*). Other temporal operators, such as F (*eventually*) and G (*always*), can be defined in terms of X and U as follows: for every formula φ , $G\varphi = \neg F\neg\varphi$, and $F\varphi = U(true, \varphi)$.

Definition 4 (LTL Formulas). Given a set $Elem$ of elementary properties, the syntax of the LTL formulas φ is as follows:

$$\varphi ::= e \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2, \quad \text{where } e \in Elem.$$

Now we define the satisfaction relation $\mathcal{K}, \pi \models \varphi$, which tells us when an LTL formula φ holds on a computation path π of the Kripke structure \mathcal{K} . We assume that the structure \mathcal{K} and the formula φ share the same set $Elem$ of elementary properties.

Definition 5 (Satisfaction Relation for LTL). Given a Kripke structure $\mathcal{K} = \langle \Sigma, I, \rho, \lambda \rangle$, a computation path π of \mathcal{K} , and an LTL formula φ , we inductively define the relation $\mathcal{K}, \pi \models \varphi$ as follows:

$$\begin{aligned} \mathcal{K}, \pi \models e & \quad \text{iff } \pi = s_0 s_1 \dots \text{ and } e \in \lambda(s_0) \\ \mathcal{K}, \pi \models \neg\varphi & \quad \text{iff } \mathcal{K}, \pi \not\models \varphi \\ \mathcal{K}, \pi \models \varphi_1 \wedge \varphi_2 & \quad \text{iff } \mathcal{K}, \pi \models \varphi_1 \text{ and } \mathcal{K}, \pi \models \varphi_2 \\ \mathcal{K}, \pi \models X\varphi & \quad \text{iff } \mathcal{K}, \pi_1 \models \varphi \\ \mathcal{K}, \pi \models \varphi_1 U \varphi_2 & \quad \text{iff there exists } i \geq 0 \text{ such that } \mathcal{K}, \pi_i \models \varphi_2 \\ & \quad \text{and, for all } 0 \leq j < i, \mathcal{K}, \pi_j \models \varphi_1 \end{aligned}$$

$\mathcal{K} \models \varphi$ iff for all computation paths π in \mathcal{K} we have that $\mathcal{K}, \pi \models \varphi$.

Graded Alternating-Time Temporal Logic

Marco Faella¹, Margherita Napoli², and Mimmo Parente²

¹ Università di Napoli “Federico II”, Via Cintia, 80126 - Napoli, Italy

² Università di Salerno, Via Ponte don Melillo, 84084 - Fisciano (SA), Italy

June 5, 2009

Abstract. Recently, graded modalities have been added to the semantics of two of the logics most commonly used by the computer science community: μ -calculus, in [KSV02], and CTL in [FNP08]. In both these settings, graded modalities enrich 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 .

Both μ -calculus and CTL naturally apply as specification languages for *closed* systems: in this paper, we study how graded modalities may affect specification languages for *open* systems. To this aim, we consider the Alternating-time Temporal Logic (ATL), a derivative of CTL that is interpreted on *games*, rather than transition systems.

We present two different semantics: the first seems suitable to off-line synthesis applications while the second may find application in the verification of fault-tolerant controllers.

For both these semantics, we efficiently solve the model checking problem, computing the truth values of graded ATL formulas on the states of a given game.

1 Introduction

Graded modalities are logical operators, especially well-known in the knowledge representation field, allowing to express quantitative bounds on the set of individuals satisfying a certain property [Fin72]. Similar notions are the counting quantifiers in classical logic [GOR97] and the number restrictions in description logics [HB91]. Recently, such notions have received renewed attention by the theoretical computer science community [CDL99,KSV02,FNP08,FMP08]. In these works, the authors have introduced graded modalities into the logics commonly used in computer science, such as the temporal logic CTL and the μ -calculus. In this paper, we make progress along these lines by adding graded modalities to the so-called Alternating-time Temporal Logic, ATL, and providing efficient model-checking algorithms for the resulting logic. ATL was introduced by Alur et al. [AHK02] as a derivative of CTL that is interpreted on *games*, rather than transition systems. Since its inception, ATL has been quickly adopted in different areas of computer science dealing with multi-agent systems, it has been implemented in several tools for the analysis of such systems [AHM⁺98,LR06], and it has provided the basis for further extensions [ÁGJ07,vdHW03].

The temporal part of ATL coincides with the one of CTL, while the path quantifiers of CTL are replaced by *team* quantifiers, that quantify over the strategies of a given team. For instance, for a suitable subformula θ , the ATL formula $\langle\langle 1 \rangle\rangle\theta$ expresses the fact that the team composed of Player 1 alone can ensure that θ holds. More in detail, said formula hides two classical quantifiers over strategies: there exists a strategy of Player 1, such that, for all strategies of Player 2, θ holds in the resulting outcome. Notice that, by a well-known result by Martin [Mar75] on the determinacy of games with Borel objectives, the two quantifiers above can be exchanged with no effect on the semantics.

Standard CTL path quantifiers can be obtained as special cases of ATL quantifiers. In a game with two players¹, the ATL formula $\langle\langle 1, 2 \rangle\rangle\theta$ states that the two players together can cooperate to ensure θ . Since this formula puts both players in the same team, it is equivalent to the CTL formula $\exists\theta$.

In this paper, we enrich the ATL quantifiers with an integral *grade*, as follows. For a natural number k , the graded ATL formula $\langle\langle 1 \rangle\rangle^k\theta$ affirms that Player 1 has k different strategies for ensuring θ . This corresponds to applying the grade k to the classical existential quantifier hidden in the team quantifier $\langle\langle 1 \rangle\rangle$. Now, simple examples show that the two classical quantifiers *cannot* be exchanged if one of them (i.e., the existential one) has been enriched with a grade. Therefore, we end up with two alternative semantics for the graded quantifier $\langle\langle 1 \rangle\rangle^k$. In the first semantics, that we already presented and that we call the *off-line* semantics, the existential quantifier comes before the universal one, leading to the following interpretation: “Player 1 has k different strategies such that for all Player 2’s strategies...”. In the second semantics, that we call the *on-line* semantics, the universal quantifiers comes first, leading to “For all strategies of Player 2, Player 1 has k different strategies such that...”.

The first semantics seems suitable to off-line synthesis applications. In this context, the game is a model of a control system, and the two players represent the controller and its environment, respectively. Verifying the property $\langle\langle 1 \rangle\rangle^k\theta$, and possibly computing k witnessing strategies for Player 1, corresponds to synthesizing k different controllers, that may later (i.e., off-line) be compared w.r.t. some external criterion.

On the other hand, the second semantics may find application in the verification of fault-tolerant controllers. In this case, we do not wish to restrict the moves of Player 1 (i.e., synthesize a controller), but rather we assume that the controller may take any of the (redundant) actions that are present in the game, and we just want to evaluate how many faults the controller can tolerate before violating its specification, where a fault is represented by the absence at runtime of a move that is present in the model. In this semantics, if the formula $\langle\langle 1 \rangle\rangle^k\theta$ holds true, then, for all behaviors of the environment, the controller has k different ways to achieve its goal. This can be interpreted as a rough estimate of the amount of fault-tolerance of the controller, in the following sense: The controller tolerates *some* sets of k faults, even if a single fault may invalidate all

¹ The original definition of ATL allows for multiple-player games. In this paper, for simplicity we treat two-player games only.

k ways to achieve the goal. We call this semantics “on-line” because it is related to the ability of the player to dynamically alter its behavior to overcome such faults.

The rest of the paper is organized as follows. Section 2 presents the basic definitions, including the two alternative semantics for graded ATL. Section 3 presents a fixpoint characterization of the two semantics. Section 4 performs a comparison between the two semantics. Section 5 describes the model-checking algorithms, computing the truth values of ATL formulas on the states of a given game. Finally, in Section 6 we give some conclusions.

2 Definitions

We treat games that are played by two players on a finite graph, where each state belongs to one of the players. The game starts in a state of the graph, and at each turn the player who owns the current state chooses one of its outgoing edges. As a consequence, the game *moves* to the destination state of that edge. The game continues in this fashion, until an infinite path in the game is formed. For an introduction to the general setting of game theory, see [OR94], while for a more specific introduction to games played on graphs, see [Tho95]. Throughout the paper, we consider a fixed set Σ of *atomic propositions*. The following definitions make this framework formal.

Games. A *game* is a tuple $G = (S_1, S_2, \delta, [\cdot])$ such that: S_1 and S_2 are disjoint finite sets of states; let $S = S_1 \cup S_2$, we have that $\delta \subseteq S \times S$ is the transition relation and $[\cdot] : S \rightarrow 2^\Sigma$ is the function assigning to each state s the set of atomic propositions that are true at s . We assume that games are non-blocking, i.e. each state has at least one successor in δ . In the following, unless otherwise noted, we consider a fixed game $G = (S_1, S_2, \delta, [\cdot])$.

A (finite or infinite) path in G is a (finite or infinite) path in the directed graph $(S_1 \cup S_2, \delta)$. Given a path ρ , we denote by $\rho(i)$ its i -th state, by $first(\rho)$ the first state, and by $last(\rho)$ the last state, when ρ is finite.

Strategies. A *strategy* in G is a pair (X, f) , where $X \subseteq \{1, 2\}$ is the *team* to which the strategy belongs, and $f : S^* \rightarrow S$ is a function such that for all $\rho \in S^*$, $(last(\rho), f(\rho)) \in \delta$. Our strategies are deterministic, or, in game-theoretic terms, *pure*. For a team $X \subseteq \{1, 2\}$, we denote by S_X the set of states belonging to team X , i.e. $S_X = \bigcup_{i \in X} S_i$, and we denote by $\neg X$ the opposite team, i.e. $\neg X = \{1, 2\} \setminus X$.

We say that an infinite path $s_0 s_1 \dots$ in G is *consistent* with a strategy $\sigma = (X, f)$ if for all $i \geq 0$, if $s_i \in S_X$ then $s_{i+1} = f(s_0 s_1 \dots s_i)$. We denote by $Out_G(s, \sigma)$ the set of all infinite paths in G which start from s and are consistent with σ . For two strategies $\sigma = (X, f)$ and $\tau = (\neg X, g)$, and a state s , we denote by $Out_G(s, \sigma, \tau)$ the unique infinite path which starts from s and is consistent with both σ and τ .

2.1 Logic

The logic ATL is defined in [AHK02]. We extend it with graded quantifiers.

Syntax. Consider the sets of *path formulas* Ψ and *state formulas* Φ defined via the inductive clauses below. Graded ATL is the set of the state formulas generated by the rules:

$$\begin{aligned}\Psi &::= \bigcirc\Phi \mid \Phi\mathcal{U}\Phi \mid \Box\Phi; \\ \Phi &::= q \mid \neg\Phi \mid \Phi \vee \Phi \mid \langle\langle X \rangle\rangle^k \Psi,\end{aligned}$$

where $q \in \Sigma$ is an atomic proposition, $X \subseteq \{1, 2\}$ is a team, and k is a natural number. The operators \mathcal{U} (until), \Box (globally) and \bigcirc (next) are the temporal operators. As usual, also the operator \diamond (eventually) can be introduced using the equivalence $\diamond p \equiv \text{true}\mathcal{U}p$. The syntax of ATL is the same as the one of graded ATL, except that the team quantifier $\langle\langle \cdot \rangle\rangle$ exhibits no natural superscript.

Semantics. We present two alternative semantics for graded ATL, called *off-line semantics* and *on-line semantics* for reasons explained in Section 4. Their satisfaction relations are denoted by \models^{off} and \models^{on} , respectively, and they only differ in the interpretation of the team quantifier $\langle\langle \cdot \rangle\rangle$.

We start with the operators whose meaning is invariant in the two semantics. Let ρ be an infinite path in the game, s be a state, and φ_1, φ_2 be state formulas. Denote by \mathbb{N} the set of non-negative integers. For $x \in \{\text{on}, \text{off}\}$, the satisfaction relations are defined as follows.

$$\begin{aligned}\rho \models^x \bigcirc\varphi_1 & \quad \text{iff } \rho(1) \models^x \varphi_1 \\ \rho \models^x \Box\varphi_1 & \quad \text{iff } \forall i \in \mathbb{N}(i) \models^x \varphi_1 \\ \rho \models^x \varphi_1\mathcal{U}\varphi_2 & \quad \text{iff } \exists j \in \mathbb{N}(j) \models^x \varphi_2 \text{ and } \forall 0 \leq i < j(i) \models^x \varphi_1 \quad (\dagger) \\ \\ s \models^x q & \quad \text{iff } q \in [s] \\ s \models^x \neg\varphi_1 & \quad \text{iff } s \not\models^x \varphi_1 \\ s \models^x \varphi_1 \vee \varphi_2 & \quad \text{iff } s \models^x \varphi_1 \text{ or } s \models^x \varphi_2.\end{aligned}$$

In order to state the meaning of the team quantifier, we need to introduce the following definitions. We say that two finite paths ρ and ρ' are *dissimilar* iff there exists $0 \leq i \leq \min\{|\rho|, |\rho'|\}$ such that $\rho(i) \neq \rho'(i)$. Observe that if ρ is a prefix of ρ' , then ρ and ρ' are not dissimilar. For a path ρ and an integer i , we denote by $\rho_{\leq i}$ the prefix of ρ comprising $i + 1$ states, i.e. $\rho_{\leq i} = \rho(0), \rho(1), \dots, \rho(i)$.

Given a path formula ψ and $x \in \{\text{on}, \text{off}\}$, we say that two infinite paths ρ and ρ' are (ψ, x) -*dissimilar* iff:

- $\psi = \bigcirc\varphi$ and $\rho(1) \neq \rho'(1)$, or
- $\psi = \Box\varphi$ and $\rho(i) \neq \rho'(i)$ for some i , or
- $\psi = \varphi_1\mathcal{U}\varphi_2$ and there are two integers j and j' such that:
 - $\rho(j) \models^x \varphi_2$,
 - $\rho'(j') \models^x \varphi_2$,
 - for all $0 \leq i < j$, $\rho(i) \models^x \varphi_1$,
 - for all $0 \leq i' < j'$, $\rho'(i') \models^x \varphi_1$, and
 - $\rho_{\leq j}$ and $\rho'_{\leq j'}$ are dissimilar.

Finally, two sets of infinite paths are (ψ, x) -dissimilar iff one set contains a path which is (ψ, x) -dissimilar to all the paths in the other set.

As explained in the following, graded ATL formulas have the ability to count how many different paths (in the on-line semantics) or strategies (in the off-line semantics) satisfy a certain property. However, it is not obvious when two paths should be considered “different”. For instance, consider the formula $p\mathcal{U}q$, and two infinite paths that start in the same state s , where s satisfies q and not p . Both paths satisfy $p\mathcal{U}q$, but only due to their initial state (i.e., $j = 0$ is the only witness for the definition (†)). Thus, we claim that these two paths should not be counted as two different ways to satisfy $p\mathcal{U}q$, because they only become different *after* they have satisfied $p\mathcal{U}q$. The notion of dissimilar (sets of) paths captures this intuition.

Off-line semantics. The meaning of the team quantifier is defined as follows, for a state s and a path formula ψ .

$$s \models^{\text{off}} \langle\langle X \rangle\rangle^k \psi \quad \text{iff there exist } k \text{ strategies } \sigma_1 = (X, f_1), \dots, \sigma_k = (X, f_k) \text{ s.t.}$$

$$\text{for all } i \neq j, \text{ Outc}(s, \sigma_i) \text{ and Outc}(s, \sigma_j) \text{ are } (\psi, \text{off})\text{-dissimilar}$$

$$\text{and } \forall \rho \in \text{Outc}(s, \sigma_i), \rho \models^{\text{off}} \psi .$$

On-line semantics. The meaning of the team quantifier is defined as follows, for a state s and a path formula ψ .

$$s \models^{\text{on}} \langle\langle X \rangle\rangle^k \psi \quad \text{iff for all strategies } \tau = (\neg X, f)$$

$$\text{there exist } k \text{ pairwise } (\psi, \text{on})\text{-dissimilar paths } \rho \in \text{Outc}(s, \tau)$$

$$\text{s.t. } \rho \models^{\text{on}} \psi .$$

For an ATL formula φ , a tag $x \in \{\text{on}, \text{off}\}$, and a state s , we set $\text{grad}^x(s, \varphi)$ to be the greatest integer k such that $s \models^x \varphi^k$ holds, where φ^k is obtained from φ by assigning grade k to the outmost team quantifier. If such integer does not exist, we set $\text{grad}^x(s, \varphi) = \infty$. We say that a state is a *decision point* if it belongs to Player 1 and it has at least two successors.

3 Fixpoint Characterization

In this section we provide a fixpoint characterization of the function $\text{grad}(s, \varphi)$ when the state s is given.

Let $T \subseteq S$ be the set of states of G where the ATL formula $\langle\langle 1 \rangle\rangle \theta$ holds, for $\theta = \Box q$ or $\theta = p\mathcal{U}q$, and let $\hat{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$. Consider the following functor $F^{\text{on}} : (T \rightarrow \hat{\mathbb{N}}) \rightarrow (T \rightarrow \hat{\mathbb{N}})$.

$$F^{\text{on}}(f)(s) = 1 \sqcup \begin{cases} \sum_{(s, s') \in \delta \text{ and } s' \in T} f(s') & \text{if } s \in S_1 \\ \min_{(s, s') \in \delta \text{ and } s' \in T} f(s') & \text{if } s \in S_2, \end{cases} \quad (1)$$

where $x \sqcup y$ denotes $\max\{x, y\}$.

Lemma 1. *Given $\theta = \Box q$ or $\theta = p\mathcal{U}q$, let T be the set of states where $\langle\langle 1 \rangle\rangle\theta$ holds. The function $f : s \in T \rightarrow \text{grad}^{\text{on}}(s, \langle\langle 1 \rangle\rangle\theta) \in \mathbb{N}$ is the least fixpoint of F^{on} .*

Proof. First, we prove that f is a fixpoint of F^{on} . Let $s \in T$. If $\theta = \Box q$, since $s \models^{\text{on}} \langle\langle 1 \rangle\rangle\theta$, we have that $s \models^{\text{on}} q$. If instead $\theta = p\mathcal{U}q$, either $s \models^{\text{on}} p$ or $s \models^{\text{on}} q$.

For all successors s_i of s , let $k_i = \text{grad}^{\text{on}}(s_i, \langle\langle 1 \rangle\rangle\theta)$. For each strategy τ of Player 2, there are k_i dissimilar paths starting from s_i , consistent with τ , and satisfying θ . Therefore, if $s \in S_1$, by adding state s in front of each of these paths, we obtain $\sum_i k_i$ dissimilar paths starting from s , consistent with τ , and satisfying θ . If instead $s \in S_2$, let $i = \arg(\min_j k_j)$. Consider the memoryless strategy τ of Player 2 that picks s_i when the game is in s . Under τ , there are k_i dissimilar paths starting from s and satisfying θ . From the choice of i , it follows that all strategies of Player 2 have at least as many dissimilar paths from s .

Next, we prove that f is the *least* fixpoint of F^{on} . Precisely, we prove by induction on n the following statement: Let g be a fixpoint of F^{on} and let $s \in T$, if $g(s) \leq n$ then $f(s) \leq g(s)$. Assume for simplicity that $\theta = \Box q$, as the other case can be proved along similar lines.

If $n = 1$, by hypothesis $g(s) = 1$. Considering the definition of operator (1), there are the following three possibilities: (i) s has no successors in T ; (ii) s belongs to S_1 and has only one successor in T ; (iii) s belongs to S_2 and has a successor t in T such that $g(t) = 1$. Option (i) can be discarded because T is the set of states where $\langle\langle 1 \rangle\rangle\Box q$ holds, and thus each state in T has at least one successor in T . Given the remaining two options, one can see that Player 2 can force the game in a loop where all states x have value $g(x) = 1$, and Player 1 cannot exit this loop. Accordingly, we have $f(s) = 1$, as requested.

If $n > 1$, by contradiction, let g be a fixpoint of F^{on} which is smaller than f . I.e., there is a state $s \in T$ such that $g(s) < f(s)$. Clearly, it must be $f(s) > 1$. Starting from s , build a path in the game in the following way. Let t be the current last state of the path (at the beginning, $t = s$): if $t \in S_2$, pick as the next state of the path a successor $u \in T$ of t such that $g(u) = g(t)$ (notice that $f(u) \geq f(t)$); if $t \in S_1$ and t has only one successor u in T , pick u as the next state (notice that $g(u) = g(t)$ and $f(u) = f(t)$); finally, if $t \in S_1$ and t has more than one successor in T , stop. If the above process continues forever, it means that Player 2 can force the game in a loop from which Player 1 cannot exit. As before, this means that $f(s) = 1$, which is a contradiction.

Otherwise, the above process stops in a state $t \in S_1$, such that $g(t) = g(s)$ and $f(t) \geq f(s)$. Therefore, $f(t) > g(t)$. Since t has more than one successor in T , by (1), for all successors u of t we have $g(u) < g(t) = g(s) \leq n$ and thus $g(u) \leq n - 1$. Moreover, there is a successor u^* of t such that $g(u^*) < f(u^*)$. On the other hand, by induction $g(u^*) \geq f(u^*)$, which is a contradiction. ■

Let us observe the following proposition that immediately follows from the Lemma and that will be used in the following.

Proposition 1. *Let $G = (S_1, S_2, \delta, [\cdot])$ be a game and let $\varphi = \langle\langle 1 \rangle\rangle^k \theta$, for $k > 1$ be a graded formula where $\theta = \Box q$ or $\theta = p\mathcal{U}q$. Given $s \in S$, if $s \models^{\text{on}} \varphi$, then:*

- if $s \in S_1$ then there exist t successors s_1, \dots, s_t of s in δ , and t integers k_1, \dots, k_t , such that for all $i = 1, \dots, t$, we have $s_i \models^{\text{on}} \langle\langle 1 \rangle\rangle^{k_i} \theta$, and $\sum_{i=1}^t k_i = k$.
- if $s \in S_2$ then all successors s' of s in δ are such that $s' \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \theta$.

Let T be the set of states where $\langle\langle 1 \rangle\rangle \theta$ holds. Consider the following operator $F^{\text{off}} : (T \rightarrow \hat{\mathbb{N}}) \rightarrow (T \rightarrow \hat{\mathbb{N}})$.

$$F^{\text{off}}(f)(s) = 1 \sqcup \begin{cases} \sum_{(s,s') \in \delta \text{ and } s' \in T} f(s') & \text{if } s \in S_1 \\ \prod_{(s,s') \in \delta \text{ and } s' \in T} f(s') & \text{if } s \in S_2. \end{cases} \quad (2)$$

Lemma 2. *Given $\theta = \square q$ or $\theta = p\mathcal{U}q$, let T be the set of states where $\langle\langle 1 \rangle\rangle \theta$ holds. The function $f : s \in T \rightarrow \text{grad}^{\text{off}}(s, \langle\langle 1 \rangle\rangle \theta) \in \mathbb{N}$ is the least fixpoint of F^{off} .*

Proof. First, we prove that f is a fixpoint of F^{on} . Let T be the set of states where $\langle\langle 1 \rangle\rangle \theta$ holds, and let $s \in T$. For all successors s_i of s in T , let us define $k_i = \text{grad}^{\text{off}}(s_i, \langle\langle 1 \rangle\rangle \theta)$. That is, k_i strategies of Player 1 exist which determine k_i (θ , off)-dissimilar sets of paths, $\text{Outc}(s_i, \cdot)$, consistent with the strategies and satisfying θ .

If $s \in S_1$, then the total number of strategies is the sum of $\text{grad}^{\text{off}}(s_i, \langle\langle 1 \rangle\rangle \theta)$, for each successor s_i of s in T . In fact, each path consistent with a strategy of Player 1 starting from s , can be obtained by adding the state s in front of one of the k_i paths consistent with a strategy starting from s_i .

If $s \in S_2$, then Player 1 has no choices in s and thus all the strategies starting from s are obtained by choosing one of the k_i strategies which start from each successor s_i .

Next, we prove that f is the *least* fixpoint of F^{off} . Similarly to the proof of Lemma 1, we prove by induction on n the following statement: Let g be a fixpoint of F^{off} and let $s \in T$, if $g(s) \leq n$ then $f(s) \leq g(s)$. Assume as before that $\theta = \square q$ (the other case is similar).

The case for $n = 1$ can be proved similarly to the proof of Lemma 1. If $n > 1$, by contradiction, let g be a fixpoint of F^{off} which is smaller than f . I.e., there is a state $s \in T$ such that $g(s) < f(s)$. Clearly, it must be $f(s) > 1$. Assume w.l.o.g. that also $g(s) > 1$, otherwise proceed as in the case for $n = 1$. Starting from s , build a path in the game in the following way. Let t be the current last state of the path (at the beginning, $t = s$): if t has only one successor u in T , pick u as the next state (notice that $g(u) = g(t)$ and $f(u) = f(t)$); if $t \in S_2$ and t has more than one successor in T , pick as the next state of the path a successor $u \in T$ of t such that $g(u) < f(u)$ (it is a simple matter of algebra to show that such a state exists); finally, if $t \in S_1$ and t has more than one successor in T , stop. If the above process continues forever, it means that Player 2 can force the game in a loop from which Player 1 cannot exit. As before, this means that $f(s) = 1$, which is a contradiction.

Otherwise, the above process stops in a state $t \in S_1$, such that $g(t) \leq g(s)$ and $g(t) < f(t)$. Since t has more than one successor in T , by (2), for all successors u of t we have $g(u) < g(t) \leq g(s) \leq n$ and thus $g(u) \leq n - 1$. Moreover, there

is a successor u^* of t such that $g(u^*) < f(u^*)$. On the other hand, by induction $g(u^*) \geq f(u^*)$, which is a contradiction. ■

4 Comparing the Two Semantics

The following examples show that in general the two semantics are different. In the figure, states of S_1 are circles and those of S_2 are squares.

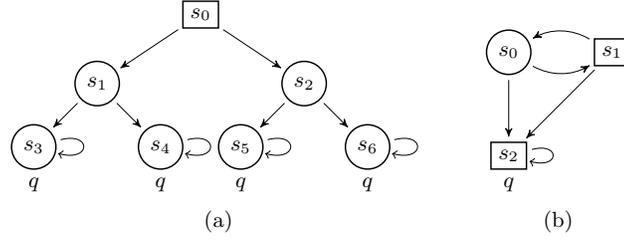


Fig. 1: Two games where the two semantics differ.

Example 1. Consider the game in Figure 1a, where the goal for Player 1 is to reach the proposition q , which is true in the leaves of the tree. According to the off-line semantics, there are 4 possible strategies to achieve that goal. Namely, there are two choices from s_1 and two choices from s_2 . The total number of strategies is then given by multiplying the two. Thus, we have $s_0 \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \diamond q$, $k \leq 4$ and $s_0 \not\models^{\text{off}} \langle\langle 1 \rangle\rangle^5 \diamond q$. On the other hand, according to the online semantics, for all strategies of Player 2, there are only two paths satisfying $\diamond q$. Thus, $s_0 \models^{\text{on}} \langle\langle 1 \rangle\rangle^2 \diamond q$ and $s_0 \not\models^{\text{on}} \langle\langle 1 \rangle\rangle^3 \diamond q$.

For a more extreme case, consider also the following.

Example 2. Consider the game in Figure 1b, where the goal for Player 1 is again to reach the proposition q , which is true in s_2 . According to the off-line semantics, there are infinitely many strategies to achieve that goal. One strategy goes directly from s_0 to s_2 . Another one goes first from s_0 to s_1 and then from s_0 to s_2 , if the Player 2 moves back to s_0 . Essentially, for all $i > 0$, there is a strategy of Player 1 that tries i visits to s_1 before going to s_2 . Thus, we have $s_0 \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \diamond q$, for all $k > 0$. On the other hand, according to the online semantics, for all strategies of Player 2, there are only two paths leading to victory. Thus, $s_0 \models^{\text{on}} \langle\langle 1 \rangle\rangle^2 \diamond q$ and $s_0 \not\models^{\text{on}} \langle\langle 1 \rangle\rangle^3 \diamond q$.

The following theorem states that the two semantics coincide when all quantifiers have grade 1.

Theorem 1. *For all games G , states s in G , and ATL state formulas φ , it holds that*

$$s \models^{\text{on}} \varphi \quad \text{iff} \quad s \models^{\text{off}} \varphi.$$

Proof. When all team quantifiers have grade 1, the classical quantifiers embedded in the ATL formula φ can be exchanged due to the well-known result by Martin on the determinacy of games with Borel objectives [Mar75]. Exchanging the classical quantifiers leads from one semantics to the other. ■

This theorem allows us to simply say in the following that a state s satisfies an ATL formula, without specifying whether the semantics is referred to, is the on-line or the off-line.

Now we prove that the on-line satisfaction of a formula implies its off-line satisfaction as well. First, let us give a technical lemma.

Lemma 3. *Let $G = (S_1, S_2, \delta, [\cdot])$ be a game and let $\varphi = \langle\langle 1 \rangle\rangle^k \theta$, for $k > 1$ be a graded formula where $\theta = \Box q$ or $\theta = p\mathcal{U}q$. Let $s \in S$:*

1. *if $s \models^{\text{on}} \varphi$, then there exists a finite path ρ with $\text{first}(\rho) = s$, $\text{last}(\rho) \in S_1$, and $\rho(i) \models^{\text{on}} \langle\langle 1 \rangle\rangle^1 \theta$ for all i . Moreover, $\text{last}(\rho)$ has at least two successors s_j in δ , such that $s_j \models^{\text{on}} \langle\langle 1 \rangle\rangle^1 \theta$, $j = 1, 2$;*
2. *if there exists a finite path ρ with $\text{first}(\rho) = s$ and $\text{last}(\rho) \in S_1$ and such that $\rho(i) \models^{\text{off}} \langle\langle 1 \rangle\rangle^1 \theta$, for all i , and $\text{last}(\rho) \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \theta$, then $\text{first}(\rho) \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \theta$.*

Proof. Let us first prove Item 1. From the definition of on-line semantics, at least two paths ρ_1 e ρ_2 exist such that they are (θ, on) -dissimilar and both satisfy θ according to the on-line semantics. From this dissimilarity of ρ_1 and ρ_2 , there exists an $i > 1$ such that $\rho_1(i) = \rho_2(i) \in S_1$ but $\rho_1(i+1) \neq \rho_2(i+1)$ and thus $\rho = \text{first}(\rho_1) \dots \rho_1(i)$ satisfies the Lemma. Note that since $k > 1$, s cannot belong to a connected component of G which does not have edges outgoing from it and that does not contain states of S_1 .

Consider now Item 2. Let ρ be a path satisfying the hypothesis. We prove that $\rho(i) \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \theta$, for all $0 \leq i < |\rho|$. Since $\rho(i) \models^{\text{off}} \langle\langle 1 \rangle\rangle^1 \theta$, then $\text{grad}^{\text{off}}(s', \langle\langle 1 \rangle\rangle \theta) > 0$, for each successor s' of $\rho(i)$ in δ and thus $\text{grad}^{\text{off}}(\rho(i+1), \langle\langle 1 \rangle\rangle \theta) \leq \text{grad}^{\text{off}}(\rho(i), \langle\langle 1 \rangle\rangle \theta)$, from Lemma 2. Hence, if $\rho(i+1) \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \theta$ then, $k \leq \text{grad}^{\text{off}}(\rho(i+1), \langle\langle 1 \rangle\rangle \theta) \leq \text{grad}^{\text{off}}(\rho(i), \langle\langle 1 \rangle\rangle \theta)$ and the thesis follows. ■

Theorem 2. *For all games G , states s in G , and a graded ATL formula φ*

$$\text{if } s \models^{\text{on}} \varphi \text{ then } s \models^{\text{off}} \varphi.$$

Proof. Let us first consider formulas $\varphi = \langle\langle 1 \rangle\rangle^k \theta$ con $\theta = \Box q$ or $\theta = p\mathcal{U}q$, where $p, q \in \Sigma$. The proof proceeds by induction on k . The base case for $k = 1$ follows from Theorem 1.

If $k > 1$, from Item 1 of Lemma 3 a path ρ of length $h \geq 0$ exists, whose states satisfy $\langle\langle 1 \rangle\rangle^1 \theta$ and with $\rho(0) = s$ and $\rho(h) \in S_1$, and this latter having two successors satisfying $\langle\langle 1 \rangle\rangle^1 \theta$. Without loss of generality suppose that each $\rho(j)$, $0 \leq j < h$ either belongs to S_2 or has just one successor satisfying $\langle\langle 1 \rangle\rangle^1 \theta$ (that is $\rho(h)$ is the first state occurring in ρ which belongs to S_1 and which has two successors satisfying $\langle\langle 1 \rangle\rangle^1 \theta$). This implies, by Proposition 1, that if $\rho(j) \models^{\text{on}} \varphi$ then also $\rho(j+1) \models^{\text{on}} \varphi$. Hence since $\rho(0) \models^{\text{on}} \varphi$, then $\rho(h) \models^{\text{on}} \varphi$, as well.

On the other hand, again from Proposition 1 also follows that there are s_1, \dots, s_t successors of $\rho(h)$ in δ , and k_1, \dots, k_t such that $\sum_{i=1}^t k_i = k$ e $s_i \models^{\text{on}} \langle\langle 1 \rangle\rangle^{k_i} \theta$. Since $\rho(h)$ has at least two successors satisfying $\langle\langle 1 \rangle\rangle^1 \theta$, then $t > 0$ and $k_i < k$. Thus, from the inductive hypothesis, $s_i \models^{\text{off}} \langle\langle 1 \rangle\rangle^{k_i} \theta$. Moreover from Lemma 2 $k = \sum_{i=1}^t k_i \leq \sum_{i=1}^t \text{grad}^{\text{off}}(s_i, \langle\langle 1 \rangle\rangle \theta) = \text{grad}^{\text{off}}(\rho(h), \langle\langle 1 \rangle\rangle \theta)$ and thus $\rho(h) \models^{\text{off}} \varphi$. Now noticing that the path ρ satisfies the hypothesis of Item 2 of Lemma 3 and $s = \rho(0)$, we can conclude that $s \models^{\text{off}} \varphi$.

To complete the proof of our statement we proceed by structural induction on a generic graded ATL formula. The base is trivial for the atomic propositions and for the non-temporal operators. Let φ be a graded ATL formula for which we inductively suppose that if $r \models^{\text{on}} \varphi$ then $r \models^{\text{off}} \varphi$, for all states r of G . If $s \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \bigcirc \varphi$, the statement trivially follows. Suppose now that $s \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \square \varphi$ and let \hat{G} be a new game obtained from G by simply renaming the states as follows: $\hat{S}_i = \{\hat{s} \mid s \in S_i\}$, $i = 1, 2$ and adding the new atomic proposition q_φ , holding true in all the states \hat{r} such that $r \models^{\text{on}} \varphi$. Clearly, $\hat{s} \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \square q_\varphi$ and, as shown above, $\hat{s} \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \square q_\varphi$. It is easy to see that this implies that $s \models^{\text{off}} \langle\langle 1 \rangle\rangle^k \square \varphi$, as well. The proof for the \mathcal{U} operator is similar. ■

5 Model Checking

Given a state s in G and a graded ATL formula φ , the model checking problem asks whether $s \models \varphi$. Notice that we only need to treat the $\langle\langle 1 \rangle\rangle$ quantifier. The $\langle\langle \emptyset \rangle\rangle$ and $\langle\langle 1, 2 \rangle\rangle$ quantifiers coincide with the \forall and \exists quantifiers of CTL, respectively. Therefore, they can be model-checked using the results of [FNP08]. Finally, the $\langle\langle 2 \rangle\rangle$ quantifier is the dual of $\langle\langle 1 \rangle\rangle$, and can therefore be evaluated using the algorithms developed for $\langle\langle 1 \rangle\rangle$ on a game where the roles of Player 1 and Player 2 have been reversed.

In this section, we provide algorithms for solving a stronger form of model checking: we compute $\text{grad}^x(s, \varphi)$, that is the greatest k such that $s \models^x \langle\langle 1 \rangle\rangle^k \theta$ holds, for a path formula θ and $x \in \{\text{on}, \text{off}\}$.

5.1 Off-line Semantics

Given a path formula $\theta = \square q$ or $\theta = p\mathcal{U}q$, we describe an algorithm for computing $\text{grad}^{\text{off}}(s, \langle\langle 1 \rangle\rangle \theta)$ for all states $s \in S$ (the \bigcirc operator is a simple case). Then, we show how to solve the model checking problem using said algorithm. In the following, we say that a strongly connected component of a graph is a *sink* if there are no outgoing edges from it.

Lemma 4. *For each state s , Algorithm 1 computes $\text{grad}^{\text{off}}(s, \langle\langle 1 \rangle\rangle \theta)$, for $\theta = \square q$ or $\theta = p\mathcal{U}q$. The algorithm runs in linear time.*

Proof. The algorithm first computes, as a base step, the states satisfying the ATL formula $\langle\langle 1 \rangle\rangle \theta$, and removes states and moves which do not contribute to winning this game. Then, it computes in the new game the strongly connected

Algorithm 1 The algorithm computing $grad^{off}(\cdot, \langle\langle 1 \rangle\rangle\theta)$, given a path formula $\theta = \Box q$ or $\theta = p\mathcal{U}q$.

1. Using standard ATL algorithms, compute the set of states T where $\langle\langle 1 \rangle\rangle^1\theta$ holds, and assign 0 to the states in $S \setminus T$. Then, compute the corresponding sub-game, i.e., remove the states not in T and the moves of Player 1 that leave T . If $\theta = p\mathcal{U}q$, remove also all moves leaving the states where $\neg p$ holds.
 2. On the sub-game, compute the strongly connected components.
 3. Proceed backwards starting from the sink components, according to the following rules.
 - (a) Sink components having more than one state and containing a decision point are assigned grade ∞ .
 - (b) Sink components which do not fall in case 3a are assigned grade 1.
 - (c) Non-sink components having more than one state and containing a decision point are assigned grade ∞ .
 - (d) Non-sink components which have more than one state and do not fall in case 3c are assigned ∞ if they have a successor component with grade greater than 1; otherwise, they are assigned 1.
 - (e) Non-sink components containing only one state: if this state is of Player 1 then it is assigned the sum of the grades of the successor components; while if the state is of Player 2, then it is assigned the product of the grades of the successor components.
-

components (we assume that there exists at least one such component, otherwise the statement trivially holds).

The algorithm computes the greatest grade k for each state s , such that $s \models \langle\langle 1 \rangle\rangle^k\theta$. It is immediate to observe that all the states belonging to a strongly connected component have the same grade, as they all have the same number of strategies, thus the same value is assigned to the whole component.

The algorithm proceeds as follows: first it examines all sink components, that is components from which there are no edges outgoing to other components. If such a component contains a decision point, then the value ∞ is assigned to it, otherwise the value 1 is assigned. Let us prove that this is correct. Let r be a decision point and call r_1, r_2 two of its successors. Informally speaking, for all $h > 0$ there is a strategy of Player 1 that, in the state r , chooses to visit r_1 h times, before visiting r_2 . More precisely, let $\alpha = r_1 \dots r$ be a finite path in G , define the strategy $\sigma_h = (\{1\}, f)$ where $f(\alpha^h) = r_2$ and for $j < h$, $f(\alpha^j) = r_1$. Clearly, for each $h > 0$, the strategies σ_i , $i \leq h$, determine the pairwise (θ, off) -dissimilar $Outc(r_2, \sigma_i)$ and thus, we have $r_2 \models^{off} \langle\langle 1 \rangle\rangle^k\theta$, for all $k > 0$. On the contrary, if there is not such a decision point, there is only one strategy in the connected component. Thus, steps 3a and 3b are correct and the correctness of step 3c can be derived in a similar way.

Consider now a non-sink component C having more than one state and not containing a decision point. If the algorithm has assigned 1 to all the successor components of C , then there is only one strategy for the team $\{1\}$. Otherwise, suppose that there is a state r in C of Player 2 having a successor r' in another

component and that there exist two strategies of Player 1 starting from r' . Then, for any way of alternating these two strategies, whenever the state r' is entered, there is a strategy of Player 1 from r , and thus the algorithm correctly assigns grade infinite.

Case 3e refers to singleton connected components. The algorithm correctly assigns the values of $grad^{off}(\cdot, \langle\langle 1 \rangle\rangle\theta)$ from Lemma 2.

Observe that the algorithm is complete as all cases have been examined and assuming an adjacency list representation for the game, the above algorithm runs in linear time. ■

To solve the model checking problem for graded ATL, we can follow the standard approach for the non-graded operators and design a trivial algorithm for the \bigcirc operator. For the other temporal graded operators we can use Lemma 4 as follows. Suppose that G has been model-checked against a given graded ATL formula φ . Then, to check whether $s \models^{off} \langle\langle 1 \rangle\rangle^{\hat{k}} \square \varphi$, for a given grade \hat{k} , Algorithm 1 can determine the greatest grade k such that $s \models \langle\langle 1 \rangle\rangle^k \square q_\varphi$ holds, for a new atomic proposition q_φ , holding true in each state r such that $r \models^{off} \varphi$. Similarly for the \mathcal{U} operator. Thus, the following theorem holds.

Theorem 3. *Given a game $G = (S_1, S_2, \delta, [\cdot])$, a state s in G and a graded ATL formula φ , the graded model checking problem, $s \models^{off} \varphi$, can be solved in time $\mathcal{O}(|\delta| \cdot |\varphi|)$, where $|\varphi|$ is the number of operators occurring in φ .*

The above complexity result assumes that each basic operation on integers is performed in constant time. Under this assumption, notice that the complexity of the model checking problem is independent of the integer constants appearing in the formula.

5.2 On-line Semantics

Similarly to the previous section, we describe an algorithm for computing $grad^{off}(s, \langle\langle 1 \rangle\rangle\theta)$ for $\theta = \square q$ or $\theta = p\mathcal{U}q$, and for all states $s \in S$.

Algorithm 2 The algorithm for computing $grad^{on}(\cdot, \langle\langle 1 \rangle\rangle\theta)$, given a path formula $\theta = \square q$ or $\theta = p\mathcal{U}q$.

1. Using standard ATL algorithms, compute the set of states T where $\langle\langle 1 \rangle\rangle^1\theta$ holds. The following steps are performed in the subgame with state-space T . In other words, the moves of Player 1 that leave T are removed. If $\theta = p\mathcal{U}q$, remove also all moves leaving the states where $\neg p$ holds. Assign grade 0 to the states in $S \setminus T$.
 2. Let d be a new atomic proposition which holds in the decision points (of the subgame). Find the states where $\langle\langle 2 \rangle\rangle^1 \square \neg d$ holds, and assign grade 1 to them.
 3. Find the states where $\langle\langle 1 \rangle\rangle^1 \square \diamond d$ holds, and assign grade ∞ to them.
 4. For the remaining states, compute their value by inductively applying equation (1) to those states whose successors have already been assigned a value.
-

Given a path formula $\theta = \Box q$ or $\theta = pUq$, Algorithm 2 computes $\text{grad}^{\text{on}}(s, \langle\langle 1 \rangle\rangle\theta)$, for all states $s \in S$. The complexity of the algorithm is dominated by step 3, which involves the solution of a Büchi game [Tho95]. This task can be performed in time $\mathcal{O}(|S| \cdot |\delta|)$, i.e., quadratic in the size of the adjacency-list representation of the game.

It is not obvious that the algorithm assigns a value to each state in the game. Indeed, step 4 assigns a value to a state only if all of its successors have already received a value. If, at some point, each state that does not have a value has a successor that in turn does not have a value, the algorithm stops. For the above situation to arise, there must be a loop of states with no value. The following lemma shows that the above situation cannot arise, and therefore that the algorithm ultimately assigns a value to each state.

Lemma 5. *At the end of step 3 of Algorithm 2, there is no loop of states with no value.*

Proof. By contradiction, assume that the thesis is false. We proceed by proving three intermediate claims, finally leading to a contradiction.

(a) First, we prove that all loops with no value contain a state which is labeled with d . If, during step 2, no state in the loop is labeled with d , each state in the loop either belongs to Player 2 or has one successor only. Therefore, all states in the loop satisfy $\langle\langle 2 \rangle\rangle^1 \Box \neg d$ and receive value 1 during step 2, which is a contradiction. Thus, at least one state in the loop is labeled with d (i.e., it is a decision point).

(b) It is clear that a state of Player 2 with no value cannot have a successor with value 1, otherwise by step 2 it would have value 1 too.

(c) Consider the set A of all states belonging to a loop of states with no value. Each state in A has a successor in A , with no value. If Player 1 always chooses to remain in A , by (b) we obtain an infinite path that either (i) remain forever in A , or (ii) eventually reaches a state with value ∞ . In the first case, by (a) the infinite path contains infinitely many occurrences of d . In the second case, again Player 1 can enforce infinitely many visits to d . This proves that each state in A satisfies $\langle\langle 1 \rangle\rangle^1 \Box \Diamond d$, which is a contradiction. Therefore, A must be empty and we obtain the thesis. ■

Lemma 6. *Given a path formula $\theta = \Box q$ or $\theta = pUq$, at the end of Algorithm 2, each state s has value $\text{grad}^{\text{on}}(s, \langle\langle 1 \rangle\rangle\theta)$. The algorithm runs in quadratic time.*

Proof. We proceed by examining the four steps of the algorithm. If state s receives its value (zero) during step 1, it means that $s \not\models^{\text{on}} \langle\langle 1 \rangle\rangle\theta$. Therefore, zero is indeed the largest integer k such that $s \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \theta$ holds.

If s receives its value (one) during step 2, it means that $s \models^{\text{on}} \langle\langle 2 \rangle\rangle^1 \Box \neg d$. Consider the strategy of Player 2 ensuring the truth of $\Box \neg d$. According to this strategy, Player 1 can never choose between two different successors. Therefore, there is a unique infinite path consistent with this strategy of Player 2. This implies that 1 is the greatest integer k such that $s \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \theta$ holds.

If s receives its value (infinity) during step 3, it means that $s \models^{\text{on}} \langle\langle 1 \rangle\rangle^1 \Box \Diamond d$. Consider any strategy τ of Player 2, and a strategy σ of Player 1 ensuring $\Box \Diamond d$. The resulting infinite path ρ contains infinitely many decision points for Player 1. For each decision point $\rho(i)$, let σ_i be a strategy of Player 1 with the following properties: (i) σ_i coincides with σ until the prefix $\rho_{\leq i}$ is formed, (ii) after $\rho_{\leq i}$, σ_i picks a different successor than σ , and then keeps ensuring θ . It is possible to find such a σ_i because $\rho(i)$ is a decision point in the subgame. For all $i \neq j$ such that $\rho(i)$ and $\rho(j)$ are decision points, the outcome of τ and σ_i is dissimilar from the outcome of τ and σ_j . Therefore, $s \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \theta$ holds for all $k > 0$.

Finally, if s receives its value during step 4, the correctness of the value is a consequence of Lemma 1. The complexity of the algorithm is discussed previously in this section. ■

Due to the above complexity result, and the discussion already made for the off-line semantics, we obtain the following conclusion.

Theorem 4. *Given a game $G = (S_1, S_2, \delta, [\cdot])$, a state s in G and a graded ATL formula φ , the graded model checking problem, $s \models^{\text{on}} \varphi$, can be solved in time $\mathcal{O}(|S| \cdot |\delta| \cdot |\varphi|)$, where $|\varphi|$ is the number of operators occurring in φ .*

As before, under the constant-time assumption for basic integer operations, the above complexity is independent of the integer constants appearing in the formula.

6 Conclusions

In this paper, we explored the consequences of adding counting capabilities to the team quantifiers of the game logic ATL. Such capability naturally leads to two different interpretations, according to the respective order of the two classical quantifiers hidden within each ATL team quantifier.

Given an ATL formula, an interesting computational question is then to determine the value of the maximum grade for which that formula is true on a given state of a game. For both interpretations, we provide a fixpoint characterization for that value. A fixpoint characterization also suggests a straightforward method for computing such value, namely Picard iteration: start with the lowest possible value and repeatedly apply the fixpoint operator. However, in our case, two issues prevent Picard iteration from being applied effectively. First, the maximum grade of a formula can be infinity. Second, even if grade infinity was to be treated separately, Picard iteration would still require a number of iterations proportional to the integer value being computed. For these reasons, we provide ad-hoc algorithms, that compute the maximum grade of a formula in polynomial time, avoiding the above-mentioned issues while still exploiting the fixpoint characterization.

Future work along these lines includes an investigation into the practical applications of these formalisms and algorithms, possibly in the field of fault-tolerant systems.

References

- [ÅGJ07] T. Ågotnes, V. Goranko, and W. Jamroga. Alternating-time temporal logics with irrevocable strategies. In *TARK '07: Proceedings of the 11th conference on Theoretical aspects of rationality and knowledge*, pages 15–24. ACM, 2007.
- [AHK02] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49:672–713, 2002.
- [AHM⁺98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In *CAV 98: Proc. of 10th Conf. on Computer Aided Verification*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 521–525. Springer-Verlag, 1998.
- [CDL99] F. Corradini, R. De Nicola, and A. Labella. Graded modalities and resource bisimulation. In *Proceedings of FST/TCS'99*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 381–393. Springer-Verlag, 1999.
- [Fin72] K. Fine. In so many possible worlds. *Notre Dame journal of Formal Logic*, 13(4):516–520, 1972.
- [FMP08] A. Ferrante, A. Murano, and M. Parente. Enriched μ -calculi module checking. *Logical Methods in Computer Science*, 4(3), 2008.
- [FNP08] A. Ferrante, M. Napoli, and M. Parente. Ctl model-checking with graded quantifiers. In *ATVA '08: Proc. of the 6th International Symposium on Automated Technology for Verification and Analysis*. (A preliminary version was presented at CILC '08), volume 5311 of *Lect. Notes in Comp. Sci.*, pages 18–32, 2008.
- [GOR97] E. Gradel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Proc. 12th IEEE Symp. Logic in Comp. Sci.*, 1997.
- [HB91] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 335–346, 1991.
- [KSV02] O. Kupferman, U. Sattler, and M.Y. Vardi. The complexity of the graded μ -calculus. In *Proc. 18th Conference on Automated Deduction*, volume 2392 of *Lect. Notes in Comp. Sci.*, 2002.
- [LR06] A. Lomuscio and F. Raimondi. Memas: A model checker for multi-agent systems. In *TACAS 06: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lect. Notes in Comp. Sci.*, pages 450–454. Springer-Verlag, 2006.
- [Mar75] D.A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975.
- [OR94] M.J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.
- [Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, volume 900 of *Lect. Notes in Comp. Sci.*, pages 1–13. Springer-Verlag, 1995.
- [vdHW03] W. van der Hoek and M. Wooldridge. Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Studia Logica*, 75(1):125–157, 2003.

A CLP engine for a general purpose configuration tool

Andrea Calligaris¹, Dario Campagna², Christian De Rosa³,
Agostino Dovier¹, Angelo Montanari¹, and Carla Piazza¹

¹ Dept. of Mathematics and Computer Science University of Udine
(dovier|montana|piazza)@dimi.uniud.it

² Dept. of Mathematics and Computer Science University of Perugia
dario.campagna@dipmat.unipg.it

³ Acritas s.r.l., Martignacco (UD), Italy derosa@acritas.it

Abstract. Product configurators are an emerging software technology, developed for helping companies in deploying mass customization strategies. Among the various approaches, constraint-based techniques look particularly encouraging, for both their modeling capabilities and their efficiency. In this paper, we present our work on a product configurator based on Constraint Logic Programming. The system we propose integrates an existing configurator module with a reasoning engine written in SICStus Prolog.

1 Introduction

The current competitive pressure leads many companies to offer to customers an increasing product variety. Such a flexibility, however, results into an increase in companies' costs, because of the high complexity introduced by product variety in production processes.

In current markets, customers ask for products closer to their needs, preserving high quality, short delivery times, and affordable costs. The attempt at overcoming the trade-off between costs and delivery times, on the one hand, and product variety, on the other hand, is commonly called *mass customization*. To operate according to mass customization means to sell products satisfying customer's needs, preserving as much as possible the advantages of *mass production* in terms of efficiency and productivity. This operating mode involves a series of difficulties that companies that want to adopt it struggle to resolve using traditional software tools, designed for repetitive productions.

A few years ago, software systems designed for supporting the order cycle of products realized in many variants, both standard and customizable, appeared on the market. These systems are called *software product configurators* and can be used by companies to successfully operate

according to mass customization. A software product configurator is a software tool that allows one to effectively and efficiently deal with problems related to product customization, automatically guaranteeing the satisfaction of all requirements. In addition, it must allow one to determine realization times and costs, generating a detailed plan about all the production process phases, from the order of raw materials to the delivery of the final product. A product configurator can thus be viewed as a sophisticated expert system and the need to study formalisms and to develop algorithms that make it possible to realize efficient and versatile configurators becomes apparent.

The configuration problem has recently attracted a significant amount of attention not only from the application point of view, but also from the methodological one [13]. In particular, significant approaches based on computational logic [6,14] and constraint programming [9,4,12] have emerged. In this paper, we describe the main achievements of our research on product configuration based on Constraint Logic Programming (CLP). We focused our attention on the development of a new CLP-based (product) configuration engine, called MCE, for the existing commercial configuration platform Morphos. In the proposed framework, the description of a product consists of a tree, whose nodes represent the components of a product, and a set of rules. Each node is paired with a set of properties, which express configurable characteristics of the component it represents. Each property is endowed with a finite domain (typically, a finite set of integers or strings), which represents the set of its possible values. The set of rules defines the compatibility relations between properties of product components.

The choice of the CLP paradigm has various motivations. First, many CLP systems providing constraint solvers with different efficiency degrees and constraint expressiveness characteristics are available, e.g., SICStus Prolog [15] and ECLⁱPS^e [2]. Moreover, the translation of a partial configuration (a partially configured product consisting of a set of nodes, with the associated properties, and a set of constraints about them) into a CLP program does not present any relevant difficulty and it produces a “natural” encoding of the original description. Finally, given a CLP program that represents a partial configuration, the CLP constraint solver makes it possible to verify the consistency of the partial configuration as well as to infer information about consequences of user’s choices.

In its current version, the system offers tools that support product configuration only (not product modeling). It takes advantage of a model of the product and of user choices about property values to create a pro-

gram in SICStus Prolog [15] encoding a Constraint Satisfaction Problem (CSP). Each variable of the CSP represents a property of the product being configured, while the constraints of the CSP encode the rules that specify the relationships among the properties of the product. Once created, the program is executed using the `clpfd` solver of SICStus Prolog. If all constraints are satisfiable, the execution of `clpfd` on the given program returns for each variable the associated, possibly restricted domain; otherwise, it certifies the inconsistency of the encoded CSP. Once the solver computation ends, MCE communicates to Morphos the domain (computed by the solver) of each variable whose domain has been restricted. Using these pieces of information, Morphos is able to prevent user’s choices that violate the given constraints.

The paper is organized as follows. In Section 2, we briefly describe the product configuration problem and existing software product configurators; moreover, we summarize the state of the art of research about the application of constraint programming to product configuration. In Section 3, we present the distinctive features of the configuration platform *Morphos*. The main contributions of the paper are illustrated in Section 4 and Section 5. Some comparisons between MCE and the original script-based Morphos configuration engine are reported in Section 6. Conclusions are drawn in Section 7.

2 Product configuration problem

The product configuration problem is fundamental for a large class of companies which offer products in different variants and options. In general, it requires a complex interaction with the customer to find, among the available characteristics and functions, those that best meet his/her needs. Such an interaction can result in combinations that have never been realized before. Products subject to configuration are products whose basic structure is predefined and that can be customized by combining together a series of available components and options (modules, accessories, etc.) or by specifying suitable parameters (lengths, tensions, etc.). The class of “products subject to configuration” includes products of different types, like, for instance, computers, cars, industrial machinery, shoes, etc.. With the term *configurable product* we refer to a type of product offered by a company. Hence, it does not correspond to a specific physical object, but it identifies a set of physical objects that the company can realize. A *configured product* is a single variant of the configurable product, which corresponds to a fully-specified physical object. A configured product is

obtained by customizing a configurable product, that is, by specifying the value of each customizable attribute of the configurable product. The series of activities and operations ranging from the acquisition of information regarding the particular variant of the product requested by the customer to the generation of data for the realization of the requested variant is called *configuration process*.

Companies offering customizable products face a series of management difficulties involving different functional areas. Many problems are related to an inadequate flow of information between the different areas. Traditional solutions adopted for the acquisition of customizable product orders are not, in general, satisfactory remedy to these difficulties. Recently, the major management software vendors have begun to develop and distribute software systems for product configuration. These systems aim at significantly improving the general management of configurable products. Software product configurators make available to the user a set of tools for the management of the customization of products. The “heart” of a software product configurator is the *product model*. It is a logical structure that formally represents the characteristics of the types of product offered by a company; moreover, it specifies a number of constraints on the relationships among these characteristics. The main modules of a product configurators are the *modeler* and the *configuration engine*. The modeler supports the modeling processes. It allows one to create and modify product models by defining characteristics and constraints of configurable products. The configuration engine supports the configuration processes. It facilitates the work of the seller, helping him/her in organizing and managing the acquisition of information about the product variant to be realized. In particular, it makes it possible to immediately check the validity and compatibility of inserted data.

A significant number of contributions witnesses the adequacy of constraint programming as a tool for product configuration, e.g., [3,10], in particular for knowledge representation and problem solving. Generally speaking, a configuration problem can be expressed as a Constraint Satisfaction Problem (CSP) whose solutions represent the possible configured products. A clear example of such an encoding can be found in [5], where the authors takes advantage of the classical N -queens problem to present the distinctive features of a product configurator based on constraint programming and preferences. Two applications of constraint programming to concrete instances of the configuration problem are reported in [7] (ILOG (J)Configurator system) and [4] (Lava system).

A common issue discussed in the literature is that of computational efficiency: suitable elements must be introduced to improve the efficiency of constraint solving. Other desirable system features are the ability to organize components in a hierarchy, the availability of tools that provide explanations on system behaviors, and the presence of interactive methods to express preferences and modify choices already made. To support at least in part these features, different variants of the standard CSP model have been proposed in the literature. An extension of standard CSP, called *Conditional CSP*, that makes it possible to manage optional components during the modeling phase in an effective way has been proposed in [9]. Another variant of CSP, close to Conditional CSP, called *Generative CSP*, has been developed in [4]. The handling of CSP at various levels of abstraction (*Composite CSP*) has been illustrated in [12]. A configurator that takes advantage of CSP and soft constraints to implement an iterative approach to the configuration problem is described in [5]. Finally, problems and advantages of configurators based on CSP have been discussed in detail in [1], where the CSP technology is integrated with that of *Truth Maintenance Systems* [8], which originates the so-called *assumption-based CSP*.

3 Morphos System

Morphos is a configuration platform developed by Acritas S.r.l. It is simple to use (being as simple as possible was one of the design principles of *Morphos*) and it can be easily integrated with the information system of a company. The user is guided in the process of creating of a new configured product: using a graphical user interface, he/she can provide a detailed definition of the components of the product; any given answer can be constrained through previously-defined rules that impose or restrict the possible choices. The process of maintaining configurations (storage of product revisions, creation of new configurations from existing ones) also reduces to a little number of immediate and intuitive steps.

The constraints on the characteristics of a product are specified by means of suitable JavaScript scripts. They are exploited by the configuration engine to infer information on product parameters after user's choices during the configuration process. On the positive side, the use of scripts gives a high degree of freedom in constraints definition and it provides an effective tool to manage computational procedures. On the negative side, it makes it difficult to maintain the constraints, it does not allow one to distinguish between constraints and computational procedures, and it

makes constraints consistency verification extremely involved and error-prone. In addition, no specific techniques are available to tackle complex constraints and no user-friendly tools help people having no specific skills in constraint definition.

The awareness of the difficulties inherent to the use of scripts and the intention of extending Morphos with new features and techniques for product modeling and configuration process management lead Acritas S.r.l. to start a collaboration with the Department of Mathematics and Computer Science of the University of Udine. The following sections describe the main achievements of such a collaboration.

4 Morphos Product Model

As already pointed out, a *Morphos product model* is defined by a tree, whose nodes represent well-defined components of a product, and a set of rules. Each node has associated a set of properties that describe configurable characteristics of the corresponding component. Each *node property* is characterized by a name and a set of integers or strings representing the set of values it can assume. This set can be given by explicitly enumerating its elements or by specifying a range (interval) of values, defined by a minimum value, a maximum value, and, possibly, a step (such an option is not supported by other systems, such as, for instance, Kumbang [11], where the elements of the domain of a variable must be explicitly given). Figure 1 shows the structure of the tree of a simple product model for a bicycle and the properties of the nodes **Fork**, **Wheel**, and **Crank**.

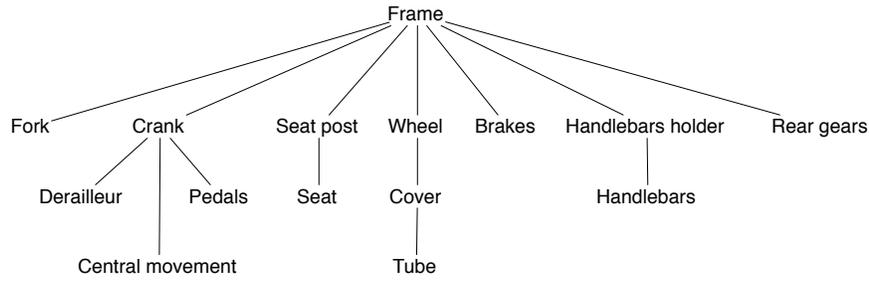
The constraints that define the compatibility relations between the properties of the components of a product can be of three types. To define them, we preliminarily introduce the notions of term, primitive constraint, and node constraint:

Term. A property of a node is a term. A constant (integer number or string) is a term. If t_1 and t_2 are terms, then $t_1 \oplus t_2$ is a term, where $\oplus \in \{+, -, *, /\}$ (notice that all terms are ground).

Primitive constraint. A primitive constraint has the form $p \text{ op } t$, where p is a node property, t a term, and $\text{op} \in \{<, \leq, =, \geq, >, \neq\}$.

Node constraint. A primitive constraint is a node constraint. If F and G are node constraints, then $F \wedge G$, $F \vee G$, and $\neg F$ are node constraints.

We distinguish three different types of constraint:



Node	Property	Type	Value
Fork	Type	String	Racing, MTB
	Material	String	Aluminum, Carbon
	ShockAbsorber	String	80 mm, 100mm, 85-130 mm, SingleChassis
Crank	Type	String	Racing, MTB
	Central movement	String	Integrated, None
	Gears	String	46/36, 48/38, 50/34, 44/32/22
Wheel	Type	String	Racing, MTB
	Material	String	Aluminum, Carbon
	Diameter	String	22",24",26",28",29"
	Spokes number	Integer	From 18 to 24 with step 2
	Cover	String	Traditional, Tubeless

Fig. 1. Bicycle: tree structure and node properties.

Rule. A rule is defined by a *condition*, a node constraint, and a series of *actions* (commands) to be executed by Morphos when the condition holds or a set of node constraints that must be satisfied when the condition holds. An example of rule for the bicycle product model is the following one:

Condition: Fork.Type = Racing

Action: Fork.ShockAbsorber = SingleChassis

Constraint. A constraint is a rule whose condition always holds (True condition), and whose action consists of a node constraint, that is, a constraint defines a condition that must be satisfied by all configurations. An example of constraint for the bicycle product model is the following one:

Condition: True

Action: Wheel.Cover = Cover.Typology

Relation: A relation is a table that relates two or more properties. It defines a set of admissible tuples of values, that is, an admissible

subset of the Cartesian product of the set of values of the properties it relates. An example of relation for the bicycle product model is given in Table 1.

Crank.Type	Crank.Gears
Racing	46/36
Racing	48/38
Racing	50/34
MTB	44/32/22

Table 1. Bicycle: an example of relation.

As we will show in Section 5, in the current version of the system, product model constraints behave as *global rules*: they hold for all the ordered tuples of instance properties they involve. However, taking advantage of the tree structure of the product model one may think of introducing *local rules*, that is, constraints which hold only for tuples of instance properties belonging to the nodes of a specific subtree. Such an ability of dealing with local rules will be incorporated in the next release of the engine.

The above-described solution is quite different from the one supported by the original script-based Morphos configuration engine: a script was associated with each node, which was executed whenever a change in a property of the node took place. No special mechanisms to control script execution were available. As a consequence, the original Morphos engine was unable to prevent unacceptable behaviors, such as infinite computations, from occurring, as shown by the following simple example. Let us assume that we have two nodes *Node1* and *Node2*, which respectively contain properties *A* and *B*. Both properties have the set $\{0,1\}$ as their domain. In addition, *Node1* and *Node2* are tied up with the scripts Script1 and Script2, respectively, which are reported in Table 2. The resulting behavior can be summarized as follows. Whenever property *A* changes, then Script1 is executed setting the value of *B* accordingly to that of *A*. Similarly, whenever property *B* changes, Script2 sets a new value for *A*, but this time the value for *A* is the negation of that for *B*. It can be easily checked that this can lead to an infinite computation. For instance, let us assign value 0 to *A*. Since *A* has changed, Script1 is executed. Due to the first rule of Script1, value 0 is assigned to *B*. The change in the value of *B* causes the execution of Script2, whose first rule assigns value 1 to

$A = 0 \Rightarrow B := 0$	$B = 0 \Rightarrow A := 1$
$A = 1 \Rightarrow B := 1$	$B = 1 \Rightarrow A := 0$

Table 2. Script1 (left) and Script2 (right)

A. Such a change forces the engine to execute Script1 again and, since the value of A is 1, it assigns value 1 to B . The change in the value of B forces the execution of Script2 that assigns value 0 to A , the value that was assigned to A at the very beginning. Hence, whenever we assign the value 0 (resp., 1 to A (resp., B)) an infinite cycle is generated.

5 Morphos Configuration Engine (MCE)

To improve the capabilities of the Morphos engine we developed a new configuration engine, called *Morphos Configuration Engine (MCE)*, taking advantage of CLP techniques. Figure 2 pictorially shows how the Morphos MCE system manages the configuration process. First, Mor-

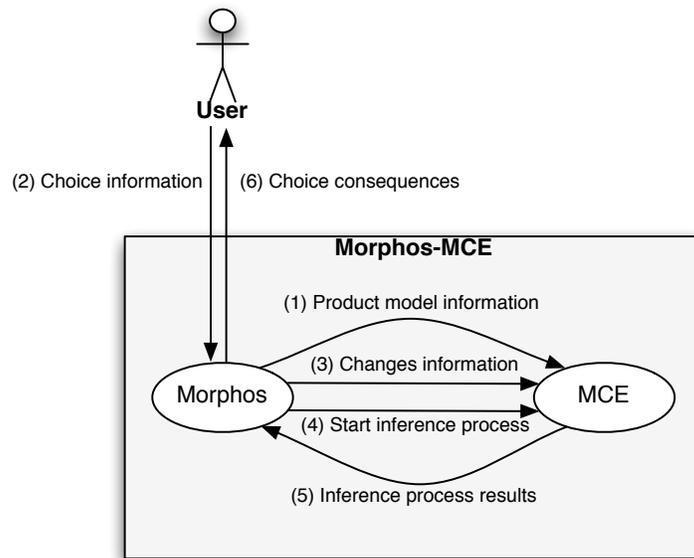


Fig. 2. Morphos-MCE: the configuration process.

phos initializes MCE (1) sending to it information about the model of the product to be configured, that is, nodes, properties, and constraints

defined by the product model. After such an initialization phase, the interaction with the user begins. The user makes his/her choices using the Morphos interface (2). He/she can add or remove node instances (product components) and set the values of node properties (product component characteristics). Morphos communicates to MCE each data variation specified by the user (3) and MCE updates the current partial configuration accordingly. MCE can then be exploited to infer information from the resulting partial configuration (4). Once the inference process ends, MCE returns to Morphos the results of its computation (5). Morphos in its turns shows to the user the consequences of his/her choices on the (partial) configuration (6).

The Morphos MCE system also allows the user to modify past choices about property values. Figure 3 shows how the Morphos MCE system manages the assignment revision process. We illustrate such a capabil-

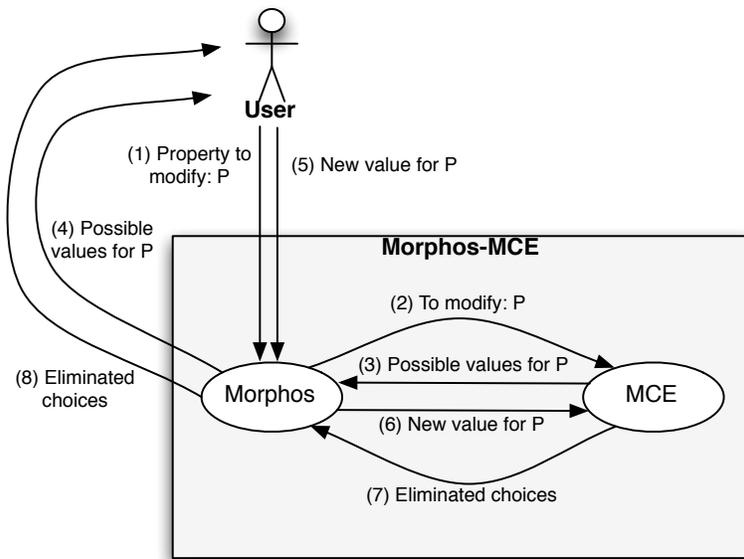


Fig. 3. Morphos-MCE: the assignment revision process.

ity of the Morphos MCE system by means of a simple example. Let us assume that the user would like to modify the value he/she previously assigned to property P. The user selects property P using Morphos (1). Morphos communicates to MCE that the user would like to modify the

value of P (2). MCE computes the set of values that can be assigned to P without incurring in conflicts with constraints and choices made before the one concerning property P , and it communicates the result to Morphos (3). Morphos shows to the user the values he/she can assign to P (4). The user chooses the new value for P (5). Morphos communicates it to MCE (6). MCE computes a maximal subset of the choices made after the one concerning P that can be maintained without generating any conflict with constraints, previous choices, and the new choice for P , and it communicates to Morphos the choices that have to be withdrawn (7). Morphos shows to the user which choices must be withdrawn to keep the configuration consistent (8). The user can change the subset of maintained choices according to his/her own preferences. In doing that, he/she can impose to the system to reintroduce some choices it withdrawn, thus forcing it to recompute the subset of choices to withdraw.

Let us describe now the CLP encoding of the product configuration problem. Starting from information about the product model and the partial configuration (node instances and choices on property values), MCE defines a SICStus Prolog program encoding a CSP. The variables of the CSP represent the properties of nodes instances in the partial configuration, the domains of the variables are the sets of possible values for the properties defined in the product model, the constraints of the CSP are the rules, the constraints, and the relations about the properties of node instances in the partial configuration (choices of property values are viewed as primitive constraints). Rules, constraints, and relations are translated into SICStus Prolog constraints as follows. Let us consider first the case of node constraints.

- A primitive constraint $p \text{ op } t$ is translated into a constraint $P \text{ RelOp } t$, where RelOp is the SICStus Prolog relational operator corresponding to op .
- A node constraint $F \wedge G$ (resp., $F \vee G$, $\neg F$) is translated into a constraint $F \ \#\wedge \ G$, (resp., $F \ \#\vee \ G$, $\#\neg F$), where F and G are the translations in SICStus Prolog of F and G , respectively.

Constraints defining compatibility relations among the properties of product components are translated in SICStus Prolog as follows.

- A rule with condition defined by a node constraint C and actions defined by node constraints A_1, \dots, A_n is translated into a SICStus Prolog constraint $C \ \#\Rightarrow \ A_1 \ \#\wedge \ \dots \ \#\wedge \ A_n$, where C , A_1 , A_2 , etc. are the translations in SICStus Prolog of the node constraints C, A_1, \dots, A_n .

- A constraint with action defined by node constraint A is translated into a SICStus Prolog constraint A .
- A relation on properties p_1, \dots, p_n consisting of the tuples $\langle t_{1,1}, \dots, t_{1,n} \rangle, \dots, \langle t_{m,1}, \dots, t_{m,n} \rangle$ is translated into a set of SICStus Prolog constraints of the form $p_i \# = t_j(i) \# => (p_1 \text{ in } \{\dots\} \#/\ \dots \#/\ p_i - 1 \text{ in } \{\dots\} \#/\ p_i + 1 \text{ in } \{\dots\} \#/\ \dots \#/\ p_n \text{ in } \{\dots\})$.

It is worth pointing out that a configuration may include multiple instances of each node defined in the product model. As an example, a bicycle configuration contains two instances for node `Wheel` and two instances for node `Cover`. Each constraint in the product model must hold for all the ordered tuples of instance properties it involves, e.g., the constraint with action `Wheel.Cover = Cover.Typology` must hold for all the four couples of instance properties `Wheel.Cover` and `Cover.Typology`. As a consequence, for each constraint in the product model and each ordered tuple of instance properties it involves, a constraint is inserted in the SICStus Prolog program defined by MCE.

Given a program with the above-described characteristics, the `clpfd` solver of SICStus Prolog computes the possibly restricted domain associated with each variable, provided that all constraints are satisfied; otherwise, it detects the inconsistency of the encoded CSP. When the computation of the solver ends, MCE communicates to Morphos the domain (computed by the solver) of each variable whose domain has been restricted. By exploiting information about (restricted) domains, Morphos can prevent user's choices that would violate the constraints, that is, user's choices incompatible with previous ones. Moreover, if there are commands to be executed, MCE communicates them to Morphos as well.

We conclude the section with a short description of how MCE uses `clpfd` during the assignment revision process. Let us consider again the case when the user would like to modify the value he/she assigned to property P . To compute the set of alternative values that can be assigned to P without generating conflicts with constraints and choices made before the one concerning property P , MCE defines a SICStus Prolog program encoding a CSP as in the configuration process, but without considering the primitive constraints representing choices on property values made after the one the user would like to modify. To compute a maximal subset of the choices made after the one concerning P that can be maintained without generating conflicts with constraints, previous choices, and the new choice for P , MCE defines a SICStus Prolog program encoding a CSP as in the configuration process, but with the following additional features. A variable N_i with domain $\{0, 1\}$ is associated with each assignment to

property values A_i made after the one for P . Let m be the number of these assignments. For each of them, a constraint of the form $N_i \#=> A_i$, where A_i is the translation of the primitive constraint representing A_i , is inserted in the program (instead of A_i). Moreover, a constraint of the form $N_1 + N_2 + \dots + N_m \#= T$, where T is a variable with domain $\{0, \dots, m\}$, is inserted in the program. A maximal subset of the assignment A_i that maintains the configuration consistent is computed using a predicate of the form `maximize(labeling([], [N_1, . . . , N_m, T]), T)`.

6 Comparisons with Morphos original engine

MCE has been developed in C# inside the .NET 3.5 framework. It provides several methods to solve the configuration problem. In particular, it supports the following functionalities:

- it communicates with Morphos through XML messages;
- it records and manages both information received from Morphos and solutions returned to Morphos;
- it creates and executes CLP programs.

The execution of CLP programs exploits the *SICStus Runtime Environment*⁴

Compared with the original configuration engine, MCE presents several advantages. We would like to point out a couple of them.

A first advantage of the CLP-based engine of MCE with respect to Morphos script-based one is represented by the simplicity of the translation of rules, constraints, and relations into a SICStus Prolog program. This allows one to easily check whether a partial configuration is satisfiable or not. On the contrary, the script-based engine makes it necessary to define an ad-hoc script for each constraint on properties that must be satisfied. With the CLP-based engine, it is sufficient to define the constraints within the model; the engine automatically inserts all constraints about properties involved in the partial configuration in the SICStus Prolog program.

A second advantage is given by the use of a constraint solver to remove from the domains the values that are incompatible with already made

⁴ SICStus Prolog allows one to create a standalone environment where CLP programs can run, without requiring the installation of the SICStus suite on the final user's computer.

choices. Consider, for instance, the following rule for the bicycle product model:

Condition: `Fork.Type = Racing`

Action: `Fork.ShockAbsorber = SingleChassis`

This rule states that whenever the value `Racing` is assigned to property `Type` of node `Fork`, property `ShockAbsorber` of node `Fork` can only assume the value `SingleChassis`. At the same time, it states that if a value different from `SingleChassis` is assigned to property `ShockAbsorber` of node `Fork`, then property `Type` of node `Fork` cannot assume the value `Racing`. The single SICStus Prolog constraint we obtain from this rule imposes both conditions. On the contrary, in the case of the original script-based engine, Morphos makes use of two different scripts to impose the two conditions. This makes the modeling phase much more complex, since the programmer has both to determine all possible implications of rules and to encode them by means of suitable scripts. Moreover, the complexity of the encoding complicates the maintenance phase.

7 Conclusions

In this paper we briefly introduced the important problem of product configuration and we described an approach to it based on Constraint Logic Programming. In particular, we illustrated the distinctive features of a CLP-based configuration engine, called MCE, which has been incorporated in the product configurator system Morphos developed by Acritas S.r.l. Besides an intuitive account of the behavior of MCE, we pointed out its advantages with respect to the original script-based configuration engine of Morphos. A simple case of product configuration referring to the domain of bicycles has been used as a source of exemplification.

Besides experimenting the proposed tool on different real-world application domains, we are currently refining/extending it in several directions. On the one hand, we are refining MCE by adding the possibility of distinguishing between local and global rules, as well as that of dealing with different numerical domains (e.g., integers and reals); on the other hand, we are extending Morphos with model and process configuration capabilities.

Acknowledgments. This work has been partially supported by Acritas S.r.l. and Regione Friuli Venezia Giulia. We would like to thank Maurizio Piani, Claudio Buble, Andrea Formisano, and Andrea Schiavinato for useful discussions and suggestions.

References

1. J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs-Application to configuration. *Artificial Intelligence*, 135:199–234(36), February 2002.
2. K.R. Apt and M.G. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, 2006.
3. F.Frayman and S. Mittal. COSSACK: A Constraint-Based Expert System for Configuration Tasks. *Knowledge-Based Expert Systems in Engineering: Planning and Design*, pages 143–166, 1987.
4. G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.
5. E. Freuder, C. Likitvivatanavong, M. Moretti, F. Rossi, and R. Wallace. Computing explanations and implications in preference-based configurators. In Barry O’Sullivan, editor, *Recent Advances in Constraints*, volume 2627 of LNAI, pages 76–92., 2003.
6. G. Friedrich and M. Stumptner. Consistency-based configuration. In *AAAI-99 Workshop on Configuration*, pages 35–40. AAAI Press, 1999.
7. U. Junker. The Logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic. In *IJCAI-03 Workshop on Configuration*, pages 13–20, 2003.
8. J. P. Martins. The truth, the whole truth, and nothing but the truth: An indexed bibliography to the literature of truth maintenance systems. *AI Mag.*, 11(5):7–25, 1991.
9. S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *AAAI*, pages 25–32, 1990.
10. S. Mittal and F. Frayman. Making Partial Choices in Constraint Reasoning Problems. *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 631–636, 1987.
11. V. Myllärniemi, T. Asikainen T. Männistö, and T. Soinen. Kumbang configurator - a configuration tool for software product families. In *IJCAI-05 Workshop on Configuration*, 2005.
12. D. Sabin and E. C. Freuder. Configuration as Composite Constraint Satisfaction. In George F. Luger, editor, *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996, 1996.
13. D. Sabin and R. Weigel. Product configuration frameworks - a survey. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 13(4):42–49, Jul/Aug 1998.
14. T. Soinen, I. Niemelä, J. Tiihonen, and R. Sulonen. Unified configuration knowledge representation using weight constraint rules. In *ECAI-00 Configuration Workshop*, pages 79–84, 2000.
15. Swedish Institute of Computer Science, Intelligent Systems Laboratory. *SICStus Prolog User’s Manual*, 4.0.3 edition, May 2008.

Bottom-up Evaluation of Finitely Recursive Queries

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone

Department of Mathematics, University of Calabria, I-87036 Rende (CS), Italy
e-mail: {calimeri, cozza, ianni, leone}@mat.unical.it

Abstract. The support for function symbols in logic programming under Answer Set Programming semantics (ASP) allows to overcome some modeling limitations of traditional ASP systems, such as the inability of handling infinite domains. On the other hand, admitting function symbols in ASP makes inference undecidable in the general case. Thus, the research is lately focusing on finding proper subclasses of ASP programs with functions for which decidability of inference is guaranteed. The two major proposals so far, finitary programs and finitely-ground programs, are complementary, to some extent; indeed, the former are conceived for allowing decidable querying (using a top-down evaluation strategy), while the latter for allowing finite model computation (using a bottom-up evaluation strategy). One of the main advantages of finitely-ground programs is that they can be directly evaluated by current ASP systems. However, many programs lie outside this class, such as, in general, positive finitary programs. Indeed, ground queries over such programs can be easily answered by means of top-down techniques; bottom-up approaches, instead, are, in general, unsuitable. In this work we present a proper adaptation of the magic-sets technique that aims at making query answering over positive finitary (normal) programs feasible by means of bottom-up techniques, i.e., those all current ASP systems rely on.

1 Introduction

Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP) [1–5], evolved significantly during the last decade, and has been recognized as a convenient and powerful method for declarative knowledge representation and reasoning. Lately, the ASP community has clearly perceived the strong need to extend ASP by functions, and many relevant contributions have been done in this direction [6–11]. Supporting function symbols allows to overcome one of the major limitation of traditional ASP systems, i.e. the ability of handling finite sets of constants only. On the other hand, admitting function symbols in ASP makes the common inference tasks undecidable or not computable.

Finitary programs [11] is a class of logic programs that allows function symbols yet preserving decidability of ground querying by imposing restrictions both on recursion and on the number of potential sources of inconsistency. Recursion is restricted by requiring each ground atom to depend on finitely many ground atoms; such programs are called *finitely recursive*. Moreover, the number of odd-cycles (cycles of recursive calls involving an odd number of negative subgoals) is required to be finite, so restricting the potential source of inconsistency. Thanks to these two restrictions, consistency checking and ground queries are decidable while nonground queries are semidecidable.

The class of *finitely-ground* (\mathcal{FG}) programs [6], recently proposed, can be seen as a “dual” notion of the class of finitary programs. Indeed, while the latter is suitable for a top-down evaluation, the former allows a bottom-up computation. Basically, for each program P in this class, there exists a finite subset P' of its instantiation, called *intelligent instantiation*, having precisely the same answer sets as P . Importantly, such a subset P' is computable for \mathcal{FG} programs. Both finitary programs and finitely-ground programs can express any computable function, and

preserve decidability for ground queries. However, answer sets and non-ground queries are computable on finitely-ground programs, while they are not computable on finitary programs. Furthermore, the bottom-up nature of the notion of finitely-ground programs allows an immediate implementation in ASP systems (as ASP instantiators are based on a bottom-up computational model). Indeed, the DLV system [12], for instance, has already been adapted to deal with finitely-ground program by properly extending its instantiator [13].

The two above mentioned classes are not comparable. In particular, the class of finitely-ground programs does not include programs for which ground querying could be trivially decided by following a top-down approach, i.e., positive finitely recursive programs. These are some of the simplest finitary programs; still, they are not, in general, suited for a bottom-up evaluation. However, this class is of clear interest; indeed, it includes many significant programs¹, such as most of the standard predicates on lists. For instance, the following program, performing the check for membership of an element in a list, is finitely recursive and positive.

$$\begin{aligned} &member(X, [X|Y]). \\ &member(X, [Y|Z]) :- member(X, Z). \end{aligned} \tag{1}$$

This work aims at overcoming this shortage by finding a strategy that allows a bottom-up evaluation of queries on positive finitely recursive normal logic programs. It is worth noting that a working translation module, actually not considering the peculiar optimization techniques herein presented for finitely recursive queries, is described in [14].

The main contributions of this work can be summarized as follows.

- We design a suitable adaptation of the *magic-sets* rewriting technique for programs with functions.
- We define the class of finitely recursive queries as the queries that can be answered taking into account only a finitely recursive fragment of a program.
- Given a positive finitely recursive program P and a (ground) query Q , let $RW(Q, P)$ denote the logic program obtained by our magic-sets rewriting technique. Then, we present the following main results:
 - $P \models Q$ if and only if $RW(Q, P) \models Q$;
 - given a finitely-recursive query Q on a program P , the size of $RW(Q, P)$ is linear in the size of the input program;
 - given a ground query Q on a program P , if Q is finitely-recursive on P , then $RW(Q, P)$ is finitely-ground.

Thus, the herein presented technique paves the way for the evaluation of finitely recursive queries by the grounder of existing ASP systems, such as DLV and Smodels [15], simply by applying a light-weight rewriting on the (non-ground) input program.

The remainder of the paper is structured as follows. Section 2 motivates our work by means of few significant examples; for the sake of completeness, in Section 3 we report some needed preliminaries; Section 4 illustrates our adaptation of the magic-sets rewriting technique to the class of positive finitary programs; in Section 5 we present a number of theoretical results of the rewritten programs; eventually, Section 6 draws our conclusions and depicts the future work.

¹ Note that, in general, the class of finitely recursive Horn programs strictly includes those Horn programs whose evaluation terminates under standard Prolog SLD resolution.

2 Motivation

Positive finitely-recursive programs might be seen as the simplest subclass of finitary programs. As finitary programs, they enjoy all nice properties of this class. In particular, consistency checking is decidable as well as reasoning with ground queries (while reasoning is semi-decidable in case of non ground queries). Unfortunately, even if a program P is finitely-recursive, it is not suited for the bottom-up evaluation for two main reasons:

1. A bottom-up evaluation of a finitely-recursive program would generate some new terms at each iteration, thus iterating for ever.

Example 1. Consider the following program P_2 defining the natural numbers:

$$\begin{aligned} & nat(0). \\ & nat(s(X)) :- nat(X). \end{aligned} \tag{2}$$

The program is positive and finitely-recursive, so every ground query (such as for instance $nat(s(s(s(0))))?$) can be answered in a top-down fashion; but its bottom-up evaluation would iterate for ever, as, for any positive integer n , the n -th iteration would derive the new atom $nat(s^n(0))$.

2. Finitely-recursive programs do not enforce the range of an head variable to be restricted by a body occurrence (i.e., bottom-up safety is not required). A bottom-up evaluation of these unsafe rules would cause the derivation of non-ground facts; such a case is not admissible by present grounding algorithms.

Example 2. Consider the following program P_3 , defining the reachability among vertices of a graph:

$$\begin{aligned} & reachable(X, X). \\ & reachable(X, Y) :- reachable(X, Z), arc(Z, Y). \end{aligned} \tag{3}$$

The program is positive and finitely-recursive; thus, any ground query can be computed top-down, while a bottom-up evaluation is unfortunately unfeasible: the first iteration would generate $\{reachable(X, X)\}$, representing an infinite set of atoms. In this case, $node(X)$ could be added to the body of the first rule, rendering safe the variable X and then making possible the program bottom-up evaluation. But, this is not always the case, as shown in the next example.

Example 3. The following program P_4 defines the comparison operator ‘less than’ between two natural numbers (the function symbol s represents the successor of a natural number):

$$\begin{aligned} & lessThan(X, s(X)). \\ & lessThan(X, s(Y)) :- lessThan(X, Y). \end{aligned} \tag{4}$$

The program is positive and finitely-recursive, thus any ground query can be easily answered top-down. For instance, the query $lessThan(s(0), s(s(0)))?$ results true, whereas the query $lessThan(s(s(0)), s(s(0)))?$ is false. Bottom-up evaluation of this program is unfeasible, since the first iteration would generate the set consisting of an infinite number of atoms having the form $\{lessThan(X, s(X))\}$.

3 Preliminaries

This section reports the formal specification of the ASP language with function symbols, followed by some basics on the magic-sets technique. The subclass of ASP programs herein considered are positive normal logic programs (i.e., disjunction and negation are not allowed).

3.1 Syntax

A *term* is either a *simple term* or a *functional term*.² A *simple term* is either a constant or a variable. If $t_1 \dots t_n$ are terms and f is a function symbol (*functor*) of arity n , then $f(t_1, \dots, t_n)$ is a *functional term*; $t_1 \dots t_n$ are subterms $f(t_1, \dots, t_n)$. The subterm relation is assumed to be reflexive and transitive, that is:

- each term is also a subterm of itself;
- if t_1 is a subterm of t_2 and t_2 is subterm of t_3 then t_1 is also a subterm of t_3 .

Each predicate p has a fixed arity $k \geq 0$. If t_1, \dots, t_k are terms and p is a *predicate* of arity k , then $p(t_1, \dots, t_k)$ is an *atom*. Let A be a set of atoms and p be a predicate. With small abuse of notation we say that $p \in A$ if there is some atom in A with predicate name p .

A *rule* r is of the form:

$$\alpha \text{ :- } \beta_1, \dots, \beta_n. \quad (5)$$

where $\alpha, \beta_1, \dots, \beta_n$ are atoms.

Atom α is called *head* of r , while the conjunction β_1, \dots, β_n , is the *body* of r . We denote the head atom by $H(r)$, and denote by $B(r)$ the set of body atoms. If the body of r is empty (i.e., $n = 0$ and then $B(r) = \emptyset$) we usually omit the “:-” sign; and if it contains no variables, then it is referred to as a *fact*.

An ASP program P is a finite set of rules. The ASP programs considered in this paper are also called *Horn programs*, as negation is forbidden. Horn programs where functional terms are not allowed are generally called *Datalog programs*.

Given a predicate p , a *defining rule* for p is a rule r such that the predicate p occurs in the head atom $H(r)$. If all defining rules of a predicate p are facts, then p is an *EDB predicate*; otherwise p is an *IDB predicate*.³ The set of all facts of P is denoted by $Facts(P)$; the set of instances of all *EDB* predicates is denoted by $EDB(P)$ (note that $EDB(P) \subseteq Facts(P)$). The set of all head atoms in P is denoted by $Heads(P) = \bigcup_{r \in P} H(r)$.

A *query* Q is an *IDB* atom.⁴ As usual, a program (a rule, a term, a query) is said to be *ground* if it contains no variables.

3.2 Semantics

Given a program P , the *Herbrand universe* of P , denoted by U_P , consists of all (ground) terms that can be built combining constants and functors appearing in P . The *Herbrand base* of P , denoted by B_P , is the set of all ground atoms obtainable from the atoms of P by replacing variables with elements from U_P .⁵ A *substitution* for a rule $r \in P$ is a mapping from the set of variables of r to the set U_P of ground terms. A *ground instance* of a rule r is obtained applying a substitution to r . Given a program P the *instantiation (grounding) $grnd(P)$* of P is defined as the set of all ground instances of its rules. Given a ground program P , an *interpretation* I for P is a subset of B_P . An atom a is true w.r.t. I if $a \in I$; it is false otherwise. Given a ground rule r ,

² We will use traditional square-bracketed list constructors as shortcut for the representation of lists by means of nested functional terms (see, for instance, [6]). The usage “à la prolog”, or any different, is only a matter of syntactic sugar.

³ EDB and IDB stand for Extensional Database and Intensional Database, respectively.

⁴ Note that this definition of a query is not as restrictive as it may seem, as one can include appropriate rules in the program for expressing unions of conjunctive queries (and more).

⁵ With no loss of generality, we assume that constants appearing in any query Q always appear in P . Since we focus on query answering, this allows us to restrict to Herbrand universe/base.

we say that r is satisfied w.r.t. I if its head atom is true w.r.t. I or some body atom appearing in $B(r)$ is false w.r.t. I .

Given a ground program P , we say that I is a *model* of P , iff all rules in $grnd(P)$ are satisfied w.r.t. I . A model M of P is an *answer set* of P if there is no model N for P such that $N \subset M$. Each program P considered in this paper (positive and non-disjunctive) has a unique answer set, which is denoted by $AS(P)$.

3.3 The Magic-Sets Technique

The *magic-sets* method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of additional rules, which narrow the computation to what is relevant for answering the query. Intuitively, the goal of the magic-sets method is to use the constants appearing in the query to reduce the size of the instantiation by eliminating ‘a priori’ a number of ground instances of the rules which cannot contribute to the (possible) derivation of the query goal.

This method has originally been defined in [16] for non-disjunctive Datalog (i.e., with no function symbols) queries only. Afterwards, many generalizations have been proposed. In the context of ASP, the generalization to the disjunctive case [17] and to Datalog with (possibly unstratified) negation [18] are worth remembering.

We next provide a brief and informal description of the magic-sets rewriting technique. The reader is referred to [19] for a detailed presentation. The method is structured in four main phases which are informally illustrated below by example, considering the query $path(1, 5)$ on the following program P_6 :

$$\begin{aligned} path(X, Y) &:- edge(X, Y). \\ path(X, Y) &:- edge(X, Z), path(Z, Y). \end{aligned} \tag{6}$$

1. Adornment Step: The key idea is to materialize, by suitable adornments, binding information for *IDB* predicates which would be propagated during a top-down computation. Adornments are strings consisting of the letters b and f , denoting ‘bound’ and ‘free’ respectively, for each argument of an *IDB* predicate. First, adornments are created for query predicates so that an argument occurring in the query is adorned with the letter b if it is a constant, or with the letter f if it is a variable. The query adornments are then used to propagate their information into the body of the rules defining it, simulating a top-down evaluation. It is worth noting that adorning a rule may generate new adorned predicates. Thus, the adornment step is repeated until all adorned predicates have been processed, yielding the *adorned program*.

For simplicity of presentation, we next adopt the “basic” magic-sets method as defined in [16], in which binding information within a rule comes only from the adornment of the head predicate, from *EDB* predicates in the (positive) rule body, and from constants. In other words, an adornment of type ‘ b ’ is induced by a constant, or by a variable occurring either as an argument in a position of type b in the head predicate or in an *EDB* predicate. On the contrary, in the so-called “generalized” magic-sets method [20], bindings may also be generated by *IDB* predicates in rule bodies. In particular, an appropriate Sideways Information Passing Strategy (SIPS) has to be specified for each rule, fixing the body ordering and the way in which bindings are generated. In this respect, the basic method uses a particular, predetermined SIPS for all rules.

Example 4. Adorning the query $path(1, 5)$ generates the adorned predicate $path^{bb}$ since both arguments are bound, and the adorned program P_7 is:

$$\begin{aligned} path^{bb}(X, Y) &:- edge(X, Y). \\ path^{bb}(X, Y) &:- edge(X, Z), path^{bb}(Z, Y). \end{aligned} \tag{7}$$

2. Generation Step: The adorned program is used to generate *magic rules*, which simulate the top-down evaluation scheme and single out the atoms which are relevant for deriving the input query. Let the *magic version* $\text{magic}(p^\alpha(\bar{t}))$ for an adorned atom $p^\alpha(\bar{t})$ be defined as the atom $\text{magic}.p^\alpha(\bar{t}')$, where \bar{t}' is obtained from \bar{t} by eliminating all arguments corresponding to an f label in α , and where $\text{magic}.p^\alpha$ is a new predicate symbol obtained by attaching the prefix ‘*magic.*’ to the predicate symbol p^α . Then, for each adorned atom A in the body of an adorned rule r_a , a magic rule r_m is generated such that (i) the head of r_m consists of $\text{magic}(A)$, and (ii) the body of r_m consists of the magic version of the head atom of r_a , followed by all the (*EDB*) atoms of r_a which can propagate the binding on A .

Example 5. Let us consider the adorned program P_7 . First rule does not produce any magic rule, since it does not contain any adorned predicate in its body. Hence, we only generate the following magic rule:

$$\text{magic_path}^{bb}(Z, Y) \text{ :- } \text{magic_path}^{bb}(X, Y), \text{edge}(X, Z). \quad (8)$$

3. Modification Step: The adorned rules are subsequently modified by including magic atoms generated in Step 2 in the rule bodies, which limit the range of the head variables avoiding the inference of facts which cannot contribute to deriving the query. The resulting rules are called *modified rules*. Each adorned rule r_a is modified as follows. Let H be the head atom of r_a . Then, atom $\text{magic}(H)$ is inserted in the body of the rule, and the adornments of all non-magic predicates are stripped off.

Example 6. Rules resulting from the modification of the adorned program P_7 are:

$$\begin{aligned} \text{path}(X, Y) & \text{ :- } \text{magic_path}^{bb}(X, Y), \text{edge}(X, Y). \\ \text{path}(X, Y) & \text{ :- } \text{magic_path}^{bb}(X, Y), \text{edge}(X, Z), \text{path}(Z, Y). \end{aligned} \quad (9)$$

4. Processing of the Query: Let the query goal be the adorned *IDB* atom g^α . Then, the magic version (also called *magic seed*) is produced as $\text{magic}(g^\alpha)$ (see step 2 above). For instance, in our example we generate $\text{magic_path}^{bb}(1, 5)$.

The complete rewritten program consists of the magic, modified, and query rules.

Example 7. The complete rewriting of our example program is:

$$\begin{aligned} & \text{magic_path}^{bb}(1, 5). \\ \text{magic_path}^{bb}(Z, Y) & \text{ :- } \text{magic_path}^{bb}(X, Y), \text{edge}(X, Z). \\ \text{path}(X, Y) & \text{ :- } \text{magic_path}^{bb}(X, Y), \text{edge}(X, Y). \\ \text{path}(X, Y) & \text{ :- } \text{magic_path}^{bb}(X, Y), \text{edge}(X, Z), \text{path}(Z, Y). \end{aligned} \quad (10)$$

In this rewriting, $\text{magic_path}^{bb}(X, Y)$ represents the start- and end-nodes of all potential sub-paths of paths from 1 to 5. Therefore, when answering the query, only these sub-paths will be actually considered in bottom-up computations.

4 Rewriting Finitely-Recursive Queries

In this Section the definition of finitely-recursive queries is given first; then, a suitable adaptation of the *magic-sets* rewriting technique for programs with functions is presented, that allows a bottom-up evaluation of such queries over positive programs.

4.1 Finitely-Recursive Queries

Given a ground program P , we say that a ground atom a depends on another ground atom b if there is a rule $r \in \text{grnd}(P)$ such that a is the head of r and either $b \in B(r)$ or c depends on b for some atom $c \in B(r)$. A finitely-recursive program [11] is such that every ground query on it depends only on a finite set of other ground atoms.

Interestingly, even for some non finitely-recursive program, there may exist a subset of all possible ground queries for which the above mentioned property holds.

Example 8. Consider the following program P_{11} :

$$\begin{aligned}
 & \text{lessThan}(X, s(X)). \\
 & \text{lessThan}(X, s(Y)) \text{ :- } \text{lessThan}(X, Y). \\
 & q(f(f(0))). \\
 & q(X) \text{ :- } q(f(X)). \\
 & r(X) \text{ :- } \text{lessThan}(X, Y), q(X).
 \end{aligned} \tag{11}$$

If the whole program is considered, then it is not finitely-recursive. In fact, atoms such as $q(c)$ or $r(c)$ (where c is a whatsoever constant) depend on an infinite number of other atoms. In the former case, because of a never terminating recursion; in the latter, because of both the local variable Y and the dependency on q atoms. Nevertheless, query atoms having as predicate lessThan continue to depend only on a finite set of other atoms.

More formally, queries depending on a finite set of other atoms can be defined as follows.

Definition 1. Given a program P and a ground query Q over P we say that Q is *finitely-recursive* on P if and only if Q depends only on a finite set A of other ground atoms in $\text{grnd}(P)$.

In other words, we can say that a ground query Q on a program P is finitely-recursive if and only if the “relevant subprogram” of P for Q ($R_{(P,Q)}$) is positive and finitely-recursive. For instance, considering the program P_{11} of Example 8, all atoms like $\text{lessThan}(c_1, c_2)$, where c_1 and c_2 are constant values, are examples of finitely-recursive queries.

4.2 Rewriting Algorithm

We restrict our attention to finitely-recursive queries that cannot be safely bottom-up evaluated (it makes sense to think of a standard bottom-up evaluation for others). In such a case, some steps of the magic-sets technique reported in Section 3.3 can be significantly simplified. In particular, the adornment phase is no longer needed, given that the *IDB* predicates involved in the query evaluation would have a completely bound adornment. Indeed:

- the query is ground, so it would have a completely bound adornment;
- all rules involved in a top-down evaluation of the query cannot have local variables (i.e. variables appearing only in the body of the rule) since the relevant subprogram is supposed to be finitely-recursive.⁶ Thus, starting from a ground query, a complete bound adornment from the head to all the *IDB* predicates of the body would be propagated.

In the generation step, it is no longer necessary to include any other atom in the body of the generated magic rule, apart from the magic version of the head atom. Again, this is due to the absence of local variables, so that all the needed bindings are provided through the magic version of the head atom.

⁶ Indeed, function symbols make the universe infinite, and local variables in a rule would make its head depend on an infinite number of other ground atoms. Local variables could obviously appear in a function-free program, but this could be easily bottom-up evaluated.

```

Input: a program  $P$  and a finitely-recursive query  $Q = g(\bar{c})?$  on  $P$ 
Output: the rewritten program  $RW(Q, P)$ .
Main Vars:  $S$ : stack of predicates to rewrite;
                $modifiedRules(Q, P), magicRules(Q, P)$ : set of rules;
                $Done$ : set of predicates;
begin
1.  $modifiedRules(Q, P) := \emptyset$ ;  $Done := \emptyset$ 
2.  $magicRules(Q, P) := \{magic\_g(\bar{c})\}$ ;
3.  $S.push(g)$ ;
4. while  $S \neq \emptyset$  do
5.    $u := S.pop()$ ;
6.   if  $u \notin Done$  then
7.      $Done := Done \cup \{u\}$ ;
8.     for each  $r \in P$  :  $r$  is a defining rule for  $u$  do
9.       if  $B(r) \neq \emptyset$  or  $Vars(r) \neq \emptyset$  then
10.        // let  $r$  be  $u(\bar{c}) :- v_1(\bar{c}_1), \dots, v_n(\bar{c}_n)$ .
11.         $modifiedRules(Q, P) := modifiedRules(Q, P) \cup$ 
12.           $\{u(\bar{c}) :- magic\_u(\bar{c}), v_1(\bar{c}_1), \dots, v_n(\bar{c}_n)\}$ ;
13.        for each  $v_i : v_i \in B(r)$  and  $v_i \in IDB(P)$  do
14.           $magicRules(Q, P) := magicRules(Q, P) \cup$ 
15.             $\{magic\_v_i(\bar{c}_i) :- magic\_u(\bar{c})\}$ ;
16.         $S.push(v_i)$ ;
17.      end for
18.    else
19.       $modifiedRules(Q, P) := modifiedRules(Q, P) \cup r$ 
20.    end if
21.  end for
22. end while
23.  $RW(Q, P) := magicRules(Q, P) \cup modifiedRules(Q, P)$ ;
24. return  $RW(Q, P)$ ;
end.

```

Fig. 1. Magic Sets rewriting algorithm for finitely-recursive queries

The algorithm MS_{FR} in Figure 1 implements the magic-sets method for finitely-recursive queries. Starting from a program P and a finitely-recursive ground query Q on P , the algorithm outputs a program $RW(Q, P)$ consisting of a set of *modified* and *magic* rules (denoted by $modifiedRules$ and $magicRules$, respectively), which are generated on a rule-by-rule basis. To this end, it exploits a stack S for storing all predicates that are still to be used for propagating the query binding. At first, the set of magic rules is initialized with the magic version of the query (line 2) and the predicate of the query atom is pushed on S (line 3). At each step, an element u is removed from S (line 5). If the predicate u has not been already considered (the auxiliary variable $Done$ is used to check this) (line 6), all the rules defining u are processed one-at-a-time (lines 8 – 18). Given one of such rules r , if the body of r is not empty or there is at least a variable in r (line 9, where $Vars(r)$ is used to denote the set of variables occurring in the rule r), then a modified version of r is created (line 10) and a set of magic rules are generated (one for each *IDB* atom in the body) (lines 11 – 14). Moreover, every *IDB* predicate appearing in the body of r is pushed on the stack S (line 13). In case r is a fact, i.e. its body is empty and there are no

variables, it is added to the *modifiedRules* set as it is (line 16). Finally, once all the predicates involved in the query evaluation have been processed (S is thus empty), the algorithm outputs the program $RW(Q, P)$ obtained as the union of all modified rules and generated magic rules (lines 21 – 22).

Some rewriting example are reported next.

Example 9. Consider the finitely-recursive query $Q = nat(s(s(0)))?$ on the program P_2 of Example 1. For this example, we will depict, step by step, the execution performed by the MS_{FR} algorithm. After the initialization of variables, the algorithm (lines 1–2) generates the first magic rule deriving from the query:

$$magic_nat(s(s(0))). \quad (12)$$

The predicate nat is then pushed onto the stack S (line 3) and the first iteration of the cycle (line 4) starts. The predicate nat is extracted from S and is marked as already done (lines 5 – 7). In this case, this is the only predicate to be considered. All defining rules for nat are then processed (lines 8 – 18). The first rule defining nat is a fact ($nat(0)$): both conditions of line 9 are false, so the rule is added to the *ModifiedRules* set (line 16) unchanged. The second rule defining nat is a recursive rule. First of all, the modified rule:

$$nat(s(X)) :- magic_nat(s(X)), nat(X). \quad (13)$$

is added to the *ModifiedRules* set (line 10). Then, the following magic rule for the nat atom occurring in the body is generated, and the nat predicate is pushed onto the stack S (lines 11 – 14):

$$magic_nat(X) :- magic_nat(s(X)). \quad (14)$$

Then, the second iteration starts but it immediately ends, as the predicate extracted from the stack S is the already considered predicate nat . Finally, S is found empty, and there are no further iterations needed. The algorithm outputs the following complete rewritten program $P_{15}=RW(Q, P)$:

$$\begin{aligned} &magic_nat(s(s(0))). \\ &magic_nat(X) :- magic_nat(s(X)). \\ &nat(0). \\ &nat(s(X)) :- magic_nat(s(X)), nat(X). \end{aligned} \quad (15)$$

Example 10. Considering the query $Q = lessThan(s(s(0)), s(0))?$ on the program P_4 of Example 3, the algorithm outputs the following rewritten program $P_{16}=RW(Q, P)$:

$$\begin{aligned} &magic_lessThan(s(s(0)), s(0)). \\ &magic_lessThan(X, Y) :- magic_lessThan(X, s(Y)). \\ &lessThan(X, s(X)) :- magic_lessThan(X, s(X)). \\ &lessThan(X, s(Y)) :- magic_lessThan(X, s(Y)), lessThan(X, Y). \end{aligned} \quad (16)$$

As we can see in the following, most of the common queries on programs for manipulating lists are finitely-recursive and then can be rewritten using the MS_{FR} algorithm.

Example 11. Let us consider the following program P_{17} , that works on lists:

$$\begin{aligned} &reverse(L, R) :- sup_reverse(L, [], R). \\ &sup_reverse([], R, R). \\ &sup_reverse([X|T_1], L, R) :- sup_reverse(T_1, [X|L], R). \end{aligned} \quad (17)$$

For the query $Q = \text{reverse}([a, b, c, d], [d, c, b, a])?$, the rewritten program $P_{18} = RW(Q, P)$ is:

$$\begin{aligned}
& \text{magic_reverse}([a, b, c, d], [d, c, b, a]). \\
& \text{magic_sup_reverse}(L, [], R) \text{ :- } \text{magic_reverse}(L, R). \\
& \text{magic_sup_reverse}(T_1, [X|L], R) \text{ :- } \text{magic_sup_reverse}([X|T_1], L, R). \\
& \text{reverse}(L, R) \text{ :- } \text{magic_reverse}(L, R), \text{sup_reverse}(L, [], R). \\
& \text{sup_reverse}([], R, R) \text{ :- } \text{magic_sup_reverse}([], R, R). \\
& \text{sup_reverse}([X|T_1], L, R) \text{ :- } \text{magic_sup_reverse}([X|T_1], L, R), \\
& \quad \text{sup_reverse}(T_1, [X|L], R)
\end{aligned} \tag{18}$$

5 Properties of Rewritten Programs

Let $RW(Q, P)$ denote the output of the MS_{FR} algorithm, having as input a program P and a finitely-recursive query Q . Next, we are going to prove some relevant results about the $RW(Q, P)$ program. First of all we give a query equivalence property.

Theorem 1. Given a ground query Q on a program P , if Q is finitely-recursive on P , then $P \models Q$ if and only if $RW(Q, P) \models Q$.

Proof. (Sketch) Query equivalence has already been proved for the ‘standard’ magic-sets technique (see e.g. [19]). The algorithm presented in Section 4.2 differs from the standard one for some aspects but all of them have no consequences on the correctness of the transformation. We next recall the differences against the standard magic-sets technique, and illustrate why the introduced changes do not affect the query equivalence result:

1. Adornment is not performed because the structure of finitely-recursive queries implies that only a completely bound adornment would be derived. Anyway, all the *IDB* predicates involved in the top-down query evaluation are correctly identified and processed, likewise the standard algorithm does. Both rules modification and magic rules generation are performed considering all processed *IDB* predicate as having an implicit completely bound adornment.
2. Only the magic version of the head atom is included in the body of each generated magic rule. According to the standard technique, all *EDB* atoms which can propagate the binding on variables occurring in the currently processed atom should be added to the body. In case of a finitely-recursive query, all variables occurring in the body of a rule necessarily occurs also in its head (no local variables are admitted). Hence, we know ‘a priori’ that no further atom is needed.
3. The MS_{FR} algorithm acts on a rule-by-rule basis, instead of performing the different phases on the entire program. This approach, adopted also in [17] and [18], allows us to improve efficiency, as each rule of the original program is processed just once. The resulting rewritten program is not affected by this change.

Next, we are going to prove a result about the efficiency of the rewriting algorithm. To this aim, we need to introduce the definition of what we mean for size of a program.

Definition 2. Let P be a (non-ground) logic program. The *size* $\|t\|$ of a term t is 1, if the term is a constant or a variable; the size of a functional term $f(t_1, \dots, t_n)$ is defined as $1 + \|t_1\| + \dots + \|t_n\|$. The *size of an atom* is given by the sum of the size of its terms; if the atom has arity 0, size is 1. The *size of the program* P , denoted by $\|P\|$, is the sum of the sizes of all atoms occurring in P . It is worth noting that multiple occurrences of the same atom in P are taken into account.

Example 12. The program P_{17} in Example 11 has size $\|P\| = 18$.

Theorem 2. Given a finitely-recursive query Q on a program P , the size of $RW(Q, P)$ is linear in the size of P and Q . In symbols: $\|RW(Q, P)\| = O(\|P\| + \|Q\|)$. Also, $MS_{FR}(Q, P)$ outputs $RW(Q, P)$ in time linear in the size of P .

Proof. (Sketch) The size of the rewritten program may increase w.r.t. the original one, because new atoms (“magic” atoms) and new rules (“magic” rules) are introduced. Nevertheless, a magic atom $magic.p(\bar{t})$ might be added only as the counterpart of an atom $p(\bar{t})$ already existing in P ; it is worth noting that they have exactly the same terms (and thus, the same size). Hence, it is enough to consider the number of atoms in $RW(Q, P)$ w.r.t. the number of atoms in P .

The program $RW(Q, P)$ is obtained as the union of the two sets of rules $modifiedRules(Q, P)$ and $magicRules(Q, P)$. In the worst case, the number of atoms in the first set is given by the number of atoms in P plus as many atoms as the number of rules in P (at most one magic atom is added for each rule in P). Thus, $\|modifiedRules(Q, P)\|$ is definitely $O(\|P\|)$.

Let us consider now the $magicRules(Q, P)$ program. At first, the magic version of the query is added. Then, for each *IDB* atom occurring in the body of a rule in P , at most one magic rule with exactly two atoms is generated. Then, in the worst case, the number of atoms in $magicRules(Q, P)$ is not greater than $2 \cdot \|P\| + \|Q\|$.

As for the time complexity of MS_{FR} , note that each rule $r \in P$ is processed exactly once. Processing r requires to scan, and process once, all the atoms appearing in it.

Thus, from the considerations above, the statements immediately follows.⁷

5.1 Relationships with Finitely-Ground Programs

We point out next the relationship between finitely-recursive queries and finitely-ground programs [6]. For the sake of clarity, we roughly recall here some key concepts therein introduced;⁸ for formal definitions, more details, and examples, we refer the reader to the aforementioned paper.

Some graphs are defined, namely *Dependency Graph* and *Component Graph*, in order to properly split a given program P into modules, each one corresponding to a strongly connected component (SCC)⁹ of the dependency graph. An ordering relation is then defined among modules/components: a *component ordering* γ for P is a total ordering such that the instantiation performed one module at a time according to γ is equivalent to the one obtained by processing the whole program altogether. An new operator (Φ) is defined, that properly instantiates a given module of P ; such operator exploits the instantiation of previous modules (according to a component ordering) in order to generate a subset of the theoretical instantiation, by adding only those ground rules whose heads have a chance to be true in some answer set. By properly composing consecutive applications of the least fixpoint of Φ to the modules of P according to a component ordering $\gamma = \langle C_1, \dots, C_n \rangle$, a sequence $S_0 \dots S_n$ of sets of ground rules is generated, such that many useless rules w.r.t. answer sets computation are dropped. The *intelligent instantiation* P^γ of P for γ is the last element S_n of the sequence above. A program P is *finitely-ground (FG)* if P^γ is finite, for every component ordering γ for P .

We are now ready to introduce an important property of the rewritten programs resulting from the magic-set technique presented in this paper.

⁷ Indeed, the final size of the rewritten program actually depends also on the query, since rules that are not relevant for answering are not considered at all; nevertheless, we perform here a worst-case analysis (which is faced when all original rules are relevant).

⁸ It is worth noting that the class of programs therein considered allow also default negation in the bodies and disjunction in the heads.

⁹ We recall here that a strongly connected component of a directed graph is a maximal subset S of the vertices, such that each vertex in S is reachable from all other vertices in S .

Theorem 3. Given a ground query Q on a program P , if Q is finitely-recursive on P , then $RW(Q, P)$ is finitely-ground.

Proof. (Sketch) Let P_{magic} be the set of predicates defined by rules in $magicRules(Q, P)$ and let P_{mod} be the set of predicates defined by $modifiedRules(Q, P)$. We observe that $P_{magic} \cap P_{mod} = \emptyset$ and every component ordering γ for $RW(Q, P)$ (Definition 4 in [6]) will be such that: if $p \in P_{magic}$ and p belongs to the component C_i , then C_i will precede every component C_j featuring a predicate q s.t. $q \in P_{mod}$. This means that, in the modular bottom-up evaluation performed by the intelligent instantiation (Definition 8 in [6]), all rules in $magicRules(Q, P)$ will precede rules in $modifiedRules(Q, P)$.

We know that Q is finitely-recursive, i.e. it depends only on a finite set of other ground atoms in $grnd(P)$. But rules in $magicRules(Q, P)$ are properly built in order to exclusively derive atoms which the query Q depends on. So, starting from the ground query atom Q , each element of the sequence S_i in the intelligent instantiation definition is necessarily finite for modules of $magicRules(Q, P)$. But, magic atoms give binding to all variables occurring in the head of rules in $modifiedRules(Q, P)$, restricting their domain of possible values, so elements S_i will be finite also for all modules of $modifiedRules(Q, P)$. This is enough to prove the statement.

This last result is relevant because it implies that all nice properties of finitely-ground programs hold for rewritten finitely-recursive queries too. This includes, in particular, bottom-up computability of the answer set and hence full decidability of reasoning.

Example 13. If we consider the rewritten program P_{15} of Example 9, we can observe that its resulting intelligent instantiation is finite:

$$\begin{aligned}
& magic_nat(s(s(0))). \\
& magic_nat(s(0)) \text{ :- } magic_nat(s(s(0))). \\
& magic_nat(0) \text{ :- } magic_nat(s(0)). \\
& nat(0). \\
& nat(s(s(0))) \text{ :- } magic_nat(s(s(0))), nat(s(0)). \\
& nat(s(0)) \text{ :- } magic_nat(s(0)), nat(0).
\end{aligned} \tag{19}$$

The above ground program has the following unique finite answer set: $\{magic_nat(s(s(0))), magic_nat(s(0)), magic_nat(0), nat(0), nat(s(0)), nat(s(s(0)))\}$. The answer to the query Q is then 'yes'.

Example 14. Let us now consider the rewritten program (P_{16}) of Example 10. Also this program, differently from the originating P program, can be safely bottom-up evaluated. Its intelligent instantiation is finite and results to be:

$$\begin{aligned}
& magic_lessThan(s(s(0)), s(0)). \\
& magic_lessThan(s(s(0)), 0) \text{ :- } magic_lessThan(s(s(0)), s(0)).
\end{aligned} \tag{20}$$

The above ground program has the unique finite answer set $\{magic_lessThan(s(s(0)), s(0)), magic_lessThan(s(s(0)), 0)\}$. Thus, the answer to the query Q is 'no'.

6 Conclusions

We presented an adaptation of the magic-sets technique that allows query answering over positive finitary normal programs also by means of standard bottom-up techniques. This allows us to

enrich the collection of logic programs with function symbols for which ground query answering can be performed by all current ASP solvers.

Future work will focus on overcoming the limitation of considering only ground queries, by identifying the minimal set of variables required to be bound in order to preserve decidability. Next step will then deal with negation.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080
3. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
4. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: Proceedings of the 16th International Conference on Logic Programming (ICLP'99), Las Cruces, New Mexico, USA, The MIT Press (November 1999) 23–37
5. 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
6. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Proceedings of the 24th International Conference on Logic Programming (ICLP 2008). Volume 5366 of *Lecture Notes in Computer Science.*, Udine, Italy, Springer (December 2008) 407–424
7. Baselice, S., Bonatti, P.A., Criscuolo, G.: On Finitely Recursive Programs. In: 23rd International Conference on Logic Programming (ICLP-2007). Volume 4670 of *LNCS.*, Springer (2007) 89–103
8. Simkus, M., Eiter, T.: FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In: Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR2007). Volume 4790 of *Lecture Notes in Computer Science.*, Springer (2007) 514–530
9. Lin, F., Wang, Y.: Answer Set Programming with Functions. In: Proceedings of Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR2008), Sydney, Australia, AAAI Press (September 2008) 454–465
10. Syrjänen, T.: Omega-restricted logic programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, Vienna, Austria, Springer-Verlag (September 2001)
11. Bonatti, P.A.: Reasoning with infinite stable models. *Artificial Intelligence* **156**(1) (2004) 75–111
12. 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) (July 2006) 499–562
13. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: `DLV-Complex` homepage (since 2008) <http://www.mat.unical.it/dlv-complex>.
14. Marano, M., Ianni, G., Ricca, F.: A Magic Set Implementation for Disjunctive Logic Programming with Function Symbols. Submitted to CILC 2009.
15. 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 (April 2000) Online at <http://xxx.lanl.gov/abs/cs/0003033v1>.
16. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Massachusetts (1986) 1–15

17. Cumbo, C., Faber, W., Greco, G., Leone, N.: Enhancing the magic-set method for disjunctive datalog programs. In: Proceedings of the the 20th International Conference on Logic Programming – ICLP'04. Volume 3132 of Lecture Notes in Computer Science. (2004) 371–385
18. Faber, W., Greco, G., Leone, N.: Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences* **73**(4) (2007) 584–609
19. Ullman, J.D.: *Principles of Database and Knowledge Base Systems. Volume 2.* Computer Science Press (1989)
20. Beeri, C., Ramakrishnan, R.: On the Power of Magic. In: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '87), New York, NY, USA, ACM (1987) 269–284

Extending Soft Arc Consistency Algorithms to Non-Invertible Semirings, with an Application to Multi-Criteria Problems*

Stefano Bistarelli^{1,2,3}, Fabio Gadducci⁴, Emma Rollon⁵ and
Francesco Santini^{2,3}

¹ Dipartimento di Matematica e Informatica Università di Perugia, Italy
`bista@dipmat.unipg.it`

² Dipartimento di Scienze, Università “G. d’Annunzio” di Chieti-Pescara, Italy
`bista@sci.unich.it`, `santini@sci.unich.it`

³ Istituto di Informatica e Telematica (IIT-CNR) Pisa, Italy
`stefano.bistarelli@iit.cnr.it`, `francesco.santini@iit.cnr.it`

⁴ Dipartimento di Informatica, Università di Pisa, Italy
`gadducci@di.unipi.it`

⁵ Department of Software, Technical University of Catalonia, Barcelona, Spain
`erollon@lsi.upc.edu`

Abstract. We extend arc consistency algorithms proposed in the literature in order to deal with not necessarily invertible semirings. As a result, consistency algorithms can now be used as a preprocessing procedure in soft CSPs defined over a larger class of semirings: either partially ordered, or with non idempotent \times , or not closed \div operator, or constructed as cartesian product or Hoare power sets of any semiring (which can be used for multicriteria CSPs). To reach this objective, we first show that each semiring can be transformed into a new one where the $+$ is instantiated with the Least Common Divisor (LCD) between the elements of the semiring. The LCD value corresponds to the amount we can “safely move” from the binary constraint to the unary one in the consistency algorithm (when \times is not idempotent). We then propose an arc consistency algorithm which takes advantage of this operator.

1 Introduction and Motivations

Constraint propagation embeds any reasoning which consists in explicitly forbidding values or combinations of values for some variables of a problem because a given subset of its constraints cannot be satisfied otherwise. A very important mean to accomplish this task is represented by local consistency algorithms.

Local consistency [1] is an essential component of a constraint solver: a local property with an enforcing algorithm, often polynomial time, that transforms a

* Research partially supported by the EU FP6-IST IP 16004 SENSORIA and by the Spanish Min. of Science and Innovation through the projects TIN2006-15387-C03-0

classical constraint network into an equivalent network that satisfies the property. If this equivalent network is empty, then the initial problem/network is obviously inconsistent, allowing to detect some inconsistencies very efficiently.

The idea of the semiring-based framework [1, 2] was to further extend the classical constraint notion, and to do it with a formalism that could encompass most of the existing extensions, with the aim to provide a single environment where properties could be proven once and for all, and inherited by all the instances. At the technical level, this was done by adding to the usual notion of *Constraint Satisfaction Problem* [1] (CSP) the concept of a structure representing the levels of satisfiability of the constraints. Such a structure is a set with two operations (see Sec. 2 for further details): one (written $+$) is used to generate an ordering over the levels, while the other one (\times) is used to define how two levels can be combined and which level is the result of such combination. Because of the properties required on such operations, this structure is similar to a semiring (see Sec. 2): from here the terminology of “semiring-based soft constraint” [2–4], that is, constraints with several levels of satisfiability, and whose levels are (totally or partially) ordered according to the semiring structure. In general, problems defined according to the semiring-based framework are called *Soft Constraint Satisfaction Problems* (soft CSPs or SCSPs).

In the literature, many local consistency algorithms have been proposed for soft CSPs: while classical consistency algorithms [5] aim at reducing the size of constraint problems, soft consistency algorithms work by explicating the error information that is originally implicit in the problem. The most recent ones exploit *invertible* semirings (see Sec. 2), providing, under suitable conditions, an operator \div that is the inverse of \times , i.e., such that $(a \div b) \times b = a$ (see two alternative proposals in [6] and [7, 8], respectively).

In this paper we aim at generalizing the previous consistency algorithms to not necessarily invertible semirings. In particular, we first show how to distill from a semiring a novel one, such that its $+$ operator corresponds to the *Least Common Divisor (LCD)* operator (see Sec. 3) of the elements in the semiring preference set. We then show, and this represents the practical application of the first theoretical outcome, how to apply the derived semiring inside soft arc consistency algorithms in order to extend their use to not necessarily invertible semiring structures, thus leading to a further generalization of soft arc consistency techniques. In words, the value represented by the LCD corresponds to the amount we can “safely move” from the binary constraint to the unary one in the consistency algorithm. Summing up, this paper extends the use of consistency algorithms beyond the limits imposed by the proposals in [6, 7]. As a practical and very important example, the new consistency algorithms can be applied to multicriteria problems, where the Hoare Powerdomain of the Cartesian product of multiple semirings represents the set of partially ordered solutions [9].

The paper is organized as follows: Sec. 2 summarizes the background notions about semirings and soft constraints. Sec. 3 shows how to assemble the new LCD operator by transforming a semiring, while Sec. 4 proposes its use inside a local

consistency algorithm for soft CSPs. Then, Sec. 5 shows the algorithm execution over a multicriteria problem. The final remarks are provided in Sec. 6.

2 Preliminaries

Semirings provide an algebraic framework for the specification of a general class of combinatorial optimization problems. Outcomes associated to variable instantiations are modeled as elements of a set A , equipped with a sum and a product operator. These operators are used for combining constraints: the intuition is that the sum operator induces a partial order $a \leq b$, meaning that b is a better outcome than a ; whilst the product operator denotes the aggregation of outcomes coming from different soft constraints.

2.1 The Algebra of Semirings

This section reviews the main concepts, adopting the terminology used in [3, 6]. A (commutative) semiring is a five-tuple $\mathcal{K} = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that A is a set, $\mathbf{1}, \mathbf{0} \in A$, and $+, \times : A \times A \rightarrow A$ are binary operators making the triples $\langle A, +, \mathbf{0} \rangle$ and $\langle A, \times, \mathbf{1} \rangle$ commutative monoids (semigroups with identity), satisfying distributivity ($\forall a, b, c \in A. a \times (b + c) = (a \times b) + (a \times c)$) and with $\mathbf{0}$ as annihilator element for \times ($\forall a \in A. a \times \mathbf{0} = \mathbf{0}$). A semiring is *absorptive* if additionally $\mathbf{1}$ is an annihilator element for $+$ ($\forall a \in A. a + \mathbf{1} = \mathbf{1}$).¹

Let \mathcal{K} be an absorptive semiring. Then, the operator $+$ of \mathcal{K} is idempotent. As a consequence, the relation $\langle A, \leq \rangle$ defined as $a \leq b$ if $a + b = b$ is a partial order and, moreover, $\mathbf{1}$ is its top element. If additionally \mathcal{K} is also *idempotent* (that is, the product operator \times is idempotent), then the partial order is actually a *lattice*, since $a \times b$ corresponds to the greatest lower bound of a and b . For the rest of the paper, we fix the absorbing semiring $\mathcal{K} = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.

An absorptive semiring is *invertible* if whenever $a \leq b$, there exists an element $c \in A$ such that $b \times c = a$ or, in other words, if the set $Div(a, b) = \{c \mid b \times c = a\}$ is not empty. It is *uniquely* invertible if that set is actually a singleton whenever $b \neq \mathbf{0}$. All classical soft constraint instances (i.e. *Classical CSPs*, *Fuzzy CSPs*, *Probabilistic CSPs* and *Weighted CSPs*) are indeed invertible, and uniquely so.

A *division* operator \div is a kind of weak inverse of \times , such that $a \div b$ returns an element chosen from $Div(a, b)$. There are currently two alternatives in the literature: the choice in [6] favours the maximum of such elements (whenever it exists), the choice in [7] favours the minimum (whenever it exists). While the latter is better computationally, the former encompasses more semiring instances.² All of our results in the following sections can be applied for any choice of \div .

¹ The absorptiveness property is equivalent to $\forall a, b \in A. a + (a \times b) = a$, that is, any element $a \times b$ is actually “absorbed” by either a or b .

² For example, *complete* semirings, i.e., those which are closed wrt. infinite sums, and the distributivity law holds also for an infinite number of summands. See [6] for a throughout comparison between the two alternatives.

Example 1. Let us consider the *weighted* semiring $\mathcal{K}_w = \langle \mathbb{N} \cup \{\infty\}, \min, +, 0, \infty \rangle$. The $+$ and \times operators are \min and $+$ with their usual meaning over the naturals. The ∞ value is handled in the usual way (i.e. $\min\{\infty, a\} = a$ and $\infty + a = a$ for all a). This semiring is widely used to model and solve a variety of combinatorial optimization problems [10]. Note that the induced order is total and corresponds to the inverse of the usual order among naturals (e.g. $9 \leq 6$ because $\min\{9, 6\} = 6$). Note as well that \mathcal{K}_w is uniquely invertible. The division corresponds to the usual subtraction over the naturals (e.g. $a \div b = a - b$). The only case that deserves special attention is $\infty \div \infty$, because any value of the semiring satisfies the division condition. In [6] and [7] it is defined as 0 and ∞ , respectively.

Consider now semiring $\mathcal{K}_b = \langle \mathbb{N}^+ \cup \{\infty\}, \min, \times, \infty, 1 \rangle$, where $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. The $+$ and \times operators are now \min , \times with the usual meaning over the naturals. This semiring is a slightly modified version of a semiring proposed in [11] in order to deal with bipolar preferences. As before, the semiring is totally ordered. However, it is not invertible: for example, even if $9 \leq 6$, clearly there is no $a \in \mathbb{N}^+ \cup \{\infty\}$ such that $6 \times a = 9$. Intuitively, $\mathbb{N}^+ \cup \{\infty\}$ is not closed under the arithmetic division, the obvious inverse of the arithmetic multiplication.

2.2 Soft Constraints Based on Semirings

The aim of this section is to briefly recall the main concepts of the semiring-based approach to the soft CSPs framework.

Definition 1 (constraints). *Let $\mathcal{K} = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ be an absorptive semiring; let V be an ordered set of variables; and let D be a finite domain of interpretation for V . Then, a constraint $(V \rightarrow D) \rightarrow A$ is a function associating a value in A to each assignment $\eta : V \rightarrow D$ of the variables.*

We then define $C = \eta \rightarrow A$ as the set of all constraints that can be built starting from \mathcal{K} , V and D . The application of a constraint function $c : (V \rightarrow D) \rightarrow A$ to a mapping $\eta : V \rightarrow D$, is noted $c\eta$. Note that even if a constraint involves all the variables in V , it can depend on the assignment of a finite subset of them, called its support. For instance, a binary constraint c with $\text{supp}(c) = \{x, y\}$ is a function $c : (V \rightarrow D) \rightarrow A$ which depends only on the assignment of variables $\{x, y\} \subseteq V$. The support corresponds to the classical notion of scope of a constraint. We often refer to a constraint with support I as c_I . Moreover, an assignment over a support I of size k is concisely represented by a tuple t in D^k and we often write $c_I(t)$ instead of $c_I\eta$.

We now present the extension of the basic operations (namely, combination, division and projection) to soft constraints.

Definition 2 (combination and division). *The combination operator $\otimes : C \times C \rightarrow C$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ for any two constraints c_1, c_2 .*

The division operator $\oplus : C \times C \rightarrow C$ is defined as $(c_1 \oplus c_2)\eta = c_1\eta \div c_2\eta$ for any two constraints c_1, c_2 such that $c_1 \sqsubseteq c_2$ (i.e., such that $c_1\eta \leq c_2\eta$ for all η).

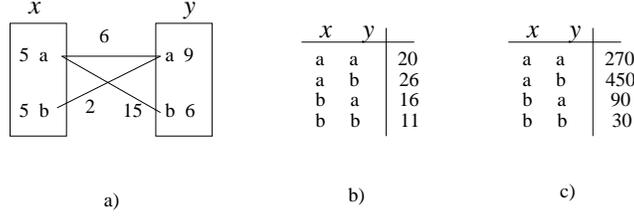


Fig. 1. A soft CSP and the combination of its constraints wrt. semirings \mathcal{K}_w and \mathcal{K}_b .

Informally, performing the \otimes or the \oplus 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.

Definition 3 (projection). Let $c \in C$ be a constraint and $v \in V$ a variable. The projection of c over $V - \{v\}$ is the constraint c' such that $c'\eta = \sum_{d \in D} c\eta[v := d]$.

We denote such projection as $c \downarrow_{(V - \{v\})}$. The projection operator is inductively extended to a set of variables $I \subseteq V$ by $c \downarrow_{(V - I)} = c \downarrow_{(V - \{v\})} \downarrow_{(V - \{I - \{v\}\})}$. Informally, projecting means eliminating variables from the support.

Definition 4 (soft CSPs). A soft constraint satisfaction problem is a pair $\langle C, Y \rangle$, where C is a set of constraints and $Y \subseteq V$.

The set Y contains the variables of interest for the constraint set C .

Definition 5 (solutions). The solution of a soft CSP $P = \langle C, Y \rangle$ is defined as the constraint $Sol(P) = (\otimes C) \downarrow_Y$.

The solution of a soft CSP is obtained by combining all constraints, and then projecting over the variables in Y . In this way we get the constraint with support (not greater than) Y which is “induced” by the entire soft CSP.

A tightly related notion, one which is quite important in combinatorial optimization problems, is the best level of consistency.

Definition 6. The best level of consistency of a soft CSP problem $P = \langle C, Y \rangle$ is defined as the constraint $blevel(P) = (\otimes C) \downarrow_{\emptyset}$.

Example 2. Figure 1.a) shows a soft CSP with variables $V = \{x, y\}$ and values $D = \{a, b\}$. For the purpose of the example, all the variable in V are of interest. The problem has two unary soft constraints c_x, c_y and one binary constraint c_{xy} . Each rectangle represents a variable and contains its domain values. Besides each domain value there is a semiring value given by the corresponding unary

constraint (for instance, c_y gives value 9 to any labeling in which variable y takes value a). Binary constraints are represented by labeled links between pairs of values. For instance, c_{xy} gives value 15 to the labeling in which variable x and y take values a and b . If a link is missing, its value is the unit $\mathbf{1}$ of the semiring.

Figure 1.b) shows the combination of all constrains (i.e. $c_x \otimes c_y \otimes c_{xy}$) assuming the weighted semiring K_w . Each table entry is the sum of the three corresponding semiring values. In this case, the best level of solution is 11 the minimum over all the entries. Figure 1.c) shows the combination of all constrains assuming the bipolar semiring K_b . It is different wrt. the previous case, because semiring values are now multiplied. As before, the best level of solution is the minimum over all the entries which, in this case, is 30.

2.3 Local Consistency

Soft local consistencies are properties over soft CSPs that a given instance may or may not satisfy. For the sake of simplicity, in this paper we restrict ourselves to the simplest consistencies. However, all the notions and ideas presented in subsequent sections can be easily generalized to more sophisticated ones.

In the sequel, we assume that soft CSPs are binary (i.e., no constraint has arity greater than 2). We also assume the existence of a unary constraint c_x for every variable x , and of a zero-arity constraint (i.e. a constant), noted c_0 : if such constraints are not defined, we consider dummy ones: $c_x(a) = \mathbf{1}$ for all $a \in D$ and $c_0 = \mathbf{1}$. One effect of local consistency is to detect and remove unfeasible values. For that purpose we define the current domain variable x as $D_x \subseteq D$.

Definition 7. Let $P = \langle C, Y \rangle$ be a binary soft CSP.

- *Node consistency (NC).* Value $a \in D_x$ is NC if $c_0 \times c_x(a) > \mathbf{0}$. Variable x is NC if $\sum_{a \in D_x} c_x(a) = \mathbf{1}$. P is NC if every variable is NC.
- *Arc consistency (AC).* Value $a \in D_x$ is AC wrt. $c_{x,y}$ if $\sum_{b \in D_y} c_{x,y}(a, b) = \mathbf{1}$. Variable x is AC if all its values are AC wrt. every binary constraint such that x is in its support. P is AC if every variable is AC and NC.

Each property should come with an enforcing algorithm that transforms the problem into an equivalent one that satisfies the property. Enforcing algorithms are based on the concept of local consistency rule.

Definition 8 (local consistency rule [6]). Let c' and c be two constraints such that $\text{supp}(c') \subset \text{supp}(c)$ and let $Y = \text{supp}(c) \setminus \text{supp}(c')$. A local consistency rule involving c' and c , denoted $CR(c', c)$, consists of two steps

- Aggregate to c' information induced by c

$$c' := c' \otimes (c \downarrow_Y)$$

- Compensate in c the information aggregated to c' in the previous step

$$c := c \oplus (c \downarrow_x).$$

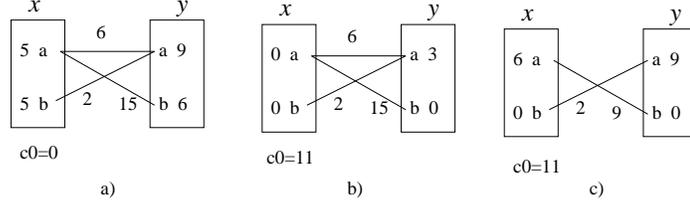


Fig. 2. Three equivalent soft CSP instances (the semiring being \mathcal{K}_w). The first one is arc and node inconsistent. The second one is node consistent and arc inconsistent. The third one is node and arc consistent.

It is important to note that local consistency rules are only defined for soft CSPs with invertible semirings, because \oplus is only defined for them. The fundamental property of the above local consistency rule is that it does not change the solution of soft CSPs defined on invertible semirings [6].

Example 3. Figure 2.a) shows the soft CSP we have chosen as our running example. Assuming the \mathcal{K}_w semiring, the problem is not NC. It can be made NC by applying the local consistency rule twice, to c_0, c_x and c_0, c_y , which increases c_0 and decreases the unary constraints. The resulting equivalent soft CSP is depicted in Figure 2.b). However, it is not AC. It can be made arc consistent by applying the local consistency rule to $c_x, c_{x,y}$. The resulting equivalent soft CSP is depicted in Figure 2.c). This problem is more explicit than the original one. In particular, the zero-arity constraint c_0 contains the best level of consistency.

3 Defining an LCD-based Semiring Transformation

Many absorptive semirings are not invertible: see e.g. [6] for some references. Nevertheless, we would like to apply some consistency rules also to these cases.

The aim of the section is to show how to distill from a semiring \mathcal{K} a new semiring $LCD_{\preceq}(\mathcal{K})$ such that the latter is invertible. In this section we show the construction of the structure $LCD_{\preceq}(\mathcal{K})$, prove it to be a semiring, and state a conservativity result, namely, that \mathcal{K} and $LCD_{\preceq}(\mathcal{K})$ coincide, if \mathcal{K} is invertible.

Technically, we exploit a notion of *least common divisor*: for any two elements of a semiring, we consider the set of common divisors, and we assume the existence of at least one minimal element among such divisors: intuitively, these are the “worst” elements according to the ordering on the semiring.

Definition 9 (common divisor). *Let \mathcal{K} be an absorptive semiring and let $a, b \in A$. The divisors of a is the set $Div(a) = \{c \mid \exists d. c \times d = a\}$; the common divisors of a and b is the set $CD(a, b) = Div(a) \cap Div(b)$.*

Clearly, the set $CD(a, b)$ is never empty, since it contains at least $\mathbf{1}$. Moreover, its elements form a partial order, from which we may identify the minimal ones.

Definition 10 (least common divisor). Let \mathcal{K} be an absorptive semiring and $a, b \in A$. The least common divisors of a, b are the minimal of the set $CD(a, b)$, i.e., the elements of the set $LCD(a, b) = \{c \in CD(a, b) \mid \nexists d \in CD(a, b). d < c\}$.

We say that \mathcal{K} admits bound least common divisors is the set $LCD(a, b)$ is finite and not empty for any $a, b \in A$.

We can characterize the choice of a least common divisor if a linearization of the partial order associated to the semiring is found.

Definition 11 (linearized least common divisor). Let \mathcal{K} be an absorptive semiring. A linearization of the partial order \leq_K associated to \mathcal{K} is a total order \preceq_K which is compatible with \leq_K , i.e., such that $\forall a, b \in A. a \leq_K b \implies a \preceq_K b$.

If \mathcal{K} has bound least common divisors, $LCD_{\preceq_K}(a, b)$ denotes the minimum of the set $LCD(a, b)$ according to \preceq_K for any $a, b \in A$.

In other words, the $LCD_{\preceq}(a, b)$ operator returns a single element of $LCD(a, b)$, according to a given ordering. In the following, for any finite set $E \subseteq A$, $LCD(E)$ and $LCD_{\preceq}(E)$ denote the obvious associative extensions.

We can finally prove the main result of this section, i.e., the existence of a semiring whose sum operator is based on LCD_{\preceq} . This new semiring is used by the consistency algorithm in Sec. 4.

Theorem 1 (LCD-based semiring). Let \mathcal{K} be an absorptive semiring with bound least common divisors, and let \preceq be one of its linearizations. Then, the tuple $LCD_{\preceq}(\mathcal{K}) = \langle A, LCD_{\preceq}, \times, \mathbf{0}, \mathbf{1} \rangle$ is an absorptive and invertible semiring.

Proof. As for proving that $LCD_{\preceq}(\mathcal{K})$ is an absorptive semiring, it just suffices to check out each single property, noting that they hold since some choices are forced by the linearization. As an example, $LCD(a, \mathbf{0})$ coincides with $Div(a)$, and the linearization tells us that $LCD_{\preceq}(a, \mathbf{0}) = a$. Similar considerations hold for associativity and distributivity. As for invertibility, by definition $a \leq_{LCD_{\preceq}} b$ implies that $LCD_{\preceq}(a, b) = b$. \square

As a final result, we need to check out what is the outcome of the application of the LCD_{\preceq} construction to an already invertible semiring.

Proposition 1 (conservativity). Let \mathcal{K} be an absorptive and invertible semiring. Then, \mathcal{K} and $LCD_{\preceq_K}(\mathcal{K})$ are the same semiring for any linearization \preceq_K .

The proof is immediate: note that if \mathcal{K} is invertible, then by definition $a + b \in CD(a, b)$, and the element surely is the greatest lower bound of the set.

Example 4. Recall the non invertible semiring $\mathcal{K}_b = \langle \mathbb{N}^+ \cup \{\infty\}, \min, \times, \infty, 1 \rangle$. Since this semiring is totally ordered, we ignore linearizations. By definition, the least common divisor of a and b is $LCD(a, b) = \max\{c \mid \exists d. c \times d = a, \exists e. c \times e = b\}$ which corresponds with the arithmetic notion of greatest common divisor. The LCD transformation of \mathcal{K}_b is $LCD(\mathcal{K}_b) = \langle \mathbb{N}^+ \cup \{\infty\}, LCD(a, b), \times, \infty, 1 \rangle$. The partial order induced by $LCD(\mathcal{K}_b)$ is $a \leq b$ if $LCD(a, b) = b$, i.e., if b is a divisor of a . Finally, the division operator of $LCD(\mathcal{K}_b)$ is the usual arithmetic division. However, it is closed because it is restricted to divisible pairs of numbers.

4 LCD-based Local Consistency

This section generalizes local consistencies to soft CSPs with non invertible semirings. The leading intuition is to replace the $+$ operator by the LCD_{\succeq} operator also in the original definition of the local consistency rule. The value represented by the found LCD corresponds to the amount we can “safely move” from the binary constraint to the unary one in order to enforce consistency. This very intuitive idea can be applied also over non invertible semirings.

Definition 12. Let $P = \langle C, Y \rangle$ be a binary soft CSP defined over semiring \mathcal{K} , and let LCD_{\succeq} be a linearized least common divisor operator.

- *LCD Node consistency (LCD-NC).* Value $a \in D_x$ is LCD-NC if $c_0 \times c_x(a) > \mathbf{0}$. Variable x is LCD-NC if $LCD(\{c_x(a) \mid a \in D_x\}) = \mathbf{1}$. P is LCD-NC if every variable is LCD-NC.
- *LCD Arc consistency (LCD-AC).* Value $a \in D_x$ is LCD-AC wrt. $c_{x,y}$ if $LCD(\{c_{x,y}(a, b) \mid b \in D_y\}) = \mathbf{1}$. Variable x is LCD-AC if all its values are LCD-AC wrt. every binary constraint such that x is in its support. P is LCD-AC if every variable is LCD-AC and LCD-NC.

Definition 13 (LCD local consistency rule). Let $P = \langle C, Y \rangle$ be a soft CSP defined over semiring \mathcal{K} . A LCD local consistency rule involving c' and c , noted $LCD-CR(c', c)$, is like a classical local consistency rule where all operations (i.e., combination, projection and division) are done using the $LCD(\mathcal{K})$ semiring.

Theorem 2. Let P be a soft CSP. The application of the LCD local consistency rules of Definition 13 does not change its solution.

Example 5. Figure 3.a) shows our running example. Assuming the \mathcal{K}_b semiring, the problem is not LCD-NC. It can be made LCD-NC by applying the LCD consistency rule twice, to c_0, c_x and c_0, c_y . The resulting equivalent soft CSP is depicted in Figure 2.b). Note that it is LCD-NC but not NC. It is not LCD-AC and can be made so by applying the LCD consistency rule twice, first to $c_x, c_{x,y}$ (producing Figure 3.c)) and then to $c_y, c_{x,y}$ (producing Figure 3.d). The resulting problem is not LCD-NC (due to variable y). It can be made LCD-NC and LCD-AC by applying the LCD consistency rule to c_0, c_y , resulting the problem in Figure 3.e). This problem is more explicit than the original one. In particular, the zero-arity constraint c_0 contains the best level of consistency.

The previous LCD node and arc consistency properties can be enforced by applying appropriated LCD consistency rules. Figure 4 shows a LCD-AC enforcing algorithm. It is based on the AC algorithm of [10]. For simplicity, we assume that no empty domain is produced. It uses two auxiliary functions: $PruneVar(x)$ prunes not LCD-NC values in D_x and returns *true* if the domain is changed; $LCD-CR(c', c)$ iteratively applies the LCD consistency rule to c' and c until reaching a fixed point. Note that if the original semiring \mathcal{K} is invertible, $LCD-CR(c', c)$ iterate only once. The main procedure $AC()$ uses a queue Q containing those variables that may not be LCD-AC. Q should be initialized with all the variables to check at least once their consistency.

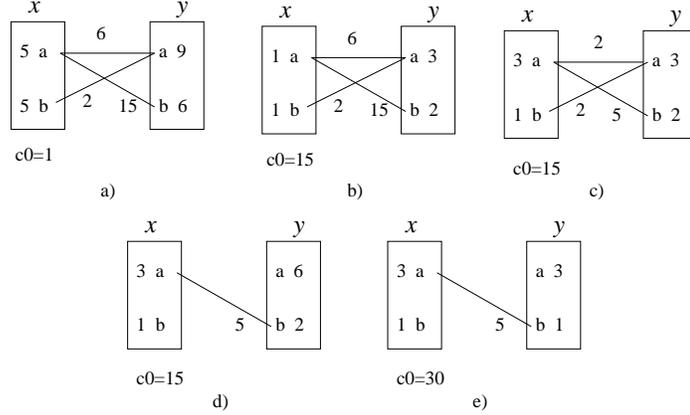


Fig. 3. Five equivalent soft CSP instances (the semiring being \mathcal{K}_b). The first one is LCD arc and node inconsistent. The second one is LCD node consistent and LCD arc inconsistent. The last one is LCD node and arc consistent.

5 Dealing with Multi-Objective Optimization

A *multi-criteria CSP* (MC-CSP) is a soft CSP composed by a family of p soft CSPs $\{P_i\}_{i=1}^p$. Each soft CSP P_i , which is called a criterion, is defined over a semiring $\mathcal{K}_i = \langle A_i, +_i, \times_i, \mathbf{0}_i, \mathbf{1}_i \rangle$. As shown in [9], a MC-CSP problem is defined over a semiring \mathcal{K}_{mo} obtained via the so-called Hoare powerset of the cartesian product of the components. Before defining \mathcal{K}_{mo} we review some basic concepts.

Let $\mathbf{A} = A_1 \times \dots \times A_p$ be the set of all vectors with p components. Given two values $\mathbf{a}, \mathbf{b} \in \mathbf{A}$, let \times and $+$ be the pointwise combination and addition of its elements (i.e., $\mathbf{a} \times \mathbf{b} = (a_1 \times_1 b_1, \dots, a_p \times_p b_p)$ and $\mathbf{a} + \mathbf{b} = (a_1 +_1 b_1, \dots, a_p +_p b_p)$), respectively. The comparison among vectors, denoted by \leq , is defined as $\mathbf{a} \leq \mathbf{b}$ iff $\mathbf{a} + \mathbf{b} = \mathbf{b}$. If $\mathbf{a} \leq \mathbf{b}$, we say that \mathbf{b} is *better* or *dominates* \mathbf{a} .

Finally, let S be a set of vectors: we define its set of *non-dominated elements* $\|S\|$ as $\{\mathbf{u} \in S \mid \nexists \mathbf{v} \in S, \mathbf{u} < \mathbf{v}\}$

Lemma 1. Let $\{\langle A_i, +_i, \times_i, \mathbf{0}_i, \mathbf{1}_i \rangle\}_{i=1}^p$ be a family of semirings. Then, also the structure $\mathcal{K}_{mo} = \langle A_{mo}, +_{mo}, \times_{mo}, \mathbf{0}_{mo}, \mathbf{1}_{mo} \rangle$ is a semiring, for

- $A_{mo} = \{S \in \mathbf{A} \mid S = \|S\|\}$
- $S \times_{mo} T = \|\{\mathbf{u} \times \mathbf{v} \mid \mathbf{u} \in S, \mathbf{v} \in T\}\|$
- $S +_{mo} T = \|\{S \cup T\}\|$
- $\mathbf{0}_{mo} = \{(\mathbf{0}_1, \dots, \mathbf{0}_p)\}$
- $\mathbf{1}_{mo} = \{(\mathbf{1}_1, \dots, \mathbf{1}_p)\}$

If each semiring of the family is absorptive, then also \mathcal{K}_{mo} is so.

The best level of consistency of a soft CSP problem P induced by semiring \mathcal{K}_{mo} is the maximal set of non-dominated vectors, each one being the valuation

```

function PruneVar( $x$ )
1.  $change := false$ ;
2. for each  $a \in D_x$  do
3.   if  $c_x(a) \times c_0 = \mathbf{0}$  then
4.      $D_x := D_x - \{a\}$ ;
5.      $change := true$ ;
6. return  $change$ ;
procedure LCD-AC( $P, V$ )
1.  $Q := V$ ;
2. while ( $Q \neq \emptyset$ ) do
3.    $y := pop(Q)$ ;
4.   for each  $c_{x,y} \in C$  do LCD-CR( $c_x, c_{xy}$ );
5.   for each  $x \in V$  do LCD-CR( $c_0, c_x$ );
6.   for each  $x \in V$  do
7.     if PruneVar( $x$ ) then  $Q := Q \cup \{x\}$ ;

```

Fig. 4. Algorithm LCD-AC.

associated with one complete assignment. In the multi-objective literature, this set is called *efficient frontier* of P , denoted as $\mathcal{E}(P)$.

In general, the semiring \mathcal{K}_{mo} is not invertible, even if each semiring of the family is so. However, thanks to the LCD transformation, we can apply the local consistencies described in the previous section as follows.

Consider a multi-criteria soft CSP $P = \langle C, con \rangle$ composed by two soft CSPs, both defined on the weighted semiring \mathcal{K}_w . For our purposes, $con = V = \{x, y\}$ with domain values $D_x = \{a, b, c\}$ and $D_y = \{a, b\}$. The set of constraints C is composed by two unary constraints $c_x(x)$ and $c_y(y)$, and one binary constraint $c_{xy}(x, y)$. We can express these constraints extensionally with the tables below

x	
a	$\{(2, 1)\} + \{(0, 2), (1, 0)\} + \{(0, 1), (1, 0)\}$
b	$\{(0, 2), (1, 0)\}$
c	$\{(2, 1)\} + \{(0, 2), (1, 0)\}$

y	
a	$\{(0, 0)\}$
b	$\{(0, 0)\}$

x	y	
a	a	$\{(0, 0)\}$
a	b	$\{(0, 0)\}$
b	a	$\{(2, 1)\} + \{(0, 2), (3, 0)\}$
b	b	$\{(2, 1)\} + \{(3, 0)\}$
c	a	$\{(2, 1)\} + \{(0, 2), (1, 0)\}$
c	b	$\{(2, 1)\} + \{(1, 1)\} + \{(0, 2), (1, 0)\} + \{(0, 6), (2, 0)\}$

where the valuation associated to each possible assignment of its variables is expressed as a decomposition of divisors that cannot be further decomposed. Moreover, we have the constraint $c_0 = \{(0, 0)\}$.

First, let us see if the problem is node consistent (NC). The only variable that may not be NC is x . Operator $LCD(c_x(a), c_x(b), c_x(c))$ returns $\{\{(0, 2), (1, 0)\}\}$, which is the only common divisor among $c_x(a)$, $c_x(b)$ and $c_x(c)$. To make variable x NC, this valuation must be combined with the current c_0 and divided from $c_x(a)$, $c_x(b)$ and $c_x(c)$. The unary constraint c_x is as follows,

x	
a	$\{(2, 1)\} + \{(0, 1), (1, 0)\}$
b	$\{(0, 0)\}$
c	$\{(2, 1)\}$

and constraint $c_0 = \{(0, 0)\} + \{(0, 2), (1, 0)\} = \{(0, 2), (1, 0)\}$.

Let us see if the problem is arc consistent. We have to verify that both variables x and y are AC. Consider variable x . Domain value $a \in D_x$ is AC because $LCD(c_{xy}(a, a), c_{xy}(a, b)) = \{\{(0, 0)\}\}$. Domain value $b \in D_x$ is not AC because $LCD(c_{xy}(b, a), c_{xy}(b, b)) = \{\{(2, 1)\}\}$. According to the AC enforcement algorithm, we combine divisor $\{(2, 1)\}$ with $c_x(b)$ and divide $c_{xy}(ba)$ and $c_{xy}(bb)$ by it. Similarly, domain value $c \in D_x$ is not AC because $LCD(c_{xy}(c, a), c_{xy}(c, b)) = \{\{(2, 1)\} + \{(0, 2), (1, 0)\}\}$. Again, it may be added to $c_x(c)$ and divided from $c_{xy}(ca)$ and $c_{xy}(cb)$. After these changes, the constraint tables are

x	
a	$\{(2, 1)\} + \{(0, 1), (1, 0)\}$
b	$\{(2, 1)\}$
c	$\{(2, 1)\}$

x	y	
a	a	$\{(0, 0)\}$
a	b	$\{(0, 0)\}$
b	a	$\{(0, 2), (3, 0)\}$
b	b	$\{(3, 0)\}$
c	a	$\{(0, 0)\}$
c	b	$\{(1, 1)\} + \{(0, 6), (2, 0)\}$

Since we have modified the valuations in c_x , we have to revise if variable x is still NC. It is easy to see that there exists one common divisor among $c_x(a)$, $c_x(b)$ and $c_x(c)$, that is, $\{(2, 1)\}$. Then, we must divide c_x by $\{(2, 1)\}$ and combine this valuation with the current c_0 . Now, constraint c_x is

x	
a	$\{(0, 1), (1, 0)\}$
b	$\{(0, 0)\}$
c	$\{(0, 0)\}$

The current c_0 is $\{(2, 1)\} + \{(0, 2), (1, 0)\} = \{(2, 3), (3, 1)\}$.

Now, consider variable y : $LCD(c_{xy}(aa), c_{xy}(ba), c_{xy}(ca)) = \{(0, 0)\}$, which means that no common divisor exists, except for trivial divisor $\{(0, 0)\}$. As a consequence, variable y is already AC. Since both the variables in the problem are AC, also the problem is so.

Finally, consider the variable ordering $\{y, x\}$. Since the valuation of each of the domain values of variable y in c_y is $\{(0, 0)\}$, the problem is also directional arc consistent (DAC) wrt. that order. It can be verified that also with the other direction, that is, wrt. variable ordering $\{x, y\}$, the problem is still DAC.

6 Conclusions and Further Works

We presented a technique for transforming any semiring into a novel one, whose sum operator corresponds to the *LCD* of the set of preferences. This new semiring can be cast inside local consistency algorithms and allows to extend their use to problems dealing with not invertible semirings. A noticeable application case is represented by multicriteria soft CSPs, where the (Hoare Powerdomain of the) Cartesian product of semirings represents the set of partially ordered solutions.

In the future we plan to extend existent libraries on crisp constraints in order to deal also with propagation for soft constraints, as proposed in this paper. Moreover, we would like to develop approximated algorithms following the ideas in [12], and focusing on multi-criteria problems.

References

1. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier (2006)
2. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. *Journal of ACM* **44**(2) (1997) 201–236
3. Bistarelli, S.: Semirings for Soft Constraint Solving and Programming. Volume 2962 of LNCS. Springer (2004)
4. Bistarelli, S., Rossi, F.: Semiring-based soft constraints. In Degano, P., Nicola, R.D., Meseguer, J., eds.: Concurrency, Graphs and Models. Volume 5065 of LNCS., Springer (2008) 155–173
5. Mackworth, A.K.: Consistency in networks of relations. *Artificial Intelligence* **8**(1) (1977) 99–118
6. Bistarelli, S., Gadducci, F.: Enhancing constraints manipulation in semiring-based formalisms. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: ECAI'06. Volume 141 of Frontiers in Artif. Intell. and Applications., IOS Press (2006) 63–67
7. Cooper, M., Schiex, T.: Arc consistency for soft constraints. *Artificial Intelligence* **154**(1–2) (2004) 199–227
8. Cooper, M.: High-order consistency in valued constraint satisfaction. *Constraints* **10**(3) (2005) 283–305
9. Bistarelli, S., Gadducci, F., Larrosa, J., Rollon, E.: A soft approach to multi-objective optimization. In de la Banda, M.G., Pontelli, E., eds.: ICLP'08. Volume 5366 of LNCS., Springer (2008) 764–768

10. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence* **159**(1-2) (2004) 1–26
11. Bistarelli, S., Pini, M.S., Rossi, F., Venable, K.B.: Bipolar preference problems: Framework, properties and solving techniques. In Azevedo, F., Barahona, P., Fages, F., Rossi, F., eds.: *CSCLP'06*. Volume 4651 of LNCS., Springer (2007) 78–92
12. Larrosa, J., Meseguer, P.: Exploiting the use of DAC in MAX-CSP. In Freuder, E.C., ed.: *CP'96*. Volume 1118 of LNCS., Springer (1996) 308–322

Constraint Based Languages for Biological Reactions

Stefano Bistarelli^{1,2,3}, Marco Bottalico³ and Francesco Santini^{2,3}

¹ Dipartimento di Matematica Informatica, Università di Perugia, Italy
bista@dipmat.unipg.it

² Istituto di Informatica e Telematica (CNR), Pisa, Italy
[stefano.bistarelli, francesco.santini]@iit.cnr.it

³ Università G. d'Annunzio, Pescara, Italy
[bista, bottalico, santini]@sci.unich.it

Abstract. In this paper, we study the modelization of biochemical reaction by using concurrent constraint programming idioms. In particular we will consider the stochastic concurrent constraint programming (sCCP), the Hybrid concurrent constraint programming languages (Hcc) and the Biochemical Abstract Machines (BIOCHAM).

Keywords: Biochemical Reactions, Blood coagulation, Concurrent Constraint Programming (CCP), stochastic Concurrent Constraint Programming (sCCP), Hybrid concurrent constraint programming languages (Hcc), Biochemical Abstract machines (BIOCHAM).

1 Introduction

Systems biology is a science integrating experimental activity and mathematical modeling. They study the dynamical behaviors of biological systems. While current genome project provide a huge amount of data on genes or proteins, lots of research is still necessary to understand how the different parts of a biological system interact in order to perform complex biological functions. Mathematical and computational techniques are central in this approach to biology, as they provide the capability of formally describing living systems and studying their properties.

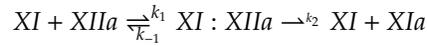
A variety of formalisms for modeling biological systems has been proposed in the literature. In [13], the author distinguishes three basic approaches: discrete, stochastic, continuous, additionally we have various combinations between them. Discrete models are based on discrete variables and discrete state changes; continuous models are based on differential equations that typically model biochemical reactions; finally in the stochastic the probabilities may appear explicitly in random variables and random numbers, or implicitly like in kinetics laws. In the latest approach we have a simplified representation of processes, an integration of stochastic noise in order to get more realistic models. The need to capture both discrete and continuous phenomena, motivates the study of dynamical systems [6].

The goal of this paper is to show different kinds of languages to model biochemical reactions in order to compare and to use their appropriate features in different ways.

2 Background on Biochemical reactions and Blood Coagulation

In this paper, we want to examine the Biochemical Reactions; they are chemical reactions involving mainly proteins. In a cell, there are many different proteins, hence, the number of reactions that can take place, can be very high. All the interactions that take place in a cell, can be used to create a diagram, obtaining a biochemical reaction network.

We will examine in the following, one of the thirteen enzymatic reaction of the blood coagulation [14], in the generic form, through the Michaelis-Menten kinetics:



where XI is the enzyme (E) that binds substrate (S) = $XIIa$, to form an enzyme-substrate complex (ES) = $XI : XIIa$. After we have the formation of product (P) = XIa and the release of the unchanged enzyme (E) = XI , ready for a new reaction (see Appendix A for further details).

We are interesting at the Blood Coagulation [14]. It is part of an important host defense mechanism termed hemostasis (the cessation of blood loss from a damaged vessel). Blood clotting is a very delicately balanced system; when hemostatic functions fail, hemorrhage or thromboembolic phenomena results. The chemical reactions that constitute all the process, can be see as a decomposition of many kinds of enzymatic reactions, involved reactants, products, enzymes, substrates, stoichiometric coefficients, proteins, inhibitors and chemical accelerators.

3 Concurrent Constraint Programming

The Concurrent Constraint (cc) programming paradigm [10] concerns the behaviour of a set of concurrent agents with a shared store, which is a conjunction of constraints. Each computation step possibly adds new constraints to the store. Thus information is monotonically added to the store until all agents have evolved. The final store is a refinement of the initial one and it is the result of the computation. The concurrent agents do not communicate directly with each other, but only through the shared store, by either checking if it entails a given constraint (*ask* operation) or adding a new constraint to it (*tell* operation).

For the CCP's syntax, we have that 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, and x is a tuple of variables. Each procedure is defined (at most) once, thus nondeterminism is expressed via the $+$ combinator only. We also assume that, in $p(x) :: A$, we have $vars(A) \subseteq x$, where $vars(A)$ is the set of all variables occurring free in agent A . In a program $P = F.A$, A is the initial agent, to be executed in the context of the set of declarations F . This corresponds to the language considered in [10] which allows only guarded nondeterminism.

4 Stochastic Concurrent Constraint Programming

sCCP [2] is obtained by adding a stochastic duration to the instruction interacting with the constraint store C , i.e. *ask* and *tell*. The most important feature added in the sCCP is the continuous random variable T , associated with each instruction. It represents the time needed to perform the corresponding operations in the store. T is exponentially distributed, and its probability density function is $f(\tau) = \lambda e^{-\lambda\tau}$ where λ is a positive real number (rate of the exponential random variable) representing the expected frequency per unit of time. The duration of an ask or a tell can depend on the state of the store at the moment of the execution.

The main difference of sCCP with classical cc, is the presence of two different actions with temporal duration, *ask* and *tell*, identified by a rate function λ : $tell_\lambda(c)$ and $ask_\lambda(c)$, following the probability law. It means that the reaction occurs in a stochastic time T , $f(\tau) = \lambda e^{-\lambda\tau}$ whose mean is $1/\lambda$; i.e. $tell_\infty$ is an instantaneous execution while $tell_0$ never occurs.

Other functions are the same that in CCP, except for the variables, that in CCP are rigid, in the sense that, whenever they are instantiated, they keep that value forever. Time-varying variables (called stream variables) can be easily modeled in sCCP as growing lists with a unbounded tail: $X = [a_1, \dots, a_n|T]$. When the quantity changes, we simply need to add the new value, say b , at the end of the list by replacing the old tail variable with a list containing b and a new tail variable: $T = [b|T']$. When we need to know the current value of the variable X , we need to extract from the list, the value immediately preceding the unbounded tail. The stream variables are denoted with assignment $\$=$.

We model the biochemical equation [5], in sCCP, with the following recursively defined method:

$$\begin{aligned} & \text{react}(XIIa, XIa, K_M, V_0) : - \\ & \quad \text{ask}_{r_{MM}(K_M, V_0, XIIa)}(XIIa > 0). \\ & (tell_\infty(XIIa \$= XIIa - 1) || tell_\infty(XIa \$= XIa + 1)). \\ & \quad \text{react}(XIIa, XIa, K_M, V_0) \end{aligned}$$

Where the rate λ of the *ask*, is computed by the Michaelis-Menten kinetics: $r_{MM}(K_M, V_0, XIIa) = \frac{V_0 \times XIIa}{XIIa + K_M}$. Roughly the program inserts in the store the current value for the variables $K_M, V_0, XIIa$; it checks the value of the factor $XIIa$, then, with an immediate effect, it updates the values for the factors $XIIa$ (reagent) and XIa (product) with the new values. Subsequently it executes a new instance of the program.

5 Hybrid cc

Hybrid concurrent constraint programming languages (Hybrid cc [8, 1]) is a powerful framework for modeling, analyzing and simulating hybrid systems, i.e., systems that exhibit both discrete and continuous change. It is an extension

of Timed Default cc [12] over continuous time. One the major difficulty in the original cc framework is that cc programs can detect only the presence of information, not the absence [13]. Default cc extends cc by a negative *ask* combinator (*if a else A*) which imposes the constraint *a* at the program *A*.

```

e = 100, s = 10, es = 0, p = 0, k1 = 1, km1 = 0.1, k2 = 0.01,
{ hence cont(e), cont(s), cont(es), cont(p),
  if (e > 0 && s > 0.1) then
  { e = (((km1+k2) * prev(es)) - (k1 * (prev(e) * prev(s))))),
    s = ((km1 * prev(es)) - (k1 * (prev(e) * prev(s)))) ,
    es = ((k1 * (prev(e) * prev(s))) - ((km1+k2) * prev(es))),
    p = (k2 * prev(es))
  } else e' = 0, s' = 0, es' = 0, p' = 0 },
sample(e, s, es, p)

```

We can observe that in the first row, we have the initial conditions with the quantity of Enzyme (100), Substrate (10), Enzyme-Substrate (0), Product (0), and the rate of the three reactions: $k_1 = 1$ for the first one, $k_{-1} = 0.1$ for the second one and $k_2 = 0.01$ for the third reaction. With the syntax $cont(e, s, es, p)$ we assert that the rates of e, s, es, p are continuous. Subsequently we control the current values of e and s then we can start the reaction; the $\&\&$ and $=$ operators, has the usual meaning of “and” and “equal”. With the derivation rules in the Appendix A (Par.9), we can easily obtain the quantity of e, s, es, p in the next time instant. If the “if condition” doesn’t hold, we obtain the previous amount of factors.

6 Biochemical Abstract Machine

Biochemical Abstract Machines (BIOCHAM [9]) is a software environment for modeling complex cell processes, making simulations (i.e. in silico experiments), formalizing the biological properties of the system know from real experiments, checking them and using them as specification when refining a model. BIOCHAM [3] is based on two aspects: the analysis and simulation of boolean, kinetic and stochastic model and the simulation of biological proprieties in temporal logic. For kinetics model, BIOCHAM can search for appropriate parameter values in order to reproduce a specific behavior observed in experiments and formalized in temporal logic.

We can use the Michaelis-Menten kinetics to represent the first enzymatic reaction of blood coagulation. To explain the language used to model the reaction in the BIOCHAM [4] language, we can translate the syntax in the following way:

```

(k1*[E]*[S], km1*[ES]) for E + S <=> ES.
k2*[ES] for ES => E + P.
parameter(k1, 1).
parameter(km1, 0.1).
parameter(k2, 0.01).
present(E, 100).

```

present(S, 10) .
 absent(ES) .
 absent(P) .

There are two different syntax operator, used to model the different kinds of reaction: \rightleftharpoons and \Rightarrow . The first one model the reversible reaction, involved in the *ES* formation, this reaction can be reversible. The second one model the irreversible reaction which produce the *P* factor. The *for* operator show us for which substances, the reaction is performed: the first *for* is for the $E + S \xrightleftharpoons[k_{-1}]{k_1} E : S$ reaction, the second one for the $E : S \xrightarrow{k_2} E + P$ reaction.

7 Difference and similarity among four languages

In the following table, we encode the main differences and the similarities of the previous three languages to describe biological reactions and their simulation methods.

Language name	Time intervals	Use ODE's	Simulation program plotting	Formal specification
<i>sCCP</i>	continuous	no	no	constraints
<i>Hcc</i>	both	yes	no	constraints
<i>BIOCHAM</i>	continuous	yes	yes, GNU Plot	logic programming

8 Related and future works

Most important features are represented in [11, 7]. In [11] the authors suggests to model biomolecular process i.e. protein networks, by using the pi-Calculus, while in [7] is shown that there are two formalisms for mathematically describing the time behavior of a spatially homogeneous chemical systems: the deterministic approach and the stochastic one. The first regards the time evolution as a continuous and predictable process which is governed by a set of ordinary differential equations (the "reaction-rate equations"), while the seconds regards the time evolution as a kind of random-walk process which is governed by a single differential-difference equation (the "master equation").

From the application point of view, the examined languages allows the biologist to model biological systems in a high-level and declarative way, using different kinds of applications and languages construct that capture directly a variety of biological phenomena.

The aim of many researches in the bioinformatics field is to improve the modeling features of data in order to describe biological behaviors in a more accurate way, such that they can be used in in silico modeling in the drug sperimentations (preclinic) area; we aim to save time and money in the last sperimental phase on humans. Biological reactions are the first step towards a description of cycles cellular-based mechanisms.

9 Appendix A

Under the **Michaelis-Menten** hypotheses [5], the most important equations are: $K_M = \frac{k_{-1}+k_2}{k_1}$ Michaelis constant. It measures the affinity of the enzyme for the substrate: if K_M is small there is a high affinity, and viceversa.

$V_{MAX} = V_0 = k_2[E_0]$. This is the maximum rate, would be achieved when all of the enzyme molecules have substrate bound (Hp1). $[E_0]$ is the starting quantity of enzyme E . k_2 is also called k_{cat} .

$\frac{d[P]}{dt} = \frac{V_{MAX}[S]}{K_M + [S]}$. This final equation, is usually called the “Michaelis-Menten equation”. It shows the speed of the formation of the product.

References

1. Panos J. Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors. *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer, 1995.
2. Luca Bortolussi and Alberto Policriti. Modeling biological systems in stochastic concurrent constraint programming. *Constraints*, 13(1-2):66–90, 2008.
3. Laurence Calzone, François Fages, and Sylvain Soliman. Biocham: an environment for modeling biological systems and formalizing experimental knowledge. *Bioinformatics*, 22(14):1805–1807, 2006.
4. Nathalie Chabrier-Rivier, François Fages, and Sylvain Soliman. The biochemical abstract machine biocham. In *Proc. CMSB*, pages 172–191, 2004.
5. Thomas Delvin. *Textbook of biochemistry with clinical correlations*. McGraw Hill Book co., 2001.
6. Schaft A.J. van der and J.M. Schumacher. *Introduction to Hybrid Dynamical Systems*. Springer-Verlag, 1999.
7. Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. In *The Journal of Physical Chemistry*, pages 2340–2352, 1977.
8. Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Computing with continuous change. *Sci. Comput. Program.*, 30(1-2):3–49, 1998.
9. Patricia M. Hill, editor. *Logic Based Program Synthesis and Transformation, 15th International Symposium, LOPSTR 2005, London, UK, September 7-9, 2005, Revised Selected Papers*, volume 3901 of *Lecture Notes in Computer Science*. Springer, 2006.
10. Ugo Montanari and Francesca Rossi, editors. *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, volume 976 of *Lecture Notes in Computer Science*. Springer, 1995.
11. Aviv Regev, William Silverman, and Ehud Y. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Proc. Pacific Symposium on Biocomputing*, pages 459–470, 2001.
12. Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Timed default concurrent constraint programming. *J. Symb. Comput.*, 22(5/6):475–520, 1996.
13. Peter J. Stuckey, editor. *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings*, volume 2401 of *Lecture Notes in Computer Science*. Springer, 2002.
14. Williams. *Williams Hematology*. McGraw Hill Book co., 2006.

nfn2dlp: A Normal Form Nested Programs Compiler*

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. Normal Form Nested (NFN) programs have recently been introduced in order to allow for enriching the syntax of disjunctive logic programs under the answer sets semantics. In particular, heads of rules can be disjunctions of conjunctions, while bodies can be conjunctions of disjunctions. Different to many other proposals of this kind, NFN programs may contain variables, and a notion of safety has been defined for guaranteeing domain independence. Moreover, previous results show that there is a polynomial translation from NFN programs to standard disjunctive logic programs (DLP).

In this paper we present the tool `nfn2dlp`, a compiler for NFN programs, which implements an efficient translation from safe NFN programs to safe DLP programs. The answer sets of the original NFN program can be obtained from the answer sets of the transformed program (which in turn can be obtained by using a DLP system) by a simple transformation. The system has been implemented using the object-oriented programming language Ruby and Treetop, a language for Parsing Expression Grammars (PEGs). It currently produces DLP programs in the format of DLV and can use DLV as a back-end.

1 Introduction

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. The heads (resp. the bodies) of DLP rules are disjunctions (resp. conjunctions) of simple constructs, viz. atoms and literals. In [1], we proposed Normal Form Nested programs that are an extension of Disjunctive Logic Programs with variables. In particular the head of an NFN rule is a formula in disjunctive normal form while the body is a formula in conjunctive normal form. We provided also a polynomial translation from NFN programs to DLP programs. The main idea of the algorithm is to introduce new atoms, which represent conjunctions appearing in the head of the rules and disjunctions appearing in the bodies. This translation allows for evaluating NFN programs using DLP systems, such as DLV [2], GnT [3], or Cmodels3 [4].

In this paper we describe a tool implementing the efficient translation from safe NFN programs to safe DLP programs presented in [1], called `nfn2dlp`. The system provides an NFN parser and safety checker, and an efficient translation to an equivalent

* 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.”

DLP program. The output program is in the format of DLV, state-of-the-art implementation for disjunctive logic programs under the answer set semantics, and thus allows for effective answer set computation of NFN programs.

2 Normal Form Nested Programs

In this section, we briefly introduce syntax, semantics and safety of NFN programs. For a more detailed discussion, we refer to [1].

Syntax We consider a first-order language without function symbols. NFN programs are finite sets of rules of the form

$$C_1 \vee \dots \vee C_n \text{ :- } D_1, \dots, D_m. \quad n, m \geq 0$$

where each of C_1, \dots, C_n is a positive basic conjunction (a_1, \dots, a_k) of atoms and each of D_1, \dots, D_m is a basic disjunction $(l_1 \vee \dots \vee l_j)$ of literals. The parentheses around basic conjunctions and disjunctions may be omitted. $C_1 \vee \dots \vee C_n$ is the *head*, and D_1, \dots, D_m is the *body* of a rule. An NFN program is called *standard* if all basic conjunctions and disjunctions are singleton literals.

Safety Let r be an NFN rule. A variable X in r is *restricted* if there exists a positive basic disjunction D in the body of r , such that, for each $a \in D$, X occurs in a ; we also say that D saves X and X is made safe by D . A rule is safe if each variable appearing in the head and each variable that appears in a negative body literal are restricted. An NFN program is safe if each of its rules is safe.

As shown in [1], safe programs have the important property of domain independence, that is, their semantics is invariant with respect to the considered universe.

Semantics We consider ground instantiations of NFN programs with respect to a given universe. When considering safe NFN programs, the Herbrand universe is sufficient. An *interpretation* for a safe NFN program P can therefore be denoted as a subset of the Herbrand base. The satisfaction of ground rules by interpretations is defined in the classical way, interpreting rules as implications. An interpretation that satisfies a program is called a *model*.

The *reduct* of a ground program P with respect to an interpretation I , denoted by P^I , is obtained by (1) deleting all false literals w.r.t. I from rule bodies, and (2) deleting all rules s.t. any basic disjunction becomes empty after (1). An interpretation I is an *answer set* for P iff I is a subset-minimal model for P^I . We denote the set of answer sets for P by $AS(P)$.

3 An Efficient Translation from NFN to DLP

In this section we will review the rewriting algorithm *rewriteNFN* from [1]. The basic structure of *rewriteNFN* is shown in Fig. 1. The input for *rewriteNFN* is a safe NFN

program P and it builds and eventually returns a safe standard DLP program, P_{DLP} . The algorithm transforms one rule at a time. For each NFN rule, it constructs one *major rule*, which maintains the structure of the NFN rule, replacing complex head and body structures by appropriate labels. Head and body of the major rule are built independently by means of functions *buildHead* and *buildBody*, respectively, which will be described in the sequel of this section. These functions may also create a number of auxiliary rules, for defining labels and auxiliary predicates which are needed mostly for guaranteeing safety of the transformed program.

```

begin rewriteNFN
Input: NFN program  $P$ 
Output: DLP program  $P_{DLP}$ .
var  $B$ : conjunction of literals;  $H$ : disjunction of atoms;
     $P_{DLP} := \emptyset$ ;
for each rule  $r \in P$  do
     $H := \text{buildHead}(H(r), P_{DLP})$ ;  $B := \text{buildBody}(B(r), P_{DLP})$ ;
     $P_{DLP} := P_{DLP} \cup \{H :- B.\}$ ;
return  $P_{DLP}$ ;

```

Fig. 1. Algorithm *rewriteNFN*

3.1 Head Transformation

Function *buildHead* takes as input an NFN rule head and builds a corresponding standard rule head (disjunction of atoms). For each non-singular basic conjunction $C = a_1, \dots, a_n$ (i.e. $n > 1$), the function builds an auxiliary atom representing C . The predicate name of the new atom is aux_C^r and its arguments are all variables occurring in C . In order to act as a substitute for C , the function also creates auxiliary rules $aux_C^r(\dots) :- a_1, \dots, a_n.$ and $a_i :- aux_C^r(\dots), i = 1, \dots, n$. It is easy to see that these auxiliary rules are safe.

3.2 Body Transformation

More care has to be taken in function *buildBody*. Since not all variables in a safe NFN rule body have to be restricted, just replacing body disjunctions by labels as for NFN heads may result in unsafe auxiliary rules because of unrestricted variables. If the variable in question occurs only in its body disjunction, it can be safely dropped from the label atom, but if this variable occurs also elsewhere in the rule, the values it represents must match in each of its occurrences. Therefore, *buildBody* focuses on *shared variables*, where a variable X is *shared* in a rule r , if it appears in two different body disjunctions of r , or if X appears in both head and body of the rule. As a further complication, in some literals a shared variable may not be bound to any value.

For creating the body of the major rule, *buildBody* replaces each body disjunction D of a rule r containing more than one literal by a label atom $aux_D^r(V_1, \dots, V_n)$, where aux_D^r is a fresh symbol and V_1, \dots, V_n are the shared variables of r occurring in D . An auxiliary rule for defining $aux_D^r(V_1, \dots, V_n)$ is added for each literal in D , where variables not occurring in the respective literal are replaced by the special constant $\#u$,

representing that the respective variable is unbound. Moreover, if the literal is negative, some new *universe* atoms are added to the body which in turn are defined by appropriate auxiliary rules. Since $\#u$ has to match with any other constant, matching has to be made explicit in the body of the major rule by adding dedicated *matching* atoms, also defined by auxiliary rules.

3.3 Properties of the Algorithm

Let P a safe NFN program, $P_{DLP} = \text{rewriteNFN}(P)$, and \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, $I \in AS(P)$ if and only if there exists a unique $J \in AS(P_{DLP})$ such that $I = J \cap \mathcal{A}_N$. As mentioned previously, all rules generated by *rewriteNFN* are safe. Moreover, the complexity of the algorithm is a small polynomial.

4 The nfn2d1p System

Algorithm *rewriteNFN*, along with an NFN parser and safety checker has been implemented as a front-end to DLP systems. Currently, the syntax of the system DLV is supported, but the implementation is decoupled from DLV and can easily be modified for supporting other DLP systems such as Gt or Cmodels3. The resulting tool, called *nfn2d1p*, is publicly available at

<http://www.mat.unical.it/software/nfn2d1p/>.

In the following we provide some information about issues in the implementation of *nfn2d1p*. Moreover, we give a description of the usage of *nfn2d1p*.

4.1 Implementation of nfn2d1p

The tool *nfn2d1p* has been implemented using the language Ruby [5], an object-oriented language rooted also in functional and scripting languages.

The tool comprises an NFN parser, implemented using the tool *treetop* [6], which provides a parser generator for Parsing Expression Grammars (PEGs) [7] for Ruby. PEGs are a novel concept for parser specification, which look similar to BNF grammars but differ in semantics; most importantly these grammars avoid ambiguity and allow for modular language specification. These properties simplify the development and allow for reuse of the parser.

The tool *nfn2d1p* has been constructed using an object-oriented design: For all language constructs, such as atoms, literals, basic disjunctions, basic conjunctions and rules, appropriate Ruby classes have been designed. The respective objects are created during parsing. The safety check has been implemented as a method of the rule class.

Moreover, two classes for handling rewriting have been defined, *RewriteHead* and *RewriteBody*, respectively. These classes contain as attributes the respective NFN structure (head and body, respectively), a corresponding DLP structure for constructing the major rule, and a set of auxiliary DLP rules. The methods of these classes effectively implement *buildHead* and *buildBody*. Finally, a basic command-line interface is provided, which we overview in Section 4.2.

4.2 Using `nfn2d1p`

The interface of `nfn2d1p` is via the command-line. By default, `nfn2d1p` reads the files provided as arguments, treats their contents as one NFN program, analyzes its well-formedness and safety, and eventually translates it into a DLP program, which will be provided on standard output.

Example 1. Consider the program P represented in the text file `example.txt` as

$$a, b(X) :- c(X) \vee d(X, Y). \quad c(1). \quad d(2, 3).$$

In order to test for safety and to transform P into a DLP program, we issue

```
$ nfn2d1p.rb example.txt
```

on the command line. Since the program is safe, the rewritten program is printed on standard output:

```
a :- auxh1.0(X). \quad b(X) :- auxh1.0(X). \quad auxh1.0(X) :- a, b(X). \quad c(1).
aux1.0(X) :- c(X). \quad aux1.0(X) :- d(X, Y). \quad auxh1.0(X) :- aux1.0(X). \quad d(2, 3).
```

The answer sets of the NFN program can be computed by pipelining the output into DLV using the command

```
$ nfn2d1p.rb example.txt | DLV -- -filter=a,b,c,d
```

yielding the answer set $\{c(1), d(2, 3), a, b(1), b(2)\}$. The distribution also provides a script `nfnsolve` which automates the call to DLV with the appropriate filter.

5 Future Work

We plan to modify `nfn2d1p` in order to output the resulting DLP program in formats acceptable to other systems (`lparse` and `gringo` acting as grounders for `Cmodels3`, `GnT`, and `ClaspD`). Furthermore, we are working on extensions of the NFN language in order to allow for a richer (not necessarily normal-form) syntax in heads and bodies. This will require an adaptation of the safety definition and an extension of the translation algorithm.

References

1. Bria, A., Faber, W., Leone, N.: Normal form nested programs. In: Proceedings of the 11th European Conference on Logics in Artificial Intelligence (JELIA 2008). LNCS 5293, (2008) 76–88
2. 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
3. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923, (2004) 331–335
4. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
5. Flanagan, D., Matsumoto, Y.: *The Ruby Programming Language*. O'Reilly (2008)
6. Sobo, N.: `treetop` homepage <http://treetop.rubyforge.org/>.
7. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004). (2004) 111–122

Solving CSPs with Naming Games

Stefano Bistarelli^{1,2,3} and Giorgio Gosti^{3,4}

¹ Dipartimento di Scienze, Università “G. d’Annunzio” di Chieti-Pescara, Italy
`bista@sci.unich.it`

² Institute of Informatics and Telematics (IIT-CNR) Pisa, Italy
`stefano.bistarelli@iit.cnr.it`

³ Dipartimento di Matematica e Informatica, Università degli Studi di Perugia,
`giorgio.gosti@dipmat.unipg.it`.

⁴ Institute for Mathematical Behavioral Sciences, University Of California, Irvine.

Abstract. Constraint solving problems (CSPs) represent a formalization of an important class of problems in computer science. We propose here a solving methodology based on the naming games. The naming game was introduced to represent N agents that have to bootstrap an agreement on a name to give to an object. The agents do not have a hierarchy and use a minimal protocol. Still they converge to a consistent state by using a distributed strategy. For this reason the naming game can be used to untangle distributed constraint solving problems (DCSPs). Moreover it represents a good starting point for a systematic study of DCSP methods, which can be seen as further improvement of this approach.

1 Introduction

The goal of this research is to generalize the naming game model in order to define a distributed method to solve CSPs. In the study of this method we want to exploit the power of distributed calculation, by letting the CSP solution emerge, rather than being the conclusion of a hierarchical search.

In DCSP protocols we design a distributed architecture of processors, or more generally a group of agents, to solve a CSP instantiation. In this framework we see the problem as a dynamic system and we set the stable states of the system as one of the possible solutions to our CSP. To do this we design each agent so that it will move towards a stable local state. The system is called “self-stabilizing” whenever the global stable state is obtained starting from a local stable state [4]. When the system finds the stable state the CSP instantiation is solved. A protocol designed in this way is resistant to damage and external threats because it can react to changes in the problem instance.

It is important to notice the fundamental difference with the Distributed CSP Algorithms designed by Yokoo [13], and Sipper [11]. Yokoo addresses three fundamental types of distributed CSP Algorithms: Asynchronous Backtracking, Asynchronous Weak-commitment Search, and Distributed Breakout Algorithm. Although these algorithms share the propriety of being asynchronous they require a pre-agreed agent/variable ordering. The algorithm that is presented in

this paper does not need this initial condition, and despite the analogies with a distributed local search algorithm, the solution search is based on the competition between the domains of agents, in which by domain we define a set of agents in a local solution. Furthermore, the domain competition dynamics is driven by the agents change in the local solution and by the agent option to belong to more than one partial solution. Sipper defines the Non-uniform Cellular Automata as a distributed agent system immersed in a discrete time and space lattice, in which each agent is able to evolve a specific deterministic behavior. His approach to distributed CSP "[...] is one in which a grid of rules locally co-evolves to perform a given task", such as finding the solution. Besides, in our algorithm the agents have a predefined behavior, although this behavior is determined by random variables, such as the drawn speaker and the drawn assignment.

The algorithm described in this paper uses a central scheduler that randomly draws the speakers, this may be interpreted as a "central orchestrator" scheme, although we evince that this central scheduler has no information on the CSP instance, and has no pre-determined agent/variable ordering. Moreover, if we consider the case in which there is no central scheduler, and the agents have a fix probability P to speak at a certain turn t , then we see that if this probability satisfies the relation $P \ll 1/N$, the probability that there be more than one speaker at the time t will be approximately equal to zero. Under these conditions our system can be seen as an approximation of a distributed system with no central scheduler.

In Section 2 we illustrate the naming game formalism and we make some comparisons with the distributed CSP (DCSP) architecture. Then we describe the language model that is common to the two formalizations and introduce an interaction scheme to show the common framework. At last we state the definition of Self-stabilizing system [4].

In Section 3 we explicitly describe our generalization and formalize the protocol that our algorithm uses and test it on different CSPs. Moreover, for particular CSPs instantiations we analytically describe the multi-agent algorithm evolution that makes the system converge to the solution.

2 Background

2.1 The distributed constraint satisfaction problem (DCSP)

In CSPs we consider a set of N variables x_1, x_2, \dots, x_N , their definition domains D_1, D_2, \dots, D_N and a set of constraints on the values of these variables. Solving the CSP means finding a at least one of the CSP solutions. A CSP solution is a particular assignment tuple $\bar{X} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_N)$ for the variables x_1, x_2, \dots, x_N that satisfy all the constraints C .

In the DCSP [13], the variables of the CSP are distributed among the agents. These agents are able to communicate between themselves and know all the constraint predicates that are relevant to their own variables. The agents through interaction find the appropriate values to assign to the variables and solve the CSP.

2.2 Introduction to Naming Games

The naming games [10, 1, 9, 6] describe a set of problems in which a number N of agents bootstrap a commonly agreed name for one or more objects. Each naming game is defined by an interaction protocol. There are two important aspects of the naming game: the agents randomly interact and use simple deterministic or probabilistic rules to update their state; the agents converge to a consistent state in which all the objects of the set have a uniquely assigned name, by using a distributed social strategy.

Generally, two agents are randomly extracted at each turn to perform the role of the speaker and the listener (or hearer as used in [10, 1]). The interaction between the speaker and the listener determines the agents' update of their internal state. DCSP and the naming game share a variety of common features [5], moreover, we will show in Section 3 that the naming game can be seen as a set of particular DCSP instances.

2.3 The Communication Model

In this framework we define a general model that describes the communication procedures between agents both in naming games and in DCSPs. The communication model consists of N agents (also called processors) arranged in a network.

We will use a central scheduler that at each turn randomly extracts the agents that will be interacting. Two agents connected by an edge in the network are neighbors. We define a broadcast register in which only the speaker i can write and can be read by all the neighboring listeners. At each interaction the speaker broadcasts the same variable assignment (word) b_s to all the neighbors by assigning the value b_s to the broadcast register. For each edge of the *communication graph* we allocate a register on which the listener can upload the communication outcome feedback f_{ij} using a predetermined signaling system.

The interaction scheme can be represented in three steps:

1. *Broadcast* The speakers broadcast information related to the proposed assignment for the variable;
2. *Feedback* The listeners feedback the interaction outcome expressing some information on the speaker assignment by using a standardized signal system;
3. *Update* The speakers and the listeners update their state regarding the overall interaction outcome.

In this scheme we see that at each turn the agents update their state. The update reflects the interaction they have experienced. We have presented the general interaction scheme, wherein each naming game and DCSP algorithm has its own characterizing protocol.

2.4 Self-Stabilizing Algorithms

A self-stabilizing protocol [4] has some important properties. First, the global stable states are the wanted solutions to our problem. Second, the system configurations are divided into two classes: legal associated to solutions and illegal

associated to non-solutions. We may define the protocol as self-stabilizing if in any infinite execution the system finds a legal system configuration that is a global equilibrium state. Moreover, we want the system to converge from any initial state. These properties make the system fault tolerant and able to adapt its solutions to changes in the environment.

To make a self-stabilizing algorithm we program the agents of our distributed system to interact with the neighbors. The agents through these interactions update their state trying to find a stable state in their neighborhood. Since the algorithm is distributed many legal configurations of the agents' states and its neighbors' states start arising sparsely. Not all of these configurations are mutually compatible and so form inconsistent legal domains. The self-stabilizing algorithm must find a way to make the global legal state emerge from the competition between these domains. Dijkstra [4] and Collin [3] suggest that an algorithm designed in this way can not always converge and a special agent is needed to break the system symmetry. In this paper we will show a different strategy based on the concept of random behavior and probabilistic transition function that we will discuss in the next sections.

3 Generalization of the naming game to solve DCSP

In the naming game, the agents want to agree on the name given to an object. This can be represented as a DCSP, where the name proposed by each agent is the assignment of the CSPs variable controlled by the agent, and where an equality constraint connects all the variables. On the other hand, we can generalize the naming game to solve DCSPs.

We attribute an agent to each variable of the CSP as in [13]. Each agent $i = 1, 2, \dots, N$, names its own variable x_i in respect to the *variable domain* D_i . We restrict the constraints to binary relation C_{ij} between variable x_i and x_j . If $x_i C_{ij} x_j$ is true, then the values of the variables x_i and x_j are consistent. We define two agents as neighbors if their variables are connected by a constraint.

The agents have a *list*, which is a continuously updated subset of the domain elements. The difference between the *list* and the domain is that the domain is the set of values introduced by the problem instance, and the *list* is the set of variable assignments that the agent subjectively forecasts to be in the global solution, on the basis of its past interactions. When the agent is a speaker, it will refer to this *list* to choose the value to broadcast and when it is a listener, it will use this *list* to evaluate the speaker broadcasted value.

At turn $t = 0$ the agents start an empty *list*, because they still do not have information about the other variable assignments. At each successive turn $t = 1, 2, \dots$ an agent is randomly drawn by the central scheduler to cover the role of the speaker, and all its neighbors will be the listeners. The communication between the speaker s and a single listener l can be a *success*, a *failure*, or a *consistency failure*. At the end of the turn all the listeners feedback to the speaker the *success*, the *failure*, or the *consistency failure* of the communication.

If all the interaction sessions of the speakers with the neighboring listeners are successful, we will have a *success update*. If there was one or more *consistency failure* we will have a *consistency failure update*. If there was no *consistency failure* and just one or more *failures*, there will be a *failure update*.

The interaction, at each turn t , is represented by this protocol:

1. *Broadcast*. If the speaker *list* is empty it extracts an element from its *variable domain* D_s , puts it in its *list* and communicates it to the neighboring listeners. Otherwise, if its *list* is not empty, it randomly draws an element from its *list* and communicates it to the listeners. We call this the broadcast assignment d_s .
2. *Feedback*. Then the listeners calculate the consistent assignment subset K and the consistent domain subset K' :
 - *Consistency evaluation*. Each listener uses the constraint defined by the edge, which connects it to the speaker, to find the consistent elements d_l to the element d_s received from the speaker. The elements d_l that it compares with d_s are the elements of its *list*. These consistent elements form the *consistent elements subset* K . We define $K = \{d_l \in list | d_s C_{s_l} d_l\}$. If K is empty the listeners compare each element of its variable domain D_l with the element d_s , to find a *consistent domain subset* K' . We define $K' = \{d_l \in D_l | d_s C_{s_l} d_l\}$.

The *consistent elements subset* K and the *consistent domain subset* K' determine the following feedback:

- *Success*. If the listener has a set of elements d_l consistent to d_s in its *list* (K is not empty), there is a *success*.
 - *Consistency failure*. If the listener does not have any consistent elements d_l to d_s in its *list* (K is empty), and if no element of the listener *variable domain* is consistent to d_s (K' is empty), there is a *consistency failure*.
 - *Failure*. If the listener does not have any d_l consistent elements to d_s in its *list* (K is empty), and if a non empty set of elements of the listener *variable domain* are consistent to d_s (K' is not empty), there is a *failure*.
3. *Update*. Then we determine the overall outcome of the speaker interaction on the basis of the neighbors' feedback:
 - *Success update*. This occurs when all interactions are successful. The speaker and the neighbors cancel all the elements in their *list* and update it in the following way: the speaker stores only the successful element d_s and the listener stores the consistent elements in K .
 - *Consistency failure update*. This occurs when there is at least one *consistency failure* interaction. The speaker must eliminate the element d_s from its *variable domain* (this can be seen as a step of local consistency pruning). The listeners do not change their state.
 - *Failure update*. This occurs in the remaining cases. The speaker does not update its list. The listeners update their *lists* by adding the set K' of all the elements consistent with d_s to the elements in the *list*.

We can see that in the cases where the constraint $x_i C_{ij} x_j$ is an equality, the subset of consistent elements to x_i is restricted to one assignment of x_j . For this

assignment of the constraint $x_i C_{ij} x_j$ we obtain the naming game as previously described. Our contribution to the interaction protocol is to define K and K' . As a matter of fact, in the naming game the consistent listener assignment d_l to the speaker's assignment d_s is one and only one ($d_l = d_s$), unlike the CSP instances, in which there may be more consistent listener assignments d_l for each speaker's assignment d_s . This observation is fundamental to solve general CSP instances. Moreover, in the naming game there is only one speaker and one listener at each turn, but we observed that under this hypothesis the agents were not always able to enforce local consistency (e.g.: graph coloring of a completely connected graph). Thus we had to extend the interaction to all the speaker neighbors and let all the neighbors be listeners.

At each successive turn the system evolves through the agents' interactions in a global equilibrium state. In the equilibrium state all the agents have only one element for their variable and this element must satisfy the constraint $x_i C_{ij} x_j$ with the element chosen by the neighboring agents. Clearly, this state is a solution of the CSP instance, and we call this a *global consensus state*. Once in this state the interactions are always successful. The probability to transit to a state different from the *global consensus state* is zero, for this reason the *global consensus state* is referred to as an absorbing state. We call the turn at which the system finds global consensus *convergence turn* t_{conv} .

Simple Algorithm Execution The N -Queens Puzzle is the problem of placing N queens on a $N \times N$ chessboard without having them be mutually capturable. This means that there can not be two queens that share the same row, column, or diagonal. This is a well known problem and has been solved linearly by specialized algorithms. Nevertheless it is considered a classical benchmark and we use it to show how our algorithm can solve different instances. To reduce the search space we assign the variables of a different column to each queen. We can do this because, if there were more than one queen in a column, they would have been mutually capturable. In this way each of our agents will have as its domain the values of a distinct column and all the agents will be mutually connected by an edge in the graph representing all the constraints. In the example we show a N -Queens Puzzle with $N = 4$. Each agent (queen) is labeled after its column with numbers from zero to three from left to right. The rows are labeled from the bottom with numbers from zero to three. In Figure 1 we show how the algorithm explores the solution space randomly and how it evolves at each turn t . We write the speaker s that is extracted at each turn and its broadcasted value b . Then we write the listeners l , their respective K , K' , and the feedbacks. At the end we write the updates. The picture represents graphically the evolution of the agents' list at the end of each turn.

At turn $t = 1$ (see Fig.1) speaker $s = 3$ is randomly drawn. The *variable* controlled by this speaker is the position of the queen on the last column of the chessboard. The speaker has an empty *list*, hence it draws from its *variable domain* the element $d_s = 3$ which corresponds to the highest row of its column. The speaker adds this new element to its *list*. Since all the agents are connected, all

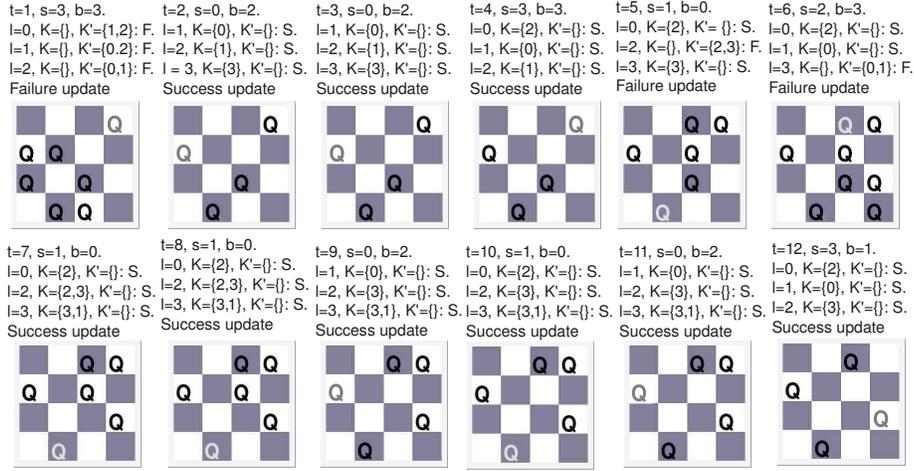


Fig. 1. Single algorithm run for the N -Queens Puzzle with $N = 4$.

the agents apart from the speaker are listeners. Their *lists* are empty therefore K is empty. Thus they compute K' from the *variable domain*. The listeners feedback failure, thus the speaker replies with a failure update. The listeners add the elements of their respective K' to their *lists*. The picture in fig.1 shows the elements in the agents' *lists* at the end of the turn. At turn $t = 2$ speaker $s = 0$ is drawn and it broadcasts the element $d_s = 2$. All the listeners have a consistent element in their list, therefore, their K s are not empty, and they feedback a success. The listeners delete their *lists* and add the elements in their K . At turn $t = 3$ the speaker $s = 0$ speaks again and broadcasts the same element $d_s = 2$. Therefore, the listener computes the same K s as before, and feedbacks a success. Thus we have a success update, but since the K s are the same the system does not change. At turn $t = 4$ the speaker $s = 0$ is drawn and broadcasts the same variable $d_s = 3$ that it had broadcasted at the first turn. Since all the elements in the listeners' *lists* are still consistent to this broadcast, the algorithm has a success update and the agents' *lists* remain the same. At turn $t = 5$ a new speaker is drawn $s = 1$, it broadcasts $d_s = 0$. The listeners zero and three have a consistent element to this broadcast, therefore their K is not empty. Furthermore, listeners two has no consistent elements to put in K , and finds the rows two and three from its *variable domain* to be consistent to this broadcast. The overall outcome is a failure and thus we have a failure update. The listeners zero and two have empty K' s so they do not change their *lists* and listener two adds two new elements to its lists. At turn $t = 6$ agent two speaks and broadcasts the element $d_s = 3$. Agent three does not have consistent elements to this broadcast and thus feedbacks a failure. Then we have a failure update and agent two adds two elements to its *list*. At turn $t = 7$ agent one speaks and broadcasts the element $d_s = 0$. All the listeners have consistent elements

therefore their K s are not empty. We get a success update. The agents two and three both delete an element from their *lists* which is not consistent to the speaker broadcast. At turn $t = 8$ agent one speaks again and broadcasts the same element $d_s = 0$. The system is unchanged. At turn $t = 9$ agent zero speaks and broadcasts the element $d_s = 2$. All listeners have consistent elements, therefore, there is a success update. Listener two deletes an element that was not consistent with the speaker broadcast. At turn $t = 10$ agent one speaks and broadcasts the element $d_s = 0$. All listeners have consistent elements to this broadcast, there is a success update, and the system is unchanged. At turn $t = 11$ agent zero speaks and broadcasts the element $d_s = 2$. All listeners have consistent elements to this broadcast, there is a success update, and the system is unchanged. At turn $t = 12$ agent three speaks and broadcasts the element $d_s = 1$. All listeners have consistent elements to this broadcast, there is a success update. Since the speaker had a different element in its list from the broadcasted element $d_s = 1$, he deletes this other element from its *list*. At this point all the elements in the agents *lists* are mutually consistent. Therefore, all the successive turns will have success updates and the system will not change any more. The system has found its global equilibrium state that is a solution of the puzzle we intended to solve.

3.1 Difference with Prior Self-Stabilizing DCSPs

In the prior DCSPs the agent is a finite state-machine. Furthermore, its evolution in time is represented by a *transition function*, which depends on its state and its neighbors' states in the current turn. Let the local state s_i be the union of the agent state a_i and its neighbors' state. The *transition function* determines the communication outcome, and the agents states update once we know the local state s_i . Moreover, the *transition function* of an agent, which state is a_i , in the local state s_i is one and one only, and we can forecast exactly its next state a_{i+1} and the next local state s_{i+1} .

Let uniform protocols be distributed protocols, in which all the nodes are logically equivalent and identically programmed. It has been proved that in particular situations uniform self-stabilizing algorithms can not always solve the CSPs ([3]), especially if we consider the ring ordering problems. In ring ordering problems we have N numbered nodes $\{n_1, n_2, \dots, n_N\}$ ordered on a cycle graph. Each node has a variable, the variable assignment of the $i + 1$ -th node n_{i+1} is the consecutive number of the variable assignment of the i -th node n_i in modulo N . The *variable domain* is $\{0, 1, \dots, N - 1\}$ and every link has the constraint $\{n_i = j, n_{i+1} = (j + 1) \bmod N | 0 \leq j \leq N\}$. Dijkstra [4] and Collin [3] propose dropping the uniform protocol condition to make the problem solvable.

Our protocol overcomes this by introducing a random behavior *probabilistic transition function* $T(P_s, P_l, P_b)$. In our algorithm the agent state is defined by an array that attributes a zero or a one to each element of the agent domain. The array element will be zero if the element is not in the *list*, and one if the element is in the list. This array determines a binary number a_i , which defines the state of the agent. If we know the states of all the agents, the transition of each agent from state a_i to a_j is uniquely determined once we know the agent

that will be the speaker, the agents that will be the listeners, and the element that will be broadcasted by the speaker (Fig.2(a)).

Since the speaker will be chosen randomly, we can compute the probability for each agent being a speaker P_s . From this information, since we know the underlying graph and that all its neighbors will be listeners, we can compute the probability for each agent to be a listener P_l . Knowing the speaker state, we can compute the probability for each element to be broadcast P_b . At this point we may be able to compute the *probabilistic transition function* $T(P_s, P_l, P_b)$, which will depend on the probabilities that we have just defined (Fig.2(b)).

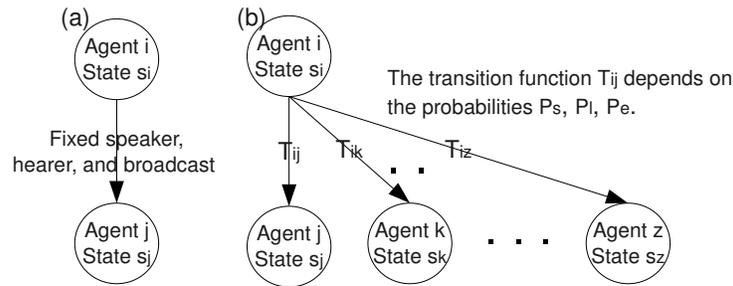


Fig. 2. (a) Shows that once we determine the speaker state, the broadcast, and the listener state we are able to determine the speaker and listeners' transitions. (b) Shows that since we have the speaker probability P_s , the broadcast probability P_b , and the listener probability P_l we can determine the probabilistic transition function $T(P_s, P_l, P_b)$.

In this setting the agent state a_t at turn t can now be represented by a discrete distribution function and the *transition function* is now a Markovian Chain, the arguments of which are the transition probabilities p_j between the local states s_i and s_j . Thus we speak of a *probabilistic transition function*, which represents the probability of finding the system in a certain state s_j starting from s_i at time t . This behavior induces the algorithm to explore the state space randomly, until it finds the stable state that represents our expected solution. In the following plot we show the *convergence turn* t_{conv} scaling with the size N of the ring ordering problem. We average the *convergence turn* t_{conv} on ten runs of our algorithm for a set size N . Then we plot this point in a double logarithmic scale to evince the power law exponent of the function. We found that $t_{conv} \propto N^{3.3}$.

3.2 Analytical Description

In this section we are going probabilistically analyze how our algorithm solves graph coloring problems for the following graph structures: path graph and completely connected graph. These are simple limiting cases that help us to picture

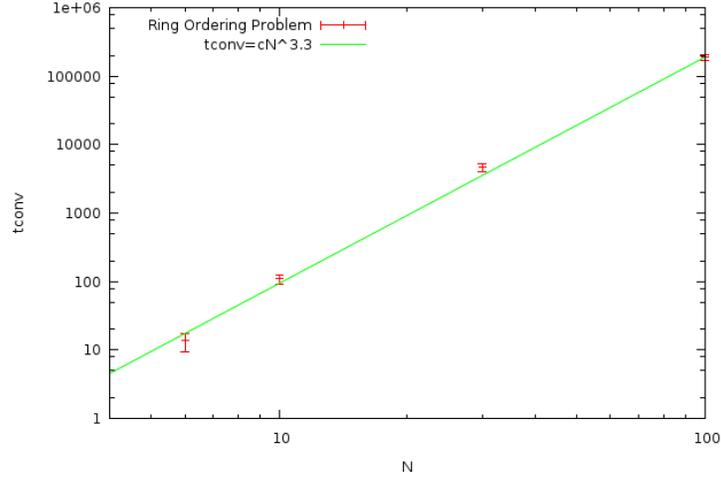


Fig. 3. The plot shows the ring ordering problem with N nodes, we see for the convergence turn t_{conv} the scaling proportion: $t_{conv} \propto N^{3.3}$.

how our algorithm evolves in more general cases. In graph coloring the variable domain elements represent colors, and the edges between nodes represent inequality constraints.

Path graph The way our algorithm solves a path graph coloring instance can be described through analytical consideration; similar observations can then be extended to the cycle graph. The system dynamics are analogous to the naming game on a one dimensional network [2]. To each node of the path graph we attribute a natural number in increasing order, from the first node of the path, to which we attribute 1, to the last node, to which we attribute N . The path graph coloring needs minimum two colors to be solved, therefore we imposed the agents' variable domain to two colors. We see that there are only two solutions: with odd number nodes of one color and even number nodes of the second color in one solution; and inverted in the other solution. At the beginning, when $t < N/3$, the system is dominated by new local consistent nodes' neighborhoods, which emerge sparsely and propagate to the connected nodes. At each turn a speaker is randomly drawn. This speaker has an empty *list* and it has to draw the assignment from the two element *variable domain*. In this way it selects one of the two solutions to which it starts to belong. Broadcasting its assignment to its neighbors it makes them converge on the same solution. In this way a small consistent domain of three agents that agree on the final state emerges.

Since the speakers are chosen by the scheduler randomly, after some time, $t > N/3$, all the agents have been speakers or listeners at least once. Thus we

find approximately $N/3$ domains dispersed in little clusters of generally three agents. Each of these domains belong to one of the two final solutions.

At this point the domains start to compete. Between two domains we see an overlapping region appear. This region is constituted by agents that have more than one element in their lists. We can refer to them as undecided agents that belong to both domains, since the agents are on a path graph this region is linear. By probabilistic consideration we can see that this region tends to enclose less than two agents. For this reason we define the region that they form as a *border*, for a path graph of large size N the border width is negligible. So we approximate that only one agent is within this *border*. Under this hypothesis we can evaluate the evolution of the system as a diffusion problem, in which the borders move in a random walk on the path graph. When a domain grows over another domain, the second domain disappears. Thus the relation between the cluster growth and time is $\Delta x \propto (\frac{t}{\xi})^{1/2}$, where ξ is the time needed for the random walk to display a deviation of ± 1 step from its position. The probability that the border will move one step right or left on the path graph is $\propto 1/N$, proportional to the probability that an agent on the border or next to the border is extracted. Thus we can fix the factor $\xi \propto 1/N$. Since the lattice is N steps long we find the following relation for the average convergence turn $t_{conv} \propto N^3$. The average convergence turn t_{conv} is the average time in which the system finds global consistency. Let the average convergence turn t_{conv} be defined as the sum of the weighted convergence turns of all the possible algorithm runs, where the weights are the probabilities of the particular algorithm run.

Completely connected graph Since all the variables in the graph coloring of a completely connected graph are bound by a inequality constraint, these variables must all be different. Thus the agent domain must have N elements.

At the beginning all the agents' lists are empty. The first speaker chooses a color, and since all the agents are neighbors, it communicates with all of them. The listeners place in their *lists* the colors from the *variable domain* consistent with the color picked by the speaker. In the following turns the interactions are always successful. Two cases may be observed: the speaker has never spoken so it selects a color from its *list* and it broadcasts it, and deletes all the other elements, while the listeners cancel the broadcasted color from their lists; or the speaker has already spoken once, so there are no changes in the system because it already has only one color and all the other agents have already deleted this assignment. Since at each turn only one agent is a speaker, to let the system converge all the agents have to speak once.

Let N be the number of agents and t_{conv} the system convergence time. Let t_i be the turn t in which i agents have spoke one or more times, and $N - i$ have never spoke. Let s_i be the waiting time between the turn t_{i-1} and the turn t_i . It is useful to notice that we can represent t_{conv} by $t_{conv} = \sum_{i=1}^N s_i$ and s_i is a geometric random variable. We use this probability to compute the weighted average turn at which the system converges. This will be the weighted average convergence turn t_{conv} . Moreover, we use the property that the expected

value of the sum of two random variables is equal to the sum of the respective expected values, $E[t_{conv}] = \sum_{i=1}^N E[s_i]$. Since s_i is a geometric random variable, $E[s_i] = i/(N - i)$. Thus:

$$t_{conv} = N \sum_{i=1}^{N-1} \frac{1}{N-i} = N \sum_{j=1}^{N-1} \frac{1}{j} \sim N \log(N-1) \quad (1)$$

This, as we stated above corresponds to the time of convergence of our system, when it is trying to color a completely connected graph.

3.3 Algorithm Test

We have tested the above algorithm in the classical CSP problems, graph coloring. We plotted the graph of the convergence turn t_{conv} scaling with the number N of the CSP variables, each point was measured by ten runs of our algorithm. We considered four types of graphs for the graph coloring: path graphs, cycle graphs, completely connected graphs, and Mycielsky graphs.

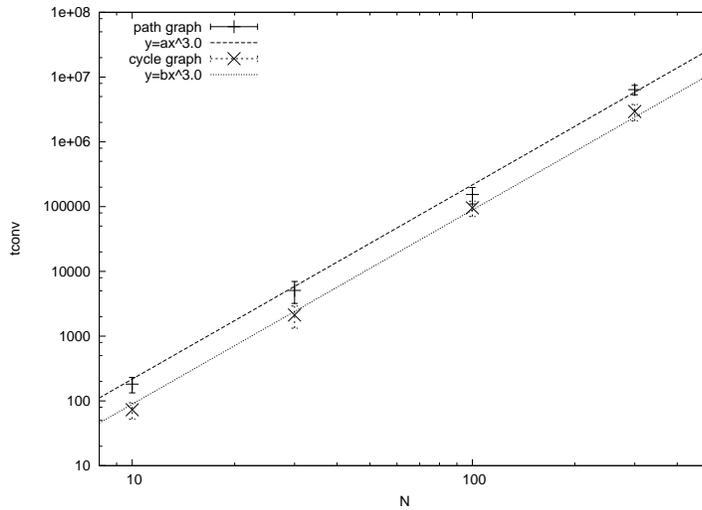


Fig. 4. The plot shows the graph coloring in the case of path graphs and special 2-colorable cycle graphs with 2 colors. The convergence turn t_{conv} of the path graphs and cycle graphs exhibit a power law behavior $t_{conv} \propto N^{3.0}$. The cycle graph exhibits a faster convergence. The points on this graph are averaged on ten algorithm runs.

Graph Coloring In the study of graph coloring we presented four different graph structures:

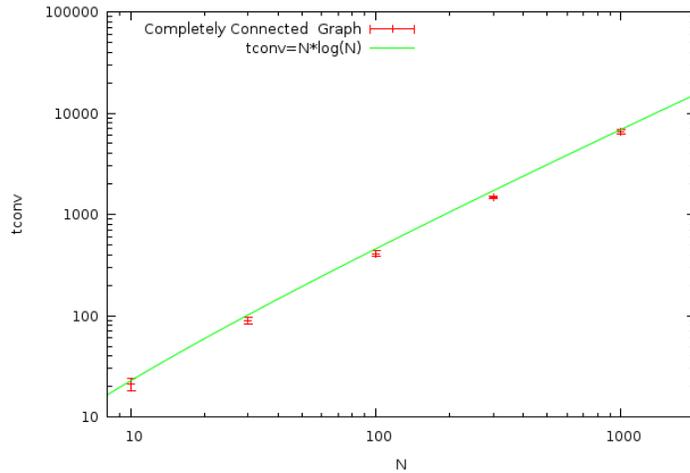


Fig. 5. The plot shows the graph coloring in the case of a completely connected graph with N colors: in this case we find that the convergence turn is $t_{conv} \sim N \log(N)$. The points on this graph are averaged on ten algorithm runs.

- path graphs
- cycle graphs
- completely connected graph
- Mycielski graphs.

In the study of the path graph and the cycle graph we have restricted ourselves to the 2 – *chromatic* cases: all the path graphs and only the even number node cycle graphs. Thus we imposed the agent variable domain to two colors. In this context the convergence turn t_{conv} of the path graph and the cycle graph exhibit a power law behavior $t_{conv} \propto N^{3.0}$. The cycle graph exhibits a faster convergence (Fig. 4). We see from these measurements that the power law of the convergent time scaled with the number of nodes N is compatible with our analytical considerations.

The graph coloring in the case of a completely connected graph always needs at least N colors: in this case we find that the convergence turn is $t_{conv} \propto N \log(N)$ (Fig. 5).

The Mycielski graph [8] of an undirected graph G is generated by the Mycielski transformation on the graph G and is denoted as $\mu(G)$. Let the N number of nodes in the graph G be referred to as v_1, v_2, \dots, v_N . The Mycielski graph is obtained by adding to graph G $N+1$ nodes: N of them will be named u_1, u_2, \dots, u_N and the last one w . We will connect with an edge all the nodes u_1, u_2, \dots, u_N to w . For each existing edge of the graph G between two nodes v_i and v_j we include an edge in the Mycielski graph between v_i and u_j and between u_i and v_j .

M_i	N	E	k optimal coloring	t_{conv}
M_4	11	20	4	32 ± 2
M_5	23	71	5	170 ± 20
M_6	47	236	6	3300 ± 600
M_7	95	755	7	$(1.1 \pm 0.2) \cdot 10^6$

Table 1. Convergence turn t_{conv} of the Mycielski graph coloring. M_i is the Mycielski graph identification, N is the number of nodes, E is the number of edges, k the optimal coloring, and t_{conv} the convergence turn.

The Mycielski graph of graph G of N nodes and E edges has $2N + 1$ nodes and $2E + N$ edges.

Iterated Mycielski transform applications starting from the null graph, generates the graphs $M_i = \mu(M_{i-1})$. The first graphs of the sequence are M_0 the null graph, M_1 the one node graph, M_2 the two connected nodes graph, M_3 the five nodes cycle graph, and M_4 the Grötzsch graph with 11 vertices and 20 edges (see Fig. 6). The number of colors k needed to color a graph M_i of the Mycielski sequence is, $k = i$ ([8]).

These graphs are particularly difficult to solve because they do not possess triangular cliques, moreover, they have cliques of higher order and the coloring number increases each Mycielski transformation ([7]). We ran our algorithm to solve the graph coloring problem with the known optimal coloring. Table 1 shows for each graph of the Mycielski sequence M_i , the number of nodes N , the number of edges E , the minimal number of colors needed k and the convergence turn t_{conv} of our algorithm.

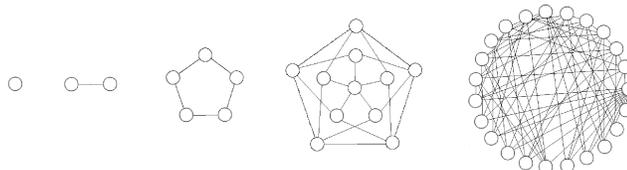


Fig. 6. Mycielski graph sequence M_1, M_2, M_3, M_4 , and M_5 [12].

4 Conclusions and Future Work.

Our aim is to develop a probabilistic algorithm able to find the solution of a CSP instance. In the study of this method we are trying to fully exploit the power of distributed calculation. To do this we generalize the naming game algorithm, by letting the CSP solution emerge, rather than being the conclusion of a sequence

of statements. As we saw in Subsection 3.2, our algorithm is based on the random exploration of the system state space. Our algorithm travels through the possible states until it finds the absorbing state, where it stabilizes. These ergodic features guarantee that the system has a probability equal to one to converge for long times. Unfortunately this time, depending on the particular CSP instance, can be too long for practical use. This goal was achieved through the union of new topics addressed in statistical physics (the naming game), and the abstract framework posed by constraint solving.

In future work we will test the algorithm on a uniform random binary CSP to fully validate this method. We also expect to generalize the communication model to let more than one agent speak at the same turn. Once we have done this we can let the agents speak spontaneously without a central scheduler.

References

1. Baronchelli, A., Felici, M., Caglioti, E., Loreto, V., Steels, L.: Sharp Transition Toward Shared Vocabularies in Multi-Agent Systems. In: *Journal of Statistical Mechanics*, (2006).
2. Baronchelli, A., Dall'Asta, L., Barrat, A. and Loreto, V.: Topology Induced Coarsening in Language Games. In: *Phys. Rev. E* 73, (2006).
3. Collin, Z., Dechter, R., and Katz, S.: On the Feasibility of Distributed Constraint Satisfaction. In: *Proceedings of the Twelfth International Joint Conference of Artificial Intelligence (IJCAI-91)*, (1991).
4. Dijkstra E.W.: Self-Stabilizing Systems in Spite of Distributed Control. In: *Communications of the ACM*, (1974).
5. Gosti, G.: Resolving CSP with Naming Games. In: *24th International Conference on Logic Programming*, (2008).
6. Komarova, N. L., Jameson, K. A., Narens, L.: Evolutionary Models of Color Categorization Based on Discrimination. In: *Journal of Mathematical Psychology*, (2007)
7. Trick M.: Network Resources for Coloring a Graph. <http://mat.gsia.cmu.edu/COLOR/color.html>.
8. Mycielski, J.: Sur le coloriage des graphes. In: *Colloq. Math.* 3, (1955).
9. Nowak, M.A., Plotkin, J.B., Krakauer, J.D.: The Evolutionary Language Game. In: *Journal of Theoretical Biology*, (1999).
10. Steels, L.: Self-Organizing Vocabularies. In: *Artificial Life V: Proceeding of the Fifth International Workshop on the Synthesis and Simulation of Living Systems* (1996).
11. Sippers, M.: Co-evolving Non-Uniform Cellular Automata to Perform Computations. In: *Physica D*, 92:193-208, (1996).
12. Weisstein, E. W.: Mycielski Graph. In: *MathWorld—A Wolfram Web Resource*. <http://mathworld.wolfram.com/MycielskiGraph.html>.
13. Yokoo, M., Katsutoshi H.: Algorithms For Distributed Constraint Satisfaction: A Review. In: *Autonomous Agents and Multi-Agent Systems*, Vol.3, No.2, pp.198-212, (2000).

An Heuristics for Load Balancing and Granularity Control in the Parallel Instantiation of Disjunctive Logic Programs

Simona Perri, Francesco Ricca, and Marco Sirianni

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{perri,ricca,sirianni}@mat.unical.it

Abstract. In this paper we present a dynamic heuristics that allows for improving the performance of a parallel instantiator algorithm based on the DLV system. In this system, each rule is rewritten in several “splits” of the same size that are assigned to a number of parallel instantiator subprocesses. The new heuristics allows for dynamically determining an optimal amount of work that has to be assigned to each parallel instantiator, and, thus, it improves the overall efficiency of the parallel evaluation. We implemented our heuristics and performed an experimental analysis that confirms the efficacy of the proposed method.

1 Introduction

In the last few years, entry-level computer systems have started to implement multi-core/multi-processor SMP (Symmetric MultiProcessing) architectures. In a modern SMP computer two or more identical processors can connect to a single shared main memory, and the operating system supports multithreaded programs for exploiting the available CPUs [1]. However, most of the available software was devised for single-processor machines and is unable to exploit the power of SMP architectures. Recently [2, 3], such technology has been exploited for implementing faster evaluation systems in the field of Answer Set Programming (ASP). ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [5–10] which features a declarative nature combined with a relatively high expressive power [11, 12].

Traditionally, the kernel modules of ASP systems work on a ground instantiation of the input program. Therefore, 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 (see [10, 12]); thus, having an efficient instantiation procedure is, in general, crucial for the performance of the entire ASP systems. Moreover, some recent applications of ASP (see e. g. , [13–16]), have evidenced the practical need for faster instantiators. It is easy to see that the exploitation of SMP technology in the grounding process can bring significant performance improvements. Indeed, an effective technique for the parallel instantiation of ASP programs was proposed in [2]. However, the efficacy of this method was limited to programs with many rules, since it roughly allows for instantiating independent rules in parallel; but, in [3] a rewriting technique has been proposed that modifies the input program in such a way that the technique of [2] becomes applicable also in case of programs with few rules. The following example explains the idea behind this rewriting

(the reader is referred to [3] for a detailed description of the technique). Consider the program encoding of the well-known 3-colorability problem:

$$\begin{aligned} (r) \quad & col(X, red) \vee col(X, yellow) \vee col(X, green) :- node(X). \\ (c) \quad & :- edge(X, Y), col(X, C), col(Y, C). \end{aligned}$$

In this case, the technique of [2] is unable to produce a parallel evaluation, since it proceeds by first instantiating (r) (thus, computing the extension of col), and, then, only once this is done, by processing the constraint (c) .

However, one may rewrite this program in an equivalent one which is more amenable for parallel evaluation. Basically, since each single rule of the input program is processed by one processing unit, one may think of rewriting it into an equivalent program containing several rules. For instance, the program can be rewritten as follows [3]:

$$\begin{aligned} (r_1) \quad & col(X, red) \vee col(X, yellow) \vee col(X, green) :- node_1(X). \\ (r_2) \quad & col(X, red) \vee col(X, yellow) \vee col(X, green) :- node_2(X). \\ \dots & \\ (r_n) \quad & col(X, red) \vee col(X, yellow) \vee col(X, green) :- node_n(X). \\ (c_1) \quad & :- edge_1(X, Y), col(X, C), col(Y, C). \\ \dots & \\ (c_n) \quad & :- edge_n(X, Y), col(X, C), col(Y, C). \end{aligned}$$

The sets of nodes and edges are *split up* into subsets by splitting the extension of predicates $node$ and $edge$, respectively. The resulting program is equivalent to the original one modulo renaming, but its instantiation can be done in parallel.

This rewriting was implemented [3] in a parallel ASP instantiator based on DLV [11]. In the first version of the system the number of rule splits was set to a global user-defined value (the same for each rule in input). However, this naïve strategy is not optimal in most cases. Indeed, if each instantiator receives a “very small” amount of work, then the costs added by parallel execution are larger than the benefits (because of the overhead introduced by thread creation and scheduling); on the other hand, if the amount of work assigned to threads is “too big” then a resulting bad workload distribution will reduce the advantages of parallel evaluation. Note also that, the optimal setting may be different for each rule.

In this paper, we propose a dynamic heuristics that is able to improve the overall efficiency of the parallel evaluation of [3] by automatically determining, rule by rule, an optimal amount of work that has to be assigned to each parallel instantiator. Moreover, we implemented our heuristics, and we report here the results of an experimental analysis that confirms the efficacy of the proposed method.

2 An Heuristics for Load Balancing and Granularity Control

A real implementation of a parallel system has to deal with two important issues that strongly affect the performance: load balancing and granularity. Indeed, if the workload is not uniformly distributed to the available processors then the benefits of parallelization are not fully obtained; moreover, if the “amount” of work assigned to each parallel processing unit is too small then the (unavoidable) overheads due to creation and scheduling of parallel tasks might overcome the advantages of parallel evaluation (in a corner case, adopting a sequential evaluation might be preferable).

The parallel grounder described in [3] implements a naïve strategy: basically each rule is rewritten in a globally fixed number of “splits” (specified by the user). Here, a crucial role is played by the number of splits allowed for each rule, which is (usually) the main source of concurrently running threads, and it directly determines the “amount” of work (and, thus, the “size of the split”) assigned to instantiators. It is easy to see that the best possible fixed setting for the number of splits might be not optimal, since the evaluation of different rules in the same program may require significantly different execution times. In our setting, (i. e. shared memory processor) developing a sophisticated granularity-control strategy is not essential (as also observed in [4]) as for other parallel architectures (like, e. g. , clusters); rather it is sufficient to set the size of the splits for each rule to an adequate value. The size of the split should be sufficiently large to avoid thread management overhead; and sufficiently small to exploit the preemptive multitasking scheduler of the operating system for obtaining a good workload distribution. Clearly, also the number of running threads has to be controlled.

In order to satisfy both requirements, in our implementation: (i) we let the user set the number of concurrently running thread (an adequate value is given by a multiple of the number of available CPUs so that preemptive multitasking is exploited for load balancing); and (ii) we implemented and tuned an heuristics that allows for selecting an optimal split size for each rule. Note that the second task is not trivial, since the time needed for evaluating each rule is not known a priori.

More in detail, our method computes an estimation $e(r)$ of the amount of work required for evaluating each rule r of the program (just before r has to be instantiated) then, it exploits $e(r)$ for associating to r its split size among three empirically-defined values: small, large, and no split (i.e. sequential evaluation)¹; Basically, very easy rules are evaluated sequentially, since the overhead introduced by threads is higher than their expected evaluation time (granularity control); whereas, for hard rules a small split size is employed for obtaining a finer distribution of work; finally, easy rules, whose computation can still exploit some parallelism, are evaluated using a large split size for minimizing the overheads. The estimation $e(r)$ is obtained by combining (actually summing) two factors: the number of comparisons made by our algorithm and the number of operations needed to compute and print the output (“size of the corresponding join”). The latter is motivated by the similarities between the rule instantiation process and the evaluation of a join in a database system. In the following, we describe how the two factors have been estimated.

Size of the join. The size of the join between two relations R and S with one or more common variables can be estimated, according to [17], as follows:

$$T(R \bowtie S) = \frac{T(R) \cdot T(S)}{\prod_{X \in \text{var}(R) \cap \text{var}(S)} \max\{V(X, R), V(X, S)\}}$$

where $T(R)$ is the number of tuples in R , and $V(X, R)$ (called selectivity) is the number of distinct values assumed by the variable X in R . For joins with more relations one can repeatedly apply this formula to couple of body predicates according to a given evaluation order.

¹ Note that in case of recursive rules, a new distribution is performed each time they are processed (according semi-naïve evaluation [2]); thus obtaining a dynamic load balancing.

Problem	small	medium	large	Heuristics
$3col_1$	11.50 (0.11)	8.57 (0.01)	8.57 (0.07)	8.91 (0.04)
$3col_2$	14.42 (0.15)	11.68 (0.13)	11.89 (0.05)	11.29 (0.20)
$3col_3$	23.01 (0.23)	19.68 (1.06)	19.57 (0.04)	18.46 (0.19)
$reach_2$	60.73 (0.17)	40.73 (0.18)	39.92 (0.32)	39.70 (0.12)
$reach_3$	129.78 (0.77)	84.67 (0.30)	82.78 (0.32)	82.44 (0.52)
$ramsey_1$	44.00 (0.77)	91.04 (0.65)	95.92 (0.12)	43.19 (0.31)
$ramsey_2$	72.38 (0.40)	170.51 (1.28)	236.82 (2.76)	71.96 (0.21)
$ramsey_3$	112.66 (0.71)	286.64 (0.74)	294.51 (2.51)	111.02 (0.13)

Table 1. Result for different size of the split - result for heuristics choice.

Number of comparisons. An approximation of the number of comparison done for instantiating a rule r is: $C(r) = \sum_{x \in X(r)} \prod_{l \in L(r,x)} V(x,l)$, where $X(r)$ is the set of variables that appear in at least two literals in the body of r , $L(r,x)$ is the set of body literals in which x occurs; and $V(x,l)$ is the selectivity of x in the extension of l . Roughly, the number of comparisons is approximated by sum of the products of the number of distinct values assumed by each join variable in the body of r .

3 Experiments

We implemented our heuristics and tested its efficacy in a collection of benchmark programs already used for assessing ASP instantiators performance ([11, 18, 19]). In particular, we considered the following well-known problems: *3-colorability*, *Reachability* and *Ramsey Numbers*; for space reasons we do not report the encodings (they are available at <http://www.mat.unical.it/ricca/downloads/cilc09.zip>), for a detailed problems description refer to [11, 19, 3]. The machine used for the experiments is a two-processor Intel Xeon “Woodcraracteriest” (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. We set the number of concurrent splits to 32 (the quadruple of available processors). Actually, an experimental analysis (not reported here for space reasons) confirmed that this fixed setting is optimal for the available hardware.

We measured the performance of the system in the case of some growing (fixed) split sizes (small=1 tuple, medium=50 tuples, large=100 tuples)², and when the new heuristics is employed. In order to obtain more trustworthy results, each single experiment was repeated three times, and both the average and standard deviation of the results are reported in Table 1. In particular, the first three columns contain the average instantiation times (and standard deviation) for the different split sizes, while last column reports the performance obtained by applying the heuristics. The fixed setting that obtains the best result is reported in bold face for each instance.

First of all we notice that *the performance of the heuristics version is always optimal and overcomes the better fixed setting in most cases*, since it benefits from the selection of a different split size for each rule of the program.³

About Ramsey, since the encoding is composed of few “very easy” rules and two “hard” constraints, the best split size choice for the entire program is the smaller one (see Table 1). We verified that the version with the heuristics evaluates sequentially the

² More experiments have been done on different split size; we reported here the most significant.

³ In addition, we observed that performance gains (w.r.t. serial execution) range from about 700% up to 790% in the best setting, which is near to the theoretical limit for eight-processors.

rules and selects a small split size for the constraints, thus resulting the best performer. Dual considerations hold for 3col, where the best split size for the entire program is the largest (third column), since the evaluation of rules requiring a medium/large split sizes dominates the computation time. Still, the version equipped with the heuristics, overcomes the results of the fixed split size, as it chooses a large split size for the rule, and a small split size for the constraints. The application of the heuristics to reachability (the encoding is composed of two “easy” rules, one of which is recursive) results in the selection of large split size and sequential evaluation respectively for the two input rules, and this choice results to be still more appropriate.

As far as future work is concerned, we are experimenting for a finer tuning of the heuristics, and we are considering a larger set of problems. Moreover, we planned to test the system behavior when more than 8 processors are available (probably by exploiting a simulation environment).

References

1. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
2. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. *Journal of Algorithms in Cognition, Informatics and Logics* **63**(1–3) (2008) 34–54
3. Vescio, S., Perri, S., Ricca, F.: Efficient Parallel ASP Instantiation via Dynamic Rewriting. In: ASPOCP 2008, Udine, Italy (2008)
4. Lopez, P., Hermenegildo, M., Debray, S.: A Methodology for Granularity Based Control of Parallelism in Logic Programs. In: *J. Symbolic Computation* (1996), 22, 715–734
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
6. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: (ICLP’99) 23–37
7. Marek, V. W. , Truszczyński, M. : Stable Models and an Alternative Logic Programming Paradigm. In: *The Logic Programming Paradigm-A 25-Year Perspective*. (1999) 375–398
8. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
9. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence* **138**(1–2) (2002) 3–38
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. *ACM Transactions on Database Systems* **22**(3) (September 1997) 364–418
11. 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
12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys* **33**(3) (2001) 374–425
13. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kalka, 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: (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
14. Curia, R., Ettore, M., Gallucci, L., Iiritano, S., Rullo, P.: Textual Document Pre-Processing and Feature Extraction in OLEX. In: *Data Mining VI*, WIT Pres, 2005, pp. 163-173
15. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Non-monotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar (2005)

16. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. Proceedings ASP05, Bath, UK (July 2005) 248–262
17. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
18. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR 2007,LCNCS 4483,3-17
19. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Annals of Mathematics and Artificial Intelligence **51**(2–4) (2007) 195–228

A Magic Set Implementation for Disjunctive Logic Programming with Function Symbols

Marco Marano, Francesco Ricca, and Giovambattista Ianni

Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy.
mmarano@deis.unical.it, ianni@mat.unical.it, ricca@mat.unical.it

Abstract. The Magic Sets rewriting technique (MS) consists in rewriting a logic program P with respect to a query Q in such a way that, the bottom-up evaluation of the rewritten program simulates the top-down evaluation of Q in the original program. In this way, only a restricted part of the ground program of P is evaluated, thus obtaining valuable efficiency improvements. Many extensions and modifications of the base technique have been proposed in literature, but most of them are confined to the Datalog realm. In this paper we present an implementation of the Magic Set rewriting technique that is applicable to positive disjunctive logic programs with function symbols under the answer set semantics. We show how the base technique has to be modified when both disjunction in the rule heads and function symbols occur contemporarily in the input program. Finally, we describe our implementation of the technique, which is able to produce magic programs compatible with DLV syntax.

1 Introduction

The Magic Sets rewriting technique takes a significant place in the literature about logic programming and deductive database systems, since its early definition in [1]. Given a logic program P and a query Q over its vocabulary, this technique consists in rewriting P with respect to Q , by adding some predicates and some newly created rules: these latter are introduced in order to simulate the top-down computation of the program. By using Magic Sets it is possible to reduce the amount of unnecessary computation, due to portions of the ground version of P which cannot alter the answer to Q , but are however evaluated if a pure bottom-up scheme is used. Many extensions and modifications of the base technique have been proposed in literature, aimed at improving or extending it to more specific cases. Among them, we mention here the extensions to disjunctive logic programs in [2, 6], and the one realized for programs with (possibly unstratified) negation in [3]. In this paper we focus our attention on positive disjunctive logic programs with function symbols, applying the magic set technique to this kind of programs. Some particular issues arise when considering this language, due to the presence of function symbols along with disjunction. The main contributions of this work are: (i) we extend the magic set technique to the case of positive disjunctive programs with function symbols by devising an appropriate transformation algorithm; (ii) we give an implementation of the algorithm, and we show how it works by example.

In the following, we first recall both syntax and stable model semantics of disjunctive logic programs; then, we present by means of an example the classic Magic Sets transformation for Datalog programs, followed by the detailed description of our “magic” algorithm; some notes regarding the implementation and comments about future work conclude the paper.

2 DLP with function symbols

Let \mathcal{C} be a denumerable set of distinguished constant and function symbols. Let \mathcal{X} be a set of variables. Let \mathcal{P} be a set of predicate symbols. We conventionally denote variables with uppercase first letter, while constants will be denoted by lowercase first letter. A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant symbol from \mathcal{C} or a variable from \mathcal{X} . A *functional term* is of the form $f(t_1, \dots, t_n)$ where t_1, \dots, t_n are *terms* and f is a function symbol of arity n .

An *atom* is of the form $p(t_1, \dots, t_n)$, where t_1, \dots, t_n are *terms*, and $p \in \mathcal{P}$ represents the *predicate name* of the atom.

A *program* is a set of *rules* of the form $a_1 \vee \dots \vee a_n :- b_1, \dots, b_m$ where a_1, \dots, a_n and b_1, \dots, b_m are atoms, and $n \geq 0, m \geq 0$. The disjunction $a_1 \vee \dots \vee a_n$ is the head of r , denoted by $H(r)$, while b_1, \dots, b_m is a conjunction denoted as $B(r)$ (the body of r). A rule with empty body will be called *fact*. A predicate appearing only in rule bodies and in facts is referred to as *EDB predicate*, otherwise as *IDB predicate*. In the following, we will assume to deal with *safe* programs, that is, programs in which each variable appearing in a rule r appears in at least one atom in $B(r)$.

Given the program P , the *Herbrand Universe* U_P is the set of ground terms which can be constructed using the symbols of \mathcal{C} appearing in P . A *substitution* for a rule r of P is a mapping θ from the set of variables of r to U_P . We denote $r\theta$ as the *ground rule* obtained by substituting variable occurring in r with elements of U_P according to θ . A ground rule contains only ground atoms; the set of all possible ground atoms that can be constructed combining predicates of P and terms in U_P is usually referred to as *Herbrand Base* (B_P). We denote by $grnd(r)$ the set of ground rules obtained by applying all the possible substitutions to r . Given a plain program P , its ground version $grnd(P)$ is the union of all the sets $grnd(r)$ for $r \in P$.

An *interpretation* for P is a set of ground atoms, that is, an interpretation is a subset $I \subseteq B_P$. We define the following entailment notion with respect to an interpretation I .

For a a ground atom: $I \models a$ iff $a \in I$; For a_1, \dots, a_n ground atoms:

$I \models a_1, \dots, a_n$ iff $I \models a_i$, for each $1 \leq i \leq n$; $I \models a_1 \vee \dots \vee a_n$ iff $I \models a_i$ for at least one $i, 1 \leq i \leq n$. For a rule r : $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$;

A *model* for P is an interpretation M for P such that every rule $r \in grnd(P)$ is such that $M \models r$. A model M for P is *minimal* if no model N for P exists such that N is a proper subset of M . The set of all minimal models for P is denoted by $MM(P)$.

An interpretation I for a program P is an *answer set* for P if $I \in MM(P)$ (i. e., I is a minimal model for the positive program P). The set of all answer sets for P is denoted by $ans(P)$. We say that $P \models a$ for an atom a , if $M \models a$ for all $M \in ans(P)$.

3 Informal Overview

The *Magic Sets* rewriting technique consists of a simulation of the top-down evaluation of a query Q by modifying an original program P and producing a rewritten program $M(P, Q)$ which comprises additional rules, and updates to the original ones. $M(P, Q)$ is conceived in order to reduce computation to what is actually relevant for answering the query. In fact, $grnd(P)$ contains, in general, many ground rules that have no impact in answering Q as they are related to atoms which Q does not depend on. In general, it is expected that $grnd(M(P, Q))$ has smaller size than $grnd(P)$.

The original magic sets method was first described in [1] for the case of Datalog, i. e. logic programs without function symbols. Following work considered the presence of functional terms, yet not explicitly taking disjunction also into account (see e.g. []).

Concerning the stable model semantics, it is known how to apply this rewriting technique to Datalog Programs with disjunction [2, 6] and also (with some restricting assumption) to unstratified programs [3].

To give an intuition about the general magic set technique for Datalog programs, we can consider the following (traditional) example. Let us consider the query $Q = path(1, 5)$? on the following program P_1 :

$$path(X, Y) :- edge(X, Y). \quad path(X, Y) :- edge(X, Z), path(Z, Y).$$

As a first step, head predicates are “adorned”. Basically, we simulate the top-down computation and annotate the way how the variable bindings are propagated from the head atom to body atoms. Each rule of the input program is replaced by an “adorned” one in which the name of each predicate is modified by appending the binding information. Given an *IDB* predicate, we denote a bound argument with the b letter, while a free one is labeled with f . For instance $path^{bf}$ is a predicate which is in principle a subset of $path$: in particular its first argument is restricted to a set of values (the *magic set* of $path^{bf}$) which is usually much smaller than the range of $path$ on its first argument. The adornment process starts from the query Q . This latter is adorned in a very simple manner: all constants in the query become bound, all variables are marked as free (we obtain in this case the predicate $path^{bb}$). Adornment is propagated to rules’ heads in which $path$ appears, and subsequently from the head to the body. If a new adorned predicate is created (as it is present in the head or the body of the rule), this is processed in turn in the same way of the original adorned query, until no more adorned predicates have to be processed. SIPs (Sideways Information Passing Strategies) are used in order to establish the adornment policy: for space reasons we refer the reader to [7], for a detailed explanation about SIPs. In our example, the arguments of the given query are both constants, and thus bound; we will build the adorned program according to $path^{bb}$:

$$path^{bb}(X, Y) :- edge(X, Y). \quad path^{bb}(X, Y) :- edge(X, Z), path^{bb}(Z, Y).$$

Note that EDB predicates are excluded from adornment. The next step of the transformation consists in generating magic rules starting from the adorned program. These rules define *magic predicates*. A *magic predicate* defines the allowed range of values for bound arguments of a predicate. We start from the head of the rule.

Given an adorned head atom $a(\mathbf{t})$, we obtain the set of terms \mathbf{t}' , derived from \mathbf{t} by removing all the terms corresponding to free arguments, and generate the magic atom $magic_a(\mathbf{t}')$. Then, for each atom b in the body, we create its magic version $magic_b(\dots)$. Subsequently, we generate a magic rule having $magic_b$ in the head and $magic_a$ in the body, followed by all the atoms of the adorned rule which can propagate the binding.

The third step consists of the modification of the adorned rules. In this step we add to the bodies of the rules the magic atoms which have been generated in the previous step. For each rule with head h , an according magic atom $magic_h$ is inserted in the body of the rule.

$$\begin{aligned} &magic_path^{bb}(1, 5). \quad magic_path^{bb}(Z, Y) :- magic_path^{bb}(X, Y), edge(X, Z). \\ &path(X, Y) :- magic_path^{bb}(X, Y), edge(X, Y). \\ &path(X, Y) :- magic_path^{bb}(X, Y), \quad edge(X, Z), \quad path(Z, Y). \end{aligned}$$

Finally, in the last step the query is processed by adding a magic fact $magic_q_ad$ if q is the query and ad its adornment; In our example we add $magic_path^{bb}(1, 5)$.

The resulting program is then evaluated w.r.t. the query.

4 Magic Sets for DLP with Function Symbols

In this section we present a new version of the standard Magic Set technique presented above, which works on *disjunctive* programs with *function symbols*. The algorithm is sketched in Figure 1. The main procedure is called **magify**.

```

Program magify(Program P, Query Q)
{
  Program M(P,Q)=∅;
  if(Q.isconjunctive())
  {
    Rule R',Query Q';
    (Q',R') = normalizeQuery(Q);
    P.addRule(R');
    Q=Q';
  }
  Program AP = createAdornedProgram(P,Q);
  Program MR = createMagicRules(AP);
  Program MP = addMagicAtoms(P);
  Fact MF = createMagicFact(Q);
  M(P,Q)=removeAdornments(MP∪MR∪MF);
  return M(P,Q);
}

```

Fig. 1. Function createAdornedProgram

```

Program createAdornedProgram( Program P,
Query Q )
{
  Stack S = ∅; Program AP = ∅;
  S.push(createAdornedVersionOf(Q));
  while(S.size > 0)
  {
    Atom x= S.pop();
    for(Rule r in P)
    Rules adornedRules = adornRule(r,x);
    AP.add(adornedRules);setDone(X);
    for(Rule ar in adornedRules)
    for(Atom a in ar)
    if(!done(a)) S.push(a);
  }
  return AP;
}

```

Fig. 2. Function createAdorned-Program

The function **magify** takes a program P and a query Q as input, and applies the Magic Sets Transformation, generating $M(P, Q)$ (the *magified program*). **magify** is made of other subprocedures, detailed in the following. Let us assume it is given the query: $Q_2 = a(f(1))?$ and the program P_2 :

$$r1 : a(X) \vee b(X) :- c(X), e(X). \quad r2 : c(f(X)) :- c(X). \quad r3 : e(1). \quad r4 : c(1).$$

When a query is conjunctive, it is transformed into a rule, having in the head a new atom which contains all the variables from the atoms in the original query. The original query is replaced by a new one which consists of the head of the newly created rule. This procedure is performed by the function **normalizeQuery(Query Q)**.

The next step consists of creating the adorned program **AP**, by means of the function **createAdornedProgram(Program P, Query Q)**, reported in Figure 2. A stack S is used in order to keep the atoms scheduled for adornment. The query is adorned using the function **createAdornedVersionOf** and pushed in S at first. The main cycle pops out from S a given atom a and accordingly adorns each rule having in the head an atom whose name matches with it. When a certain adornment is generated for the first time for a predicate, this is pushed into S , in order to be processed. The algorithm iterates until S is empty.

The adornment of each rule is actually performed by the inner function **adornRule(r, x)** which returns a set of adorned rules according to the labels of x , to be added to the adorned program. If x is not in the head of r , **adornRule** returns an empty set. More in detail, the output of **adornRule** contains a set R' of adorned rules for each atom $x' \in H(r)$ which unifies with x . Each $r' \in R'$ is built according to the following strategy: per each $x' \in H(r)$ which unifies with x , x' is labelled according to x , then such labelling is propagated to $B(r)$, according to a SIP [7]. Successively, adornments are propagated from

$B(r)$ to the remaining head atoms. Moreover, from the obtained adorned disjunctive rule r' , corresponding to x' , we obtain $|H(r)|-1$ auxiliary rules obtained by leaving in the head only one atom $x \in H(r) \setminus \{x'\}$ and having $B(r) \cup (H(r) \setminus x)$ as body. The obtained set of auxiliary rules in AP will not take part in the final program $M(P, Q)$, but will be further processed in order to obtain the set of magic rules MR . In turn, magic rules are created, according with the traditional strategy, by calling the **CreateMagicRules** function. In our example, we get first from rule $r1$ and $r2$, the adorned versions $r1' : a^b(X) \vee b^b(X) :- c^b(X), e(X)$ and $r2' : c^b(f(X)) :- c^b(X)$ then **createMagicRules(Program P)** obtains from $r1'$ and $r2'$ the corresponding magic rules; and from $r1'$ we get the two rules: $a^b(X) :- c^b(X), e(X), b^b(X)$. $b^b(X) :- c^b(X), e(X), a^b(X)$., while $r2'$ is left unchanged. Now the function **createMagicRules** simply applies the normal Magic-Set strategy to these intermediate rules, as seen in previous section. In our example we obtain:

$$\begin{aligned} &magic_{\mathcal{C}}^b(X) :- magic_{\mathcal{A}}^b(X), e(X), b^b(X). \quad magic_{\mathcal{C}}^b(X) :- magic_{\mathcal{B}}^b(X), e(X), a^b(X). \\ &magic_{\mathcal{C}}^b(X) :- magic_{\mathcal{C}}^b(f(X)). \end{aligned}$$

The third rule has been obtained by applying the algorithm for the non disjunctive case. Now, the function **addMagicAtoms(P)** is called, which returns a version of **P** including magic predicates within the body of each rule of **P**. In this simple step, for each atom in the head of the rule the corresponding magic atoms are added in the body. Successively, a magic atom from the query is generated by the function **createMagicFact(Query Q)** to be added to the final output. In our example we get: $magic_{\mathcal{A}}^b(f(1))$. Finally, the function call **removeAdornments(MP \cup MR \cup MF)** removes all adornments from the non-magic predicates. This is necessary as stated in [2]. The final output for our example is the following:

$$\begin{aligned} &magic_{\mathcal{C}}^b(X) :- magic_{\mathcal{A}}^b(X), e(X), b^b(X). \quad magic_{\mathcal{C}}^b(X) :- magic_{\mathcal{B}}^b(X), e(X), a^b(X). \\ &magic_{\mathcal{C}}^b(X) :- magic_{\mathcal{C}}^b(f(X)). \quad magic_{\mathcal{A}}^b(f(1)). \\ &a(X) \vee b(X) :- c(X), e(X), magic_{\mathcal{A}}^b(X), magic_{\mathcal{B}}^b(X). \quad c(f(X)) :- c(X), magic_{\mathcal{C}}^b(f(X)). \end{aligned}$$

It must be noted here that two aspects of the class of programs we are treating, disjunction and the presence of function symbols, need a particular treatment. In particular:

Disjunction requires modifications on the adornment strategy. Let r be a rule of the form:

$$r1 : h_1(t_1) \vee \dots \vee h_n(t_n) :- b_1(p_1), \dots, b_m(p_m).$$

If we adorn the rule w.r.t. the atom $h_i(t_i)$, also other head atoms have to be taken into consideration, because they can contain variables which are actually important for the evaluation. The function acts as follows:(i) the atom $h_i(t_i)$ is adorned w.r.t. the query; (ii) the body is adorned w.r.t. the adornments of $h_i(t_i)$ by using a suitable SIP; (iii) other head atoms $h_1(t_1) \vee \dots \vee h_{i-1}(t_{i-1}) \vee h_{i+1}(t_{i+1}) \vee \dots \vee h_n(t_n)$ are adorned w.r.t. patterns found in the body.

In fact, it has been shown in [6] that if we want to keep the algorithm sound, other head predicates cannot propagate bindings, but can only receive them. In this case bindings are propagated from the body to the remaining head atoms.

Function symbols have impact on the choice of the labelling for arguments: Given an atom $a(\dots, t, \dots)$ for t a functional term t , the corresponding argument of a is labelled as bound iff all the subterms of t are set as bound at the moment of adornment of a .

Remark. Our transformation applies to programs with function symbols, thus, in general, an evaluation of the $M(P, Q)$ is not guaranteed to terminate. However, there are language restrictions that ensures termination, for instance see [9].

5 Implementation Notes and Future Work

The prototype has been implemented in the Java programming language as a preprocessor able to generate a magified program $M(P, Q)$ compatible with the DLV input format [4] from a given program P and a, possibly conjunctive, query Q . The system uses a new Library, called DLVParser, which contains a full framework of classes useful for both the parsing and the manipulation of a Disjunctive Logic Programs in standard syntax. Design patterns have been used, in order to keep the system flexible and easily extensible. In particular, the Strategy pattern has been used for allowing the implementation of multiple SIPs, so that the user of the API of our system is allowed to define his own strategy. To define a new SIP, only a few methods have to be implemented. We have implemented a default SIP, which mimics the propagation of bindings in the Prolog SLD resolution. Inclusion of other constructs such as negation and constraints are forthcoming.

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J. D. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS'86*, 1986.
2. Cumbo C., Faber W., Greco G., Leone N. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In *ICLP'04* pages 371-385, 2004.
3. Faber W., Greco G., Leone N. Magic Sets and their Application to Data Integration. In *ICDT 2005* pages 306-320, 2005.
4. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.
5. Teodor C. Przymusiński. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.
6. Greco, S. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE TKDE* 15 (2), 368-285, 2003.
7. Beeri, C., Ramakrishnan, R. On the power of magic. *JLP* 10 (1,2,3&4), 255-299, 1991.
8. Ramakrishnan, R. Magic Templates: A Spellbinding Approach To Logic Programs. *JLP* 11 (3&4), 189-216 , 1991.
9. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni and Nicola Leone. Bottom-up Evaluation of Finitely Recursive Queries. *CILC09.*, 2009.

A note on 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 constructive interpretation of description logics, we introduce the constructive description logic \mathcal{KALC} . This logic is based on a Kripke-style semantics inspired by the Kripke semantics for Intuitionistic first order logic. In the paper we present the main features of our semantics and we study its relations with other approaches. Moreover, we present a tableau calculus which turns out to be sound and complete for \mathcal{KALC} .

1 Introduction

In Computer Science it often happens that the introduction of a classically based logical system is followed by an analysis of its constructive or intuitionistic counterparts. Indeed, if the applicability of a logical system is often driven from its classical semantics, a constructive analysis allows us to take advantage of the computational properties of its formulas and proofs. In line with this consideration, one of the reasons for the success of *description logics* as a knowledge representation formalism is surely their simple classically-based semantics and only in recent works [3, 5, 6, 9, 11, 12] different proposals of a constructive reinterpretation of description logics have been motivated.

Following this line, we discuss a Kripke-style semantics for the basic description logic \mathcal{ALC} [2, 13] inspired by the Kripke semantics for Intuitionistic first order logic [14]. We call \mathcal{KALC} the logic corresponding to our semantics. Basically, we may think of a *Kripke model* as a set of worlds, representing *states of knowledge*, partially ordered by their information content. Our semantics differs from the similar semantics described in [5] by the fact that we impose a condition on the partial order. In particular, we require that every world is followed by a *classical world*, that is a world where concepts are interpreted according to the usual classical semantics for the description logic \mathcal{ALC} . As we discuss in Section 3, such a condition seems to be essential to get the finite model property, useful to define a decidable calculus for the logic. An important feature of our semantics is that such a condition can be characterized by the axiom-schema $\forall R. \neg\neg A \rightarrow \neg\neg\forall R.A$. This schema is obtained by translating in the description logics setting the *Kuroda principle* for first order logic, a principle which has been deeply studied in the literature of constructive logics [7, 14].

In the paper we also present a tableau calculus which is sound and complete with respect to our semantics. This calculus forms the base of our attempt to prove the finite model property for \mathcal{KALC} . Moreover, it allows us to directly compare to classical tableaux based algorithms that represent the main reasoning method of actual description logics implementations. In this paper we use the tableau calculus to prove that \mathcal{KALC} meets the *disjunction property*, that is, if $A \sqcup B$ belongs to \mathcal{KALC} , then either A or B belongs to \mathcal{KALC} .

The rest of the paper is organized as follows. In Section 2, we introduce the syntax and the classical semantics for \mathcal{ALC} . In Section 3 we present the constructive semantics of \mathcal{KALC} and we describe the relations between our semantics, the classical semantics for \mathcal{ALC} and the constructive semantics described in [5, 11, 12]. In Section 4, we present sound and complete tableau calculus \mathcal{T} for \mathcal{KALC} . Finally, conclusions are drawn in Section 5.

2 Syntax and classical semantics

We begin by introducing the language of the basic description logic \mathcal{ALC} [2, 13] and its classical semantics. The language \mathcal{L} of \mathcal{ALC} is based on the following denumerable sets: the set \mathbf{NR} of *role names*, the set \mathbf{NC} of *concept names*, the set \mathbf{NI} of *individual names*. A *concept* A is an expression of the kind:

$$A ::= C \mid \neg A \mid A \sqcap A \mid A \sqcup A \mid A \rightarrow A \mid \exists R.A \mid \forall R.A$$

where $C \in \mathbf{NC}$ and $R \in \mathbf{NR}$. Let \mathbf{Var} be a denumerable set of *individual variables*; the *formulas* of \mathcal{L} are defined according to the following grammar:

$$H ::= \perp \mid (s, t) : R \mid t : A \mid A$$

where $s, t \in \mathbf{NI} \cup \mathbf{Var}$, $R \in \mathbf{NR}$ and A is a concept. We write $H \in \mathcal{L}$ to mean that H is a formula of \mathcal{L} . An *atomic formula* of \mathcal{L} is a formula of the kind \perp , $(s, t) : R$ or $t : C$, where $R \in \mathbf{NR}$ and $C \in \mathbf{NC}$. A *negated formula* is a formula of the kind $t : \neg A$. A formula is *closed* if it does not contain variables. A *concept formula* is a formula of the kind $t : A$ or A , with A a concept. A *role formula* is a formula of the kind $(t, s) : R$ with $R \in \mathbf{NR}$.

We remark that in the classical setting $A \rightarrow B$ is equivalent to $\neg A \sqcup B$, but this equivalence does not hold in the constructive setting. We define the *concept inclusion* relation (*subsumption*) $A \sqsubseteq B$ as $A \rightarrow B$.

As usual, a *knowledge base* in \mathcal{ALC} consists of a *TBox* and an *ABox*. A TBox is a finite set of terminological axioms in the form $C \sqsubseteq D$ or $C \equiv D$, where C, D are concepts and $C \equiv D$ is an abbreviation for $C \sqsubseteq D$ and $D \sqsubseteq C$. An ABox is a finite set of closed concept and role assertions, of the kind $t : C$ or $(t, s) : R$, where $R \in \mathbf{NR}$ and $t, s \in \mathbf{NI}$.

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 $c \in \mathbf{NI}$, $c^{\mathcal{M}} \in \mathcal{D}^{\mathcal{M}}$; for every $C \in \mathbf{NC}$, $C^{\mathcal{M}} \subseteq \mathcal{D}^{\mathcal{M}}$; for every $R \in \mathbf{NR}$, $R^{\mathcal{M}} \subseteq \mathcal{D}^{\mathcal{M}} \times \mathcal{D}^{\mathcal{M}}$. A non atomic concept A is interpreted by a subset $A^{\mathcal{M}}$ of $\mathcal{D}^{\mathcal{M}}$ as follows:

- $(\neg A)^{\mathcal{M}} = \mathcal{D}^{\mathcal{M}} \setminus A^{\mathcal{M}}$
- $(A \sqcap B)^{\mathcal{M}} = A^{\mathcal{M}} \cap B^{\mathcal{M}}$
- $(A \sqcup B)^{\mathcal{M}} = A^{\mathcal{M}} \cup B^{\mathcal{M}}$
- $(A \rightarrow B)^{\mathcal{M}} = \{d \in \mathcal{D}^{\mathcal{M}} \mid d \notin A^{\mathcal{M}} \text{ or } d \in B^{\mathcal{M}}\}$
- $(\exists R.A)^{\mathcal{M}} = \{d \in \mathcal{D}^{\mathcal{M}} \mid \exists d' \in \mathcal{D}^{\mathcal{M}} \text{ s.t. } (d, d') \in R^{\mathcal{M}} \text{ and } d' \in A^{\mathcal{M}}\}$
- $(\forall R.A)^{\mathcal{M}} = \{d \in \mathcal{D}^{\mathcal{M}} \mid \forall d' \in \mathcal{D}^{\mathcal{M}}, (d, d') \in R^{\mathcal{M}} \text{ implies } d' \in A^{\mathcal{M}}\}$

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 H is *valid* in \mathcal{M} w.r.t. θ , and we write $\mathcal{M}, \theta \models H$, if $H \neq \perp$ and one of the following conditions holds:

- $\mathcal{M}, \theta \models t : A$ iff $t^{\mathcal{M}, \theta} \in A^{\mathcal{M}}$;
- $\mathcal{M}, \theta \models (s, t) : R$ iff $(s^{\mathcal{M}, \theta}, t^{\mathcal{M}, \theta}) \in R^{\mathcal{M}}$;
- $\mathcal{M}, \theta \models A$ iff $A^{\mathcal{M}} = \mathcal{D}^{\mathcal{M}}$.

We write $\mathcal{M} \models H$ iff $\mathcal{M}, \theta \models H$ for every assignment θ . Note that, given a concept A of \mathcal{L} , $\mathcal{M} \models A$ iff $\mathcal{M} \models x : A$, with x any variable. If Γ is a set of formulas, $\mathcal{M} \models \Gamma$ means that $\mathcal{M} \models H$ for every $H \in \Gamma$. We say that H is a *logical consequence* of Γ , and we write $\Gamma \models H$, iff, for every \mathcal{M} and every θ , $\mathcal{M}, \theta \models \Gamma$ implies $\mathcal{M}, \theta \models H$.

3 Kripke semantics

In this section we introduce a Kripke-style semantics for \mathcal{ALC} inspired by the *Kripke semantics* for Intuitionistic first order logic. A similar semantics for description logics has already been proposed in [5]: however, our semantics differs from this one by the fact that we additionally impose a condition on the partial order. As we discuss at the end of this section, such a condition seems to be essential to get the finite model property.

Given a partially ordered set (poset) (P, \leq) , we call *final* an element $\phi \in P$ such that, for every $\alpha \in P$, $\phi \leq \alpha$ implies $\phi = \alpha$. Given $\alpha \in P$, we denote with $\text{Fin}(\alpha)$ the set of final elements $\phi \in P$ such that $\alpha \leq \phi$. We call *K-poset* every poset (P, \leq) such that, for every $\alpha \in P$, $\text{Fin}(\alpha) \neq \emptyset$. We remark that every finite poset is a K-poset. A *KALC-model* is a quadruple $\underline{K} = \langle P, \leq, \rho, \iota \rangle$, where:

- (P, \leq) is a K-poset with root ρ ;
- ι is a function associating with every $\alpha \in P$ a model $\iota(\alpha) = (\mathcal{D}^{\alpha}, \cdot^{\alpha})$ for \mathcal{L} , such that, for every $\alpha, \beta \in P$ with $\alpha \leq \beta$:
 - (K1) $\mathcal{D}^{\alpha} \subseteq \mathcal{D}^{\beta}$;
 - (K2) for every $c \in \mathbf{NI}$, $c^{\alpha} = c^{\beta}$;
 - (K3) for every $C \in \mathbf{NC}$, $C^{\alpha} \subseteq C^{\beta}$;
 - (K4) for every $R \in \mathbf{NR}$, $R^{\alpha} \subseteq R^{\beta}$.

Conditions (K1)–(K4) state that the information content of any element β accessible from α is wider than the information content of α . Indeed, (K1) says that any element known at α is also known at β . (K2)–(K4) state that every fact known at α is also known at β .

Given $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ and $\alpha \in P$, we denote with \mathcal{L}_α the language obtained by adding to \mathcal{L} the elements of \mathcal{D}^α as individual names. We assume that, for every $d \in \mathcal{D}^\alpha$, $d^\alpha = d$. The introduction of a language for every $\alpha \in P$ is a technical machinery needed to treat the interpretation of quantifiers. By Condition (K1), $\alpha \leq \beta$ and $H \in \mathcal{L}_\alpha$ imply $H \in \mathcal{L}_\beta$.

Given $\underline{K} = \langle P, \leq, \rho, \iota \rangle$, $\alpha \in P$ and a closed formula H of \mathcal{L}_α , we inductively define the *forcing relation* $\alpha \Vdash H$ as follows:

- $\alpha \Vdash \perp$;
- $\alpha \Vdash t : A$ where $A \in \mathbf{NC}$, iff $t^\alpha \in A^\alpha$;
- $\alpha \Vdash (s, t) : R$ where $R \in \mathbf{NR}$, iff $(t^\alpha, s^\alpha) \in R^\alpha$;
- $\alpha \Vdash t : A \sqcap B$ iff $\alpha \Vdash t : A$ and $\alpha \Vdash t : B$;
- $\alpha \Vdash t : A \sqcup B$ iff either $\alpha \Vdash t : A$ or $\alpha \Vdash t : B$;
- $\alpha \Vdash t : A \rightarrow B$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, either $\beta \Vdash t : A$ or $\beta \Vdash t : B$;
- $\alpha \Vdash t : \neg A$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash t : A$;
- $\alpha \Vdash t : \exists R.A$ iff there exists $d \in \mathcal{D}^\alpha$ such that $\alpha \Vdash (t, d) : R$ and $\alpha \Vdash d : A$;
- $\alpha \Vdash t : \forall R.A$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$ and for every $d \in \mathcal{D}^\beta$, $\beta \Vdash (t, d) : R$ implies $\beta \Vdash d : A$;
- $\alpha \Vdash A$ iff for every $\beta \in P$ such that $\alpha \leq \beta$ and for every $d \in \mathcal{D}^\beta$, $\beta \Vdash d : A$.

By the above definition, it is easy to check that $t : \neg A$ is equivalent to $t : A \rightarrow \perp$ and hence we could avoid to assume \neg as a primitive connective.

It is easy to prove, using Conditions (K1)–(K4) and the definition of the forcing relation, the following property:

Proposition 1 (Monotonicity property). *Let $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ be a \mathcal{KALC} -model, $\alpha \in P$ and H be a closed formula of \mathcal{L}_α . If $\alpha \Vdash H$ in \underline{K} , then $\beta \Vdash H$ in \underline{K} for every $\beta \in P$ such that $\alpha \leq \beta$. \square*

Given a \mathcal{KALC} -model $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ and $\alpha \in P$, an α -substitution is a function $\sigma : \mathbf{Var} \rightarrow \mathbf{NI} \cup \mathcal{D}^\alpha$. By Condition (K1), we immediately have that an α -substitution is also a β -substitution for every $\beta \in P$ such that $\alpha \leq \beta$. Given a formula H , we denote with $H\sigma$ the formula obtained by simultaneously substituting every occurrence of a variable x in H with $\sigma(x)$. Given a set of formulas Γ , we denote with $\Gamma\sigma$ the set containing $H\sigma$ for every $H \in \Gamma$. If σ is an α -substitution and $d \in \mathcal{D}^\alpha$, $\sigma[d/x]$ is the α -substitution defined as follows:

$$\sigma[d/x](y) = \begin{cases} d, & \text{if } y = x \\ \sigma(y), & \text{otherwise} \end{cases}$$

Given a \mathcal{KALC} -model $\underline{K} = \langle P, \leq, \rho, \iota \rangle$, $\alpha \in P$ and an open formula H of \mathcal{L}_α , $\alpha \Vdash H$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$ and for every β -substitution σ , $\beta \Vdash H\sigma$. Given a formula $H \in \mathcal{L}$ and a \mathcal{KALC} -model $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ for \mathcal{L} , we write $\underline{K} \Vdash H$ (*H is valid in K*) if, for every $\alpha \in P$ and every α -substitution σ , $\alpha \Vdash H\sigma$ in \underline{K} . It is easy to check that, given an open formula $x : A$, $\underline{K} \Vdash x : A$ iff $\underline{K} \Vdash A$. Given a set of formulas $\Gamma \subseteq \mathcal{L}$ and $H \in \mathcal{L}$, H is a *K-logical consequence*

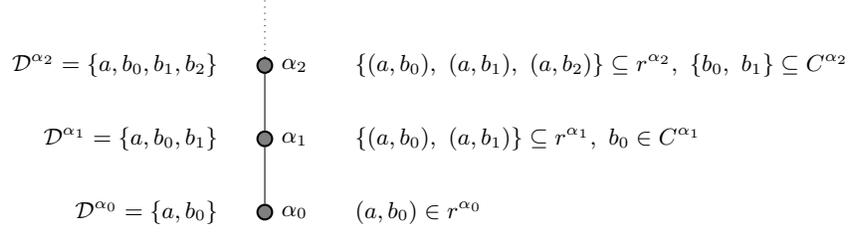


Fig. 1. A counter-model for Kur.

of Γ , denoted $\Gamma \stackrel{\underline{K}}{\models} H$, iff for every \mathcal{KALC} -model $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ for \mathcal{L} , for every $\alpha \in P$ and every α -substitution σ , $\alpha \Vdash \Gamma \sigma$ implies $\alpha \Vdash H \sigma$. A formula H is \mathcal{KALC} -valid iff $\emptyset \stackrel{\underline{K}}{\models} H$. The logic \mathcal{KALC} is the set of \mathcal{KALC} -valid formulas.

The latter definitions allow us to draw some considerations about our semantics and its relations with classical semantics for \mathcal{ALC} and the constructive semantics described in [5, 11, 12]. First of all, given a classical model \mathcal{M} for \mathcal{L} , let us consider the \mathcal{KALC} -model $\underline{K}_{\mathcal{M}} = \langle \{\rho\}, \{(\rho, \rho)\}, \rho, \iota \rangle$, where $\iota(\rho) = \mathcal{M}$. It is easy to check that $\underline{K}_{\mathcal{M}}$ is equivalent to \mathcal{M} in the following sense: for every closed formula $H \in \mathcal{L}$, $\mathcal{M} \models H$ iff $\rho \Vdash H$ in \underline{K} . Hence, if H does not hold in a classical model \mathcal{M} , $\underline{K}_{\mathcal{M}}$ is a counter-model for H . Thus, if we identify \mathcal{ALC} with the set of formulas valid in every classical model, we immediately get that $\mathcal{KALC} \subseteq \mathcal{ALC}$.

On the other hand, the final elements of a \mathcal{KALC} -model essentially coincide with classical interpretations. Indeed, given a \mathcal{KALC} -model $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ and a final element $\phi \in P$, let $\mathcal{M}_{\phi} = \iota(\phi)$. It is easy to check that, for every closed formula H of \mathcal{L}_{ϕ} , $\mathcal{M}_{\phi} \models H$ iff $\phi \Vdash H$ in \underline{K} . This implies that a formula H is satisfiable in \mathcal{KALC} iff it is classically satisfiable.

Now, let us consider the Kripke structure $\underline{K} = \langle \{\alpha_i\}_{i \in \omega}, \leq, \alpha_0, \iota \rangle$ of Figure 1 where:

- $\alpha_i \leq \alpha_j$ iff $i \leq j$;
- For every α_i :
 - $\mathcal{D}^{\alpha_i} = \{a, b_0, \dots, b_i\}$;
 - $r^{\alpha_i} = \{(a, b_0), \dots, (a, b_i)\}$;
 - $C^{\alpha_0} = \emptyset$ and $C^{\alpha_i} = \{b_0, \dots, b_{i-1}\}$ for $i > 0$.

We note that \underline{K} is not a \mathcal{KALC} -model: indeed, $\text{Fin}(\alpha_0) = \emptyset$ and hence its poset is not a K-poset. On the other hand \underline{K} satisfies Conditions (K1)–(K4) and belongs to the class of models obtainable by directly adapting the Kripke semantics for Intuitionistic first order logic to description logics. It is easy to check that $\alpha_0 \Vdash a : \forall r. \neg \neg C \rightarrow \neg \neg \forall r. C$ in \underline{K} .

Now, let us consider the axiom-schema¹:

$$(Kur) \quad \forall R. \neg\neg A \rightarrow \neg\neg\forall R.A$$

It is easy to prove the following result:

Theorem 1. *Let K be any instance of the axiom schema Kur in \mathcal{L} and let $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ be a \mathcal{KALC} -model for \mathcal{L} , then $\rho \Vdash K$. \square*

By the above theorem we immediately get that every instance of Kur is \mathcal{KALC} -valid. On the other hand, let us consider the logic \mathcal{I} consisting of the formulas valid over Kripke models with arbitrary posets. Since the model of Figure 1 is a counter-model for Kur, this principle is not valid in \mathcal{I} and, since it is valid in every finite Kripke model, \mathcal{I} fails to have the finite model property. We remark that, by now, we have not proved that \mathcal{KALC} has the finite model property: however we conjecture that this holds.

According with the above considerations, we think that the Kripke-style semantics for description logics obtained by directly translating the Kripke semantics for Intuitionistic first order logic as described in [5] is not adequate in the description logic context.

Now, let us consider the logic \mathcal{CALC}^c of [11, 12]. This logic is based on a semantics directly inspired by the Kripke style semantics for Intuitionistic modal logics. It is not easy to compare \mathcal{CALC}^c directly with \mathcal{KALC} , since the former also addresses paraconsistency issues by defining a paraconsistent negation. Moreover, as the Kur principle is related to the notion of consistency, it is not clear how to transpose it in the paraconsistent context of \mathcal{CALC}^c .

To conclude this section, we remark that the Kur axiom schema corresponds to the first order principle $\forall x. \neg\neg H(x) \rightarrow \neg\neg\forall x.H(x)$. This principle is known in the literature of constructive logics as *Kuroda principle* [7, 14]. Adding this schema to Intuitionistic first order logic **Int**, we get a proper extension of **Int**, that satisfies the *disjunction property* (if $A \vee B$ is provable either A or B is provable) and the *explicit definability property* (if $\exists x.A(x)$ is provable then also $A(t)$ is provable for some term t). An important property of this logic is that a theory T is classically consistent iff it is consistent w.r.t. Kuroda Logic.

4 Tableau calculus for \mathcal{KALC}

In this section we introduce the tableau calculus \mathcal{T} for \mathcal{KALC} . The calculus works on *signed formulas*, namely expressions of the kind **TH**, **FH** or **F_cH** where H is a closed formula of \mathcal{L} . The semantics of formulas extends to signed formulas as follows. Given a \mathcal{KALC} -model $\underline{K} = \langle P, \leq, \rho, \iota \rangle$ for \mathcal{L} , $\alpha \in P$ and a signed formula W of \mathcal{L}_α , α *realizes* W in \underline{K} , and we write $\underline{K}, \alpha \triangleright W$, iff one of the following conditions hold:

¹ Saying that Kur is an axiom schema, we mean that R and A are meta-variables ranging over role names and concepts respectively. An instance of the axiom schema in \mathcal{L} is obtained by replacing R with a role name of \mathcal{L} and A with concept of \mathcal{L} .

- $W = \mathbf{T}H$ and $\alpha \Vdash H$ in \underline{K} ;
- $W = \mathbf{F}H$ and $\alpha \Vdash \neg H$ in \underline{K} ;
- $W = \mathbf{F}_c H$ and for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash \neg H$ in \underline{K} .

We remark that, for a concept formula H , we have $\underline{K}, \alpha \triangleright \mathbf{F}_c H$ iff $\alpha \Vdash \neg H$ in \underline{K} . By the monotonicity property, we get that \mathbf{T} -signed and \mathbf{F}_c -signed formulas are *persistent*: namely, if $\alpha, \beta \in P$ and $\alpha \leq \beta$, then $\underline{K}, \alpha \triangleright \mathbf{T}A$ implies $\underline{K}, \beta \triangleright \mathbf{T}A$, and $\underline{K}, \alpha \triangleright \mathbf{F}_c A$ implies $\underline{K}, \beta \triangleright \mathbf{F}_c A$. Given a set of signed formulas S , its *persistent part* S_p consists of the persistent signed formulas of S , formally:

$$S_p = \{\mathbf{T}H \mid \mathbf{T}H \in S\} \cup \{\mathbf{F}_c H \mid \mathbf{F}_c H \in S\}$$

The tableau calculus \mathcal{T} consists of the rules in Figures 2 and 3. In the rules we use the notation S, H , with S a set of signed formulas and H a signed formula, to denote the set $S \cup \{H\}$, where we assume that $H \notin S$. We remark that the rules \mathbf{F} -concept, $\mathbf{T}\exists$, $\mathbf{F}\forall$ and $\mathbf{F}_c\forall$ (marked with *) introduce in the conclusion a *parameter* $p \in \mathbf{NI}$ which must be new. That is, applying a rule, we instantiate p with an individual name not occurring in the premise of the rule. We also notice that some of the rules, e.g. $\mathbf{F}\rightarrow$ and $\mathbf{F}\forall$, restrict the set S occurring in the premise to its persistent part S_p . The rule *At* can be applied when the premise S only contains \mathbf{F} -signed formulas.

A *proof table* for a finite set of signed formulas S is a tree τ such that:

- the root of τ is S ;
- given a node S' of τ , the successors S'_1, \dots, S'_n of S' in τ are the consequences of an instance of a rule having S' as premise.

A set S of signed formulas is *contradictory* if either $\mathbf{T}\perp \in S$, $\{\mathbf{T}H, \mathbf{F}H\} \subseteq S$, or $\{\mathbf{T}H, \mathbf{F}_c H\} \subseteq S$ for some formula H . Clearly, contradictory sets are not realizable. When all the leaves of a proof table τ are contradictory, we say that τ is *closed*. A formula H is *provable* in \mathcal{T} iff there exists a closed proof table for $\{\mathbf{F}H\}$.

We remark that the calculus \mathcal{T} is essentially inspired by the calculus for Kuroda logic of [10] and by the tableau calculi for Intuitionistic logic [1, 8], with an efficient treatment of duplications. In particular the rules of Figure 3 are related to the treatment of \mathbf{T} -signed implications avoiding duplications on the purely propositional part of the logic. This treatment of implication is a key point in the development of an “efficient” decision procedure for the logic. Finally, we remark that the rule *At* could be reformulated without the side condition without affecting the soundness and completeness of the calculus. However the restricted rule is better if we are interested in an efficient proof search strategy since it reduces the non-determinism of the calculus.

It can be proved that the calculus \mathcal{T} is sound and complete w.r.t. \mathcal{KALC} semantics: the proof can be developed in a way similar to the one of [10] for *Kuroda Logic*.

Theorem 2 (Completeness). *Let Γ be a finite set of closed formulas of \mathcal{L} and let H be a closed formula of \mathcal{L} . $\Gamma \Vdash^k H$ iff there exists a closed proof table for $\{\mathbf{T}A \mid A \in \Gamma\} \cup \{\mathbf{F}H\}$. \square*

$$\begin{array}{c}
\frac{S, \mathbf{T}(A)}{S, \mathbf{T}(t : A), \mathbf{T}(A)} \mathbf{T}\text{-concept} \quad \frac{S, \mathbf{F}(A)}{S, \mathbf{F}(p : A)} \mathbf{F}\text{-concept}^* \quad \frac{S}{S_p} \text{At} \quad \text{if the only } \mathbf{F}\text{-signed} \\
\text{formulas of } S \text{ are} \\
\text{atomic.} \\
\frac{S, \mathbf{T}(t : A \sqcap B)}{S, \mathbf{T}(t : A), \mathbf{T}(t : B)} \mathbf{T}\sqcap \quad \frac{S, \mathbf{F}(t : A \sqcap B)}{S, \mathbf{F}(t : A) \mid S, \mathbf{F}(t : B)} \mathbf{F}\sqcap \quad \frac{S, \mathbf{F}_c(t : A \sqcap B)}{S_p, \mathbf{F}_c(t : A) \mid S_p, \mathbf{F}_c(t : B)} \mathbf{F}_c\sqcap \\
\frac{S, \mathbf{T}(t : A \sqcup B)}{S, \mathbf{T}(t : A) \mid S, \mathbf{T}(t : B)} \mathbf{T}\sqcup \quad \frac{S, \mathbf{F}(t : A \sqcup B)}{S, \mathbf{F}(t : A), \mathbf{F}(t : B)} \mathbf{F}\sqcup \quad \frac{S, \mathbf{F}_c(t : A \sqcup B)}{S, \mathbf{F}_c(t : A), \mathbf{F}_c(t : B)} \mathbf{F}_c\sqcup \\
\text{See Figure 3} \quad \frac{S, \mathbf{F}(t : A \rightarrow B)}{S_p, \mathbf{T}(t : A), \mathbf{F}(t : B)} \mathbf{F}\rightarrow \quad \frac{S, \mathbf{F}_c(t : A \rightarrow B)}{S_p, \mathbf{T}(t : A), \mathbf{F}_c(t : B)} \mathbf{F}_c\rightarrow \\
\frac{S, \mathbf{T}(t : \neg A)}{S, \mathbf{F}_c(t : A)} \mathbf{T}\neg \quad \frac{S, \mathbf{F}(t : \neg A)}{S_p, \mathbf{T}(t : A)} \mathbf{F}\neg \quad \frac{S, \mathbf{F}_c(t : \neg A)}{S_p, \mathbf{T}(t : A)} \mathbf{F}_c\neg \\
\frac{S, \mathbf{T}(t : \exists R.A)}{S, \mathbf{T}((t, p) : R), \mathbf{T}(p : A)} \mathbf{T}\exists^* \\
\frac{S, \mathbf{F}(t : \exists R.A), \mathbf{T}((t, s) : R)}{S, \mathbf{T}((t, s) : R), \mathbf{F}(s : A)} \mathbf{F}\exists \quad \frac{S, \mathbf{F}_c(t : \exists R.A), \mathbf{T}((t, s) : R)}{S, \mathbf{T}((t, s) : R), \mathbf{F}_c(s : A), \mathbf{F}_c(t : \exists R.A)} \mathbf{F}_c\exists \\
\frac{S, \mathbf{T}(t : \forall R.A), \mathbf{T}((t, s) : R)}{S, \mathbf{T}((t, s) : R), \mathbf{T}(s : A), \mathbf{T}(t : \forall R.A)} \mathbf{T}\forall \\
\frac{S, \mathbf{F}(t : \forall R.A)}{S_p, \mathbf{T}((t, p) : R), \mathbf{F}(p : A)} \mathbf{F}\forall^* \quad \frac{S, \mathbf{F}_c(t : \forall R.A)}{S_p, \mathbf{T}((t, p) : R), \mathbf{F}_c(p : A)} \mathbf{F}_c\forall^*
\end{array}$$

Fig. 2. The rules of calculus \mathcal{T}

The classical problems on description logics can be restated in \mathcal{T} as follows:

- *Concept validity*
Given a concept A and a TBox Γ , determine if $\Gamma \models A$. This holds iff there exists a closed proof table for $\{\mathbf{T}(K) \mid K \in \Gamma\} \cup \{\mathbf{F}(A)\}$.
- *Subsumption*
Given a TBox Γ , determine if $\Gamma \models A \sqsubseteq B$ holds. This holds iff there exists a closed proof table for $\{\mathbf{T}(K) \mid K \in \Gamma\} \cup \{\mathbf{F}(A \rightarrow B)\}$.
- *Instance checking*
Given a knowledge base Γ and an ABox assertion H , check if $\Gamma \models H$. This holds iff there exists a closed proof table for $\{\mathbf{T}(K) \mid K \in \Gamma\} \cup \{\mathbf{F}(H)\}$.

As for the problem of *concept satisfiability*, that is the problem to determine if there exists a \mathcal{KALC} -model for a concept A given a TBox Γ , by the considerations made at the end of the previous section, it can be reduced to the analogous problem for \mathcal{ALC} .

$$\frac{S_p, \mathbf{T}(t : A \rightarrow B)}{S_p, \mathbf{F}_c(t : A) \mid S_p, \mathbf{T}(t : B)} \mathbf{T} \rightarrow \text{certain}$$

$$\frac{S, \mathbf{T}(t : A \rightarrow B)}{S, \mathbf{F}(t : A) \mid S, \mathbf{T}(t : B)} \mathbf{T} \rightarrow \text{At} \quad \text{if } A \text{ is atomic or negated}$$

$$\frac{S, \mathbf{T}(t : (A \sqcap B) \rightarrow C)}{S, \mathbf{T}(t : A \rightarrow (B \rightarrow C))} \mathbf{T} \rightarrow \sqcap \qquad \frac{S, \mathbf{T}(t : (A \sqcup B) \rightarrow C)}{S, \mathbf{T}(t : A \rightarrow C), \mathbf{T}(t : B \rightarrow C)} \mathbf{T} \rightarrow \sqcup$$

$$\frac{S, \mathbf{T}(t : (A \rightarrow B) \rightarrow C)}{S, \mathbf{F}(t : A \rightarrow B), \mathbf{T}(t : B \rightarrow C) \mid S, \mathbf{T}(t : C)} \mathbf{T} \rightarrow \rightarrow$$

$$\frac{S, \mathbf{T}(t : \exists R. A \rightarrow B), \mathbf{T}((t, s) : R)}{S, \mathbf{T}(s : A), \mathbf{T}(t : B), \mathbf{T}((t, s) : R) \mid S, \mathbf{T}((t, s) : R), \mathbf{F}(s : A), \mathbf{T}(t : \exists R. A \rightarrow B)} \mathbf{T} \rightarrow \exists$$

$$\frac{S, \mathbf{T}(t : \forall R. A \rightarrow B)}{S, \mathbf{F}(t : \forall R. A), \mathbf{T}(t : \forall R. A \rightarrow B) \mid S, \mathbf{T}(t : B)} \mathbf{T} \rightarrow \forall$$

Fig. 3. Rules for $\mathbf{T} \rightarrow$

Example 1. We consider the following knowledge base describing associations between food and wines, inspired by the classical example of [4]. The TBox \mathbb{T}_w introduces the basic properties of food and wines:

$$(Ax_1) \quad \text{FOOD} \sqsubseteq \exists \text{goesWith.COLOR}$$

$$(Ax_2) \quad \text{COLOR} \sqsubseteq \exists \text{isColorOf.WINE}$$

Ax_1 , which is equivalent to $\text{FOOD} \rightarrow \exists \text{goesWith.COLOR}$, expresses the fact that every food has associated a correct wine color and Ax_2 , equivalent to $\text{COLOR} \rightarrow \exists \text{isColorOf.WINE}$, expresses the fact that there exists at least one wine for each wine color. The ABox A_w specifies the following instances and associations:

barolo:WINE	fish:FOOD	red:COLOR
chardonnay:WINE	meat:FOOD	white:COLOR
(red,barolo):isColorOf	(fish,white):goesWith	
(white,chardonnay):isColorOf	(meat,red):goesWith	

Now, let $\bar{\mathbb{T}}_w = \{\mathbf{T}H \mid H \in \mathbb{T}_w\}$ and $\bar{A}_w = \{\mathbf{T}H \mid H \in A_w\}$. Figure 4 provides an example of instance checking for the formula:

$$\text{meat} : \text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE})$$

$\overline{T}_w, \overline{A}_w, \mathbf{F}(\text{meat}:\text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))$		$\mathbf{T}\text{-concept } (\mathbf{T}(Ax_1), \mathbf{T}(Ax_2))$
$\overline{T}_w, \overline{A}_w, W_1 \equiv \mathbf{T}(\text{meat}:\text{FOOD} \rightarrow \exists \text{goesWith} . \text{COLOR}),$ $W_2 \equiv \mathbf{T}(\text{red}:\text{COLOR} \rightarrow \exists \text{isColorOf} . \text{WINE}),$ $\mathbf{F}(\text{meat}:\text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))$		$\mathbf{F}\rightarrow$
$\overline{T}_w, \overline{A}_w, W_1, W_2, \mathbf{T}(\text{meat}:\text{FOOD}),$ $W_3 \equiv \mathbf{F}(\text{meat}:\exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))$		$\mathbf{T}\rightarrow \text{An}(W_1)$
$\overline{T}_w, \overline{A}_w, W_2, \mathbf{T}(\text{meat}:\text{FOOD}), W_3$ $\mathbf{F}(\text{meat}:\text{FOOD})$	$\overline{T}_w, \overline{A}_w, W_2, \mathbf{T}(\text{meat}:\text{FOOD}), W_3$ $\mathbf{T}(\text{meat}:\exists \text{goesWith} . \text{COLOR})$	$\mathbf{F}\exists(W_3)$
$X \mid \overline{T}_w, \overline{A}_w, W_2, \mathbf{T}(\text{meat}:\text{FOOD}), \mathbf{T}(\text{meat}:\exists \text{goesWith} . \text{COLOR}),$ $\mathbf{T}(\text{meat}, \text{red}) : \text{goesWith}, \mathbf{F}(\text{red}:\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE})$		$\mathbf{F}\sqcap$
$X \mid \overline{T}_w, \overline{A}_w, W_2, \mathbf{T}(\text{meat}:\text{FOOD}),$ $\mathbf{T}(\text{meat}:\exists \text{goesWith} . \text{COLOR}),$ $\mathbf{T}(\text{meat}, \text{red}) : \text{goesWith},$ $\mathbf{F}(\text{red}:\text{COLOR})$	$\overline{T}_w, \overline{A}_w, W_2, \mathbf{T}(\text{meat}:\text{FOOD}),$ $\mathbf{T}(\text{meat}:\exists \text{goesWith} . \text{COLOR}),$ $\mathbf{T}((\text{meat}, \text{red}) : \text{goesWith}),$ $\mathbf{F}(\text{red}:\exists \text{isColorOf} . \text{WINE})$	$\mathbf{T}\rightarrow \text{An}(W_2)$
$X \mid X \mid \overline{T}_w, \overline{A}_w, \mathbf{T}(\text{meat}:\text{FOOD}),$ $\mathbf{T}(\text{meat}:\exists \text{goesWith} . \text{COLOR}),$ $\mathbf{T}((\text{meat}, \text{red}) : \text{goesWith})$ $\mathbf{F}(\text{red}:\exists \text{isColorOf} . \text{WINE}),$ $\mathbf{F}(\text{red}:\text{COLOR})$	$\overline{T}_w, \overline{A}_w, Ax_2, \mathbf{T}(\text{meat}:\text{FOOD}),$ $\mathbf{T}(\text{meat}:\exists \text{goesWith} . \text{COLOR}),$ $\mathbf{T}((\text{meat}, \text{red}) : \text{goesWith})$ $\mathbf{F}(\text{red}:\exists \text{isColorOf} . \text{WINE}),$ $\mathbf{T}(\text{red}:\exists \text{isColorOf} . \text{WINE})$	

Fig. 4.

Another example is provided by the closed proof table of Figure 5. This proof table proves the subsumption problem for:

$$\text{FOOD} \sqsubseteq \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE})$$

We remark that, for the sake of conciseness, in the first line of the proof in Figure 4 we simultaneously apply the rules $\mathbf{T}\text{-concept}$ to $\mathbf{T}(Ax_1)$ and $\mathbf{T}(Ax_2)$. In the proofs, we specify with $r(H)$ the fact that a rule r is applied to the signed formula H whenever this is not obvious from the context. We denote with X the closed branches of the proof and we underline the signed formulas causing clashes. \diamond

As shown in Figure 6, our calculus allows to exhibit a proof of general instances of the Kuroda principle. We remark that the application of the $\mathbf{F}_c\forall$ rule is essential in order to get a closed proof table. In fact, the $\mathbf{F}_c\forall$ rule directly corresponds to the condition on the final states. That is, without the condition on the partial order, this rule would not be sound, whereas this would be the case for its intuitionistic counterpart:

$$\frac{S, \mathbf{F}_c(t : \forall R.A)}{S_p, \mathbf{T}((t, p) : R), \mathbf{F}(p : A)} \mathbf{F}_c\forall'$$

$$\begin{array}{c}
\frac{\overline{\mathbf{T}}_w, \mathbf{F}(\text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))}{\overline{\mathbf{T}}_w, \mathbf{F}(x : \text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))} \mathbf{F}\text{-concept} \\
\frac{\overline{\mathbf{T}}_w, \mathbf{F}(x : \text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))}{\overline{\mathbf{T}}_w, Z_1 \equiv \mathbf{T}(x : \text{FOOD} \rightarrow \exists \text{goesWith} . \text{COLOR})} \mathbf{T}\text{-concept}(\mathbf{T}(A_{x_1})) \\
\frac{\overline{\mathbf{T}}_w, Z_1 \equiv \mathbf{T}(x : \text{FOOD} \rightarrow \exists \text{goesWith} . \text{COLOR})}{\mathbf{F}(x : \text{FOOD} \rightarrow \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))} \mathbf{F}\rightarrow \\
\frac{\overline{\mathbf{T}}_w, Z_1, \mathbf{T}(x : \text{FOOD}),}{W_3 \equiv \mathbf{F}(x : \exists \text{goesWith} . (\text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE}))} \mathbf{T}\rightarrow \text{An}(Z_1) \\
\frac{\overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), W_3, \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), W_3,}{\mathbf{F}(x : \text{FOOD}) \quad \mathbf{T}(x : \exists \text{goesWith} . \text{COLOR})} \mathbf{T}\exists \\
\frac{X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), W_3,}{\mathbf{T}((x, p) : \text{goesWith}), \mathbf{T}(p : \text{COLOR})} \mathbf{F}\exists(W_3) \\
\frac{X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \mathbf{T}((x, p) : \text{goesWith}), \mathbf{T}(p : \text{COLOR}),}{\mathbf{F}(p : \text{COLOR} \sqcap \exists \text{isColorOf} . \text{WINE})} \mathbf{F}\sqcap \\
\frac{X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}),}{\mathbf{T}((x, p) : \text{goesWith}), \quad \mathbf{T}((x, p) : \text{goesWith}), \mathbf{T}(p : \text{COLOR}),} \mathbf{T}\text{-concept}(\mathbf{T}(A_{x_2})) \\
\frac{X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \mathbf{T}((x, p) : \text{goesWith}),}{X \quad X \quad \mathbf{T}(p : \text{COLOR}), \mathbf{F}(p : \exists \text{isColorOf} . \text{WINE})} \mathbf{T}\rightarrow \text{An}(\mathbf{T}(Z_2)) \\
\frac{X \quad X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \mathbf{T}((x, p) : \text{goesWith}),}{\mathbf{T}(p : \text{COLOR}), \mathbf{F}(p : \exists \text{isColorOf} . \text{WINE})} \mathbf{T}\rightarrow \text{An}(\mathbf{T}(Z_2)) \\
\frac{X \quad X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \mathbf{T}((x, p) : \text{goesWith}),}{\mathbf{F}(p : \exists \text{isColorOf} . \text{WINE}), \quad \mathbf{F}(p : \exists \text{isColorOf} . \text{WINE}),} \mathbf{T}\rightarrow \text{An}(\mathbf{T}(Z_2)) \\
\frac{X \quad X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \mathbf{T}((x, p) : \text{goesWith}), \mathbf{T}(p : \text{COLOR}),}{\mathbf{F}(p : \exists \text{isColorOf} . \text{WINE}), \quad \mathbf{T}(p : \exists \text{isColorOf} . \text{WINE})} \mathbf{T}\rightarrow \text{An}(\mathbf{T}(Z_2)) \\
\frac{X \quad X \quad \overline{\mathbf{T}}_w, \mathbf{T}(x : \text{FOOD}), \mathbf{T}((x, p) : \text{goesWith}), \mathbf{T}(p : \text{COLOR}),}{\mathbf{F}(p : \exists \text{isColorOf} . \text{WINE}), \quad \mathbf{T}(p : \exists \text{isColorOf} . \text{WINE})} \mathbf{T}\rightarrow \text{An}(\mathbf{T}(Z_2))
\end{array}$$

Fig. 5.

However, it can be shown that this intuitionistic version of the rule would not allow to get a closed proof table for the above Kuroda instance.

To conclude this section let us discuss the constructivity of the logic \mathcal{KALC} . First of all, we remark that the rule $\mathbf{F}\sqcup$ of \mathcal{T} can be replaced by the following rules:

$$\frac{S, \mathbf{F}(t : A \sqcup B)}{S, \mathbf{F}(t : A)} \mathbf{F}\sqcup_1 \qquad \frac{S, \mathbf{F}(t : A \sqcup B)}{S, \mathbf{F}(t : B)} \mathbf{F}\sqcup_2$$

Namely, the calculus \mathcal{T}' consisting of the rules of \mathcal{T} , where the rule $\mathbf{F}\sqcup$ is replaced by the rules $\mathbf{F}\sqcup_1$ and $\mathbf{F}\sqcup_2$, is still sound and complete with respect to the constructive consequence relation \models^k . However, we have not included these rules in \mathcal{T} as they would increase the non-determinism in the proof search. Using \mathcal{T}' , we can directly prove that \mathcal{KALC} meets the disjunction property. Indeed, if $A \sqcup B \in \mathcal{KALC}$, by the completeness theorem, there exists a closed proof table for $\mathbf{F}(A \sqcup B)$. By the rules of the calculus \mathcal{T}' , we deduce that such a proof has

$$\begin{array}{c}
\frac{\mathbf{F}(\forall R. \neg\neg A \rightarrow \neg\neg\forall R.A)}{\mathbf{F}(p : \forall R. \neg\neg A \rightarrow \neg\neg\forall R.A)} \mathbf{F}\text{-concept} \\
\frac{\mathbf{F}(p : \forall R. \neg\neg A \rightarrow \neg\neg\forall R.A)}{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{F}(p : \neg\neg\forall R.A)} \mathbf{F}\rightarrow \\
\frac{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{F}(p : \neg\neg\forall R.A)}{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{T}(p : \neg\forall R.A)} \mathbf{F}\neg \\
\frac{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{T}(p : \neg\forall R.A)}{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{F}_c(p : \forall R.A)} \mathbf{T}\neg \\
\frac{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{F}_c(p : \forall R.A)}{\mathbf{T}(p : \forall R. \neg\neg A), \mathbf{T}((p, q) : R), \mathbf{F}_c(q : A)} \mathbf{F}_c\forall \\
\frac{\mathbf{T}((p, q) : R), \mathbf{F}_c(q : A), \mathbf{T}(q : \neg\neg A)}{\mathbf{T}((p, q) : R), \mathbf{F}_c(q : A), \mathbf{F}_c(q : \neg A)} \mathbf{T}\forall \\
\frac{\mathbf{T}((p, q) : R), \mathbf{F}_c(q : A), \mathbf{F}_c(q : \neg A)}{\mathbf{T}((p, q) : R), \mathbf{F}_c(q : A), \mathbf{T}(q : A)} \mathbf{T}\neg \\
\frac{\mathbf{T}((p, q) : R), \mathbf{F}_c(q : A), \mathbf{T}(q : A)}{\mathbf{T}((p, q) : R), \mathbf{F}_c(q : A), \mathbf{T}(q : A)} \mathbf{F}_c\neg
\end{array}$$

Fig. 6.

one of the following forms:

$$\begin{array}{c}
\frac{\mathbf{F}(A \sqcup B)}{\mathbf{F}(p : A \sqcup B)} \mathbf{F}\text{-concept} \\
\frac{\mathbf{F}(p : A \sqcup B)}{\mathbf{F}(p : A)} \mathbf{F}\sqcup_1 \\
\frac{\mathbf{F}(p : A)}{\Pi_1}
\end{array}
\qquad
\begin{array}{c}
\frac{\mathbf{F}(A \sqcup B)}{\mathbf{F}(p : A \sqcup B)} \mathbf{F}\text{-concept} \\
\frac{\mathbf{F}(p : A \sqcup B)}{\mathbf{F}(p : B)} \mathbf{F}\sqcup_2 \\
\frac{\mathbf{F}(p : B)}{\Pi_2}
\end{array}$$

where Π_1 and Π_2 stand for closed proof tables. Now, let us suppose that the first case holds. This means that we can build a closed proof table for $\mathbf{F}(A)$ as:

$$\frac{\mathbf{F}(A)}{\mathbf{F}(p : A)} \mathbf{F}\text{-concept} \\
\frac{\mathbf{F}(p : A)}{\Pi_1}$$

Since we can build a similar closed proof table even in the second case, the disjunction property holds.

Theorem 3 (Disjunction property). *Let $A \sqcup B \in \mathcal{L}$. If $A \sqcup B \in \mathcal{KALC}$, then either $A \in \mathcal{KALC}$ or $B \in \mathcal{KALC}$. \square*

5 Conclusions and future work

We have presented a constructive semantics for description logics inspired by the Kripke semantics for Intuitionistic first order logic. \mathcal{KALC} semantics differs from the direct translation of the Kripke semantics for Intuitionistic first order logic described in [5], since it restricts to Kripke models that force any instance of the axiom schema Kur. This semantics has a close relationship with \mathcal{BCDL} [6],

since the latter can be compared to the fragment of \mathcal{KALC} without implication. Moreover, it can be shown that Kur also holds in \mathcal{BCDL} .

We have also introduced a tableau calculus that is sound and complete w.r.t. \mathcal{KALC} semantics. As for the future work, we plan to prove that this logic has the finite model property, in order to use this calculus to give rise to an algorithm for \mathcal{KALC} decidability. Furthermore, we are interested in a closer examination of \mathcal{KALC} relationship with \mathcal{BCDL} , in order to define a tableau calculus for this logic.

References

1. A. Avellone, G. Fiorino, and U. Moscato. Optimization techniques for propositional intuitionistic logic and their implementation. *Theoretical Computer Science*, 409(1):41–58, 2008.
2. 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.
3. L. Bozzato, M. Ferrari, C. Fiorentini, and G. Fiorino. A constructive semantics for \mathcal{ALC} . In D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, A.-Y. Turhan, and S. Tessaris, editors, *2007 International Workshop on Description Logics*, volume 250 of *CEUR Proceedings*, pages 219–226, 2007.
4. R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L.A. Resnick, and A. Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In J. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 401–456. Morgan-Kaufmann, 1991.
5. V. de Paiva. Constructive description logics: what, why and how. Technical report, Xerox Parc, 2003.
6. M. Ferrari, C. Fiorentini, and G. Fiorino. Bcdl: Basic constructive description logic. Submitted to *Journal of Automated Reasoning*, 2009.
7. D.M. Gabbay. *Semantical Investigations in Heyting's Intuitionistic Logic*. Reidel, Dordrecht, 1981.
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. K. Kaneiwa. Negations in description logic - contraries, contradictories, and sub-contraries. In *In Proceedings of the 13th International Conference on Conceptual Structures (ICCS '05)*, pages 66–79. Kassel University Press, 2005.
10. 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.
11. S.P. Odintsov and H. Wansing. Inconsistency-tolerant description logic. Motivation and basic systems. In V. Hendricks and J. Malinowski, editors, *Trends in Logic. 50 Years of Studia Logica*, pages 301–335. Kluwer Academic Publishers, Dordrecht, 2003.
12. S.P. Odintsov and H. Wansing. Inconsistency-tolerant description logic. Part II: A tableau algorithm for \mathcal{CALC}^C . *J. Applied Logic*, 6(3):343–360, 2008.
13. M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.
14. A.S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.

IDUM a Logic-Based System for e-Tourism

Giuliano Candreva¹, Gianfranco De Franco², Dino De Santo¹, Carmine Donato³,
Antonella Dimasi⁴, Giovanni Grasso⁵, Salvatore Maria Ielpa⁵, Salvatore Iiritano⁴,
Nicola Leone⁵, and Francesco Ricca⁵

¹ASPIdea Software farm, Via Ungaretti 27, 87036 Rende (CS), Italy
giuliano.candreva@infotouch.it

²Top Class srl - Travel Agency, Via Busento 21, 87036 Rende (CS), Italy
gianfranco.defranco@gmail.com, dino@topclassturismo.com

³Spin srl - Consorzio di Ricerca in Tecnologie dell'Informazione e della Comunicazione,
Via degli Stadi 22/F, 87100 Cosenza, Italy
donato@consorziospin.it

⁴Exeura Srl, Via Pedro Alvares Cabrai - C.da Lecco 87036 Rende (CS), Italy
{antonella.dimasi,salvatore.iiritano}@exeura.com

⁵Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{grasso,s.ielpa,leone,ricca}@mat.unical.it

Abstract. In this paper we present the IDUM system, a successful application of logic programming to e-tourism. IDUM exploits two technologies that are based on the state-of-the-art ASP system DLV: (i) a system for ontology representation and reasoning, called OntoDLV; and, (ii) *H₂L_EX*, a semantic information extraction tool. The core of IDUM is an ontology which models the domain of the touristic offers. The ontology is automatically populated by extracting the information contained in the touristic leaflets produced by tour operators. A set of specifically devised logic programs is used to reason on the information contained in the ontology for selecting the holiday packages that best fit the customer needs. An intuitive web-based user interface eases the task of interacting with the system for both the customers and the operators of a travel agency.

1 Introduction

In the last few years, the tourism industry strongly modified the marketing strategies with the diffusion of e-tourism portals in the Internet. Big tour operators are exploiting new technologies, such as web portals and e-mails, for both simplifying their advertising strategies and reducing the selling costs. The efficacy of e-tourism solutions is also witnessed by the continuously growing community of e-buyers that prefer to surf the Internet for buying holiday packages. On the other hand, traditional travel agencies are undergoing a progressive loss of marketing competitiveness. This is partially due to the presence of web portals, which basically exploit a new market. Indeed, Internet surfers often like to be engaged in self-composing their holiday by manually searching for flights, accommodation etc. Instead, the traditional selling process, which has its strength on both the direct contact with the customer and the knowledge about the customer habits, is experiencing a reduced efficiency. This can be explained by the increased complexity of matching demand and offer. Indeed, travel agencies receive

thousand of e-mails per day from tour operators containing new pre-packaged offers. Consequently, the employees of the agency cannot know all the available holiday packages (since they cannot analyze all of them). Moreover, customers are more exigent than in the past (e.g. the classic statement “I like the sea” might be enriched by “I like snorkeling”, or “please find an hotel in Cortina” might be followed by “featuring beauty and fitness center”) and they often do not declare immediately all their preferences and/or constraints (like, e.g., budget limits, preferred transportation mean or accommodation etc.). This knowledge of customers preferences plays a central role in the traditional selling process, but matching this information with the large unstructured e-mail database is both difficult to be carried out in a precise way and time consuming. Consequently, the seller is often unable to find in a short time the best possible solution.

The goal of the IDUM project is to devise a system that addresses above-mentioned causes of inefficiency by offering:

- (i) an automatic extraction and classification of the incoming touristic offers (so that they are immediately available for the seller), and
- (ii) an “intelligent” search that combines knowledge about users preferences with geographical information, and matches user needs with the available offers.

We reach the goal by exploiting computational logic, and, in particular, Answer Set Programming (ASP) [1]. ASP is a powerful logic programming language, which is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, ASP can represent in a fully declarative way *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [2]. In particular, the core functionalities of IDUM are based on two technologies¹ relying on the state-of-the-art ASP system DLV [3]:

- OntoDLV [4, 5] a powerful ASP-based ontology representation and reasoning system; and,
- $\text{H}\nu\text{L}\varepsilon\text{X}$ [6–8], an advanced tool for semantic information-extraction from unstructured or semi-structured documents.

More in detail, in the IDUM system, behind the web-based user interface (that can be used by both employees of the agency and customers), there is an “intelligent” core that exploits an OntoDLP ontology for both modeling the domain of discourse (i.e., geographic information, user preferences, and touristic offers, etc.) and storing the available data. The ontology is automatically populated by extracting the information contained in the touristic leaflets produced by tour operators. It is worth noting that offers are mostly received by the travel agency in a dedicated e-mail account. Moreover, the received e-mails are human-readable, and the details are often contained in email-attachments of different format (plain text, pdf, gif, or jpeg files) and structure that might contain a mix of text and images. The $\text{H}\nu\text{L}\varepsilon\text{X}$ system allows for automatically processing the received contents, and to populate the ontology with the data extracted

¹ Both systems are developed by Exeura srl, a technology company working on analytics, data mining, and knowledge management, that is working on their industrialization finalized to commercial distribution.

from touristic leaflets. Once the information is loaded on the ontology, the user can perform an “intelligent” search for selecting the holiday packages that best fit the customer needs. IDUM tries to mimic the behavior of the typical employee of a travel agency by exploiting a set of specifically devised logic programs that “reason” on the information contained in the ontology.

In the remainder of the paper, we first introduce the employed ASP-based technologies; then, in Section 3, we describe how the crucial tasks have been implemented. We show the architecture of the IDUM system in Section 4, and we draw the conclusion in Section 6.

2 Underlying Logic-based technologies

The core functionalities of the e-tourism systems IDUM were based on two technologies relying on the DLV system [3]: OntoDLV [4, 5] a powerful ASP-based ontology representation and reasoning system; and, $\text{H}\ell\text{L}\varepsilon\text{X}$ [6–8], an advanced tool for semantic information-extraction from unstructured or semi-structured documents.

2.1 The OntoDLV System

Traditional ASP is not well-suited for ontology specifications, since it does not directly support features like classes, taxonomies, individuals, etc. Moreover, ASP systems are a long way from comfortably enabling the development of industry-level applications, mainly because they lack important tools for supporting programmers. All the above-mentioned issues were addressed in OntoDLV [4, 5], a system for ontologies specification and reasoning. Indeed, by using OntoDLV, domain experts can create, modify, store, navigate, and query ontologies; and, at the same time, application developers can easily develop their own knowledge-based applications on top of OntoDLV, by exploiting a complete Application Programming Interface [9]. OntoDLV implements a powerful logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) ASP with all the main ontology constructs including classes, inheritance, relations, and axioms. In OntoDLP, a *class* can be thought of as a collection of individuals that share some features. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute has a name and a type, which is, in truth, a class. This allows for the specification of *complex objects* (objects made of other objects). Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*). Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type). OntoDLP relations are strongly typed while in ASP relations are represented by means of simple flat predicates. Importantly, OntoDLP supports two kind of classes and relations: (base) classes and (base) relations, that correspond to basic facts (that can be stored in a database); and *collection* classes and *intensional* relations, that correspond to facts that can be inferred by logic programs; in particular, *collection classes* are mainly intended for object reclassification (i.e., for classifying individuals of an ontology repeatedly). For instance, the following

statement declares a class modeling customers, which has six attributes, namely: *firstName*, *lastName*, and *status* of type string; *birthDate* of type Date; a positive integer *childNumber*, and *job* which contains an instance of another class called Job.

```
class Customer (firstName: string, lastName: string,
               birthDate: Date, status: string,
               childNumber: positive integer, job: Job).
```

As in ASP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). This way, the OntoDLP rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, logic programs are organized in *reasoning modules*, to take advantage of the benefits of modular programming. For example, with the following program we single out the pairs of customers having the same age

```
module (CustomersWithTheSameAge) {
    sameAge(C1,C2,D) :- C1: Customer(birthDate:D),
                       C2: Customer(birthDate:D).
}
```

The core of OntoDLV is a rewriting procedure [5] that translates ontologies, and reasoning modules to an equivalent standard ASP program which, in the general case, runs on state-of-the art ASP system DLV [3].

OntoDLV features an advanced persistency manager that allows one to store ontologies transparently both in text files and internal relational databases; furthermore, powerful type-checking routines are able to analyze ontology specifications and single out consistency problems. Importantly, if the rewritten program is stratified and non disjunctive [1, 10, 11] (and the input ontology resides in relational databases) the evaluation is carried out directly in mass memory by exploiting a specialized version of the same system, called DLV^{DB} [17]. Note that, since class and relation specifications are rewritten into stratified and non-disjunctive programs, queries on ontologies can always be evaluated by exploiting a DBMS. This makes the evaluation process very efficient, and allows the knowledge engineer to formulate queries in a language more expressive than SQL. Clearly, more complex reasoning tasks (whose complexity is NP/co-NP, and up to Σ_2^P/Π_2^P) are dealt with by exploiting the standard DLV system instead.

2.2 The H₂L ϵ X System

H₂L ϵ X [6–8] is an advanced system for ontology-based information extraction from semi-structured and unstructured documents, that has been already exploited in many relevant real-world applications. The H₂L ϵ X system implements a semantic approach to the information extraction problem based on a new-generation semantic conceptual model exploiting:

- ontologies as knowledge representation formalism;
- a general document representation model able to unify different document formats (html, pdf, doc, ...);

- the definition of a formal attribute grammar able to describe, by means of declarative rules, objects/classes w.r.t. a given ontology.

Most of the existing approaches do not work in a semantical way and they are not independent from the particular type of document they process. Contrariwise, the approach implemented in *H₂L₂E₂X* confirms that it is possible recognize, extract and structure relevant information from heterogeneous sources.

H₂L₂E₂X is based on OntoDLP for describing ontologies, because this language perfectly fits the definition of semantic extraction rules.

Regarding the unified document representation, the idea is that a document (un-structured or semi-structured) can be seen as a suitable arrangement of objects in a two-dimensional space. Each object has an own semantics, is characterized by some attributes and it is located in a two-dimensional area of the document called *portion*. A portion is defined as a rectangular area univocally identified by four cartesian coordinates of two opposite vertices. Each portion “contains” one or more objects and an object can be recognized in different portions.

The language of *H₂L₂E₂X* is founded on the concept of *ontology descriptor*. A “descriptor” looks like a production rule in a formal attribute grammar, where syntactic items are replaced by ontology elements, and where extensions for managing two-dimensional objects are added. Each descriptor allows to describe: (i) an ontology object in order to recognize it in a document; or (ii) how to “generate” a new object that, in turn, may be added to the original ontology.

In the following example, we report a part of an ontology and two *H₂L₂E₂X* descriptors that identify the instances ‘sunny’ (or ‘windy’) of the class `weatherTerm`.

```
// core ontology (provided by the system)
class hiToken (parent:hiBox, value:string).
class hiBox (paragraph:[hiToken]).
relation hasLemma (token:hiToken, lemma:string, tag:hiPosTag).
// domain ontology (provided by the user)
class weatherTerm (descr:string).
'sunny': weatherTerm (descr:"Sunny").
'wind': weatherTerm (descr:"Windy").

// descriptors for existing objects
<'sunny'> -> <T:hiToken(), hasLemma(token:T, lemma:"sunny")>.
<'windy'> -> <T:hiToken(), hasLemma(token:T, lemma:"wind")>.
```

Here, an instance of `weatherTerm` is discovered in a document if the latter contains a portion associated to a basic object of type *hiToken* (collecting basic objects that represent all the substrings) satisfying the condition that its lemma is “sunny” (or “windy”). Note that an object may also have more than one descriptor, thus allowing one to recognize the same kind of information when it is presented in different ways.

The *H₂L₂E₂X* system has been applied to different real-world problems [6–8] such as: (i) Information extraction from balance sheets; (ii) Information extraction from news; (iii) Information extraction from clinical documents.

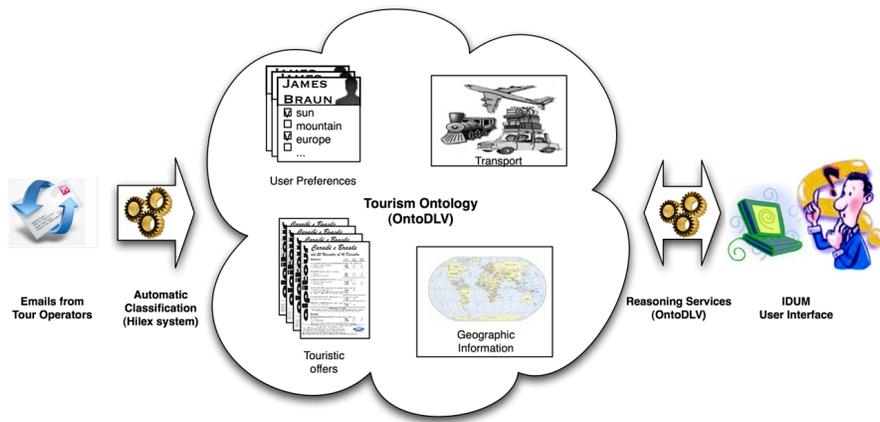


Fig. 1. The IDUM System.

3 The IDUM System

In this section we describe the core of the IDUM system and its innovative features based on ASP. IDUM is an e-tourism system, conceived for classifying and driving the search of touristic offers for both travel agencies operators and their customers.

IDUM, like other existing portals, has been equipped with a proper (web-based) user interface; but, behind the user interface there is an “intelligent” core that exploits knowledge representation and reasoning technologies based on ASP. In IDUM (see Figure 1), the information regarding the touristic offers provided by tour operators is received by the system as a set of e-mails. Each e-mail might contain plain text and/or a set of leaflets, usually distributed as pdf or image files which store the details of the offer (e.g., place, accommodation, price etc.). Leaflets are devised to be human-readable, might contain both text and images, and usually do not have the same structure. E-mails (and their content) are automatically processed by using the *H₂L₂X* system, and the extracted data about touristic offers is used to populate an OntoDLP ontology that models the domain of discourse: the “*tourism ontology*”. The resulting ontology is then analyzed by exploiting a set of reasoning modules (ASP programs) combining the extracted data with the knowledge regarding places (geographical information) and users (user preferences) already present in the tourism ontology. The system mimics the typical deductions made by a travel agency employee for selecting the most appropriate answers to the user needs.

In the following sections, we briefly describe the tourism ontology and the implementation of the above-mentioned ASP-based features.

3.1 The Tourism Ontology

The “tourism ontology” has been specified by analyzing the nature of the input (we studied the characteristics of several touristic leaflets) with the cooperation of the staff

```

class Customer (firstName: string, lastName: string,
    birthDate: Date, status: string,
    childNumber: positive integer, job: Job).
relation CustomerPrefersTrip ( cust:Customer, kind: TripKind ).
relation CustomerPrefersMeans ( cust:Customer,
    means: TransportationMean ).
...

class Place ( description:string ).
intentional relation Contains ( pl1:place, pl2:place )
{ Contains(P1,P2) :- Contains(P1,P3), Contains(P3,P2).
  Contains('Europe', 'Italy'). Contains('Italy', 'Sicily').
  Contains('Sicily', 'Palermo'). ... }

relation PlaceOffer( place: place, kind: tripKind ).
relation SuggestedPeriod ( place:place, period: positive integer ).
relation BadPeriod ( place:place, period: positive integer ).

class TouristicOffer( start: Place, destination: Place,
    kind: TripKind, means: TransportationMean,
    cost: positive integer, fromDay: Date, toDay: Date,
    maxDuration: positive integer, deadline: Date,
    uri: string, tourOperator: TourOperator ).

class TransportationMean ( description: string ).
class TripKind ( description: string ).
...

```

Fig. 2. Main entities of the touristic ontology.

of a real touristic agency, which has been repeatedly interviewed. In this way, we could model the key entities that describe the process of organizing and selling a complete holiday package. In particular, the “tourism ontology” models all the required information: user profile, geographic information, kind of holiday, transportation means, etc. In Figure 2, we report some of the most relevant classes and relations that constitute the tourism ontology. In detail, the class *Customer* allows one to model the personal information of each customer, while a number of relations is used to model user preferences, like *CustomerPrefersTrip* and *CustomerPrefersMeans*, that associate each customer to its preferred kind of trip, and its preferred transportation means, respectively. The kind of trip is represented by using a class *TripKind*. Examples of *TripKind* instances are: *safari*, *sea_holiday*, etc. In the same way, e.g. *airplane*, *train*, etc. are instances of the class *TransportationMean*. Geographical information is modeled by means of the class *place*, which has been populated with the information regarding more than a thousand of touristic places. Moreover, each place is associated to a kind of trip by means of the relation *PlaceOffer* (e.g. Kenya offers safari, Sicily offers both sea and sightseeing). Importantly, the natural a *part-of* hierarchy of places is easily modeled by using the intensional relation defining an *Contains*. This allowed us to assert manually only some basic facts and to obtain in a simple yet powerful way all the basic inclusions. Indeed,

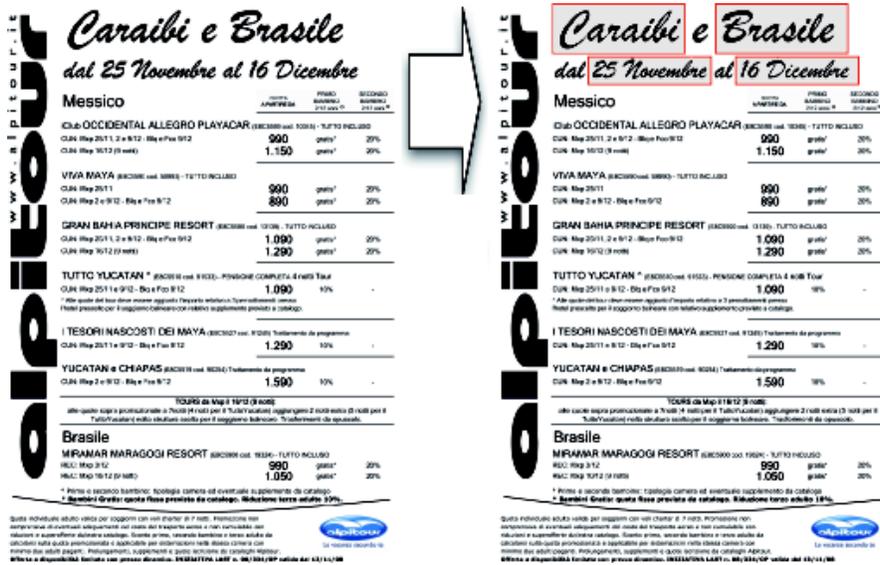


Fig. 3. Extracting offer information.

the full hierarchy is computed by evaluating a rule (that, basically, encodes the transitive closure).

The mere geographic information is, then, enriched by other information that is usually exploited by travel agency employees for selecting a travel destination. For instance, one might suggest to avoid sea holidays in winter, or to go in India during the wet monsoon period; whereas, one should be suggested to visit Sicily in summer. This was encoded by means of the two relations *SuggestedPeriod* and *BadPeriod*.

Finally, the *TouristicOffer* class contains an instance for each available holiday package. The instances of this class are added either automatically, by exploiting the $H_2L\epsilon X$ system (as described in the next section), or manually by the personnel of the agency.

3.2 Automatic Extraction of Touristic Offers

Touristic offers are mainly available in digital format, and are received via e-mail. It is usual that more than a hundred emails per day crowd the mail folder of a travel agency, and often the personnel cannot even analyze the entire in-box. This causes a loss of efficiency in the selling process, because some interesting offers might be ignored. Note that most of the information is contained in pdf, gif or jpeg files attached to the e-mail messages, and this strongly limits the efficacy of standard search tools like, e.g., the ones provided by e-mail clients.

To deal with this problem, the IDUM system has been equipped with an automatic classification system based on $H_2L\epsilon X$. Basically, after some pre-processing steps, in

which e-mails are automatically read from the inbox, and their attachments are properly handled (e.g. image files are analyzed by using OCR software), the input is sent to $H_{iL}E_X$. In turn, $H_{iL}E_X$ is able to both extract the information contained in the e-mails and populate the *TouristicOffer* class. This has been obtained by encoding several ontology descriptors (actually, we provided several descriptors for each kind of file received by the agency). For instance, the following descriptor:

```
<TouristicOffer (Destination, Period)> ->
  <X:place(XX)>{Destination:=X;}
  <X:date(XX)>{Period:=X;}
  SEPBX <X:separator()>.
```

allows to extract from the leaflet in Figure 3(a) that the proposed holiday package regards trips to both the Caribbean islands and Brazil. Moreover it also extracts the period in which this trip is offered. The extracted portions are outlined in Figure 3(b). The result of the application of this descriptor are two new instances of the *TouristicOffer* class.

3.3 Personalized Trip Search

The second crucial task carried out in the IDUM system is the personalized trip search. This feature has been conceived to make simpler the task of selecting the holiday packages that best fit the customer needs. We tried to “simulate” the deductions made by an employee of the travel agency in the selling process by using a set of specifically devised logic programs. In a typical scenario, when a customer enters the travel agency, an employee tries to understand her current desires and preferences at first; then, the seller has to match the obtained information with a number of pre-packaged offers. Actually, a number of candidate offers are proposed to the customer and, then, the employee has to understand her needs by interpreting the preferences of the customer. Customer preferences depends on her personal information (age, gender, marital status, lifestyle, budget, etc.), but also on her holiday habits (e.g. she prefers going to the mountains, or she has already been in Italy). Actually, this information has to be elicited by the seller by interviewing the customer, but most of it might be already known by the employee if she is serving an old customer. In this process, what has to be clearly understood for selecting a holiday package fitting the customer needs is resumed in the following four words: *where*, *when*, *how*, and *budget*. Indeed, the seller has to understand *where* the customer desires to go; *when* she can/wishes to leave; how much time she will dedicate to his holiday; which is the preferred transportation mean (*how*); and, finally, the available budget. However, the customer does not directly specify all this information, for example, she can ask for a sea holiday in January but she does not specify a precise place, or she can ask for a kind of trip that is unfeasible in a given period (e.g. winter holiday in Italy in August) In this case the seller has to exploit her knowledge of the world for selecting the right destination and a good offer in a huge list of proposals. This is exactly what the IDUM system does when an user starts a new search. Current needs are specified by filling an appropriate search form (see Figure 4), where some of the key information has to be provided (i.e. where and/or when and/or available money and/or how). Note that, the tourism ontology models both the knowledge of the seller (geographic information and preferred places), and the profile of customers (clearly, in the case of new customers the seller has to fill the profile information, whereas the

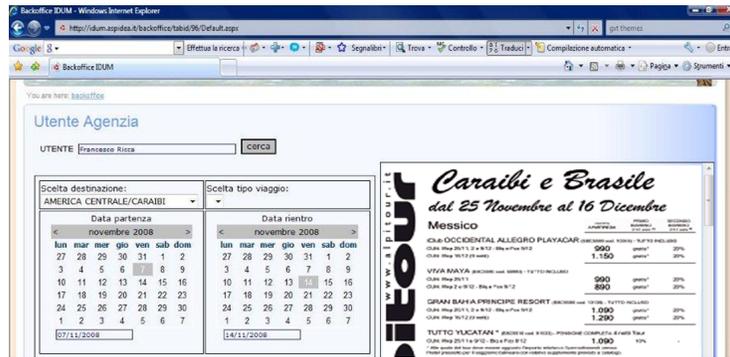


Fig. 4. The IDUM User Interface.

ontology already contains the information regarding old customers); moreover, the extraction process continuously populates the ontology with new touristic offers. Thus, the system, by running a specifically devised reasoning module, combines the specified information with the one available in the ontology, and shows the holiday packages that best fit the customer needs. For example, suppose that a customer specifies the kind of holiday and the period, then the following module selects of holiday packages:

```

module(kindAndPeriod) {
    %detect possible and suggested places
    possiblePlace(P) :- askFor(tripKind:K),
                       PlaceOffer(place:P, kind:K).

    suggestPlace(P) :- possiblePlace(P), askFor(period:D),
                       SuggestedPeriod(place:P1, period:D),
                       not BadPeriod(place:P1, period:D).

    %select possible, alternative and suggestible packages
    possibleOffer(O) :- O:TouristicOffer(destination:P),
                       possiblePlace(P).

    alternativeOffer(O) :- O:TouristicOffer(destination:P),
                           suggestPlace(P).

    suggestOffer(O) :- O:TouristicOffer(destination:P, mean:M),
                       suggestPlace(P), askFor(cust:C),
                       CustomerPrefersMean(cust:C, mean:M).
}

```

The first two rules select: possible places (i.e., the ones that offer the kind of holiday in input); and places to be suggested because they offer the required kind of holiday in the specified period. Finally, the remaining three rules search in the available holiday packages the ones that: offer an holiday that matches the original input (possible offer); are good alternatives in suggested places (alternativeOffer); or match the customer's preferred mean and can be suggested.

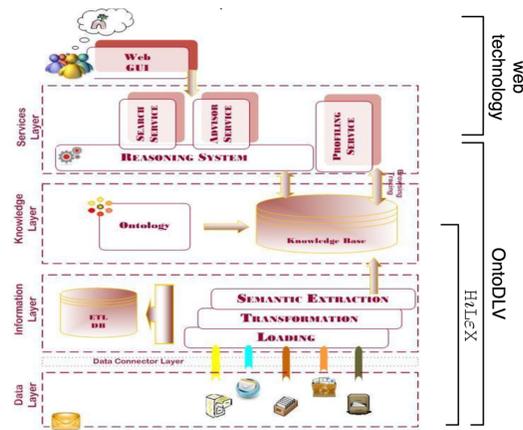


Fig. 5. System Architecture.

4 System Architecture

The architecture of the IDUM system, which is depicted in Figure 5, is made of four layers: *Data Layer*, *Information Layer*, *Knowledge Layer*, and *Service Layer*. In the *Data Layer* the input sources are dealt with. In particular, the system is able to store and handle the most common kind of sources: e-mails, plain text, pdf, gif, jpeg, and HTML files. The *Information Layer* provides ETL (Extraction, Transformation and Loading) functionalities, in particular: in the loading step the documents to be processed are stored in an auxiliary database (that also manages the information about the state of the extraction activities); whereas, in the Transformation step, semi-structured or non-structured documents are manipulated. First the document format is normalized; then, the “bi-dimensional logical representation” is generated (basically, the *H_lL_εX* portions are identified); finally, *H_lL_εX* descriptors are applied in the Semantic Extraction step and ontology instances are recognized within processed documents. The outcome of this process is a set of concept instances, that are recognized by matching semantic patterns, and stored in the core knowledge base of the system where the tourism ontology resides (*Knowledge Layer*). Domain ontology and extracted information are handled by exploiting the Persistency manager of the *OntoDLV* system (see Section 2.1). The *Services Layer* features the profiling service and the intelligent search (see Section 3.3) which implements the reasoning on the core ontology by evaluating in the *OntoDLV* system a set of logic programs. The Graphical User Interface (GUI) can access the system features by interacting with a set of web-services.

5 Related Work

The usage of ontologies for developing e-tourism applications was already studied in the literature [12, 14, 13], and the potential of the application of semantic technology recognized. The architecture of an e-tourism system able to create a touristic package

in a dynamic way is presented in [14]. It is based on an ontology written in OWL-DL. An advantage of the our approach is the possibility of developing ASP programs that reason on the data contained in the ontology for developing complex searches, while there is no accepted solution for combining logic rules and OWL-DL ontologies.

The SPETA system [15], which is based on the ontology of [14], acts as an advisor for tourists. Fundamentally, SPETA follows people who need advising when visiting a new place, and who consequently do not know what is interesting to visit. Both SPETA and IDUM exploit an ontology for building personalized solutions for the users, but the goal of SPETA (offering assistance during travels) is different from that of IDUM.

The E-Tourism Working Group at DERI [16] is developing e-tourism solutions based on the Semantic Web technology. Deri also developed a content management system: OnTourism. The solution is very similar to IDUM, but it is based on the use of Lixto Software which makes information extraction from web pages.

6 Conclusion

In this paper we described a successful example of commercial and practical use of logic programming: the e-tourism system IDUM.

The core of IDUM is an ontology modeling the domain of the touristic offers, which is automatically populated by extracting the information contained in the e-mails sent by tour operators; and an intelligent search tool based on answer set programming is able to search the holiday packages that best fits the customer needs.

The system has been developed under project "IDUM: Internet Diventa Umana" (project n. 70 POR Calabria 2000/2006 Mis. 3.16 Azione D Ricerca e Sviluppo nella Imprese Regionali - Modulo B Voucher Tecnologici) funded by the Calabrian Region. The project team involved five organizations: the Department of Mathematics of the University of Calabria (that has ASP as one of the principal research area), the consortium Spin, Exeura srl (a company working on knowledge management), Top Class srl (a travel agency), and ASPIdea (a software farm specialized in the development of web applications). The members exploited their specific knowledge for developing the innovative features of the system. The strong synergy among partners made possible to push the domain knowledge of the travel agency TopClass in both the ontology and in the reasoning modules. The result is a system that mimic the behavior of a seller of the agency and it is able to search in a huge database of automatically classified offers. IDUM combines the speed of computers with the knowledge of a travel agent for improving the efficiency of the selling process.

The IDUM system was initially conceived for solving the specific problems of a travel agency, and it is currently employed by one of the project partners: Top Class srl. Moreover, we received very positive feedbacks from the market, indeed many travel agents are willing to use the system, and the potential of IDUM has been recognized also by the chair of the Italian touring club, which is the most important Italian association of tour operators.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *NGC* **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. *ACM TODS* **22**(3) (1997) 364–418
3. 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
4. Ricca, F., Gallucci, L., Schindlauer, R., Dell’Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. *Journal of Logic and Computation* (2009)
5. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV⁺ System. *Journal of Applied Logics* **5**(3) (2007) 545–573
6. Manna, M.: Semantic Information Extraction: Theory and Practice. PhD thesis, Dipartimento di Matematica, Università della Calabria, Rende, Cosenza Italia (2008)
7. Ruffolo, M., Manna, M.: HiLeX: A System for Semantic Information Extraction from Web Documents. In: *ICEIS (Selected Papers)*. Volume 3 of LNBIP (2008) 194–209
8. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: *Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation*, Bath, UK (2005) 248–262
9. Gallucci, L., Ricca, F.: Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In: *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA’07)*. (2007) 56–70
10. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective. *AI* **138**(1–2) (2002) 3–38
11. Minker, J.: Overview of Disjunctive Logic Programming. *AMAI* **12** (1994) 1–24
12. Maedche, A., Staab, S.: Applying semantic web technologies for tourism information systems. In: *Proc. of ENTER 2002*, (2002)
13. Martin, H., Katharina, S., Daniel, B.: Towards the semantic web in e-tourism: can annotation do the trick? In: *Proc. of the 14th ECIS 2006*. (2006)
14. Jorge, C.: Combining the semantic web with dynamic packaging systems. In *Proc. of AIKED’06 (2006)*, World Scientific and Engineering Academy and Society 133–138
15. Angel, G.C., Javier, C., Ismael, R., Myriam, M., Ricardo, C.P., Miguel, G.B.J.: Speta: Social pervasive e-tourism advisor. *Telemat. Inf.* **26**(3) (2009) 306–315
16. DERI: Digital Enterprise Research Institute. Technikerstrae 21a. Innsbruck, A.: <http://e-tourism.deri.at/>.
17. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. *TPLP* **8** (2008) 129–165

A Fair Extension of (Soft) Concurrent Constraint Languages

Stefano Bistarelli^{1,2,3} and Paola Campi¹

¹ Department of Science, University “G. d’Annunzio” of Chieti-Pescara, Italy
[bista,camp]@sci.unich.it

² Department of Maths and Informatics, University of Perugia, Italy
bista@dipmat.unipg.it

³ Institute of Informatics and Telematics (IIT), CNR Pisa, Italy
stefano.bistarelli@iit.cnr.it

Abstract. The aim of this paper is to guarantee fair computations in (Soft) Concurrent Constraint languages. We present an extension of the semantics related to the operator of parallelism in order to allow a “fair” selection for the execution of concurrent agents. We define a new operator of Parallelism (\parallel_m) which is able to deal with a finite number (m) of agents. Subsequently we apply the general rule to different fairness notions.

Introduction

This paper contains an extension of the semantics of both Concurrent Constraint (CC) and Soft Concurrent Constraint (SCC) languages related to the operator of parallelism; the aim is to guarantee a fair criterion of selection among parallel agents and to restrict or remove the possible unwanted behavior of a program. In fact, the classical (S)CC parallel operator selects one agent among more parallel enabled agents at time. As the parallelism rule does not provide any criterion about the choose, some agents could be executed more time with respect to the others, or could not be executed at all. We provide a general rule to express fairness with a finite number of agents in the CC language; moreover we apply the rule in three particular notions of fairness: *Carpooling-fairness*, *Weak h-fairness* and *Strong h-fairness* (where the notion of h-fairness is given by [3]).

1 Background and overview of the existing literature

1.1 Fairness in programming languages

Some of the most common notions of fairness in computer science are given by Nissim Francez in [5]: **weak fairness** requires that if an action (or agent) is *continuously enabled*, so it can almost always proceed, then it must *eventually* do so, while **strong fairness** requires that if an action (or agent) can proceed

infinitely often then it must *eventually* proceed (be executed). These notions are available only for infinite computations.

The notions of fairness for infinite computations can be expressed also for finite computations through the notion of *Bounded Fairness* [3].

1.2 Concurrent Constraint Programming

The concurrent constraint (cc) programming paradigm [7] concerns the behavior of a set of concurrent agents with a shared store, which is a conjunction of constraints. Each computation step possibly adds new constraints to the store. The concurrent agents communicate only with the shared store, by either checking if it entails a given constraint (*ask* operation) or adding a new constraint to it (*tell* operation).

Here is the transition rule of the parallelism (see [6] for the complete CC transition system)

$$\frac{\langle A_1, \sigma \rangle \rightarrow \langle A'_1, \sigma' \rangle \quad \langle A_2, \sigma \rangle \rightarrow \langle A'_2, \sigma' \rangle}{\langle A_1 \parallel A_2, \sigma \rangle \rightarrow \langle A'_1 \parallel A'_2, \sigma' \rangle} \text{ parallelism (1)}$$

The parallelism operator works as follow: if agent A_1 with a store σ evolves in agent A'_1 with store σ' , then the parallel execution of the agents A_1 and A_2 evolves in the parallel execution between agents A'_1 and A_2 .

The rule above, does not provide a criterion to establish which agent the scheduler has to select in a parallel execution when only one agent at time can be executed. In fact, it could choose always agent A_1 and never select agent A_2 . We want to avoid these possible situations by providing a fair criterion for the selection of concurrent agents. To obtain this, we use a new parallel operator (see Section 2).

Soft Concurrent Constraint. The Soft Concurrent Constraint (SCC) language [2] uses soft constraints that generalize classical constraints by allowing several levels of satisfaction. The framework is based on semirings [1]; the formalism provides suitable operations to combine constraints (\times) and to compare ($+$) the tuples of values and constraints.

The \sqsubseteq operator. A partially ordered set is a pair (D, \sqsubseteq) where D is a set and \sqsubseteq is a binary relation over D (i.e. a subset of $D \times D$), such that: \sqsubseteq is reflexive (i.e. $d \sqsubseteq d \quad \forall d \in D$), antisymmetric (i.e. $d \sqsubseteq e$ and $e \sqsubseteq d$ imply $d = e \quad \forall d, e \in D$) and transitive (i.e. $d \sqsubseteq e \sqsubseteq d'$ implies $d \sqsubseteq d' \quad \forall d, d', e \in D$).

2 The new parallel operator

In order to obtain a Fair version of the parallel execution of a finite number of agents, we modify the parallel operator \parallel in the CC semantic by replacing it

with the \parallel_m operator, which is able to deal with a set of m agents. Since with the classical parallel operator only two concurrent agents are represented in the Parallelism rule, it is not possible to express fairness among more than 2 agents at the same level.

We develop the new transition rule, which is able to operate with different fairness definitions or scheduling algorithm with finite computations. This is possible by introducing an array \mathbf{k} and a set of conditions over \mathbf{k} , $\langle \text{cond}(\mathbf{k}) \rangle$ for the selection of the agent that we want to execute. The array \mathbf{k} (with m elements) allows us to keep track of the actions performed by each agent during the computational steps. In this way we can associate a value $\mathbf{k}[i]$ to each agent A_i . The rule considers among the m agents, the n enabled agents (that we indicate with the notation: $A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow$). This rule can be applied to different notions and metrics of fairness, as it is stated in a general way. According the algorithm or the fairness notion that we want to consider, we can use different assignments for the array \mathbf{k} . The modified parallelism rule is shown in Table 1.

Table 1. \parallel_m - parallelism (1) rule

$$\frac{\begin{array}{c} A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow \\ \langle \text{cond}(\mathbf{k}) \rangle < A_{l_i}, \sigma \rangle \rightarrow < A'_{l_i}, \sigma' \rangle \end{array}}{\langle \parallel_m(A_1, \dots, A_{l_i}, \dots, A_m), \mathbf{k}, \sigma \rangle \rightarrow \langle \parallel_m(A_1, \dots, A'_{l_i}, \dots, A_m), \mathbf{k}', \sigma' \rangle}$$

$$\mathbf{k}[x]' = \begin{cases} \text{assignment 1} & \text{if } x = 1, \dots, m \text{ for not enabled agents} \\ \text{assignment 2} & \text{if } x = l_i \quad \text{for the enabled and executed agent} \\ \text{assignment 3} & \text{if } x = l_j \quad j \neq i \text{ for the enabled and not executed agents} \end{cases}$$

In the next sections, we show how to apply this general rule to different fair algorithms or levels of fairness. The first idea we use, in order to provide a way to establish which of the agents can evolve, comes from a *Fair carpooling scheduling algorithm* [4]; subsequently we apply the new rule to obtain Strong h-fairness and Weak h-fairness (Section 2.2).

2.1 Carpooling-fairness

The fair carpool scheduling algorithm. Carpooling consists in sharing a car among a driver and one or more passengers to divide the costs of the trip. We want a scheduling algorithm that will be perceived as fair by all the members as to encourage their continued participation.

Let U a value that represents the total cost of the trip. It is convenient to take U to be the least common multiple of $[1, 2, \dots, m]$ where m is the largest number of people who ever ride together at a time in the carpool. Since in a determined day the participants can be less than m , we define n as the number of participants in the carpooling in a given day ($n \in [1 \dots m]$). Each day we calculate the

passengers and driver's scores. In the first day the score is zero for each member; in the following days the driver will increase his score of $U(n-1)/n$, while the remaining $n-1$ passengers decrease their score of U/n .

With reference to this algorithm, we represent the driver with the agent A_{l_i} , while the passengers are the remaining $(n-1)$ enabled agents.

Let $n \leq m$ represents the number of enabled agents and A_{l_i} (with $i = [1, \dots, n]$) the agent enabled and executed. A_{l_i} increases his score of $\alpha_n = U(n-1)/n$, while the $n-1$ enabled and not executed agents decrease their score of $\beta_n = U/n$. Initially all the m elements of the array \mathbf{k} are equal to 0. Subsequently, we add α_n to the previous value ($\mathbf{k}[l_i]$) of the agent A_{l_i} and we subtract β_n to the previous value ($\mathbf{k}[l_j]$) of the other agents $A_{l_j} \quad \forall j \in [1, \dots, n], \quad j \neq i$.

In this case we use the condition $\langle cond(k) \rangle = \mathbf{k}[l_i] \leq \mathbf{k}[l_j]$ that allows the (enabled) agent with a lower score to evolve. We update the scores in the variable \mathbf{k} in the following way: the agent A_{l_i} (that is the agent which performs the execution) increases his previous score of α_n ; the $(n-1)$ enabled agents decrease their score of β_n , while the scores of the not enabled agents remain unchanged. With the parameters provided by the carpooling algorithm we obtain the Carpooling-fairness rule:

Table 2. Carpooling fairness - parallelism (1) rule

$$\frac{\mathbf{k}[l_i] \leq \mathbf{k}[l_j] \quad \forall \quad j = 1, \dots, n \quad \begin{array}{c} A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow \\ \langle A_{l_i}, \sigma \rangle \rightarrow \langle A'_{l_i}, \sigma' \rangle \end{array}}{\langle ||_m(A_1, \dots, A_{l_i}, \dots, A_m), \mathbf{k}, \sigma \rangle \rightarrow \langle ||_m(A_1, \dots, A'_{l_i}, \dots, A_m), \mathbf{k}', \sigma' \rangle}$$

$$\mathbf{k}[x]' = \begin{cases} \mathbf{k}[x] & \text{if } x = 1, \dots, m \\ \mathbf{k}[x] + \alpha_n & \text{if } x = l_i \\ \mathbf{k}[x] - \beta_n & \text{if } x = l_j \quad j \neq i \end{cases}$$

At each computation we select an agent according its value in the array \mathbf{k} and then we update it in a way that at the following step, another agent (possibly) has to be executed.

2.2 Bounded-fairness

In order to make the notion of (strong and weak) fairness sensible also for finite computations, we parametrise the definition with a value h and we ask that *an event which is (infinitely often or continuously) enabled at least h times, must necessarily be executed* (this definition is based on [3]). Therefore we combine the notion of h -fairness with the classical definition of Weak and Strong fairness [5] to obtain *Weak h -fairness* and *Strong h -fairness*. These two different levels of fairness are also available for finite computations.

Strong h-fairness. Strong h-fairness follows from strong and h-fairness definitions and it requires that *if an agent (or event) is enabled at least h times, it will be eventually executed*. To guarantee this statement in the parallelism rule we can say that whatever the value h is, we want to execute the event which is enabled many times with respect to the others.

To obtain this, the array \mathbf{k} will contain the times an agent is enabled; we set to 0 the value $\mathbf{k}[l_i]$ of the agent A_{l_i} which is executed and we increase of 1 the values $\mathbf{k}[l_j]$ (for $j = 1, \dots, n$ and $j \neq i$) of the enabled agents that are not executed. The values of the not enabled agents (A_1, \dots, A_m) remains, obviously, unchanged. In this case, to be executed, an enabled agent (A_{l_i}) has to fulfill the condition $\langle cond(k) \rangle = \mathbf{k}[l_i] \geq \mathbf{k}[l_j]$, that is, it must be enabled many times than that of the others enabled j agents. The new parallelism rule is represented in table 3.

Table 3. Strong h-fairness - parallelism rule

$$\frac{\mathbf{k}[l_i] \geq \mathbf{k}[l_j] \quad \forall j = 1, \dots, n \quad A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow \quad \langle A_{l_i}, \sigma \rangle \rightarrow \langle A'_{l_i}, \sigma' \rangle}{\langle ||_m(A_1, \dots, A_{l_i}, \dots, A_m), \mathbf{k}, \sigma \rangle \rightarrow \langle ||_m(A_1, \dots, A'_{l_i}, \dots, A_m), \mathbf{k}', \sigma' \rangle}$$

$$\mathbf{k}[x]' = \begin{cases} \mathbf{k}[x] & \text{if } x = 1, \dots, m \\ 0 & \text{if } x = l_i \\ \mathbf{k}[x] + 1 & \text{if } x = l_j \end{cases}$$

Weak h-fairness. Weak h-fairness requires that *if an agent is enabled at least h consecutive times, than it must be executed*. This means that whatever the value of h is, we want to execute the agent enabled more consecutive times than the others. In order to guarantee these requirements we extend the general $||_m$ rule by including an array \mathbf{b} which contains 0 and 1 values. The i -th element $\mathbf{b}[l_i]$ has value 1 only if the agent A_{l_i} was previously enabled. Moreover we use the array \mathbf{k} that contains the number of times an agent is consecutively enabled: we increase of 1 the value $\mathbf{k}[l_j]$ of an agent A_{l_j} only if it was previously enabled (that is $\mathbf{b}[l_j] = 1$) and we set to 0 the value of the agent which performs an execution. More in detail, we select among the enabled agents ($A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow$) the one which fulfills the condition $\mathbf{k}[l_i] \geq \mathbf{k}[l_j]$, that is, the agent which is enabled consecutively more times than the others. We set to 0 the values of the arrays \mathbf{b} and \mathbf{k} for all the not enabled agents in the current step; we set to 0 the value $\mathbf{b}[l_i]$ of the enabled and executed agent A_{l_i} , and to 1 the values $\mathbf{b}[l_j]$ of the remaining enabled agents. If the agents were previously enabled (therefore $\mathbf{b}[l_j] = 1$), we increment the values $\mathbf{k}[l_j]$, otherwise we set to 1 the value of the currently but not previously (with $\mathbf{b}[l_j] = 0$) enabled agent. We obtain the rule in Table 4.

Table 4. Weak h-fairness - parallelism rule

$$\begin{array}{c}
 A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow \\
 \mathbf{k}[l_i] \geq \mathbf{k}[l_j] \quad \forall j = 1, \dots, n \quad \langle A_{l_i}, \sigma \rangle \rightarrow \langle A'_{l_i}, \sigma' \rangle \\
 \hline
 \langle \parallel_m(A_1, \dots, A_{l_i}, \dots, A_m), \mathbf{b}, \mathbf{k}, \sigma \rangle \rightarrow \langle \parallel_m(A_1, \dots, A'_{l_i}, \dots, A_m), \mathbf{b}', \mathbf{k}', \sigma' \rangle \\
 \hline
 \mathbf{b}[x]' = \begin{cases} 0 & \text{if } x = 1, \dots, m \\ 1 & \text{if } x = l_j \quad \forall j = 1, \dots, n \end{cases} \\
 \mathbf{k}[x]' = \begin{cases} \mathbf{k}[x] & \text{if } x = 1, \dots, m \quad x \neq l_j \\ 0 & \text{if } x = l_i \\ \mathbf{k}[x] + 1 & \text{if } x = l_j \quad \text{and} \quad \mathbf{b}[l_j] = 1 \quad \forall j = 1, \dots, n \quad j \neq i \\ 1 & \text{if } x = l_j \quad \text{and} \quad \mathbf{b}[l_j] = 0 \quad \forall j = 1, \dots, n \quad j \neq i \end{cases} \\
 \hline
 \end{array}$$

2.3 Using \parallel_m for Soft Concurrent Constraint

We can use the new \parallel_m operator also to deal with soft constraints. In this case, we want to select the agent with a lower level of preference; in the soft case the array \mathbf{k} previously defined in the general rule contains soft constraints (unlike values used in the crisp case) added by the Agents which perform a tell action. This array is divided into sections: for each agent A_i there is a section $\mathbf{k}[i]$ that contains the agent's constraints. These are combined through the \otimes operator. For example, if Agent i performs the action: $tell(c)$, we will modify the section $\mathbf{k}[i]$, obtaining $\mathbf{k}[i]' = \mathbf{k}[i] \otimes c$. Below we show the modified rule that uses the array of soft constraints, and as condition $\langle cond(k) \rangle = \mathbf{k}[l_i] \sqsubseteq \mathbf{k}[l_j]$.

Table 5. \parallel_m - parallelism (1) with for soft constraints

$$\begin{array}{c}
 A_{l_1} \rightarrow, \dots, A_{l_n} \rightarrow \\
 \mathbf{k}[i] \sqsubseteq \mathbf{k}[j] \quad \langle A_{l_i}, \sigma \rangle \rightarrow \langle A'_{l_i}, \sigma' \rangle \\
 \hline
 \langle \parallel_m(A_1, \dots, A_{l_i}, \dots, A_m), \mathbf{k}, \sigma \rangle \rightarrow \langle \parallel_m(A_1, \dots, A'_{l_i}, \dots, A_m), \mathbf{k}', \sigma' \rangle \\
 \hline
 \mathbf{k}[x]' = \begin{cases} \mathbf{k}[x] & \text{if } x = 1, \dots, m \\ \mathbf{k}[x] \otimes c & \text{if } x = l_i \\ \mathbf{k}[x] & \text{if } x = l_j \quad j \neq i \end{cases} \\
 \hline
 \end{array}$$

This rule means that when the agent with the lower level of preference (A_{l_i}) is selected (since the guard $\mathbf{k}[l_i] \sqsubseteq \mathbf{k}[l_j]$ permits it), it combines its constraint (c) with the element $\mathbf{k}[l_i]$ obtaining $\mathbf{k}[l_i] \otimes c$ (therefore its level of preference increases) and this constraint is inserted in the store σ ; in this way we obtain a new store σ' that corresponds to the old store combined with the new constraint: $\sigma' = \sigma \otimes c$.

In the next step another agent is selected and its constraint combined with the previous constraints of the array \mathbf{k} , then it is added to the store and so on for the successive steps. We can consider this rule as the strong fairness version with soft constraints. Furthermore we can add the array of boolean values to obtain a soft version of the weak fairness rule.

Acknowledgement

We wish to thank Paolo Baldan for his help in discussing and improving the contents of the paper.

3 Future work

We are going to use the modified parallelism rule with metrics based on other fair scheduling algorithms. Subsequently we plan to study the utility functions used in economics as level of fairness, and to associate it to a semiring structure with the aim to measure “*how much the computation is fair*”. To do this we plan to use the same indexes that are used to measure the inequality in economics, such as the Gini coefficient. We also plan to use social welfare functions to aggregate individual preferences, in a way that each agent is equally satisfied with respect to the others and according to his preferences.

References

1. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *Lecture Notes in Computer Science*. Springer, London, UK, 2004.
2. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
3. Nachum Dershowitz, D. N. Jayasimha, and Seungjoon Park. Bounded fairness. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 304–317. Springer, 2003.
4. Ronald Fagin and John H. Williams. A fair carpool scheduling algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.
5. Nissim Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
6. Vijay A. Saraswat. *Concurrent constraint programming*. MIT Press, Cambridge, MA, USA, 1993.
7. Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM.

Translating Natural Language Sentences into ASP theories using SE-DCG grammars and Lambda Calculus

Stefania Costantini and Alessio Paolucci

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
stefcost@di.univaq.it, alessiopaolucci@ieee.org

Abstract. We build upon recent work by Baral, Dzifcal and Son that define the translation into ASP of (some classes of) natural language sentences from the lambda-calculus intermediate format generated by CCG grammars. We propose to use SE-DCG grammars, and we introduce automatic generation of lambda-calculus expressions from template ones, thus improving the effectiveness and generality of the translation process.

1 Introduction

Many intelligent systems have to deal with knowledge expressed in natural language, either extracted from books, web pages and documents in general, or expressed by human users. Knowledge acquisition from these sources is a challenging matter, and many attempts are presently under way towards automatically translating natural language sentences into an appropriate knowledge representation formalism [1]. The selection of a suitable formalism plays an important role but first-order logic, that would under many respects represent a natural choice, is actually not appropriate for expressing various kinds of knowledge, i.e., for dealing with default statements, normative statements with exceptions, etc. Recent work has investigated the usability of non-monotonic logics, like Answer Set Programming (ASP)[2].

The so-called Web 3.0 [3][4], despite its definition not well-established yet, makes the important assumption that applications should accept knowledge expressed in a human-like form, transform it into a machine processable form and take this step as the basis for semantic applications. This bears a similarity with the Semantic Web objectives [4], though Web 3.0 is a much wider vision, where artificial intelligence techniques plays a central role. Also in the semantic web scenario however, automatically extracting semantic information from web pages or text documents requires to deal with natural language processing, and requires forms of reasoning.

Translating natural language sentences into a logic knowledge representation is a key point on the applications side as well. In fact, designing applications such as semantic search engines implies obtaining a machine-processable form of the extracted knowledge that makes it possible to perform reasoning on the data so

as to suitably answer (possibly in natural language) to the user’s queries, as such an engine should interact with the user like a personal agent. We have practically demonstrated in previous work [5] that for extracting semantic information from a large dataset like wikipedia, a reasoning process on the data is needed, e.g., for semantic disambiguation of concepts.

A central aspect of knowledge acquisition is related to the automation of the process. Recent work in this direction has been presented in [1] [6] and [2]. In our opinion, the latter represents a significant advancement towards automatic translation of natural language sentences into a knowledge representation format that allows for automated reasoning. This work outlines a method for translating natural language sentences into ASP, so as to be able to reason on the extracted knowledge. They try in particular to take into account sentences defining uncertain and defeasible knowledge. In this paper, we extend their approach by adopting a semantically enhanced efficient context-free parser and introducing a new more abstract intermediate representation to be instantiated on practical cases. Also, we propose a fully automated translation methodology based upon our previous work [5].

2 Background

Before entering into the details of our proposal, we need to introduce the necessary building blocks of the work of [2]. They use CCG grammars that produce a λ -calculus intermediate form of a given sentences. Then, they introduce a variant of this intermediate form so as to cope with uncertain knowledge, and finally they propose a translation into ASP. Below we shortly recall the basics of λ -calculus and CCG’s, and then illustrate the method of [2].

2.1 Lambda Calculus

λ -calculus is a formal system designed to investigate function definition, function application and recursion. Alonzo Church and Stephen Cole Kleene introduced λ -calculus in the 1930s as part of an investigation into the foundations of mathematics, but it can be seen as programming language: in fact, it is universal in the sense that any computable function can be expressed and evaluated via this formalism.

The central concept in λ calculus is the “expression”, defined recursively as follows (where a “variable”, is an identifier which can be any of the letters a, b, c, ...):

$$\begin{aligned} M &::= \langle name \rangle \mid \langle function \rangle \mid \langle application \rangle \\ \langle function \rangle &::= \lambda \langle name \rangle . M \\ \langle application \rangle &::= M M \end{aligned}$$

Parentheses can be used for clarity. λ -calculus has only two keywords: λ and the dot. A single identifier is a valid λ expression, like, e.g., $\lambda x.x$ that defines the identity function. The name after the λ is the identifier of the argument of

this function. The expression after the point is called the *body* of the definition. Functions can be applied to expressions, like, e.g., $(\lambda x.x)y$ which is the identity function applied to y . Parentheses are used to avoid ambiguity. Function applications are evaluated by substituting the value of the argument x (in this case 'y') in the body of the function definition. The names of the arguments in function definitions do not carry any meaning by themselves, they are just place holders. In λ -calculus all names are local to definitions. In the function $\lambda x.x$, x is bound since its occurrence in the body of the definition is preceded by λx . A name not preceded by a λ is called a free variable. The same identifier can occur free and bound in the same expression.

Substitution corresponds to the operation that will replace in a term all the free occurrences of x with y , like $[y/x]M$

An α -conversion allows bounded variables to change their name, like $\lambda x.M = \lambda y.[y/x]M$ where $[y/x]M$ is the result of substituting y for free occurrences of x in M and y cannot already appear in M .

β -reduction defines an axiom related to the idea of function application. The β -reduction of $((\lambda x.M)N)$ is $[N/x]M$ where $[N/x]M$ denotes the substitution of the formal parameter x with the argument N throughout the expression M . In the rest of this paper, β -reduction will be denoted by the connective $@$, where for example $\lambda x.flies(x) @ tweety$ results in $flies(tweety)$.

2.2 ASP

Answer Set Programming (ASP) is a form of logic programming based on the answer set semantics [7], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [8, 9]. Rich literature exists on applications of ASP in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see among many [10–14] and the references therein).

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

$$H \leftarrow L_1, \dots, L_m, \text{ not } L_{m+1}, \dots, \text{ not } L_{m+n}$$

where H is an atom $m \geq 0$, $n \geq 0$ and each L_i is an atom. The symbol *not* stands for negation-as-failure. Various extensions to the basic paradigm exist, that we do not consider here all of them 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.)

The semantics of ASP is expressed in terms of *answer sets* (or equivalently *stable models*, [7]). Consider first the case of a ground 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 ground 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_{m+1}, \dots, L_{m+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 or no answer sets, because conclusions are included in an answer set only if they can be justified. The following program has no answer sets: $\{a \leftarrow \text{not } b, b \leftarrow \text{not } c, c \leftarrow \text{not } a\}$. 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 [15].

Let us consider the program P consisting of the three rules $\{r \leftarrow p, p \leftarrow \text{not } q, q \leftarrow \text{not } p\}$. 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.

To find the solutions of an ASP-program, an ASP-solver is used, where several solvers have become available [16]. The reader can see [10, 18], among others, for a presentation of ASP as a tool for declarative problem-solving.

2.3 CCG-Grammars

The Combinatorial Categorical Grammars (CCGs) [19, 20] have the aim of providing high expressive power while keeping automata-theoretic complexity to a minimum. CCG is a form of lexicalized grammar in which the application of syntactic rules is entirely conditioned on the syntactic type, or category, of their inputs. No rule is structure- or derivation-dependent. Categories identify constituents as either primitive categories or functions. Primitive categories, such as N (noun), NP (noun phrase), S (sentence), and so on, may be regarded as further distinguished by features, such as number, case, inflection, and the like. Functions (such as verbs) bear categories identifying the type of their result (such as VP, verb phrase) and that of their argument(s)/complement(s) (both may themselves be either functions or primitive categories). Function categories also define the order(s) in which the arguments must combine, and whether they must occur to the right or the left of the functor. Each syntactic category is associated with a logical form whose semantic type is entirely determined by the syntactic category. The slash '/' and '\ ' operators allow a category to combine by any combinatory rule.

In summary, a CCG grammar is composed of: a set of basic categories; a set of derived categories, each constructed from the basic categories; and some syntactical rules describing via the slash operators the concatenation and determining the category of the result of the concatenation. For instance, assume

that a CCG contains the following objects: *Tweety* whose category is NP (noun phrase) and *flies* whose category is (S\NP). The category of *flies* being S\NP means that if an NP (a noun phrase) is concatenated to the left of *flies* then we obtain a string of category S, i.e., a sentence.

Categories can be regarded as encoding the semantic type of their translation. This translation can be made explicit by associating a lambda-calculus logical form with the entire syntactic category. The parsing process will then generate a first-order lambda-calculus expression with quantifiers equivalent to the whole given sentence.

2.4 Automated Translation of Natural Language into ASP

The problem CCG do not cope with is that of *approximate* linguistic expressions that involve “fuzzy” assertions like, e.g., ‘normally’, ‘most’, etc. that do not have a direct correspondence in existentially/universally quantified sentences. Thus, an intermediate form is needed that is able to take this kind of sentences into account. This intermediate form should be such as to allow a translation/transposition into some executable formalism that allows the extracted knowledge to be reasoned about. Recent relevant work presented in [2] proposes the use of an intermediate λ -ASP-calculus representation, to be then translated into ASP. This calculus is an adaptation of λ -calculus to take the ASP rule format into account, so as to be able to represent fuzzy assertions and to translate them in a standard way into ASP.

The sample language discussed as a running example in [2], that they consider as a representative of a class of languages which are sufficient to represent an interesting set of sentences (including default statements and strong exceptions), is the following:

- Most birds fly.
- Penguins are birds.
- Penguins do swim.
- Penguins do not fly.
- Tweety is a penguin.
- Tim is a bird.

The first sentence is a *normative sentence* expressing a default: namely, it states that birds normally fly. The second sentence represents a subclass relationship, while third and fourth sentences represent different properties of the class of penguins. The last two sentences are statements about individuals.

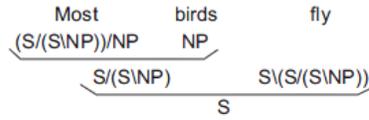
It is important to notice that none of previous approaches to automatic translation of natural language sentences is able to deal with default statements. The authors show how it is possible to automatically translate these sentences into ASP rules.

As first step, a CCG grammar for this sentences set is defined, L_{bird} . The CCG grammar is used because it gives information to “drive” the application of λ -expressions. However, as the goal is to obtain an ASP representation, the

notion of λ -representation is expanded to λ -ASP-expression which allows for construction of ASP rules. Then, the second step is the development of λ -ASP-expressions for words and categories in the language of interest. For example, for the L_{bird} grammar, the λ -ASP-expressions are (where variables denoting domain constants are indicated, as customary in ASP, in uppercase):

Word	Cat	λ -ASP-expression
fly	F1	$x.fly(x)$
	F2	$x.fly(x) \leftarrow x@X$
most	M	$\lambda v.(v@X \leftarrow \lambda x.bird(x)@X, not \neg v@X)$
birds	-	$\lambda x.bird(x)$

Given this theoretical tool an example of translation is the follow. Given the sentence “Most birds fly”, the CCG derivation tells how this sentence is constructed (where S=Sentence, NP=Noun phrase):



The word 'birds' is concatenated to the right of 'most', creating 'most birds'. This will be concatenated to the left of 'fly', creating a sentence whose category is S.

The λ -ASP-expressions for the categories of interest are:

M: $\lambda u \lambda v.(v@X \leftarrow u@X, not \neg v@X)$

B2: $\lambda x.bird(x)$

F1: $\lambda y.fly(y)$

The concatenation is driven by the CCG derivation, so the first step implies to concatenate 'birds' to 'most' and thus applying M to B2:

$(\lambda u \lambda v.(v@X \leftarrow u@X, not \neg v@X))@(\lambda x.bird(x))$ which gives

$\lambda v.(v@X \leftarrow \lambda x.bird(x)@X, not \neg v@X)$ which finally gives

$\lambda v.(v@X \leftarrow bird(X), not \neg v@X)$

The λ -ASP-expression for 'Most birds fly' is obtained by applying the above expression to F1, i.e.:

$(\lambda v.(v@X \leftarrow bird(X), not \neg v@X))@(\lambda y.fly(y))$ which gives

$\lambda y.fly(y)@X \leftarrow bird(X), not \neg \lambda y.fly(y)@X$ which finally gives

$fly(X) \leftarrow bird(X), not \neg fly(X)$.

3 Enhanced Automated Translation of Natural Language into ASP

In this section, we propose an improved fully automated methodology for generating ASP rules from natural language sentences of the kind discussed above. In our proposal, we associate to grammar rules expressions defined in a meta- λ -ASP-calculus. These expressions are 'meta' in the sense that they are associated to general categories rather than, like in CCGs and in the work of [2], to specific instances. For example, we define an expression related to the syntactic category of names, to be instantiated to the specific occurrences. We are then able to automatically generate ASP rules. This alleviates the problem, mentioned in [2], that the construction of λ -expressions requires human engineering, and it is a first step towards their automatic generation.

We do not define our method on CCG grammars. In [2] it is observed that each CCG, whose only syntactical rules are the left- and right-concatenation rules, is equivalent to a context-free grammar (see, e.g., [21]) which can be used in syntactic checking. Parsing a sentence using a context-free grammar is often more efficient, but these grammars lack the directionality information that CCG has. For example, they will not indicate how to obtain the semantics of 'most birds' from the semantics of 'most' and 'birds'. I.e., whether to apply the λ -expression of 'most' to the λ -expression of 'birds' or vice-versa. To overcome this problem, we adopt SE-DCGs, a variant (that we have defined in previous work [5]) of the well-known and widely-used DCGs. 'Per se', the DCGs do not allow for the same expressivity of CCGs, as they are fully context-free. However, the SE-DCGs are equipped with built-in constraints that may occur in rules and are verified during the parsing process whenever they are encountered. This provides the context-sensitivity that we were lacking. The built-in constraints however, as they are evaluated on a background knowledge base, also allow for semantic analysis and disambiguation of sentences. This might be useful in the CCG context as well. I.e., our proposal is not necessarily bound to the DCG realm but may in practice be transposed elsewhere.

The resulting framework is fully logical, and as we had already implemented the SE-DCG's related tools we have implemented the new methodology on top of these tools so as to directly proceed with the experiments.

3.1 SE-DCGs

In [5] we have proposed an extension, called SE-DCGs (Semantic Enhanced DCGs), to the DCGs (Definite Clause Grammars) in order to overcome their purely syntactic nature and introduce forms of 'on-the-flight' semantic analysis/reasoning. The DCGs are a well-known useful feature of prolog systems. 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. Syntax of a DCG rule is quite intuitive: a sample rule is shown below, where *non_term1* and *non_term2* are non terminal symbols and [*term1*] is a terminal symbol.

$non_term1 \rightarrow non_term2, [term1]$.

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).

The SE-DCGs can on the one hand 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-DCGs are able to operate with high efficiency because of early search space pruning due to the prompt recognition of errors and inconsistencies. The extension of DCGs to SE-DCGs is very simple, and implies limited modifications to a DCG pre-processor. The proposed extension is the following. Non terminals may have (like DCG's) one or more logical variables as arguments, that will be instantiated during the parsing process to terminal symbols. However, the right-hand side of rules is enhanced by adding expressions in brackets that express constraints on these variables (that can be seen as embedded goals in the sense of [22]). Thus, a sample SE-DCG rule looks like:

$$non_term1(X1, X2) \leftarrow non_term2(X1), non_term2(X2), [term1], \\ \{constr(X1, X2, term1)\}.$$

where there can be as many arguments as needed, and each constraint (here indicated as $constr(X1, X2, term1)$) is a formula involving variables and terminals, as well as symbols occurring in the background knowledge base.

SE-DCG's have the potential of allowing one to check and extract semantic information from sentences and periods. Syntactic-semantic analysis is performed via prolog-like queries, e.g., for the sample sentence *Most birds fly*:

$? - s(R, [most, birds, fly], [])$.

We obtain the parse tree below which correctly associates 'Most' to 'birds'.

$$R = element(\\ \quad subj(det('most'), noun('birds')), \\ \quad vp(vrb('fly')))$$

In fact, starting from the *most* lexicon, the SE-DCG analysis determines is semantic role (class), in this case *det*¹.

The extracted knowledge will be stored in the knowledge base, and used by SE-DCG's embedded goals for semantical analysis and reasoning.

Syntactic and semantic enhancements are by no means exclusive and can coexist in the same SE-DCG. In fact, constraints are evaluated in the background knowledge base. The knowledge base will include a dictionary of known objects

¹ Even though traditional English grammar does not include determiners and calls most determiners adjectives there are, however, a number of key differences between determiners and adjectives. Modern English grammars include degree (or *quantity*) determiners like *many, much, few, little* . . . See Longman English Grammar by L. G. Alexander, R. A. Close for more details.

and ontological aspects to establish correspondences. In principle however, it may contain any form of either commonsense knowledge or scientific/specialized knowledge so as to possibly integrate reasoning about language with other forms of reasoning.

Constraints in SE-DCG rules allow 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. For example, for a given sentence like 'airplanes flying in the blue sky', a constraint checking whether the subject can plausibly perform the action indicated by the verb, is performed by an SE-DCG's embedded goals like this:

$$plausible_subj(S, V), plausible_compl(V, C, Mode)$$

In the knowledge base, we might have rules such as:

$$plausible_subj(fly, V) : -flying_object(V). \\ flying_object(airplane) \dots$$

$$plausible_compl(fly, P, place) : -fly_where(P). \\ fly_where(sky) \dots$$

The constraint ensures that the sentence is plausible, while the check would fail if for instance the sentence mentioning an airplane flying *in the blue sea* as *sea* is not included in the *fly_where* list. The reasoning process can also include repair actions, related to what to do either in case of failure or in case of uncertainty.

The contribution of the SE-DCGs is more evident as the semantics complexity of sentences to be translated increases. In fact, determining the correct semantic class membership for lexical elements in complex sentences often requires reasoning (for example for disambiguation, evaluation of plausibility, etc.). For instance, consider the sentence *John often flies AILines*: through plausibility checks it is possible to establish that the subject *John* is not a *flying_object*, and thus the subject is not able to fly by himself. Then, *plausible_compl* is useful for determining that *AILines* is not an adjective but rather it somehow indicates that John will fly through (by means of) AILines. Others examples and details can be found in [5]. SE-DCG's have been applied to a complex case study, Mnemosine, which is a prototype intelligent search engine taking as dataset the Italian Wikipedia pages.

3.2 Abstract λ -ASP-expressions

In our methodology, we associate grammar rules to expressions defined in a meta- λ -ASP-calculus. These expressions are 'meta' in the sense that they are associated to categories rather than to specific instances. This alleviates the problem of the construction of λ -expressions, and is a first step towards their automatic generation.

We define λ -ASP-expressions_T (λ -ASP-Template-expressions) as a meta extension of λ -ASP-expressions. These expressions may contains lexical placeholders that are intended to be instantiated on the application context at hand. The instantiation of a λ -ASP-expression_T expression produces a λ -ASP-expression.

We define β as the λ -ASP-expression_T base.

For the running example, the λ -ASP-expression_T Base (i.e., the knowledge base containing the related expressions) is:

Lexicon	SemClass	λ -ASP-expression Template
-	noun	$\lambda x. < noun > (x)$
-	verb	$\lambda y. < verb > (y)$
most	det	$\lambda u \lambda v. (v @ X \leftarrow u @ X, \text{not } \neg v @ X)$

We have to suitably define the *instantiation* and *application* operations. The *instantiation*, denoted with the symbol @@, is the operation that will replace the lexical placeholder in a λ -ASP-expression_T with the given parameter.

The syntax of *instantiation* is the follow:

$$(\lambda - ASP - expression_T) @@ (lexicon).$$

The result of instantiation is a λ -ASP-expression. For example, given the λ -ASP-expression_T:

$$\lambda x. < noun > (x)$$

The instantiation of *noun* with 'home', is done through:

$$(\lambda x. < noun > (x)) @@ home$$

and the resulting λ -ASP-expression is:

$$\lambda x. home(x)$$

An *application* operation is the transposition of the λ -calculus application to the λ -ASP-expression realm.

The translation of a given sentence into a corresponding λ -ASP-expression starts from the leaves of the parse tree and go backwards to the root symbol. The bottom-up visit of the parse tree drives the translation execution. For each terminal symbol an *instantiation* operation is performed, while each non-terminal implies an *application* operation to be performed.

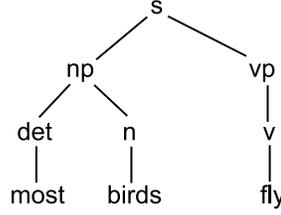
3.3 The Enhanced Methodology

The SE-DCGs play a key role, as the results of syntactic-semantics analysis are used to fully automatize the translation process. The semantic information is employed to find the correct λ -ASP-Template-expression from the λ -ASP-Template-expressions Base and thus instantiate or apply it automatically.

We illustrate the automatic translation process again with the help of the sentence *Most birds fly*. As seen above, the parse tree is represented as follows

$$R = (\text{element}(\text{subj}(\text{det}('most'), \text{noun}('birds')), \text{vp}(\text{vrb}('fly'))))$$

and, graphically, looks like:



Starting the left-to-right bottom-up visit of the parse tree, we retrieve in the λ -ASP-Template-expression Base the class 'det' (semantic match) and the exact lexical match for the 'most' lexicon. Since *most* is a terminal symbol, according to previous definitions we have to perform an instantiation operation. The λ -ASP-Template-expression for *most* is $\lambda u \lambda v.(v@X \leftarrow u@X, \text{not } \neg v@X)$

By the instantiation $(\lambda u \lambda v.(v@X \leftarrow u@X, \text{not } \neg v@X))@@\text{most}$ we obtain the λ -ASP-expression: $\lambda u \lambda v.(v@X \leftarrow u@X, \text{not } \neg v@X)$

In this case, the instantiation operation returns the same λ -ASP-expression, because no placeholders needs to be instantiated. This happens when both semantic and syntactic match occurs. This is seldom the case in complex sentences, where *instantiation* will in general perform constrains checks and syntactic manipulation.

The next leave of the parse tree is the lexicon *birds*. It has the semantic role of "noun" in the context of the sentence. We find in the template base a match for all lexicons of the semantic class *noun*. As the *birds* lexicon is a terminal symbol, according to previous definitions we perform an instantiation. The appropriate λ -ASP-Template-expression is $\lambda x. < \text{noun} > (x)$,

The instantiation operation is performed with *birds* as parameter, obtaining: $(\lambda x. < \text{noun} > (x))@@\text{birds}$. Thus, the resulting λ -ASP-expression is: $\lambda x.\text{bird}(x)$.

Going up the parse tree, we find non-terminal *np*. According to previous definitions, an *application* operations needs to be performed. In this case, semantic information drive the application of the λ -ASP-expression $(\lambda u \lambda v.(v@X \leftarrow u@X, \text{not } \neg v@X))@(\lambda x.\text{bird}(x))$ and thus we get $\lambda v.(v@X \leftarrow \lambda x.\text{bird}(x)@X, \text{not } \neg v@X)$ which produces $\lambda v.(v@X \leftarrow \text{bird}(X), \text{not } \neg v@X)$

Now, we encounter the *fly* lexicon (*verb*), thus we look for a match concerning the *verb* semantic class, applicable to all lexicons of this class. We use this λ -ASP-expression_T to *instantiate* the *fly* lexicon $(\lambda y. < \text{verb} > (y))@@\text{fly}$, and we get $\lambda y.\text{fly}(y)$.

For the *vp* class, the application is an identity, so we can skip to root symbol *s*, and thus perform the final application:

$(\lambda v.(v@X \leftarrow bird(X), not \neg v@X))@(\lambda y.fly(y))$
 which returns
 $\lambda y.fly(y)@X \leftarrow bird(X), not \neg \lambda y.fly(y)@X$
 from which we get the final ASP expression:
 $fly(X) \leftarrow bird(X), not \neg fly(X)$

4 Concluding remarks and future work

In this paper, we have introduced a fully automated process to translate natural language sentences into ASP theory, taking uncertain and defeasible knowledge into account. The method is an advancement over [2] in that it is based on an efficient semantically enhanced context-free class of grammars, and uses a more abstract intermediate representation that allows for automated translation via a visit of the parse tree. The proposed methodology has been implemented and experimented (the implementation is available from the authors).

In [2] it is observed that the class of sentences considered in their work so far lacks several constructs and in particular conjunctions (e.g. and, or, etc.), adverbs (e.g. quickly, slowly, etc.), and other auxiliary verbs (e.g. can, might, etc.) and that the problem in dealing with these constructs lies in making sure that each category can only be used in grammatically correct sentences and in ensuring that the semantical representation is proper. SE-DCGs with their 'on-line' semantic analysis can be of great use on these problems. For lack of space we cannot provide a proper explanation, we have already worked out sentences with conjunctions, like e.g. *Most birds fly and sing*.

The SE-DCG grammars that we have employed are a logic programming tool, and then we can say that the proposed methodology is fully logical, both in the definition and in the implementation. Experiments show that SE-DCGs seem to be able to retrieve enough semantic information to cope with complex sentences. However, we do not deny the superior expressive power of CCGs over DCGs. Therefore, our future work will include an attempt to merge the advantages of both.

References

1. Bos, J., Markert, K.: Recognising textual entailment with logical inference. In: HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing, Association for Computational Linguistics (2005) 628–635
2. Baral, C., Dzifcak, J., Son, T.C.: Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions. (2008) 818–823
3. Lassila, O., Hendler, J.: Embracing "web 3.0". IEEE Internet Computing **11**(3) (2007) 90–93
4. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American, issue of May, 17 (2001)

5. Costantini, S., Paolucci, A.: Semantically augmented DCG analysis for next-generation search engines. In: A. Formisano, ed., Online Proc. of CILC2008,, Italian Conference on Computational Logic. (2008) URL <http://www.dipmat.unipg.it/CILC08/programma.html>.
6. Moldovan, D.I., Harabagiu, S.M., Girju, R., Morarescu, P., Lacatusu, V.F., Novischi, A., Badulescu, A., Bolohan, O.: Lcc tools for question answering, TREC
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conference and Symposium on Logic Programming, The MIT Press (1988) 1070–1080
8. Lifschitz, V.: Answer set planning. In: Proc. of the 16th Intl. Conference on Logic Programming. (1999) 23–37
9. Marek, V.W., Truszczyński, M. In: Stable logic programming - an alternative logic programming paradigm. Springer (1999) 375–398
10. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
11. Anger, C., Schaub, T., Truszczyński, M.: ASPARAGUS – the Dagstuhl Initiative. ALP Newsletter **17**(3) (2004) See <http://asparagus.cs.uni-potsdam.de>.
12. Leone, N.: Logic programming and nonmonotonic reasoning: From theory to systems and applications. In Baral, C., Brewka, G., Schlipf, J.S., eds.: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. (2007) 1
13. Truszczyński, M.: Logic programming for knowledge representation. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007
14. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, chapter 7. Elsevier (2007)
15. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. J. of Logic Programming **19/20** (1994) 9–72
16. : Web references of (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>;
DeReS and aspps: <http://www.cs.uky.edu/ai/>;
DLV: <http://www.dbai.tuwien.ac.at/proj/dlv>;
Smodels: <http://www.tcs.hut.fi/Software/smodels>.
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. Dovier, A., Formisano, A., Pontelli, E.: A comparison of CLP(FD) and ASP solutions to NP-complete problems. In Gabbrielli, M., Gupta, G., eds.: Logic Programming, 21st International Conference, ICLP 2005, Proceedings. Volume 3668 of LNCS., Springer (2005) 67–82
19. Steedman, M.J.: Gapping as constituent coordination. Linguistics and Philosophy **13**(2) (1990) 207–263
20. Steedman, M.J., Baldridge, J.: Combinatory categorial grammar. To appear in Robert Borsley and Kersti Borjars (eds.) Constraint-based approaches to grammar: alternatives to transformational syntax. Oxford: Blackwell, draft available on the web sites of the authors (2009)
21. Gamut, L.: Logic, Language, and Meaning. University of Chicago Press (1991)
22. Pereira, F.C.N., Shieber, S.M.: Prolog and natural-language analysis. Center for the Study of Language and Information, Stanford, CA, USA (1987)