

List of CILC 2007 papers

Testing Logic Programs with Multiple Chances

Francesco Buccafurri, Gianluca Caminiti and Rosario Laurendi

Rule-based e-mail annotations

Massimo Marchi, Giacomo Fiumara, Rosamaria Pagano and Fortunato Roto

Automatic Correctness Proofs for Logic Program Transformations

Alberto Pettorossi, Maurizio Proietti and Valerio Senni

Frame logic under answer set semantics

Mario Alviano, Francesco Calimeri, Giovambattista Ianni and Alessandra Martello

Parallel Instantiation of ASP Programs

Francesco Calimeri, Simona Perri and Francesco Ricca

Implementation and Evaluation of Look-Back Techniques and Heuristics in DLV

Wolfgang Faber, Nicola Leone, Marco Maratea and Francesco Ricca

Towards Proof Theoretic Model Generation

Camillo Fiorentini, Alberto Momigliano and Mario Ornaghi

Web Site Verification: an Abductive Logic Programming Tool

Giacomo Terreni, Paolo Mancarella and Francesca Toni,

Programmazione Logica Multi-valued, Sinonimia, Controllo

Daniele Genito

Modeling CNF-formulae and formally proving the correctness of DPLL by Means of Referee

Eugenio Omodeo and Alexandru Ioan Tomescu

A Dialogue Games Framework for the Operational Semantics of Logic Languages

Stefania Costantini and Arianna Tocchio

A Heuristic Approach for P2P Negotiation

Stefania Costantini, Arianna Tocchio and Panagiota Tsintza

Using Unfounded Sets for Computing Answer Sets of Programs with Recursive Aggregates

Mario Alviano, Wolfgang Faber and Nicola Leone

Extending Agent-oriented Requirements with Declarative Business Processes: a Computational Logic-based Approach

Volha Bryl, Paola Mello, Marco Montali, Paolo Torroni and Nicola Zannone

A Tableau for Right Propositional Neighborhood Logic over Trees

Davide Bresolin, Angelo Montanari and Pietro Sala

KLM Logics of Nonmonotonic Reasoning: Calculi and Implementations

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

Answer Set Programming with Resources

Stefania Costantini and Andrea Formisano

A Goal-Directed Calculus for Standard Conditional Logics

Nicola Olivetti and Gian Luca Pozzato

Indexing Techniques for the DLV Instantiator

Gelsomina Catalano, Nicola Leone and Simona Perri

Combination Methods for Model-Checking of Infinite-State Systems

Silvio Ghilardi, Enrica Nicolini, Silvio Ranise and Daniele Zucchelli

Multivalued Action Languages with Constraints in CLP(FD)

Agostino Dovier, Andrea Formisano and Enrico Pontelli

SLDNE-Draw: a Visualisation Tool of Prolog Operational Semantics

Marco Gavanelli

Decision algorithms for fragments of real analysis. II. A theory of differentiable functions with convexity and concavity predicates

Domenico Cantone and Gianluca Cincotti

On the satisfiability problem for a 3-level quantified syllogistic

Domenico Cantone and Marianna Nicolosi Asmundo

Generalizzazione di Clausole Basata Su Un Nuovo Criterio di Similarità

Stefano Ferilli, Teresa Basile, Nicola Di Mauro, Marenglen Biba and Floriana Esposito

Constraint-based simulation of biological systems described by Molecular Interaction Maps

Luca Bortolussi, Simone Fonda and Alberto Policriti

Interoperability Mechanisms for Ontology Management Systems

Gallucci Lorenzo, Giovanni Grasso, Nicola Leone and Francesco Ricca

Some remarks on Rice's and Rice-Shapiro Theorems and related problems

Domenico Cantone and Salvatore Cristofaro

Testing Logic Programs with Multiple Chances

Francesco Buccafurri, Gianluca Caminiti, and Rosario Laurendi

DIMET, Università degli Studi Mediterranea di Reggio Calabria,
via Graziella, loc. Feo di Vito, 89122 Reggio Calabria, Italy,
{bucca, gianluca.caminiti, rosario.laurendi}@unirc.it

Abstract. The extension of Answer Set Programming (ASP) to naturally represent multi-chances reasoning is an issue recently studied in the literature by proposing logic programs with multiple chances (MC-programs). The importance of the above issue derives from the consideration that traditional ASP does not allow us to naturally express forms of reasoning where different conditions, partially alternative and hierarchically structured, are mentally grouped in order to derive some conclusion. The hierarchical nature of such knowledge concerns with the possible failure of a chance of deriving a conclusion and the necessity, instead of blocking the reasoning process, of activating a subordinate chance. However, introducing an additional abstraction layer in order to improve the Knowledge-Representation capability of ASP, sometimes might result in an intolerable increase of solver execution time. This paper is aimed to study the above issue, by implementing MC-programs, evaluating them, and comparing them with the most suitable language (for multi-chance reasoning) existing in the literature, that is Nested Logic Programming.

1 Introduction

The extension of Answer Set Programming (ASP) to naturally represent multi-chances reasoning is an issue recently studied in the literature [4]. There, the authors start from the consideration that in human-like reasoning it often happens that different conditions are mentally grouped in order to derive some conclusion. Such conditions represent different chances (i.e., possibilities), partially overlapped, exploitable as alternative ways to go further in the reasoning steps. Moreover, following different kinds of ordering, like reliability, simplicity, cost, and so on, different chances are hierarchically structured, and the failure of a chance of deriving a conclusion, caused by uncertainty conditions, enables the application of a subordinate chance. This way, the reasoning process is not necessarily blocked by uncertainty.

Consider for example the diagnostic medical reasoning. The doctor tries to recognize a classical clinical picture, in order to derive a diagnosis, but often such a picture is unclear and incomplete. In this case he typically adopts subordinate ways in order to reach the same conclusion as consulting, ex-adiuvantibus treatments, tests and so on.

Such a situation can be easily generalized to the case where multiple possibilities of deriving yield the same conclusion, as represented in Figure 1. A conclusion can be derived by the joint occurring of a given set of n basic conditions such that each, in case of uncertainty, can be substituted by a suitable second-chance condition. Clearly, it is

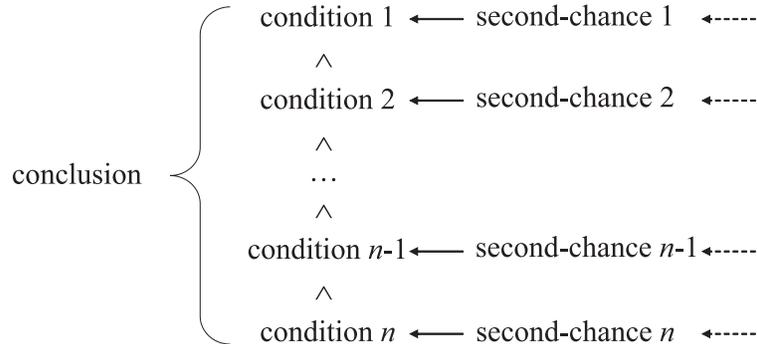


Fig. 1. Basic and second-chance conditions required to yield the same conclusion.

possible that each second-chance condition is uncertain itself, hence being substituted by a third-chance option, etc.

Answer Set Programming [7] does not allow us to express such situations in a synthetic fashion, since different chances of deriving a conclusion must be distributed over different rules, and conditions enabling the switching among chances must be explicitly represented.

In [4], a language is defined extending ASP by including a new modality, allowing us the representation, in a compact and natural fashion, of the multi-chances form of reasoning described above. Therein, the language is compared by examples with other formalisms, which are suitable to represent multiple chances. Among them we recall *Nested Logic Programming* (NLP) [11], that is a well-known framework languages allowing default reasoning with exceptions and prioritized rules [2, 14, 5, 3, 8].

An important question that is not addressed in [4], is that the high abstraction level introduced by the new modality, giving the language the capability of expressing multi-chances reasoning in a very natural and declarative fashion, has a heavy price in terms of evaluation time. Indeed, in [4], only an asymptotic complexity analysis is provided, showing that the complexity of the language is the same as plain ASP.

This paper address the above issue, by experimentally studying the performance of MC-programs and by comparing the language with the most suitable language (for multi-chance reasoning) existing in the literature, that is NLP [11].

The plan of the paper is the following. Section 2 briefly recalls syntax and semantics of Logic Programs with Multiple Chances [4]. Then, in Section 3 we discuss a number of experiments we have performed in order to evaluate the performance of the above framework, and also we discuss a comparison with NLP. Finally, we draw our conclusions in Section 4.

2 Logic Programs with Multiple Chances

In this section, we give an overview of logic programs with multiple chances [4], referenced to as *MC-programs*, by briefly recalling the syntax and the semantics of the language.

Recall [7] that an *atom* is an expression $p(t_1, \dots, t_n)$, where p is a *predicate* of arity n and t_1, \dots, t_n are constants. A *literal* is either a *positive literal* a or a *negative literal* $\neg a$, where a is an atom and \neg is the *classical negation* symbol. Given a literal a , $\neg a$ is defined as $\neg p$, if $a = p$ and p if $a = \neg p$. A set L of literals is *consistent* if $\forall l \in L, \neg l \notin L$. Given a literal a , the formula *not a* is the *negation as failure* (NAF) of a ¹.

Definition 1. Given $m \geq 0$ and $k \geq 0$, an *MC-formula* β is a formula

$$(\alpha_1[\gamma_1], \dots, \alpha_m[\gamma_m])\{T_1, \dots, T_k\}$$

where:

- (1) α_j ($1 \leq j \leq m$) is (possibly the NAF of) a literal and may appear underlined,
- (2) T_i ($1 \leq i \leq k$) is a subset of $\{\alpha_j \mid \alpha_j (1 \leq j \leq m) \text{ is either underlined, or is not the NAF of a literal}\}$, and
- (3) γ_j ($1 \leq j \leq m$) is a (possibly empty) conjunction of MC-formulas.

Each set T_i ($1 \leq i \leq k$) is referenced to as the *admissibility constraint* and the (possibly empty) set $\{T_1, \dots, T_k\}$ of admissibility constraints of β is denoted by $S(\beta)$. γ_j ($1 \leq j \leq m$) is referenced to as the *second-chance* of α_j .

Observe that the recursive definition above is a generalization of the standard definition of the conjunction of (possibly the NAF of) literals occurring in the body of an ASP logic rule, where both (i) a second-chance is (possibly) associated to each literal, (ii) literals may appear underlined and (iii) a set of admissibility constraints is associated to the conjunction.

Each second-chance may be viewed as a substitution of the α_j to which is applied. Such a substitution is executed only when α_j is *uncertain* (the notion of uncertainty will be introduced in the following – the underlining is relating with this notion). Due to the recursive structure of the above definition, a second-chance is, in turn, a conjunction of MC-formulas, and thus may contain further subordinate chances (i.e., the nesting of chances is allowed). Intuitively, second-chances are subordinate conditions that are required, in order to satisfy the MC-formula, whenever the basic conditions α_i s are uncertain. Actually, not all the possible configurations of values making α_i s uncertain are allowed, in order to activate the substitution of the second-chance, but only those satisfying the admissibility constraints.

Example 1. Consider the following MC-formulas: $\beta_1 = (a, \underline{\text{not } b})[c]$ (denoted also by $(a[c], \underline{\text{not } b[c]})$), where $S(\beta_1) = \emptyset$ and $\beta_2 = (a, b)\{\{a, b\}\}[c]$, (denoted also by $(a[c], b[c])\{\{a, b\}\}$), where $S(\beta_2) = \{\{a, b\}\}$.

¹ Observe that the NAF of negative literals is allowed.

Given a conjunction of MC-formulas Δ , in favor of simplicity, we often write

$$(\alpha_1[\gamma_1], \dots, \alpha_m[\gamma_m])\{T_1, \dots, T_k\}[\Delta]$$

to denote the following MC-formula:

$$(\alpha_1[\gamma_1[\Delta]], \dots, \alpha_m[\gamma_m[\Delta]])\{T_1, \dots, T_k\}.$$

In particular, if $\gamma_1 = \dots = \gamma_m = \Delta$, we write $(\alpha_1, \dots, \alpha_m)\{T_1, \dots, T_k\}[\Delta]$ to denote the MC-formula $(\alpha_1[\Delta], \dots, \alpha_m[\Delta])\{T_1, \dots, T_k\}$. Clearly, square brackets of empty second-chances are omitted.

Definition 2. An MC-rule r is a formula $a \leftarrow \beta_1, \dots, \beta_n$ ($n \geq 0$), where a is a positive literal and β_i is an MC-formula, for each $1 \leq i \leq n$. The set $\{a\}$, denoted by $head(r)$, is referenced to as the *head* of r , and the set $\{\beta_1, \dots, \beta_n\}$, denoted by $body(r)$, is referenced to as the *body* of r .

Note that since an MC-formula is a generalization of a standard conjunction of (the NAF of) literals, a standard ASP rule is a special case of an MC-rule r . Informally, an MC-rule is a logic rule allowing in the body the substitution of some literal with its second-chance (under uncertainty conditions).

Example 2. The following are examples of MC-rules: $a \leftarrow (\underline{b}, c, e)\{\{c, e\}, \{\underline{b}, c\}\}[not\ d]$, and $a \leftarrow b[c[d]], f[not\ g]$.

Definition 3. An MC-program is a finite set of MC-rules.

We introduce now the intended models of the semantics of MC-programs. First we need some preliminary definitions.

Given an (MC-)program P , Lit^P is the set of literals occurring in P . An *interpretation* I of an (MC-)program P is a consistent subset of Lit^P . A literal a is *true* w.r.t. I if $a \in I$, it is *false* w.r.t. I if $\neg a \in I$. A literal a is *undefined* w.r.t. I if it is neither true nor false w.r.t. I . Given a literal a , a formula *not* a is *true* w.r.t. I if $a \notin I$, it is *false* w.r.t. I otherwise (observe that the NAF of a literal cannot be undefined w.r.t. a given interpretation). From now on in this section, consider given an (MC-)program P and an interpretation I of P .

We introduce now a basic notion of the framework, that is the notion of *uncertainty* of an element of an MC-formula. Indeed the mechanism of substitution of an element of an MC-formula with its second-chance is founded on this property (this will be explained in detail in the following).

Definition 4. Given an MC-formula $\beta = (\alpha_1[\gamma_1], \dots, \alpha_m[\gamma_m])\{T_1, \dots, T_k\}$, we say that α_j ($1 \leq j \leq m$) is *uncertain* w.r.t. the interpretation I if either:

- (i) $\alpha_j = a \mid a \in Lit^P \wedge \{a, \neg a\} \cap I = \emptyset$,
- (ii) $\alpha_j = \underline{a} \mid a \in Lit^P \wedge a \notin I$, or
- (iii) $\alpha_j = \underline{not\ a} \mid a \in Lit^P \wedge a \in I$.

The definition above arises from the following reasoning. We expect that an element α_j is uncertain if it is undefined. This is captured by the item (i) of the definition. Note that, correctly, this item does not include the case of the NAF of a literal, since such a formula cannot be undefined. Now, when we want to interpret as uncertain also the false value of α_j , we require that α_j is underlined (items (ii) and (iii)). Thanks to this mechanism, also the NAF of a literal can be uncertain, whenever it appears in an underlined element $\alpha_j = \underline{\text{not } a}$, and a is true.

Example 3. For instance, given the interpretation $I = \{\neg a\}$, the MC-formula $b[\text{not } d]$ is uncertain w.r.t. I since $\{b, \neg b\} \cap I = \emptyset$. Moreover, the MC-formula $\underline{c}[f]$ is uncertain w.r.t. I since $c \notin I$. Finally, the MC-formula $a[g]$ is not uncertain w.r.t. I because $\neg a \in I$.

Moreover, Definition 4 above is extended by means of a flexible machinery such that we are able to specify which literals in an MC-formula cannot be uncertain at the same time.

Definition 5. Given an MC-formula $\beta = (\alpha_1[\gamma_1], \dots, \alpha_m[\gamma_m])\{T_1, \dots, T_k\}$, $S(\beta)$ is true w.r.t. I if for each $T_i \in S(\beta)$ ($1 \leq i \leq k$), there exists $\alpha \in T_i$ such that α is not uncertain w.r.t. I .

In words, an admissibility constraint T_i states that literals occurring in an MC-formula β cannot appear simultaneously in uncertainty configurations.

Example 4. Consider the following MC-rule: $a \leftarrow (\underline{b}, c, e)\{\{c, e\}, \{\underline{b}, c\}\}[\text{not } d]$. Given an interpretation $I = \{\neg b, e\}$, it is easy to see that $S(\beta) = \{\{c, e\}, \{\underline{b}, c\}\}$ is not true w.r.t. I , because both c and \underline{b} are uncertain w.r.t. I .

Now, we define the intended models of the semantics of MC-programs. First, we introduce a transformation for MC-programs which produces an ASP program.

Definition 6. We define the *MC-transformation* of the program P w.r.t the interpretation I as the ASP program \bar{P}^I , obtained from P by executing Algorithm 1.

Algorithm 1 (The MC-transformation algorithm)

```

repeat
  for each MC-rule  $r \in P$  do
    if  $\exists \beta \in \text{body}(r) \mid S(\beta)$  is false w.r.t.  $I$  then delete  $r$ 
    else for each  $\beta = (\alpha_1[\gamma_1], \dots, \alpha_m[\gamma_m])\{T_1, \dots, T_k\} \in \text{body}(r)$  do
      delete  $S(\beta)$ 
      for each  $1 \leq j \leq m$  do
        if  $\alpha_j$  is uncertain w.r.t.  $I$  then remove from  $\alpha_j$  the underlining (if any)
          if  $\gamma_j$  is not empty then replace  $\alpha_j[\gamma_j]$  by  $\gamma_j$ 
          end if
        else remove from  $\alpha_j$  the underlining (if any); replace  $\alpha_j[\gamma_j]$  by  $\alpha_j$ 
        end if
      end for
    end for
  end for

```

end if
end for
until $\forall r \in P, \forall \beta \in \text{body}(r) \exists a \in \text{Lit}^P \mid \beta = a \vee \beta = \text{not } a$

The MC-transformation of P consists of both (i) deleting all MC-rules of P whose body includes some MC-formula with false admissibility constraint, (ii) deleting all admissibility constraints from the remaining MC-formulas, (iii) for each uncertain element α_j in the body of every MC-rule, removing the underlining (if any), and replacing α_j by its second-chance (if any); (iv) for each remaining element α_j , removing the underlining (if any) and discarding its second-chance. The loop ends when only (possibly the NAF of) literals occur in P .

Definition 7. An interpretation J of the program P is an *answer set* of P if J is an answer set² of \bar{P}^J .

Example 5. For instance, consider the following MC-program P :

$$\begin{array}{ll} r_1 : a \leftarrow (b, c, e) \{ \{c, e\}, \{b, c\} \} [not\ d] & \\ r_2 : d \leftarrow not\ a & r_3 : \neg b \leftarrow a \\ r_4 : c \leftarrow not\ d & r_5 : e \leftarrow \end{array}$$

The intended answer sets of P are: $\{a, \neg b, c, e\}, \{d, e\}$. Indeed, given $I_1 = \{a, \neg b, c, e\}$, the MC-transformation of P w.r.t. I_1 is $\bar{P}^{I_1} = \{r'_1, r_2, r_3, r_4, r_5\}$, where r'_1 is $a \leftarrow not\ d, c, e$. Observe that I_1 is an answer set of \bar{P}^{I_1} thought as an ASP program. Likewise, it is easy to see that $I_2 = \{d, e\}$ is an answer set of $\bar{P}^{I_2} = \{r_2, r_3, r_4, r_5\}$ (r_1 is deleted due to the constraint $\{b, c\}$).

Remark. The formalism presented in [4] can be viewed as an extension of the ASP negation by failure. Indeed, the rule $h \leftarrow not\ b$ can be rewritten in the form of a MC-rule as $h \leftarrow \neg b[true]$. According to the semantics of MC-programs, h is derived if either b is false (i.e., $\neg b$ is true) or b is undefined (since, in this case the subordinate condition is activated and it corresponds to the constant *true*), exactly as the ASP rule $h \leftarrow not\ b$.

3 Experiments

In this section we discuss the results of several experiments we have done in order to evaluate the language proposed in [4]. First, we describe our testbed. We have implemented an algorithm in Java which performs a source-to-source transformation from MC-programs to Answer Set Programming. The code is based on a suitable linear-time translation algorithm, described in [4]. Briefly, a given MC-program P is translated, by means of such an algorithm, into an ASP program $\Gamma(P)$. Thus, we exploit nowadays powerful ASP engines [9, 13] in order to compute the answer sets of P in terms

² The definition of *answer sets* of an ASP program can be found in [7]. We do not report it here for space limitations. Note that, according to the definition of interpretation, we want to limit our focus only on consistent answer sets.

of answer sets of the logic program $\Gamma(P)$ that is produced by the translation. We have chosen the DLV System [9], that is widely accepted as a state-of-the-art ASP solver³.

Now, we describe two series of experiments we have performed:

1. The first series of experiments is aimed to study the impact of the features of MC-programs on the performance of the ASP solver.
2. In the second series of experiments we compare the performance of the framework of MC-programs with that of NLP.

3.1 First Series of Experiments: Evaluation of MC-Programs

Now we describe the first series of experiments we have performed. We have considered the following parameters that are associated to the syntax of MC-programs:

- (1) The number of MC-formulas per rule (n);
- (2) The maximum level of nesting of MC-formulas (l);
- (3) The number of elements (possibly MC-formulas) that compose a second-chance of a MC-formula (m); and
- (4) The activation of admissibility constraints (c).

Table 1 shows the values assigned to the parameters above. Observe that such values have been chosen since we assume that a real programmer would represent multi-chance knowledge by means of human-readable MC-programs. Hence, the parameter values are not arbitrary, but they correspond to realistic cases.

Parameter	Set of Values
n	{1,2,3,4}
l	{1,2}
m	{1,2,3,4}
c	{disabled, enabled}

Table 1. Parameter values used for the experiments.

Thus, for each different choice of the parameter values, we have generated a set of MC-program instances. Each program in a given instance set is made of ten MC-rules, where each rule has a structure that corresponds to the parameter choice. The maximum number of literals occurring per each instance is fifteen.

In the next experiment we want to evaluate the impact of admissibility constraints.

³ We have used DLV release Jul 14 2006, running on a Pentium 4 @ 3.40 Ghz.

Experiment 1. In order to evaluate the effect of the admissibility constraints, we want to compare the average times needed by the ASP solver to compute the answer sets of each program instance set, for $c = \text{disabled}$ and $c = \text{enabled}$, respectively.

First, we consider the case $l = 1$. The results of the experiment are shown in Figure 2. Each instance set is denoted by the pair $\langle i, j \rangle$, where i and j correspond to values assigned to the parameters n and m , respectively. Observe that we omit all instance sets $\langle 1, j \rangle$ such that $1 \leq j \leq 4$, since it is easy to see that in those cases there is just one MC-formula in the body of a MC-rule and thus admissibility constraints cannot be used for such instances. Clearly, if m is fixed, then the average time needed to compute the answer sets grows as n increases. Conversely, if n is fixed, different values of m produce slight differences w.r.t. the average times of computation. The reason is that, since in this case $l = 1$, for each program instance, second-chances are simply conjunctions of m (possibly NAF) literals.

Moreover, another effect caused by using only one level of nested chances is that the computational overhead introduced by admissibility constraints is, on average, very small.

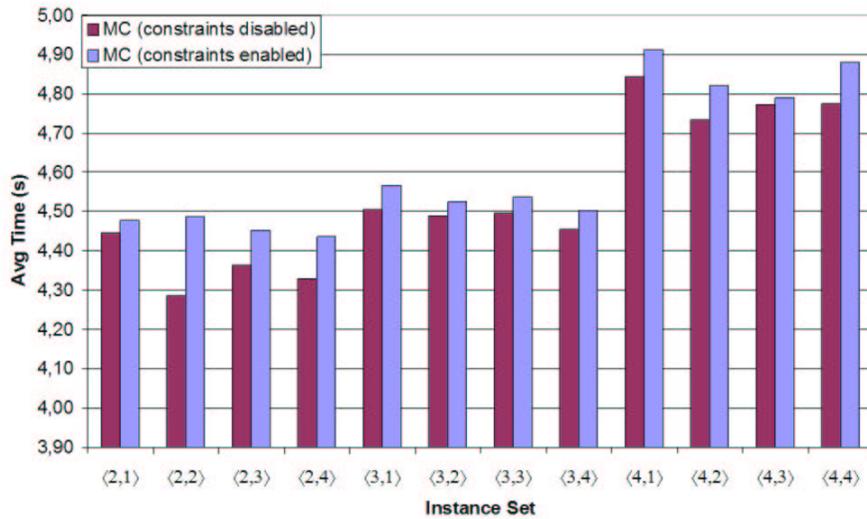


Fig. 2. Effect of admissibility constraints on MC-programs (one level of nesting).

Experiment 2. In this experiment we evaluate the performance of MC-programs, under the same setting as Experiment 1 above. The only parameter that changes here is l , i.e. $l = 2$. Figure 3 shows the results of the experiment after such a change. Considerations similar to those stated for $l = 1$ hold. However, observe that, now, the impact of admissibility constraints is, on average, more evident than the case above, since for each

level of nesting it is possible to enable the admissibility constraint. As a consequence, a bigger computational overhead is produced.

Moreover, concerning the instance sets $\langle i, j \rangle$ ($2 \leq i \leq 4, j = 1$), note that the results are comparable with those obtained for $l = 1$ (see Experiment 1). The reason is that if $j = 1$, then each second-chance in a MC-formula is composed by only one element. As a consequence, regardless the two levels of nesting, in case $j = 1$ it is possible to enable the admissibility constraints only on the first level (i.e., not inside second-chances).

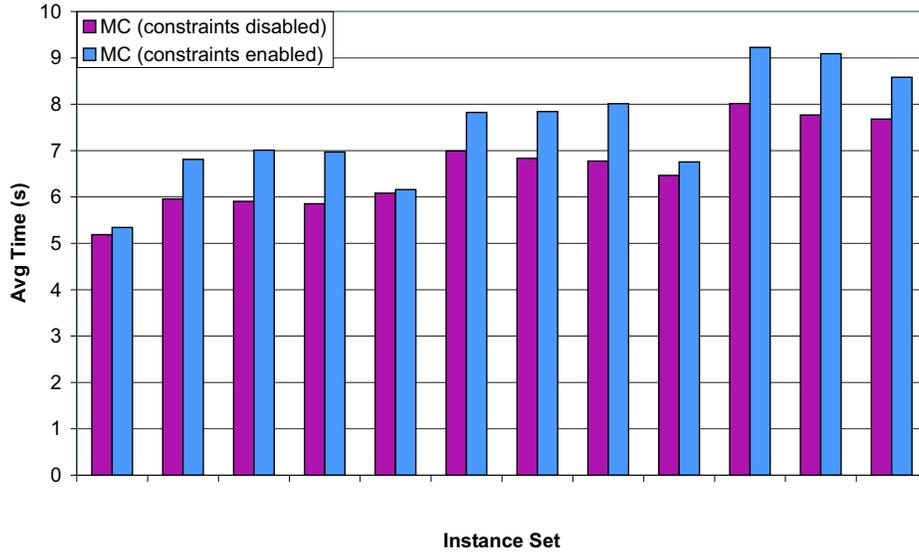


Fig. 3. Effect of admissibility constraints on MC-programs (two levels of nesting).

Experiment 3. Now, we are interested in discussing the results of an experiment that is aimed to evaluate the effect of representing multi-chance knowledge by using MC-program with more than one level of nested chances.

Figure 4 shows the average times measured on instance sets in case admissibility constraints are disabled. This is done both for one level and for two levels of nested second-chances. It is easy to see that the computational overhead due to the presence of a second level of nesting inside MC-formulas is more evident as the parameters n and m increase.

Now, observe that similar considerations hold for the results shown by Figure 5, which represents the average times measured on instance sets such that the admissibility constraints are enabled. Clearly, as mentioned in the paragraph above concerning Experiment 2, those instance sets such that no admissibility constraint can be introduced are omitted. Now, note that the presence of admissibility constraints inside MC-

formulas produces, on average, a computational overhead that is bigger than in the case such constraints are disabled. Concerning the instance sets such that $m = 1$, we recall that for such cases, the overhead is reduced because the structure imposed by the parameters to the program instances does not allow to enable admissibility constraints at the second level of nesting.

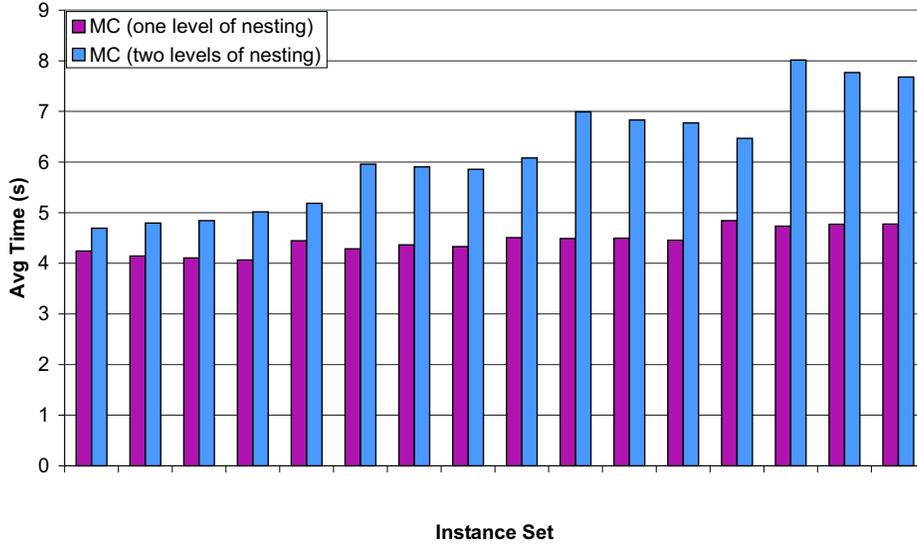


Fig. 4. Effect of the number of nesting levels on MC-programs (admissibility constraints disabled).

3.2 Second Series of Experiments: Performance Comparison with NLP

In this Section we compare the performance of MC-programs with that of Nested Logic Programming (NLP). Recall that NLP [11] is a class of logic programs, where arbitrarily nested expressions – formed from literals by using negation as failure, conjunction and disjunction ($;$) – are allowed in both the bodies and heads of rules. Given an MC-program P , a suitable translation P' into a suitable fragment of NLP (with neither NAF nor disjunction in heads of rules) exists. Informally, we can give a sketch of an easy translation scheme of an MC-rule of P into an equivalent NLP rule of P' . It suffices to write, for each literal b occurring in a conjunction of a MC-rule, the disjunction of the following two conditions: (i) The first condition represents b in the standard case (i.e., no uncertainty), (ii) The second condition models the conjunction of both all the *allowed* (by means of admissibility constraints) conditions producing the uncertainty of b and the second-chance associated to b .

Example 6. Consider, for instance, the MC-rule $a \leftarrow (b[e], c[f])\{\{b, c\}\}$. An equivalent NLP rule obtained by the above translation method is:

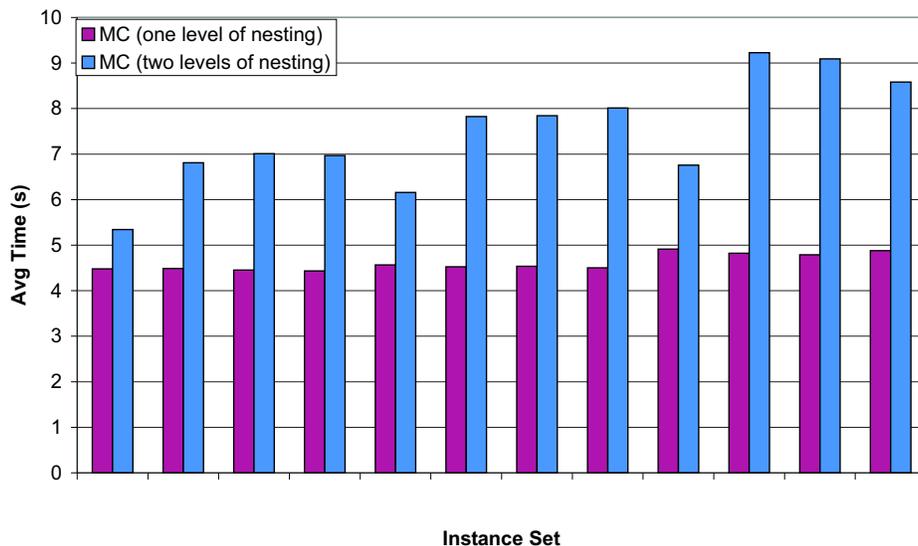


Fig. 5. Effect of the number of nesting levels on MC-programs (admissibility constraints enabled).

$$a \leftarrow (b; (\text{not } b, e, (c; \neg c))), (c; (\text{not } c, \text{not } \neg c, f, b))$$

Clearly such a translation machinery, though it is simple, it is recursive in case there is more than one level of nested second-chances inside MC-rules.

However, since in our experiments we have limited the maximum nesting level to a small value, namely 2, it is easy to see that such a translation is feasible.

Moreover, in [4] the authors have shown that encoding a multi-chance knowledge represented by MC-programs into NLP produces logic programs that are extremely tedious to write and difficult to read (i.e. they represent knowledge in fashion that is not natural). This occurs especially in case the original MC-programs have two or more levels of nested second-chances.

Now, we describe the testbed we have used for this series of experiments. For each MC-program instance that we have generated (recall Table 1), we have produced the corresponding NLP program, according to the translation scheme above. Then, we have exploited an external NLP front-end for the DLV system [15] that runs under SICStus Prolog [6] in order to obtain from each NLP instance, an equivalent ASP program that we have given in input to DLV.

Experiment 4. First we have evaluated the size of both MC- and NLP program instances after the translation into ASP. In detail, we have counted both the number of atoms and rules after the translation and we have measured the average per cent increase obtained by the NLP instances relative to the corresponding MC ones, after the translation.

The following Tables 2 and 3 summarize the results of the experiment, showing that the increase in size of NLP program instances after the translation is very big. As we

will show in the following experiments, such a big increase in the size corresponds to a better performance exhibited by the ASP solver on MC-programs.

	Constraints disabled	Constraints enabled
One level of nesting	152.18%	161.75%
Two levels of nesting	123.56%	119.04%

Table 2. MC vs NLP: Percent increase of the number of atoms

The following figures show the performance comparison between MC-programs and equivalent NLP programs, run under DLV. Observe that, for each instance set, only the average times needed to compute the answer sets are presented. We do not consider here the times required to perform the translation from either MC- or NLP program instances to ASP.

Experiment 5. First, we discuss the comparison in the case of one level of nesting ($l = 1$). The results of the experiment are reported both in case the admissibility constraints are disabled (Figure 6) and in case they are enabled (Figure 7). Clearly, we have omitted in Figure 7 the instance sets denoted by the pairs $\langle i, j \rangle$ such that $i = 1$, since such instance sets correspond to the case $n = 1$ (recall Table 1), and thus admissibility constraints cannot be enabled.

Observe that the performance of the solver is much better on the MC-programs than on the corresponding NLP instances. This occurs especially in case the admissibility constraints are enabled, since the translation from NLP into ASP produces a program whose size is much bigger than that obtained by translating the corresponding MC-program instance into ASP (recall the results of Experiment 4). As a consequence the computational overhead is bigger for NLP programs. In detail, Figure 7 shows that in case admissibility constraints are enabled, the NLP instances are slower than the corresponding MC instances by, on average, more than 200%.

Moreover, we have done the same comparison in case of two levels of nesting ($l = 2$). Figures 8 and 9 show the result of the experiment in case the admissibility constraints are disabled and enabled, respectively. Now, observe that the computational overhead due to the second level of nested chances is bigger than the case above. MC-programs exhibit better performance than NLP programs. Moreover, in case admissibility constraints are enabled, the computational overhead of NLP programs is more evident than in the case they are disabled. Finally, an interesting result is shown by Figure 9, that is, on average, in case admissibility constraints are enabled, the NLP instances are more than ten times slower than the corresponding MC instances.

	Constraints disabled	Constraints enabled
One level of nesting	336.61%	369.14%
Two levels of nesting	251.76%	282.62%

Table 3. MC vs NLP: Percent increase of the number of rules

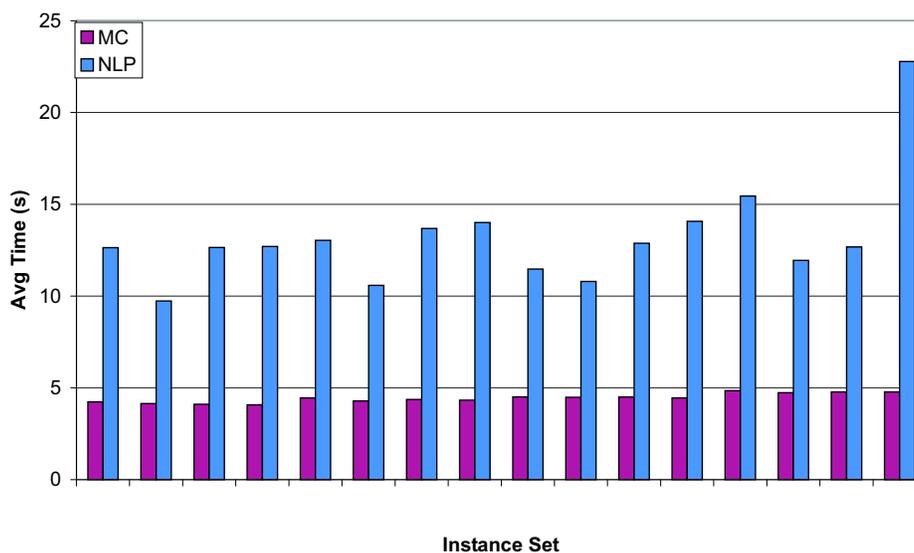


Fig. 6. MC vs NLP (admissibility constraints disabled, one level of nesting).

4 Conclusions

The paper presents experimental evaluation of the implementation of MC-logic programs. The language relies on Answer Set Programming and includes constructs useful for naturally representing some forms of reasoning where multiple chances of deriving a given conclusion occur. The results of the experiments we have performed give us valuable information about the impact that the features of MC-programs have on the performance of the ASP solver. Moreover, the results of the performance comparison between MC-programs and NLP programs (having at least the same abstraction level) show the superiority of the former in terms of running time for their evaluation.

Finally, the good performance exhibited by the ASP solver on the MC-program instances confirms us in the feeling that a very interesting direction for our future research is applying the formalism to hard problems such as planning [12, 16, 10, 1].

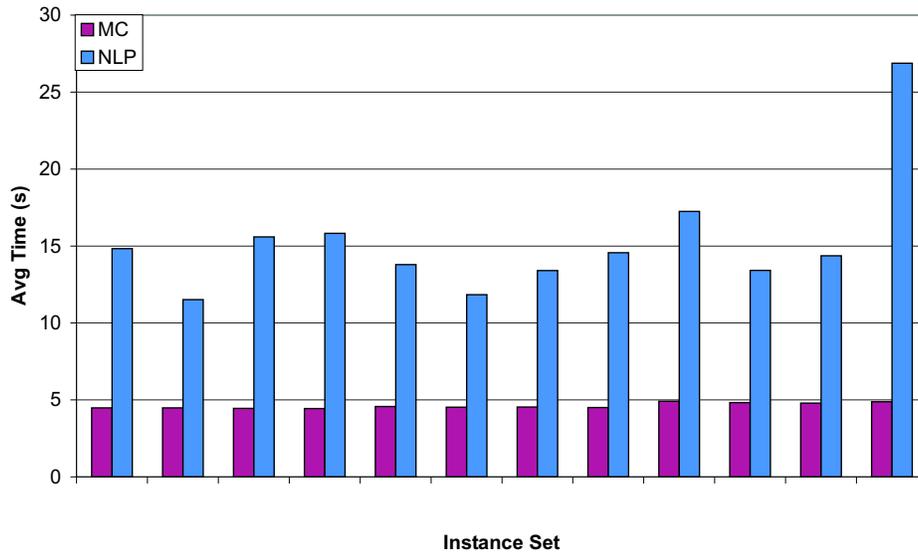


Fig. 7. MC vs NLP (admissibility constraints enabled, one level of nesting).

References

1. M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The USA-Advisor: A Case Study in Answer Set Planning. In *LPNMR*, volume 2173 of *LCNS*, pages 439–442. Springer, 2001.
2. G. Brewka and T. Eiter. Preferred Answer Sets for Extended Logic Programs. *Artif. Intell.*, 109(1-2):297–356, 1999.
3. G. Brewka, I. Niemelä, and M. Truszczynski. Prioritized Component Systems. In *AAAI05*, pages 596–601, 2005.
4. F. Buccafurri, G. Caminiti, and D. Rosaci. Logic programs with multiple chances. In G. Brewka et Al., editor, *Proceedings of ECAI - European Conference on Artificial Intelligence*, pages 347–351. IOS Press, 2006.
5. F. Buccafurri, N. Leone, and P. Rullo. Disjunctive Ordered Logic: Semantics and Expressiveness. In *Proc. of KR'98*, pages 418–431, 1998.
6. Swedish Institute for Computer Science. Sicstus prolog. <http://www.sics.se/sicstus/>.
7. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9, 1991.
8. M. Gelfond and T. C. Son. Reasoning with Prioritized Defaults. In *LPKR*, pages 164–223, 1997.
9. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ArXiv Computer Science e-prints*, pages 11004–+, 2002.
10. V. Lifschitz. Answer Set Programming and Plan Generation. *Artif. Intell.*, 138(1-2):39–54, 2002.
11. V. Lifschitz, L.R. Tang, and H. Turner. Nested Expressions in Logic Programs. *Annals of Mathematics and Artif. Intell.*, 25(3-4), 1999.

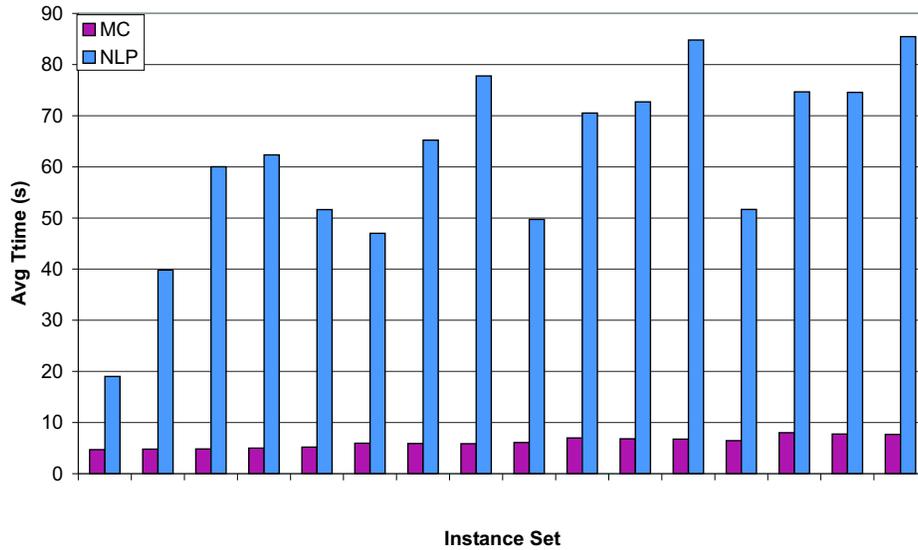


Fig. 8. MC vs NLP (admissibility constraints disabled, two levels of nesting).

12. A. Nareyek, E. C. Freuder, R. Fourer, E. Giunchiglia, R. P. Goldman, H. A. Kautz, J. Rintanen, and A. Tate. Constraints and AI Planning. *IEEE Intelligent Systems*, 20(2):62–72, 2005.
13. I. Niemelä and P. Simons. Smodels - an Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In *Proc. of the 4th LPNMR*, LNCS, pages 420–429. Springer, 1997.
14. C. Sakama and K. Inoue. Prioritized Logic Programming and Its Application to Commonsense Reasoning. *Artif. Intell.*, 123(1-2), 2000.
15. V. Sarsakov, T. Schaub, H. Tompits, and S. Woltran. nlp: A compiler for nested logic programming. In V. Lifschitz and I. Niemelä, editors, *LPNMR*, volume 2923 of LNCS, pages 361–364. Springer, 2004.
16. T. C. Son, P. H. Tu, M. Gelfond, and A. R. Morales. Conformant Planning for Domains with Constraints-A New Approach. In *Proc. of The XXth National Conf. on Artif. Intell. and the 17th Innovative Applications of Artif. Intell. Conf.*, pages 1211–1216, 2005.

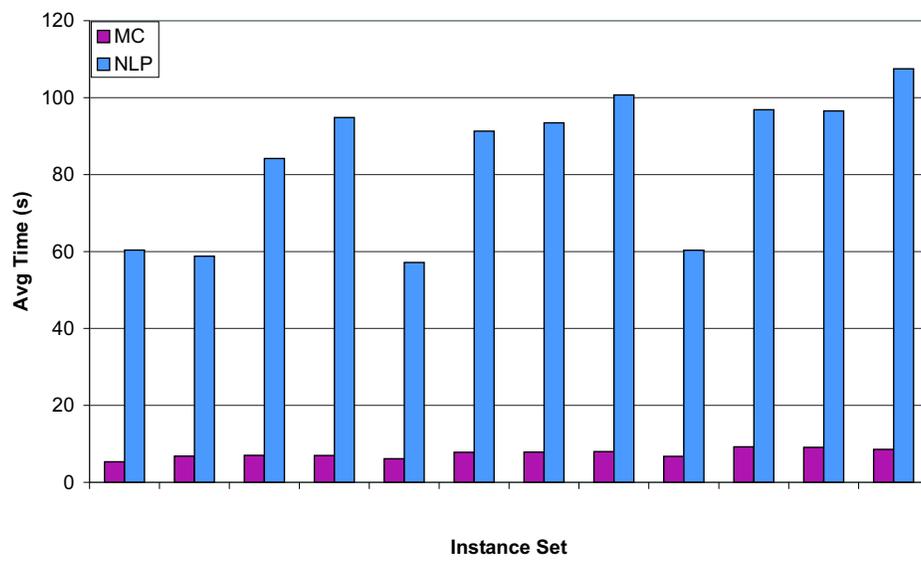


Fig. 9. MC vs NLP (admissibility constraints enabled, two levels of nesting).

Rule-based e-mail annotations^{*}

G.Fiumara¹, M.Marchi², R.Pagano¹ and F.Roto¹

¹ Dept. of Physics, University of Messina,
Sal. Sperone, 31. S. Agata di Messina. I-98166 Messina, Italy,
{*giacomo.fiumara,rosamaria.pagano*}@unime.it

² DSI - University of Milan,
Via Comelico, 39/41. I-20135 Milan, Italy,
marchi@dsi.unimi.it
<http://mag.dsi.unimi.it/>

Abstract. We present the RuBaST (Rule-Based Spam Termination) project, describe its theoretical underpinnings and some preliminary benchmark results. RuBaST shows that hand-written rules that guide the spam-likeness evaluation are effective on a relevant corpus that we have assembled by collecting e-mails that have escaped detection by SpamAssassin.

1 Introduction

This article describes a rule-based *personal* e-mail annotator that works as a spam sentinel. The annotator applies ad-hoc set of annotation rules (called anti-spam *policy*) by means of a Prolog inferential engine. Our program, called RuBaST (Rule-Based Spam Terminator) is intended as a filtering layer between a typical, site-wide SpamAssassin [1] and the user's mailbox. RuBaST takes as input e-mails and human-written annotation rules that give a *score* to certain aspects of the e-mail, e.g., the occurrence of specific words in the body text or a *suspicious* sender address. RuBaST works as follows. First, it reads the e-mail file and applies ad-hoc regular expressions that de-obfuscate suspicious words, e.g., Ciaa11is is rendered as Cialis. Next the whole e-mail is converted into a set of Prolog facts; Prolog rules representing the user's spam annotation policy are combined with those facts. The rules describe, essentially, a scoring system. The overall score is obtained as the sum of all scores given by the rules: it is compared to a user-defined threshold and the e-mail is forwarded accordingly. The annotation rules in Prolog syntax are applied by means of the SWI-Prolog inferential engine.

RuBaST has been developed in the context of our long-term research goal of applying rule-based policies to Web activities, e.g., Web service connection [2], Web data extraction [3] and Semantic Web composition [4]. It does not wish to replicate the excellent results obtained by Machine Learning and Bayesian

^{*} A companion Web site to this article, with software, results and the corpus described herewith are available at <http://mag.dsi.unimi.it/rubast/>

filtering, but rather to provide an interface where personal rules are provided and tuned to the particular type of spam one is receiving or indeed finds more obnoxious.

Even though this application is in its infancy, it shows promising results w.r.t. a small but challenging corpus that we have assembled with e-mails that have not been blocked by our campus-wide SpamAssassin filter³ then in version 3.1.1.

1.1 The idea

The basic idea of our project is to exploit rule-based reasoning to realize an anti-spam filter which assists in the onerous task of manually identifying and separating spam from *ham* (i.e., non-spam) messages. In particular, our goal is to implement an intelligent agent which would assess spam somewhat in human-like fashion⁴. RuBaST filter receives and analyzes the messages acquiring the same information perceived by the user. Every piece of information available to the user, e.g., the sender, the addressee, the object and the text of the message, is relevant and should be considered.

1.2 Validation

We have tested the first implementation of RuBaST against a corpus of e-mails that have been collected and prepared to this purpose. So far, the most e-mails in the corpus are, beyond doubt, spam. It is important to notice that the recipient also wrote the annotation policy that was given to RuBaST during the experiments. Few messages have been classified as *ham* or *maybe* by the recipient. The *maybe* case consists of unsolicited messages, e.g. conference announcements, that share, in the structure and content of the message, some similarity with real spam. For instance, the *To:* field is normally masked, the return address differs from the *From:*, the message is impersonal and almost always invites to visit a Web site. To validate the effectiveness of RuBaST in weeding out spam messages, we have posed the problem in terms of Information Retrieval (IR) [6], i.e., as a challenge for RuBaST to select all and only the messages in the corpus that were deemed spam. Then, we have compared each annotation to that given by a human (other than the recipient) and measured the degree of similarity in terms of *Precision*, *Recall* and F_1 , rather than *accuracy* and *error*⁵.

2 The Implementation

The implementation of RuBaST consist mainly of a Java application with a Prolog-based automated reasoning core. The interface between the two modules

³ <http://spamassassin.apache.org/>

⁴ In the *Logical AI* tradition of McCarthy (see, e.g., [5]), we take intelligent agents to be computer systems that acts intelligently: what the agent does is appropriate for its circumstances and its goal, it is flexible to changing environments, it deduce from experience, and it makes appropriate choices given finite computational resources.

⁵ Please refer to [7] or to [8] for an introduction to this subject.

is performed by the JPL package provided by SWI-Prolog. Figure 1 summarizes the architecture of our implementation. Such architecture addresses two main concerns that have emerged during our project. First, the current solutions that integrate inferential engines into a Java program (e.g., tuProlog [9]) were considered not viable for this project or harder to use vis-à-vis SWI Prolog. Second, we would like to retain the ability to experiment with different inferential engines in the future. For instance, in the next phase of the project we will experiment with Answer Set Programming⁶. Indeed, the annotation rules, which will be described in the sequel, hardly use any function or built-in that are specific to SWI.

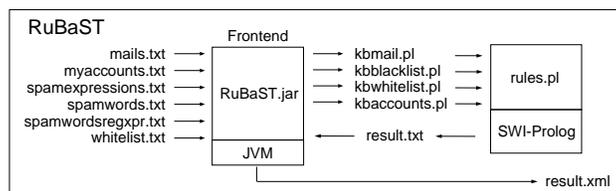


Fig. 1. The overall architecture

JPL [12] is a library that uses the SWI-Prolog⁷ foreign interface and the Java JNI interface⁸ to interleave Prolog and Java executions. The rôle of the Java module, then, is to prepare the message information for the Prolog engine. We call this preparation phase *normalization*. The main task of normalization is the so-called *de-obfuscation* of the mail body, i.e., spotting (and relative replacement) of terms that have been disguised by the spammer in order to hide them from the anti-spam softwares (please see [13] for a discussion and the latest results on de-obfuscation).

The following words are examples of obfuscation found in e-mails not detected by SpamAssassin version 3.1.1:

- *Viiagrra* (for Viagra) and
- *Ci-iallis* (for Cialis)

Our rule-based, then, applies to the normalized message the heuristic rules of the base of knowledge, assigning a positive score (spam clue) or a negative score (ham clue). The obtained value (SPAM VALUE) is compared to some thresholds and will accordingly classify the message as SPAM, MAYBE-SPAM or HAM. The prior knowledge embedded in our system consists, essentially, of two list:

⁶ Answer Set Programming, in short, gives to Prolog rules a multiple models semantics, thus allowing to model reasoning by cases. Please refer to the pioneering work of Lifschitz and Gelfond [10] and to the survey in [11] for an introduction to the subject.

⁷ <http://www.swi-prolog.org/>

⁸ <http://java.sun.com/j2se/1.4.2/docs/guide/jni/>

- *black-words* and *black-expressions* commonly used in the spam messages (commonly referred to as *blacklist*), and
- trusted email address (commonly referred to as *whitelist*).

2.1 The Java module and the de-obfuscation

The first task of the Java module is to locate and extract from the message the pieces of information that are subject of our analysis. For instance, the sender, the addressee, the object and the text of the message. Next, de-obfuscation starts. RuBaST resorts to regular expressions, to analyze the text of the message looking for words and/or expressions that are present in blacklist, yet disguised by spammers.

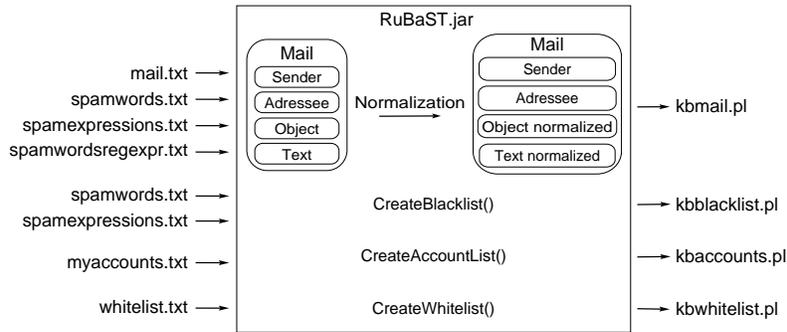


Fig. 2. Architecture of the front end

Attempts to mask the *black-expressions* are of different kind [13], for instance, misspelling, insertion, replacement (or removal) of non-alphabetical characters (e.g., extra spaces), to name a few.

RuBaST tries to pick all those obfuscations that, although complicated, are resolved by an average user *at a glance*.⁹

RuBaST constructs two *models* of regular expressions for every *black-expression*.

Example 1. The two models for the word CIALIS are:

Model 1: `[cC]{1} .{0,2} [iI]{1} .{0,2} [aA]{1} .{0,2} [lL]{1} .{0,2} [iI]{1} .{0,2} [sS]{1} .{0,2} [\\s]{1}`

Model 2: `[cC]{1} .{0,2} [\\n]+ .{0,2} [\\n]* .{0,2} [\\n]* .{0,2} [iI]{1} .{0,2} [\\n]+ .{0,2} [\\n]* .{0,2} [\\n]* .{0,2}`

...

⁹ Spammers seems to be aware that very complex obfuscation risks, in fact, to remain unclear to the human eye, thus making the attempt of spamming useless.

The second model tries to find those *black-expressions* written one character for line; we found that several spam e-mails, which we have collected, present this kind of structure.

2.2 The Prolog module

The Prolog module represents the core of our anti-spam filter. It applies two filters:

WORDS_FILTER (henceforth WF), which operates on the *Subject* and *Body* of the message, and

RULES_FILTER (henceforth RF), which operates on *envelope* informations such as sender and addressee (*From* and *To*). The final score consists of the sum of the scores recommended by each filter (see Figure 3).

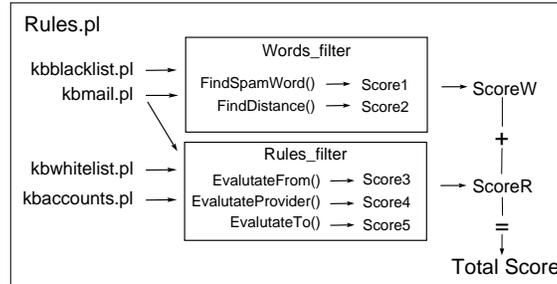


Fig. 3. Architecture of the Prolog module

The WORDS_FILTER To advertise their products, be they of commercial or financial kind, spammers are often forced to use some word or expression that is easily identifiable, e.g. the name of a popular medicine.

WF exploits this aspect to look for unsolicited offerings. To be such, spam advertisement should:

- describe the product and mention its name;
- speak favorably of the product;
- express clearly the sale offer;
- provide references, normally URLs, where the product can be purchased.

To find this information out, our module takes as input 2 files:

- *kbmail.pl* - the representation of the message;
- *kbblacklist.pl* - the internal “black words” archive.

The *kbmail.pl* file contains the relevant information of the message that are represented as Prolog facts:

- the sender:

```
from(eml_id, email_addr).  
fromProvider(eml_id, provider).
```

- the addressee:

```
to(eml_id, email_addr).
```

- the object and the text are represented as a list of words by the fact:

```
email(eml_id, [W0, W1, ..., Wk])
```

For the sake of portability, the blacklist is organized as a simple set of facts. Each *black-expression* corresponds to a value which indicates the *degree of spam* associated to its occurrence.

For instance, consider the following selection from our rules:

```
spam_word('buy', 20).  
spam_word('viagra', 100).  
spam_word('%guaranteed', 70).  
spam_word('clickhere', 70).
```

The first step of the WF execution consists of scanning the list inside the *email* facts to recognize the black-words. The list of found black-words (henceforth *hitlist*) is then analyzed to count the occurrence of every word detected and to assert in the knowledge base by:

```
assert(wFind(eml_id, expression, occurrence)).
```

A partial score is computed for every black-expression found by the formula:

$$words_score(eml_id) = \sum_{i=1}^{|\text{hitlist}|} (Value[Word_i] * Occ[Word_i]) \quad (1)$$

Where *Value* is the spam value of the black-expression and *Occ* is its occurrence in the message. This score is of course too coarse as a measure of spam likelihood.

We have then considered the relative position of black-words in the subject and in the message: the higher the distance between them, the lower the probability that they are correlated. So, in the second step we compute the distance between each two black-expressions found and to assert record these informations on the knowledge base:

```
assert(findDist(EmailID, Word1, Word2, Distance)).
```

The relative score of the position of two black-words is defined as usual:

$$distance_Score(eml_id, Word_i, Word_j) = \frac{Val_i + Val_j}{4} \frac{1}{\log(Dist + 2)} \quad (2)$$

Where Val_i and Val_j are the scores of $Word_i$ and $Word_j$, respectively, and $Dist$ is the distance between the two words¹⁰.

¹⁰ Sometimes we use the term *words* to refer to expressions. Consider for instance the expression *lowest price* that is an entry of the *blacklist*.

Rules for analyzing the envelope The information related to sender and to addressee is another element useful for the evaluation. Usually the user does not expect spam messages from familiar email addresses while expects it. The RF module analyzes the envelope informations by using the knowledge base that holds the *white-list* and the user's self-defined profile information. The annotation assigned by each rule, whenever it applies, is recorded as a set of facts of type: *rule_value(rule_name, relative_value)*.

The following scores are given by the rules that most often get applied:

rule_value('known_From', -60).

rule_value('from_myProvider', -20).

rule_value('unknown_to', 20).

The final annotation is defined as the sum of the values of the rules that did apply. The following is an example of the rules that analyze envelopes; if the sender of the mail is in the whitelist, then the rule gives a negative score (i.e. unlikely to be spam) to the e-mail.

```
/*
PROCEDURE KNOWN_FROM
it checks whether the sender is present in the white list,
represented by knownAddress(X) facts.
*/
known_from(EmailID) :- from(EmailID,X),
                        knownAddress(X),
                        rules_value(known_from,Y),
                        assert(rules_find(EmailID,known_from,Y)).
```

Rules for analyzing the content The following rules are simple examples of how the e-mail content (or the *Subject:* field) can be analyzed. The first rule below annotates the e-mail according to the occurrence of black-words:

```
/* PROCEDURE FINDWORD.
simply checks whether a backlist word appears in the message.
Use findall to find the whole hit set.
*/
word(X):-spam_word(X,_).
findWord(EmailID,X) :- email(EmailID,E),
                        word(X),
                        member(X,E).
```

The next rule captures the following reasoning: if the body or the object of the mail contains two different spam words, then we give a positive score based on the distance between them.

```
/* PROCEDURE FINDDIST2
it finds the distance between two different words_spam then
asserts (findDist(_E12,_E11,Count)where _E11 e _E12_ are different
```

```

spam_word and Count is the distance in between
*/
findDist2 :- email(EmailID,Email),
             wFind(EmailID,X,N1),
             N1>0,
             wFind(EmailID,Y,N2),
             N2>0,
             (X)\==(Y),
             not(alreadyFound(EmailID,Y,X)),
             find_dist2(EmailID,X,Y,Email).

alreadyFound(EmailID,A,B) :- findDist(EmailID,A,B,_);
                             findDist(EmailID,B,A,_).

find_dist2(EmailID,_E11,_E12,[],_,_) :- fail.

find_dist2(EmailID,_E11,_E12,[_E12|T],_E11,Count) :-
    assert(findDist(EmailID,_E11,_E12,Count)),
    C is Count+1,
    !,
    find_dist2(EmailID,_E11,_E12,T,_E11,C).

find_dist2(EmailID,_E11,_E12,[_E11|T],_E12,Count) :-
    assert(findDist(EmailID,_E12,_E11,Count)),
    C is Count+1,
    !,
    find_dist2(EmailID,_E11,_E12,T,_E12,C).

find_dist2(EmailID,_E11,_E12,[_E12|T]) :- Count is 0,
    find_dist2(EmailID,_E11,_E12,T,_E12,Count).

find_dist2(EmailID,_E11,_E12,[_E11|T]) :-
    Count is 0,
    find_dist2(EmailID,_E11,_E12,T,_E11,Count).

find_dist2(EmailID,_E11,_E12,[_|T],_E11,Count) :-
    C is Count+1,
    find_dist2(EmailID,_E11,_E12,T,_E11,C).

find_dist2(EmailID,_E11,_E12,[_|T],_E12,Count) :-
    C is Count+1,
    find_dist2(EmailID,_E11,_E12,T,_E12,C).

find_dist2(EmailID,_E11,_E12,[_|T]) :-
    find_dist2(EmailID,_E11,_E12,T).

```

We can now see how the annotation works on one example of increasingly complexity .

3 Experimental assessment

This Section describes the experimental validation of the rules w.r.t. a corpus that we assembled to this purpose. The messages collected into the corpus have been received by the same e-mail address over a period of several months between 2005 and 2006.

The considered address has been constantly shielded by the campus-wide SpamAssassin. Thus, these e-mails can be considered *hard instances* for spam detection since they escaped SpamAssassin. The SpamAssassin score, which is always attached to the meta field of the mail, has been saved for further analysis and comparison. The 131 messages have been recorded and assigned to one of these categories

- total spam;
- false positive;
- false negative, and
- valid.

Valid e-mails are only a very small selection of the e-mails received daily by the considered address. We have selected those that showed features that could indeed be considered evidence for a spam nature, thus are likely to provide a false positives. These features were: the fact that the e-mail is impersonal, that it shows multiple *To:* addresses and the fact that it has URLs in the body. We found that conference call-for-papers advertisements did have these features and could, subjectively, be considered spam. However, the large majority of the considered instances were true spams, as shown in Table 1.

ID	Class Name	# of e-mails
0	Spam	119
1	Maybe	5
2	Ham	7

Table 1. Human-made classification of the corpus

All the e-mails that compose the corpus were submitted to the human expert in one session. A simple Java interface would show the message, including the To and From fields, and ask to press a button to assign to it a score: *Spam*, *Maybe* and *Ham*. At the end of the experiment, the Java program wrote out onto a file the id and annotation of each e-mail.

3.1 Evaluation

We have evaluated the RuBaST annotation of the corpus against that of a human (other than the creator of the corpus) throughout the Information Retrieval (IR) [6] standard measures *precision*, *recall* and F_1 .

To define such measures, we need to work on four frequency scores:

- tp** RuBaST and the expert agree on the category assigned in the corpus;
- fp** RuBaST disagree with the assigned category but the expert does not;
- fn** the expert disagree with the category but RuBaST does not, and
- tn** both disagree on the assigned category.

The above scores may be understood by looking at the organization of Table 2.

		Expert	
		Relevant	Not Relevant
RuBaST	Retrieved	tp	fp
	Not Retrieved	fn	tn

Table 2. Parameters used to compute precision and recall

We can now define the three measures:

$$precision = \frac{tp}{tp + fp} \leq 1 \quad (3)$$

$$recall = \frac{tp}{tp + fn} \leq 1 \quad (4)$$

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} \quad (5)$$

The last definition is a simplification of the harmonic mean of the two measures with $\alpha = 1/2$.

3.2 Test execution

Our corpus, organized as a folder (one file per message, no classification), has been fed to RuBaST, which generated an annotation file in the same format as that described above. Next the exact classification, the human annotation and the RuBaST score were consulted and compared and we computed the needed **tp**, **fp**, and **fn** frequencies (**tn** is not really needed in what follows). The results are summarized in Table 3.

For the considered experiment the frequencies were:

	Spam	Maybe	Ham
Corpus	119	5	7
Expert	92	32	7
RuBaST	54	21	56

Table 3. Comparing the classification frequencies

$$tp = 51 \tag{6}$$

$$fp = 49 \tag{7}$$

$$fn = 7 \tag{8}$$

$$tn = 24 \tag{9}$$

Finally, the scores could be computed:

$$precision = \frac{tp}{tp + fp} = \frac{51}{51 + 49} = 0,51 \tag{10}$$

$$recall = \frac{tp}{tp + fn} = \frac{51}{51 + 7} = 0,879 \tag{11}$$

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2}{\frac{1}{0,51} + \frac{1}{0,879}} = 0,645 \tag{12}$$

That is, the percentages are (about) 51 %, 87 % and 64 %, respectively.

4 Discussion and Related Work

Even though RuBaST is a novel application of rule-based reasoning, our experimental assessment shows that it can be very effective in weeding out spam e-mails. The assessment was done on a rather small corpus (as opposed to large and diversified corpora, e.g., TREC¹¹ [14] and SpamAssassin¹²). Yet our corpus is relevant precisely for the reason that the annotation policies RuBaST enforces are personal ones and need to be evaluated against the particular spam attack one recipient is under. Even though writing the Prolog-style annotation rules can be tedious, this approach has the benefit of being compositional: rules can be changed, added a different times, copied from others etc. without any further change in the system, thus making it possible to react quickly, on a personal basis, to new classes of spam messages that would escape the standard Internet-level filters. Finally, it is possible to *learn* rules from corpora by applying Inductive Logic Programming, to the personal corpus, seen as a training set. This further activity can be incorporated into RuBaST almost effortlessly.

¹¹ <http://trec.nist.gov/>

¹² Apache Foundation: The SpamAssassin Public Mail Corpus, Apache Foundation, (<http://spamassassin.apache.org/publiccorpus/>)

5 System-wide implementation

An improvement to RuBaST we are planning is a system-wide version. The aim is to put on the server-side of the scenario the burden of the computation of Hidden Markov Models (HMM) in order to gain de-obfuscation in a significantly different way from regular expressions. Various advantages can be seen from this implementation. First, calculations are accomplished on the server, thus lightening mail clients. Secondly, the de-obfuscation challenges that are common to various clients receiving similar e-mails can be done only once and for all. In

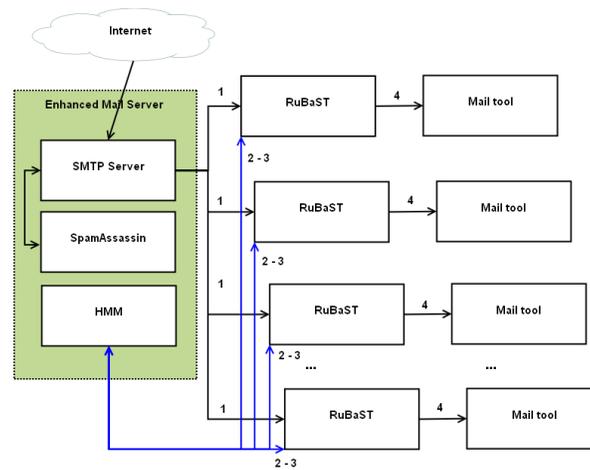


Fig. 4. Architecture of the system-wide implementation, first schema

Figure 4 we represent a possible architecture of the system-wide implementation. Incoming e-mails are processed by SMTP server and sent to SpamAssassin which re-sends the accepted e-mails to SMTP server in order to be sent to recipients (namely, to RuBaST). From here e-mails are sent to HMM (see actions 2 and 3 in Figure 4). After the de-obfuscation phase, e-mails are finally sent to RuBaST and rule-based reasoning starts (action 4 in Figure 4). Figure 5 illustrates a different architecture. The main difference from the previous schema consists in that the HMM is computed before the e-mails are sent to final recipients, that relieves client mail-tools even more than in the previous schema.

6 Acknowledgments

Thanks to Franco Salvetti for providing motivations and collaborating to the first version of this article. Rosamaria Pagano and Fortunato Roto are MSc students of Computer Science at The University of Messina.

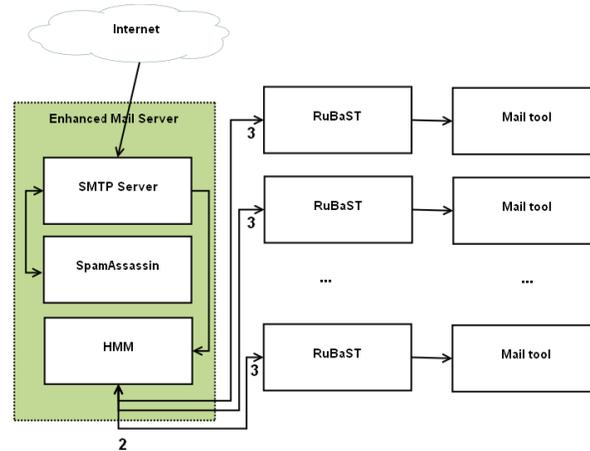


Fig. 5. Architecture of the system-wide implementation, second schema

References

1. Sergeant, M.: Internet-level spam detection and spamassassin 2.50. In: Spam Conference. (2003)
2. Marchi, M., Mileo, A., Proveti, A.: Declarative policies for web service selection. In: POLICY, IEEE Computer Society (2005) 239–242
3. Bernardoni, C., Marchi, M., Fiumara, G., Proveti, A.: Declarative web data extraction and annotation. In: 20th Workshop on Logic Programming (WLP 2006). (2005) 137–144
4. Bertino, E., Proveti, A., Salvetti, F.: Reasoning about rdf statements with default rules. In: Rule Languages for Interoperability, W3C (2005)
5. Poole, D., Macworth, A., Goebel, R.: Computational Intelligence: a Logical Approach (2nd ed). Oxford University Press (2007)
6. van Rijsbergen, C.J.: Information Retrieval (2nd ed.). Butterworths, London (1979)
7. Manning, C., Schutze, H.: Foundations of Statistical Natural Language Processing. MIT Press (1999)
8. Jurafsky, D., Martin, J.H.: Speech and Language Processing (2nd ed.). Prentice-Hall (2006)
9. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm java-prolog integration in tuprolog. *Sci. Comput. Program.* **57** (2005) 217–250
10. Lifschitz, V., Gelfond, M.: The stable model semantics for logic programming. *Proc. of 5th ILPS conference* (1988) 1070–1080
11. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm: a 25-Year Perspective*, Springer-Verlag (1999) 75–398
12. Wielemaker, J., Anjewierden, A.: An architecture for making object-oriented systems available from prolog. In: *Proc. of the 12th Int'l Workshop on Logic Programming Environments (WLPE2002)*. (2002)
13. Lee, H., Ng, A.Y.: Spam deobfuscation using a hidden markov model. In: *Proc. of the Second Conference on Email and Anti-Spam (CEAS 2005)*. (2005)

14. Cormack, G.V., Lynam, T.R.: Spam corpus creation for trec. In: Proc. of the Second Conference on Email and Anti-Spam (CEAS 2005). (2005)

Automatic Correctness Proofs for Logic Program Transformations*

Alberto Pettorossi¹ and Maurizio Proietti² and Valerio Senni¹

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

Abstract. The many approaches which have been proposed in the literature for proving the correctness of unfold/fold program transformations, consist in associating suitable well-founded orderings with the proof trees of the atoms belonging to the least Herbrand models of the programs. In practice, these orderings are given by ‘clause measures’, that is, measures associated with the clauses of the programs to be transformed. In the unfold/fold transformation systems proposed so far, clause measures are fixed in advance, independently of the transformations to be proved correct. In this paper we propose a method for the automatic generation of the clause measures which, instead, takes into account the particular program transformation at hand. During the transformation process we construct a system of linear equations and inequations whose unknowns are the clause measures to be found, and the correctness of the transformation is guaranteed by the satisfiability of that system. Through some examples we show that our method is able to establish in a fully automatic way the correctness of program transformations which, by using other methods, are proved correct at the expense of fixing sophisticated clause measures.

1 Introduction

Rule-based program transformation is a program development methodology which consists in deriving from an initial program a final program, via the application of semantics preserving transformation rules [5]. In the field of logic (or functional) programming, program transformation can be regarded as a deductive process. Indeed, programs are logical (or equational, resp.) theories and the transformation rules can be viewed as rules for deducing new formulas from old ones. The logical soundness of the transformation rules easily implies that a transformation is *partially correct*, which means that an atom (or an equation, resp.) is true in the final program only if it is true in the initial program. However, it is usually much harder to prove that a transformation is *totally correct*,

* This paper also appears in the Proceedings of the 23rd International Conference on Logic Programming (ICLP’07), Porto, Portugal, 8-13 September, 2007.

which means that an atom (or an equation, resp.) is true in the initial program if and only if it is true in the final program.

In the context of functional programming, it has been pointed out in the seminal paper by Burstall and Darlington [5] that, if the transformation rules rewrite the equations of the program at hand by using equations which belong to the same program (like the *folding* and *unfolding* rules), the transformations are always partially correct, but the final program may terminate (w.r.t. a suitable notion of termination) less often than the initial one. Thus, a sufficient condition for total correctness is that the final program obtained by transformation always terminates. This method of proving total correctness is sometimes referred to as *McCarthy's method* [13]. However, the termination condition may be, in practice, very hard to check.

The situation is similar in the case of definite logic programs, where the folding and unfolding rules basically consist in applying equivalences that hold in the least Herbrand model of the initial program. For instance, let us consider the program:

$$P: \quad p \leftarrow q \qquad r \leftarrow q \qquad q \leftarrow$$

The least Herbrand model of P is $M(P) = \{p, q, r\}$ and $M(P) \models p \leftrightarrow q$. If we replace q by p in $r \leftarrow q$ (that is, we fold $r \leftarrow q$ using $p \leftarrow q$), then we get:

$$Q: \quad p \leftarrow q \qquad r \leftarrow p \qquad q \leftarrow$$

The transformation of P into Q is totally correct, because $M(P) = M(Q)$. However, if we replace q by p in $p \leftarrow q$ (that is, we fold $p \leftarrow q$ using $p \leftarrow q$ itself), then we get:

$$R: \quad p \leftarrow p \qquad r \leftarrow q \qquad q \leftarrow$$

and the transformation of P into R is partially correct, because $M(P) \supseteq M(R)$, but it is *not* totally correct, because $M(P) \neq M(R)$. Indeed, program R does not terminate for the goal p .

A lot of work has been devoted to devise methods for proving the total correctness of transformations based on various sets of rules, including the folding and the unfolding rules. These methods have been proposed both in the context of functional programming (see, for instance, [5, 10, 17]) and in the context of logic programming (see, for instance, [3, 4, 6–9, 11, 14–16, 19–21]).

Some of these methods (such as, [3, 5, 6, 11]) propose sufficient conditions for total correctness which are explicitly based on the preservation of suitable termination properties (such as, termination of call-by-name reduction for functional programs, and universal or existential termination for logic programs).

Other methods, which we may call *implicit methods*, are based on conditions on the sequence of applications of the transformation rules that guarantee that termination is preserved. A notable example of these implicit methods is presented in [9], where integer counters are associated with program clauses. The counters of the initial program are set to 1 and are incremented (or decremented) when an unfolding (or folding, resp.) is applied. A sequence of transformations is totally correct if the counters of the clauses of the final program are all positive.

The method based on counters allows us to prove the total correctness of many transformations. Unfortunately, there are also many simple derivations

where the method fails to guarantee the total correctness. For instance, in the transformation from P to Q described above, we would get a value of 0 for the counter of the clause $r \leftarrow p$ in the final program Q , because it has been derived by applying the folding rule from clause $r \leftarrow p$. Thus, the method does not yield the total correctness of the transformation. In order to overcome the limitations of the basic counter method, some modifications and enhancements have been described in [9, 15, 16, 21], where each clause is given a *measure* which is more complex than an integer counter.

In this paper we present a different approach to the improvement of the basic counter method: instead of fixing *in advance* complex clause measures, for any given transformation we automatically generate, if at all possible, the clause measures that prove its correctness. For reasons of simplicity we assume that clause measures are non-negative integers, also called *weights*, and given a transformation starting from a program P , we look for a weight assignment to the clauses of P that proves that the transformation is totally correct.

Our paper is structured as follows. In Section 2 we present the notion of a *weighted transformation sequence*, that is, a sequence of programs constructed by applying suitable variants of the definition introduction, unfolding, and folding rules. We associate the clauses of the initial program of the sequence with some unknown weights, and during the construction of the sequence, we generate a set of constraints consisting of linear equations and inequations which relate those weights. If the final set of constraints is satisfiable for some assignment to the unknown weights, then the transformation sequence is totally correct. In Section 3 we prove our total correctness result which is based on *well-founded annotations* method proposed in [14]. In Section 4 we consider transformation sequences constructed by using also the goal replacement rule and we present a method for proving the total correctness of those transformation sequences. Finally, in Section 5 we present a method for proving predicate properties which are needed for applying the goal replacement rule.

2 Weighted Unfold/Fold Transformation Rules

Let us begin by introducing some terminology concerning systems of linear equations and inequations with integer coefficients and non-negative integer solutions.

By \mathcal{P}_{LIN} we denote the set of linear polynomials with integer coefficients. Variables occurring in polynomials are called *unknowns* to distinguish them from logical variables occurring in programs. By \mathcal{C}_{LIN} we denote the set of linear equations and inequations with integer coefficients, that is, \mathcal{C}_{LIN} is the set $\{p_1 = p_2, p_1 < p_2, p_1 \leq p_2 \mid p_1, p_2 \in \mathcal{P}_{LIN}\}$. By $p_1 \geq p_2$ we mean $p_2 \leq p_1$, and by $p_1 > p_2$ we mean $p_2 < p_1$. An element of \mathcal{C}_{LIN} is called a *constraint*. A *valuation* for a set $\{u_1, \dots, u_r\}$ of unknowns is a mapping $\sigma: \{u_1, \dots, u_r\} \rightarrow \mathbb{N}$, where \mathbb{N} is the set of natural numbers. Let $\{u_1, \dots, u_r\}$ be the set of unknowns occurring in $p \in \mathcal{P}_{LIN}$. Given a valuation σ for (a superset of) $\{u_1, \dots, u_r\}$, $\sigma(p)$ is the integer obtained by replacing the occurrences of u_1, \dots, u_r in p by $\sigma(u_1), \dots, \sigma(u_r)$, respectively, and then computing the value of the resulting arithmetic expression. A valuation

σ is a *solution* for the constraint $p_1 = p_2$ if σ is a valuation for a superset of the variables occurring in $p_1 = p_2$ and $\sigma(p_1) = \sigma(p_2)$ holds. Similarly, we define a solution for $p_1 < p_2$ and for $p_1 \leq p_2$. σ is a solution for a finite set \mathcal{C} of constraints if, for every $c \in \mathcal{C}$, σ is a solution for c . We say that a constraint c is *satisfiable* if there exists a solution for c . Similarly, we say that a set \mathcal{C} of constraints is *satisfiable* if there exists a solution for \mathcal{C} . A *weight function* for a set S of clauses is a function $\gamma : S \rightarrow \mathcal{P}_{LIN}$. A value of γ is also called a *weight polynomial*.

A *weighted unfold/fold transformation sequence* is a sequence of programs, denoted $P_0 \mapsto P_1 \mapsto \dots \mapsto P_n$, such that $n \geq 0$ and, for $k = 0, \dots, n-1$, P_{k+1} is derived from P_k by applying one of the following transformation rules: *weighted definition introduction*, *weighted unfolding*, and *weighted folding*. These rules, which will be defined below, are variants of the familiar rules without weights. For reasons of simplicity, when referring to the transformation rules, we will often omit the qualification ‘weighted’. For $k = 0, \dots, n$, we will define: (i) a weight function $\gamma_k : P_k \rightarrow \mathcal{P}_{LIN}$, (ii) a finite set \mathcal{C}_k of constraints, (iii) a set $Defs_k$ of clauses defining the new predicates introduced by the definition introduction rule during the construction of the sequence $P_0 \mapsto P_1 \mapsto \dots \mapsto P_k$, and (iv) a weight function $\delta_k : P_0 \cup Defs_k \rightarrow \mathcal{P}_{LIN}$. The weight function γ_0 for the initial program P_0 is defined as follows: for every clause $C \in P_0$, $\gamma_0(C) = u$, where u is an unknown and, for each pair C and D of distinct clauses in P_0 , we have that $\gamma_0(C) \neq \gamma_0(D)$. The initial sets \mathcal{C}_0 and $Defs_0$ are, by definition, equal to the empty set and $\delta_0 = \gamma_0$.

For every $k > 0$, we assume that P_0 and P_k have no variables in common. This assumption is not restrictive because we can always rename the variables occurring in a program without affecting its least Herbrand model. Indeed, in the sequel we will feel free to rename variables, whenever needed.

Rule 1 (Weighted Definition Introduction) Let D_1, \dots, D_m , with $m > 0$, be clauses such that, for $i = 1, \dots, m$, the predicate of the head of D_i does not occur in $P_0 \cup Defs_k$. By *definition introduction* from P_k we derive $P_{k+1} = P_k \cup \{D_1, \dots, D_m\}$.

We set the following: (1.1) for all C in P_k , $\gamma_{k+1}(C) = \gamma_k(C)$, (1.2) for $i = 1, \dots, m$, $\gamma_{k+1}(D_i) = u_i$, where u_i is a new unknown, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k$, (3) $Defs_{k+1} = Defs_k \cup \{D_1, \dots, D_m\}$, (4.1) for all D in $P_0 \cup Defs_k$, $\delta_{k+1}(D) = \delta_k(D)$, and (4.2) for $i = 1, \dots, m$, $\delta_{k+1}(D_i) = u_i$.

Rule 2 (Weighted Unfolding) Let $C: H \leftarrow G_L \wedge A \wedge G_R$ be a clause in P_k and let $C_1: H_1 \leftarrow G_1, \dots, C_m: H_m \leftarrow G_m$, with $m \geq 0$, be *all* clauses in P_0 such that, for $i = 1, \dots, m$, A is unifiable with H_i via a most general unifier ϑ_i . By *unfolding* C w.r.t. A using C_1, \dots, C_m , we derive the clauses $D_1: (H \leftarrow G_L \wedge G_1 \wedge G_R)\vartheta_1, \dots, D_m: (H \leftarrow G_L \wedge G_m \wedge G_R)\vartheta_m$, and from P_k we derive $P_{k+1} = (P_k - \{C\}) \cup \{D_1, \dots, D_m\}$.

We set the following: (1.1) for all D in $P_k - \{C\}$, $\gamma_{k+1}(D) = \gamma_k(D)$, (1.2) for $i = 1, \dots, m$, $\gamma_{k+1}(D_i) = \gamma_k(C) + \gamma_0(C_i)$, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k$, (3) $Defs_{k+1} = Defs_k$, and (4) $\delta_{k+1} = \delta_k$.

For a goal (or set of goals) G , by $\text{vars}(G)$ we denote the set of variables occurring in G .

Rule 3 (Weighted Folding) Let $C_1: H \leftarrow G_L \wedge G_1 \wedge G_R, \dots, C_m: H \leftarrow G_L \wedge G_m \wedge G_R$ be clauses in P_k and let $D_1: K \leftarrow B_1, \dots, D_m: K \leftarrow B_m$ be clauses in $P_0 \cup \text{Defs}_k$. Suppose that there exists a substitution ϑ such that the following conditions hold: (i) for $i = 1, \dots, m$, $G_i = B_i\vartheta$, (ii) there exists no clause in $(P_0 \cup \text{Defs}_k) - \{D_1, \dots, D_m\}$ whose head is unifiable with $K\vartheta$, and (iii) for $i = 1, \dots, m$ and for every variable U in $\text{vars}(B_i) - \text{vars}(K)$: (iii.1) $U\vartheta$ is a variable not occurring in $\{H, G_L, G_R\}$, and (iii.2) $U\vartheta$ does not occur in the term $V\vartheta$, for any variable V occurring in B_i and different from U .

By *folding* C_1, \dots, C_m using D_1, \dots, D_m , we derive $E: H \leftarrow G_L \wedge K\vartheta \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{C_1, \dots, C_m\}) \cup \{E\}$.

We set the following: (1.1) for all C in $P_k - \{C_1, \dots, C_m\}$, $\gamma_{k+1}(C) = \gamma_k(C)$, (1.2) $\gamma_{k+1}(E) = u$, where u is a new unknown, (2) $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \{u \leq \gamma_k(C_1) - \delta_k(D_1), \dots, u \leq \gamma_k(C_m) - \delta_k(D_m)\}$, (3) $\text{Defs}_{k+1} = \text{Defs}_k$, and (4) $\delta_{k+1} = \delta_k$.

The *correctness constraint system* associated with a weighted unfold/fold transformation sequence $P_0 \mapsto \dots \mapsto P_n$ is the set $\mathcal{C}_{\text{final}}$ of constraints defined as follows:

$$\mathcal{C}_{\text{final}} = \mathcal{C}_n \cup \{\gamma_n(C) \geq 1 \mid C \in P_n\}.$$

The following result, which will be proved in Section 3, guarantees the total correctness of weighted unfold/fold transformations. By $M(P)$ we denote the least Herbrand model of program P .

Theorem 1 (Total Correctness of Weighted Unfold/Fold Transformations). *Let $P_0 \mapsto \dots \mapsto P_n$ be a weighted unfold/fold transformation sequence constructed by using Rules 1–3, and let $\mathcal{C}_{\text{final}}$ be its associated correctness constraint system. If $\mathcal{C}_{\text{final}}$ is satisfiable then $M(P_0 \cup \text{Defs}_n) = M(P_n)$.*

Example 1. (Continuation Passing Style Transformation) Let us consider the initial program P_0 consisting of the following three clauses whose weight polynomials are the unknowns u_1, u_2 , and u_3 , respectively (we write weight polynomials on a second column to the right of the corresponding clause):

- | | |
|------------------------------|-------|
| 1. $p \leftarrow$ | u_1 |
| 2. $p \leftarrow p \wedge q$ | u_2 |
| 3. $q \leftarrow$ | u_3 |

We want to derive a continuation-passing-style program defining a predicate p_{cont} equivalent to the predicate p defined by the program P_0 . In order to do so, we introduce by Rule 1 the following clause 4 with its unknown u_4 :

- | | |
|-----------------------------------|-------|
| 4. $p_{\text{cont}} \leftarrow p$ | u_4 |
|-----------------------------------|-------|

and also the following three clauses for the unary continuation predicate cont with unknowns u_5, u_6 , and u_7 , respectively:

- | | |
|---|-------|
| 5. $\text{cont}(f_{\text{true}}) \leftarrow$ | u_5 |
| 6. $\text{cont}(f_p(X)) \leftarrow p \wedge \text{cont}(X)$ | u_6 |
| 7. $\text{cont}(f_q(X)) \leftarrow q \wedge \text{cont}(X)$ | u_7 |

where f_{true} , f_p , and f_q are three function symbols corresponding to the three predicates $true$, p , and q , respectively. By folding clause 4 using clause 5 we get the following clause with the unknown u_8 which should satisfy the constraint $u_8 \leq u_4 - u_5$ (we write constraints on a third column to the right of the corresponding clause):

$$8. \quad p_{cont} \leftarrow p \wedge cont(f_{true}) \qquad u_8 \qquad u_8 \leq u_4 - u_5$$

By folding clause 8 using clause 6 we get the following clause 9 with unknown u_9 such that $u_9 \leq u_8 - u_6$:

$$(*)9. \quad p_{cont} \leftarrow cont(f_p(f_{true})) \qquad u_9 \qquad u_9 \leq u_8 - u_6$$

By unfolding clause 6 w.r.t. p using clauses 1 and 2, we get:

$$(*)10. \quad cont(f_p(X)) \leftarrow cont(X) \qquad u_6 + u_1$$

$$11. \quad cont(f_p(X)) \leftarrow p \wedge q \wedge cont(X) \qquad u_6 + u_2$$

Then by folding clause 11 using clause 7 we get:

$$12. \quad cont(f_p(X)) \leftarrow p \wedge cont(f_q(X)) \qquad u_{12} \qquad u_{12} \leq u_6 + u_2 - u_7$$

and by folding clause 12 using clause 6 we get:

$$(*)13. \quad cont(f_p(X)) \leftarrow cont(f_p(f_q(X))) \qquad u_{13} \qquad u_{13} \leq u_{12} - u_6$$

Finally, by unfolding clause 7 w.r.t. q we get:

$$(*)14. \quad cont(f_p(X)) \leftarrow cont(X) \qquad u_7 + u_3$$

The final program is made out of clauses 9, 10, 13, and 14, marked with (*), and clauses 1, 2, and 3. The correctness constraint system \mathcal{C}_{final} is made out of the following 11 constraints.

For clauses 9, 10, 13, and 14: $u_9 \geq 1$, $u_6 + u_1 \geq 1$, $u_{13} \geq 1$, $u_7 + u_3 \geq 1$.

For clauses 1, 2, and 3: $u_1 \geq 1$, $u_2 \geq 1$, $u_3 \geq 1$.

For the four folding steps: $u_8 \leq u_4 - u_5$, $u_9 \leq u_8 - u_6$, $u_{12} \leq u_6 + u_2 - u_7$, $u_{13} \leq u_{12} - u_6$.

This system \mathcal{C}_{final} of constraints is satisfiable and thus, the transformation from program P_0 to the final program is totally correct.

3 Proving Correctness Via Weighted Programs

In order to prove that a weighted unfold/fold transformation sequence $P_0 \mapsto \dots \mapsto P_n$ is *totally correct* (see Theorem 1), we specialize the method based on *well-founded annotations* proposed in [14]. In particular, with each program P_k in the transformation sequence, we associate a *weighted program* \bar{P}_k by adding an integer argument $n (\geq 0)$, called a *weight*, to each atom $p(\mathbf{t})$ occurring in P_k . Here and in the sequel, \mathbf{t} denotes a generic m -tuple of terms t_1, \dots, t_m , for some $m \geq 0$. Informally, $p(\mathbf{t}, n)$ holds in \bar{P}_k if $p(\mathbf{t})$ ‘has a proof of weight at least n ’ in P_k . We will show that if the correctness constraint system \mathcal{C}_{final} is satisfiable, then it is possible to derive from \bar{P}_0 a weighted program \bar{P}_n where the weight arguments determine, for every clause \bar{C} in \bar{P}_n , a well-founded ordering between the head of \bar{C} and every atom in the body \bar{C} . Hence \bar{P}_n terminates for all ground goals (even if P_n need not) and the immediate consequence operator $T_{\bar{P}}$ has a unique

fixpoint [2]. Thus, as proved in [14], the total correctness of the transformation sequence follows from the *unique fixpoint principle* (see Corollary 1).

Our transformation rules can be regarded as rules for replacing a set of clauses by an equivalent one. Let us introduce the notions of implication and equivalence between sets of clauses according to [14].

Definition 1. Let I be an Herbrand interpretation and let Γ_1 and Γ_2 be two sets of clauses. We write $I \models \Gamma_1 \Rightarrow \Gamma_2$ if for every ground instance $H \leftarrow G_2$ of a clause in Γ_2 such that $I \models G_2$ there exists a ground instance $H \leftarrow G_1$ of a clause in Γ_1 such that $I \models G_1$. We write $I \models \Gamma_1 \Leftarrow \Gamma_2$ if $I \models \Gamma_2 \Rightarrow \Gamma_1$, and we write $I \models \Gamma_1 \Leftrightarrow \Gamma_2$ if ($I \models \Gamma_1 \Rightarrow \Gamma_2$ and $I \models \Gamma_1 \Leftarrow \Gamma_2$).

For all Herbrand interpretations I and sets of clauses Γ_1, Γ_2 , and Γ_3 the following properties hold:

Reflexivity: $I \models \Gamma_1 \Rightarrow \Gamma_1$

Transitivity: if $I \models \Gamma_1 \Rightarrow \Gamma_2$ and $I \models \Gamma_2 \Rightarrow \Gamma_3$ then $I \models \Gamma_1 \Rightarrow \Gamma_3$

Monotonicity: if $I \models \Gamma_1 \Rightarrow \Gamma_2$ then $I \models \Gamma_1 \cup \Gamma_3 \Rightarrow \Gamma_2 \cup \Gamma_3$.

Given a program P , we denote its associated *immediate consequence operator* by T_P [1, 12]. We denote the least and greatest fixpoint of T_P by $lfp(T_P)$ and $gfp(T_P)$, respectively. Recall that $M(P) = lfp(T_P)$.

Now let us consider the transformation of a program P into a program Q consisting in the replacement of a set Γ_1 of clauses in P by a new set Γ_2 of clauses. The following result, proved in [14], expresses the *partial correctness* of the transformation of P into Q .

Theorem 2 (Partial Correctness). *Given two programs P and Q , such that: (i) for some sets Γ_1 and Γ_2 of clauses, $Q = (P - \Gamma_1) \cup \Gamma_2$, and (ii) $M(P) \models \Gamma_1 \Rightarrow \Gamma_2$. Then $M(P) \supseteq M(Q)$.*

In order to establish a sufficient condition for the total correctness of the transformation of P into Q , that is, $M(P) = M(Q)$, we consider programs whose associated immediate consequence operators have unique fixpoints.

Definition 2 (Univocal Program). *A program P is said to be univocal if T_P has a unique fixpoint, that is, $lfp(T_P) = GFP(T_P)$.*

The following theorem is proved in [14].

Theorem 3 (Conservativity). *Given two programs P and Q , such that: (i) for some sets Γ_1 and Γ_2 of clauses, $Q = (P - \Gamma_1) \cup \Gamma_2$, and (ii) $M(P) \models \Gamma_1 \Leftarrow \Gamma_2$, and (iii) Q is univocal. Then $M(P) \subseteq M(Q)$.*

As a straightforward consequence of Theorems 2 and 3 we get the following.

Corollary 1 (Total Correctness Via Unique Fixpoint). *Given two programs P and Q such that: (i) for some sets Γ_1, Γ_2 of clauses, $Q = (P - \Gamma_1) \cup \Gamma_2$, (ii) $M(P) \models \Gamma_1 \Leftrightarrow \Gamma_2$, and (iii) Q is univocal. Then $M(P) = M(Q)$.*

Corollary 1 cannot be directly applied to prove the total correctness of a transformation sequence generated by applying the unfolding and folding rules, because the programs derived by these rules need not be univocal. To overcome this difficulty we introduce the notion of weighted program.

Given a clause C of the form $p_0(\mathbf{t}_0) \leftarrow p_1(\mathbf{t}_1) \wedge \dots \wedge p_m(\mathbf{t}_m)$, where $\mathbf{t}_0, \mathbf{t}_1, \dots, \mathbf{t}_m$ are tuples of terms, a *weighted clause*, denoted $\overline{C}(w)$, associated with C is a clause of the form:

$$\overline{C}(w): p_0(\mathbf{t}_0, N_0) \leftarrow N_0 \geq N_1 + \dots + N_m + w \wedge p_1(\mathbf{t}_1, N_1) \wedge \dots \wedge p_m(\mathbf{t}_m, N_m)$$

where w is a natural number called the *weight* of $\overline{C}(w)$. Clause $\overline{C}(w)$ is denoted by \overline{C} when we do not need refer to the weight w . A *weighted program* is a set of weighted clauses. Given a program $P = \{C_1, \dots, C_r\}$, by \overline{P} we denote a weighted program of the form $\{\overline{C}_1, \dots, \overline{C}_r\}$. Given a weight function γ and a valuation σ , by $\overline{C}(\gamma, \sigma)$ we denote the weighted clause $\overline{C}(\sigma(\gamma(C)))$ and by $\overline{P}(\gamma, \sigma)$ we denote the weighted program $\{\overline{C}_1(\gamma, \sigma), \dots, \overline{C}_r(\gamma, \sigma)\}$.

For reasons of conciseness, we do not formally define here when a formula of the form $N_0 \geq N_1 + \dots + N_m + w$ (see clause $\overline{C}(w)$ above) holds in an interpretation, and we simply say that for every Herbrand interpretation I and ground terms n_0, n_1, \dots, n_m, w , we have that $I \models n_0 \geq n_1 + \dots + n_m + w$ holds iff n_0, n_1, \dots, n_m, w are (terms representing) natural numbers such that n_0 is greater than or equal to $n_1 + \dots + n_m + w$.

The following lemma (proved in [14]) establishes the relationship between the semantics of a program P and the semantics of any weighted program \overline{P} associated with P .

Lemma 1. *Let P be a program. For every ground atom $p(\mathbf{t})$, $p(\mathbf{t}) \in M(P)$ iff there exists $n \in \mathbb{N}$ such that $p(\mathbf{t}, n) \in M(\overline{P})$.*

By erasing weights from clauses we preserve clause implications, in the sense stated by the following lemma (proved in [14]).

Lemma 2. *Let P be a program, and Γ_1 and Γ_2 be any two sets of clauses. If $M(\overline{P}) \models \overline{\Gamma}_1 \Rightarrow \overline{\Gamma}_2$ then $M(P) \models \Gamma_1 \Rightarrow \Gamma_2$.*

A weighted program \overline{P} is said to be *decreasing* if every clause in \overline{P} has a positive weight.

Lemma 3. *Every decreasing program is univocal.*

Now, we have the following result, which is a consequence of Lemmata 1, 3, and Theorems 2 and 3. Unlike Corollary 1, this result can be used to prove the total correctness of the transformation of program P into program Q also in the case where Q is not univocal.

Theorem 4 (Total Correctness Via Weights). *Let P and Q be programs such that: (i) $M(P) \models P \Rightarrow Q$, (ii) $M(\overline{P}) \models \overline{P} \Leftarrow \overline{Q}$, and (iii) \overline{Q} is decreasing. Then $M(P) = M(Q)$.*

By Theorem 4, in order to prove Theorem 1, that is, the total correctness of weighted unfold/fold transformations, it is enough to show that, given a weighted unfold/fold transformation sequence $P_0 \mapsto \dots \mapsto P_n$, we have that:

$$(P1) \ M(P_0 \cup Defs_n) \models P_0 \cup Defs_n \Rightarrow P_n$$

and there exist suitable weighted programs $\overline{P}_0 \cup \overline{Defsn}$ and \overline{P}_n , associated with $P_0 \cup Defs_n$ and P_n , respectively, such that:

$$(P2) \ M(\overline{P}_0 \cup \overline{Defsn}) \models \overline{P}_0 \cup \overline{Defsn} \Leftarrow \overline{P}_n, \text{ and}$$

$$(P3) \ \overline{P}_n \text{ is decreasing.}$$

The suitable weighted programs $\overline{P}_0 \cup \overline{Defsn}$ and \overline{P}_n are constructed as we now indicate by using the hypothesis that the correctness constraint system \mathcal{C}_{final} associated with the transformation sequence, is satisfiable. Let σ be a solution for \mathcal{C}_{final} . For every $k = 0, \dots, n$ and for every clause $C \in P_k$, we take $\overline{C} = \overline{C}(\gamma_k, \sigma)$, where γ_k is the weight function associated with P_k . For $C \in Defs_k$ we take $\overline{C} = \overline{C}(\delta_k, \sigma)$. Thus, $\overline{P}_k = \overline{P}_k(\gamma_k, \sigma)$ and $\overline{Defsk} = \overline{Defsk}(\delta_k, \sigma)$.

In order to prove Theorem 1 we need the following two lemmata.

Lemma 4. *Let $P_0 \mapsto \dots \mapsto P_k$ be a weighted unfold/fold transformation sequence. Let C be a clause in P_k , and let D_1, \dots, D_m be the clauses derived by unfolding C w.r.t. an atom in its body, as described in Rule 2. Then:*

$$M(\overline{P}_0 \cup \overline{Defsn}) \models \{\overline{C}\} \Leftrightarrow \{\overline{D}_1, \dots, \overline{D}_m\}$$

Lemma 5. *Let $P_0 \mapsto \dots \mapsto P_k$ be a weighted unfold/fold transformation sequence. Let C_1, \dots, C_m be clauses in P_k , D_1, \dots, D_m be clauses in $P_0 \cup Defs_k$, and E be the clause derived by folding C_1, \dots, C_m using D_1, \dots, D_m , as described in Rule 3. Then:*

$$(i) \ M(P_0 \cup Defs_n) \models \{C_1, \dots, C_m\} \Rightarrow \{E\}$$

$$(ii) \ M(\overline{P}_0 \cup \overline{Defsn}) \models \{\overline{C}_1, \dots, \overline{C}_m\} \Leftarrow \{\overline{E}\}$$

We are now able to prove Theorem 1. For a weighted unfold/fold transformation sequence $P_0 \mapsto \dots \mapsto P_n$, the following properties hold:

$$(R1) \ M(P_0 \cup Defs_n) \models P_k \cup (Defs_n - Defs_k) \Rightarrow P_{k+1} \cup (Defs_n - Defs_{k+1}), \text{ and}$$

$$(R2) \ M(\overline{P}_0 \cup \overline{Defsn}) \models \overline{P}_k \cup (\overline{Defsn} - \overline{Defsk}) \Leftarrow \overline{P}_{k+1} \cup (\overline{Defsn} - \overline{Defsk_{k+1}}).$$

Indeed, Properties (R1) and (R2) can be proved by reasoning by cases on the transformation rule applied to derive P_{k+1} from P_k , as follows. If P_{k+1} is derived from P_k by applying the definition introduction rule then $P_k \cup (Defs_n - Defs_k) = P_{k+1} \cup (Defs_n - Defs_{k+1})$ and, therefore, Properties (R1) and (R2) are trivially true. If P_{k+1} is derived from P_k by applying the unfolding rule, then $P_{k+1} = (P_k - \{C\}) \cup \{D_1, \dots, D_m\}$ and $Defs_k = Defs_{k+1}$. Hence, Properties (R1) and (R2) follow from Lemma 2, Lemma 4 and from the monotonicity of \Rightarrow . If P_{k+1} is derived from P_k by applying the folding rule, then $P_{k+1} = (P_k - \{C_1, \dots, C_m\}) \cup \{E\}$ and $Defs_k = Defs_{k+1}$. Hence, Properties (R1) and (R2) follow from Points (i) and (ii) of Lemma 5 and the monotonicity of \Rightarrow .

By the transitivity of \Rightarrow and by Properties (R1) and (R2), we get Properties (P1) and (P2). Moreover, since σ is a solution for \mathcal{C}_{final} and $\overline{P}_n = \overline{P}_n(\gamma_n, \sigma)$, Property (P3) holds. Thus, by Theorem 4, $M(P_0 \cup Defs_n) = M(P_n)$.

4 Weighted Goal Replacement

In this section we extend the notion of a weighted unfold/fold transformation sequence $P_0 \mapsto P_1 \mapsto \dots \mapsto P_n$ by assuming that P_{k+1} is derived from P_k by applying, besides the definition introduction, unfolding, and folding rules, also the goal replacement rule defined as Rule 4 below. The goal replacement rule consists in replacing a goal G_1 occurring in the body of a clause of P_k , by a new goal G_2 such that G_1 and G_2 are equivalent in $M(P_0 \cup Defs_k)$. Some conditions are also needed in order to update the value of the weight function and the associated constraints. To define these conditions we introduce the notion of *weighted replacement law* (see Definition 3), which in turn is based on the notion of weighted program introduced in Section 3.

In Definition 3 below we will use the following notation. Given a goal $G : p_1(\mathbf{t}_1) \wedge \dots \wedge p_m(\mathbf{t}_m)$, a variable N , and a natural number w , by $\overline{G}[N, w]$ we denote the formula $\exists N_1 \dots \exists N_m (N \geq N_1 + \dots + N_m + w \wedge p_1(\mathbf{t}_1, N_1) \wedge \dots \wedge p_m(\mathbf{t}_m, N_m))$. Given a set $X = \{X_1, \dots, X_m\}$ of variables, we will use ‘ $\exists X$ ’ as a shorthand for ‘ $\exists X_1 \dots \exists X_m$ ’ and ‘ $\forall X$ ’ as a shorthand for ‘ $\forall X_1 \dots \forall X_m$ ’.

Definition 3 (Weighted Replacement Law). Let P be a program, γ be a weight function for P , and \mathcal{C} be a finite set of constraints. Let G_1 and G_2 be goals, u_1 and u_2 be unknowns, and $X \subseteq vars(G_1) \cup vars(G_2)$ be a set of variables. We say that the *weighted replacement law* $(G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds with respect to the triple $\langle P, \gamma, \mathcal{C} \rangle$, and we write $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$, if the following conditions hold:

- (i) $M(P) \models \forall X (\exists Y G_1 \leftarrow \exists Z G_2)$, and
- (ii) for every solution σ for \mathcal{C} ,

$$M(\overline{P}) \models \forall X \forall U (\exists Y (\overline{G}_1[U, \sigma(u_1)]) \rightarrow \exists Z (\overline{G}_2[U, \sigma(u_2)]))$$

where: (1) \overline{P} is the weighted program $\overline{P}(\gamma, \sigma)$, (2) U is a variable, (3) $Y = vars(G_1) - X$, and (4) $Z = vars(G_2) - X$.

By using Lemma 2 it can be shown that, if \mathcal{C} is satisfiable, then Condition (ii) of Definition 3 implies $M(P) \models \forall X (\exists Y G_1 \rightarrow \exists Z G_2)$ and, therefore, if $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$ and \mathcal{C} is satisfiable, we also have that $M(P) \models \forall X (\exists Y G_1 \leftrightarrow \exists Z G_2)$.

Example 2. (Associativity of List Concatenation) Let us consider the following program *Append* for list concatenation. To the right of each clause we indicate the corresponding unknown.

1. $a([], L, L)$ u_1
2. $a([H|T], L, [H|R]) \leftarrow a(T, L, R)$ u_2

The following replacement law expresses the associativity of list concatenation:

$$\text{Law } (\alpha): (a(L_1, L_2, M) \wedge a(M, L_3, L), w_1) \\ \Rightarrow_{\{L_1, L_2, L_3, L\}} (a(L_2, L_3, R) \wedge a(L_1, R, L), w_2)$$

where w_1 and w_2 are new unknowns. In Example 4 below we will show that Law (α) holds w.r.t. $\langle \text{Append}, \gamma, \mathcal{C} \rangle$, where \mathcal{C} is the set of constraints $\{u_1 \geq 1, u_2 \geq 1, w_1 \geq w_2, w_2 + u_1 \geq 1\}$.

In Section 5 we will present a method, called *weighted unfold/fold proof method*, for generating a suitable set \mathcal{C} of constraints such that $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds.

Now we introduce the Weighted Goal Replacement Rule, which is a variant of the rule without weights (see, for instance, [20]).

Rule 4 (Weighted Goal Replacement) Let $C: H \leftarrow G_L \wedge G_1 \wedge G_R$ be a clause in program P_k and let \mathcal{C} be a set of constraints such that the weighted replacement law $\lambda: (G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds w.r.t. $\langle P_0 \cup \text{Defs}_k, \delta_k, \mathcal{C} \rangle$, where $X = \text{vars}(\{H, G_L, G_R\}) \cap \text{vars}(\{G_1, G_2\})$.

By applying the replacement law λ , from C we derive $D: H \leftarrow G_L \wedge G_2 \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{C\}) \cup \{D\}$. We set the following: (1.1) for all E in $P_k - \{C\}$, $\gamma_{k+1}(E) = \gamma_k(E)$, (1.2) $\gamma_{k+1}(D) = \gamma_k(C) - u_1 + u_2$, (2) $\mathcal{C}_{k+1} = \mathcal{C} \cup \mathcal{C}$, (3) $\text{Defs}_{k+1} = \text{Defs}_k$, and (4) $\delta_{k+1} = \delta_k$.

The proof of the following result is similar to the one of Theorem 1.

Theorem 5 (Total Correctness of Weighted Unfold/Fold/Replacement Transformations). *Let $P_0 \mapsto \dots \mapsto P_n$ be a weighted unfold/fold transformation sequence constructed by using Rules 1–4, and let $\mathcal{C}_{\text{final}}$ be its associated correctness constraint system. If $\mathcal{C}_{\text{final}}$ is satisfiable then $M(P_0 \cup \text{Defs}_n) = M(P_n)$.*

Example 3. (List Reversal) Let *Reverse* be a program for list reversal consisting of the clauses of *Append* (see Example 2) together with the following two clauses (to the right of the clauses we write the corresponding weight polynomials):

3. $r([], []) \leftarrow$ u_3
4. $r([H|T], L) \leftarrow r(T, R) \wedge a(R, [H], L)$ u_4

We will transform the *Reverse* program into a program that uses an accumulator [5]. In order to do so, we introduce by Rule 1 the following clause:

5. $g(L_1, L_2, A) \leftarrow r(L_1, R) \wedge a(R, A, L_2)$ u_5

We apply the unfolding rule twice starting from clause 5 and we get:

6. $g([], L, L) \leftarrow$ $u_5 + u_3 + u_1$
7. $g([H|T], L, A) \leftarrow r(T, R) \wedge a(R, [H], S) \wedge a(S, A, L)$ $u_5 + u_4$

By applying the replacement law (α), from clause 7 we derive:

8. $g([H|T], L, A) \leftarrow r(L, R) \wedge a([H], A, S) \wedge a(R, S, L)$ $u_5 + u_4 - w_1 + w_2$

together with the constraints (see Example 2): $u_1 \geq 1$, $u_2 \geq 1$, $w_1 \geq w_2$, $w_2 + u_1 \geq 1$. By two applications of the unfolding rule, from clause 8 we get:

9. $g([H|T], L, A) \leftarrow r(T, R) \wedge a(R, [H|A], L)$ $u_5 + u_4 - w_1 + w_2 + u_2 + u_1$

By folding clause 9 using clause 5 we get:

10. $g([H|T], L, A) \leftarrow g(T, L, [H|A])$ u_6

together with the constraint $u_6 \leq u_4 - w_1 + w_2 + u_2 + u_1$.

Finally, by folding clause 4 using clause 5 we get:

11. $r([H|T], L) \leftarrow g(T, L, [H])$ u_7

together with the constraint $u_7 \leq u_4 - u_5$.

The final program consists of clauses 1, 2, 3, 11, 6, and 10. The correctness constraint system associated with the transformation sequence is as follows.

For clauses 1, 2, and 3: $u_1 \geq 1, u_2 \geq 1, u_3 \geq 1.$

For clauses 11, 6, and 10: $u_7 \geq 1, u_5 + u_3 + u_1 \geq 1, u_6 \geq 1.$

For the goal replacement: $u_1 \geq 1, u_2 \geq 1, w_1 \geq w_2, w_2 + u_1 \geq 1.$

For the two folding steps: $u_6 \leq u_4 - w_1 + w_2 + u_2 + u_1, u_7 \leq u_4 - u_5.$

This set of constraints is satisfiable and, therefore, the transformation sequence is totally correct.

5 The Weighted Unfold/Fold Proof Method

In this section we present the unfold/fold method for proving the replacement laws to be used in Rule 4. In order to do so, we introduce the notions of: (i) *syntactic equivalence*, (ii) *symmetric folding*, and (iii) *symmetric goal replacement*.

A *predicate renaming* is a bijective mapping $\rho : Preds_1 \rightarrow Preds_2$, where $Preds_1$ and $Preds_2$ are two sets of predicate symbols. Given a formula (or a set of formulas) F , by $preds(F)$ we denote the set of predicate symbols occurring in F . Suppose that $preds(F) \subseteq Preds_1$, then by $\rho(F)$ we denote the formula obtained from F by replacing every predicate symbol p by $\rho(p)$. Two programs Q and R are *syntactically equivalent* if there exists a predicate renaming $\rho : preds(Q) \rightarrow preds(R)$, such that $R = \rho(Q)$, modulo variable renaming.

An application of the folding rule by which from program P_k we derive program P_{k+1} , is said to be *symmetric* if \mathcal{C}_{k+1} is set to $\mathcal{C}_k \cup \{u = \gamma_k(C_1) - \delta_k(D_1), \dots, u = \gamma_k(C_m) - \delta_k(D_m)\}$ (see Point 2 of Rule 3).

Given a program P , a weight function γ , and a set \mathcal{C} of constraints, we say that the replacement law $(G_1, u_1) \Rightarrow_X (G_2, u_2)$ holds *symmetrically* w.r.t. $\langle P, \gamma, \mathcal{C} \rangle$, and we write $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$, if the following condition holds:

(ii*) for every solution σ for \mathcal{C} ,

$$M(\overline{P}) \models \forall X \forall U (\exists Y (\overline{G}_1[U, \sigma(u_1)]) \leftrightarrow \exists Z (\overline{G}_2[U, \sigma(u_2)]))$$

where \overline{P} , U , Y , and Z are defined as in Definition 3. Note that, by Lemma 2, Condition (ii*) implies Condition (i) of Definition 3. An application of the *goal replacement rule* is *symmetric* if it consists in applying a replacement law that holds symmetrically w.r.t. $\langle P_0 \cup Defs_k, \delta_k, \mathcal{C} \rangle$. A weighted unfold/fold transformation sequence is said to be *symmetric* if it is constructed by applications of the definition and unfolding rules and by symmetric applications of the folding and goal replacement rules.

Now we are ready to present the weighted unfold/fold proof method, which is itself based on weighted unfold/fold transformations.

The Weighted Unfold/Fold Proof Method. Let us consider a program P , a weight function γ for P , and a replacement law $(G_1, u_1) \Rightarrow_X (G_2, u_2)$. Suppose that X is the set of variables $\{X_1, \dots, X_m\}$ and let \mathbf{X} denote the sequence X_1, \dots, X_m .

Step 1. First we introduce two new predicates *new1* and *new2* defined by the following two clauses: $D_1: new1(\mathbf{X}) \leftarrow G_1$ and $D_2: new2(\mathbf{X}) \leftarrow G_2$, associated with the unknowns u_1 and u_2 , respectively.

Step 2. Then we construct two weighted unfold/fold transformation sequences of the forms: $P \cup \{D_1\} \mapsto \dots \mapsto Q$ and $P \cup \{D_2\} \mapsto \dots \mapsto R$, such that the following three conditions hold:

- (1) For $i = 1, 2$, the weight function associated with the initial program $P \cup \{D_i\}$ is γ_0^i defined as: $\gamma_0^i(C) = \gamma(C)$ if $C \in P$, and $\gamma_0^i(D_i) = u_i$;
- (2) The final programs Q and R are syntactically equivalent; and
- (3) The transformation sequence $P \cup \{D_2\} \mapsto \dots \mapsto R$ is symmetric.

Step 3. Finally, we construct a set \mathcal{C} of constraints as follows. Let γ_Q and γ_R be the weight functions associated with Q and R , respectively. Let \mathcal{C}_Q and \mathcal{C}_R be the correctness constraint systems associated with the transformation sequences $P \cup \{D_1\} \mapsto \dots \mapsto Q$ and $P \cup \{D_2\} \mapsto \dots \mapsto R$, respectively, and let ρ be the predicate renaming such that $\rho(Q) = R$. Suppose that both \mathcal{C}_Q and \mathcal{C}_R are satisfiable.

(3.1) Let the set \mathcal{C} be $\{\gamma_Q(C) \geq \gamma_R(\rho(C)) \mid C \in Q\} \cup \mathcal{C}_Q \cup \mathcal{C}_R$. Then we infer:

$$\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Rightarrow_X (G_2, u_2)$$

(3.2) Suppose that the transformation sequence $P \cup \{D_1\} \mapsto \dots \mapsto Q$ is symmetric and let the set \mathcal{C} be $\{\gamma_Q(C) = \gamma_R(\rho(C)) \mid C \in Q\} \cup \mathcal{C}_Q \cup \mathcal{C}_R$. Then we infer:

$$\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$$

It can be shown that the unfold/fold proof method is sound.

Theorem 6 (Soundness of the Unfold/Fold Proof Method).

If $\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Rightarrow_X (G_2, u_2)$ then $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Rightarrow_X (G_2, u_2)$.

If $\langle P, \gamma, \mathcal{C} \rangle \vdash_{UF} (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$ then $\langle P, \gamma, \mathcal{C} \rangle \models (G_1, u_1) \Leftrightarrow_X (G_2, u_2)$.

Example 4. (An Unfold/Fold Proof) Let us consider again the program *Append* and the replacement law (α) , expressing the associativity of list concatenation, presented in Example 2. By applying the unfold/fold proof method we will generate a set \mathcal{C} of constraints such that law (α) holds w.r.t. $\langle \text{Append}, \gamma, \mathcal{C} \rangle$.

Step 1. We start off by introducing the following two clauses:

$$\begin{array}{ll} D_1. \text{ new}_1(L_1, L_2, L_3, L) \leftarrow a(L_1, L_2, M) \wedge a(M, L_3, L) & w_1 \\ D_2. \text{ new}_2(L_1, L_2, L_3, L) \leftarrow a(L_2, L_3, R) \wedge a(L_1, R, L) & w_2 \end{array}$$

First, let us construct a transformation sequence starting from $\text{Append} \cup \{D_1\}$. By two applications of the unfolding rule, from clause D_1 we derive:

$$\begin{array}{ll} E_1. \text{ new}_1([], L_2, L_3, L) \leftarrow a(L_2, L_3, L) & w_1 + u_1 \\ E_2. \text{ new}_1([H|T], L_2, L_3, [H|R]) \leftarrow a(T, L_2, M) \wedge a(M, L_3, R) & w_1 + 2u_2 \end{array}$$

By folding clause E_2 using clause D_1 we derive:

$$E_3. \text{ new}_1([H|T], L_2, L_3, [H|R]) \leftarrow \text{new}_1(T, L_2, L_3, R) \quad u_8$$

together with the constraint $u_8 \leq 2u_2$.

Now, let us construct a transformation sequence starting from $\text{Append} \cup \{D_2\}$. By unfolding clause D_2 w.r.t. $a(L_1, R, L)$ in its body we get:

$$F_1. \text{ new}_2([], L_2, L_3, L) \leftarrow a(L_2, L_3, L) \quad w_2 + u_1$$

$$F_2. \text{ new2}([H|T], L_2, L_3, [H|R]) \leftarrow a(L_2, L_3, M) \wedge a(T, M, R) \quad w_2 + u_2$$

By a symmetric application of the folding rule using clause D_2 , from clause F_2 we get:

$$F_3. \text{ new2}([H|T], L_2, L_3, [H|R]) \leftarrow \text{new2}(T, L_2, L_3, R) \quad u_9$$

together with the constraint $u_9 = u_2$.

The final programs $\text{Append} \cup \{E_1, E_3\}$ and $\text{Append} \cup \{F_1, F_3\}$ are syntactically equivalent via the predicate renaming ρ such that $\rho(\text{new1}) = \text{new2}$. The transformation sequence $\text{Append} \cup \{D_2\} \mapsto \dots \mapsto \text{Append} \cup \{F_1, F_3\}$ is symmetric.

Step 3. The correctness constraint system associated with the transformation sequence $\text{Append} \cup \{D_1\} \mapsto \dots \mapsto \text{Append} \cup \{E_1, E_3\}$ is the following:

$$\mathcal{C}_1: \{u_1 \geq 1, u_2 \geq 1, w_1 + u_1 \geq 1, u_8 \geq 1, u_8 \leq 2u_2\}$$

The correctness constraint system associated with the transformation sequence $\text{Append} \cup \{D_2\} \mapsto \dots \mapsto \text{Append} \cup \{F_1, F_3\}$ is the following:

$$\mathcal{C}_2: \{u_1 \geq 1, u_2 \geq 1, w_2 + u_1 \geq 1, u_9 \geq 1, u_9 = u_2\}$$

Both \mathcal{C}_1 and \mathcal{C}_2 are satisfiable and thus, we infer:

$$\langle \text{Append}, \gamma, \mathcal{C}_{12} \rangle \vdash_{UF}$$

$$(a(L_1, L_2, M) \wedge a(M, L_3, L), w_1) \Rightarrow_{\{L_1, L_2, L_3, L\}} (a(L_2, L_3, R) \wedge a(L_1, R, L), w_2)$$

where \mathcal{C}_{12} is the set $\{w_1 + u_1 \geq w_2 + u_1, u_8 \geq u_9\} \cup \mathcal{C}_1 \cup \mathcal{C}_2$.

Notice that the constraints $w_1 + u_1 \geq w_2 + u_1$ and $u_8 \geq u_9$ are determined by the two pairs of syntactically equivalent clauses (E_1, F_1) and (E_3, F_3) , respectively. By eliminating the unknowns u_8 and u_9 , which occur in the proof of law (α) only, and by performing some simple simplifications we get, as anticipated, the following set \mathcal{C} of constraints: $\{u_1 \geq 1, u_2 \geq 1, w_1 \geq w_2, w_2 + u_1 \geq 1\}$.

6 Conclusions

We have presented a method for proving the correctness of rule-based logic program transformations in an automatic way. Given a transformation sequence, constructed by using the unfold, fold, and goal replacement transformation rules, we associate some unknown natural numbers, called weights, with the clauses of the programs in the transformation sequence and we also construct a set of linear constraints that these weights must satisfy to guarantee the total correctness of the transformation sequence. Thus, the correctness of the transformation sequence can be proven in an automatic way by checking that the corresponding set of constraints is satisfiable over the natural numbers. However, it can be shown that our method is incomplete and, more in general, it can be shown that there exists no algorithmic method for checking whether or not any unfold/fold transformation sequence is totally correct.

As already mentioned in the Introduction, our method is related to the many methods given in the literature for proving the correctness of program transformation by showing that suitable conditions on the transformation sequence hold (see, for instance, [4, 8, 9, 16, 20, 21], for the case of definite logic programs).

Among these methods, the one presented in [16] is the most general and it makes use of *clause measures* to express complex conditions on the transformation sequence. The main novelty of our method with respect to [16] is that in [16] clause measures are fixed in advance, independently of the specific transformation sequence under consideration, while by the method proposed in this paper we automatically generate specific clause measures for each transformation sequence to be proved correct.

Thus, in principle, our method is more powerful than the one presented in [16]. For a more accurate comparison between the two methods, we did some practical experiments. We implemented our method in the MAP transformation system (<http://www.iasi.cnr.it/~proietti/system.html>) and we worked out some transformation examples taken from the literature. Our system runs on SICStus Prolog (v. 3.12.5) and for the satisfiability of the sets of constraints over the natural numbers it uses the *clpq* SICStus library.

By using our system we did the transformation examples presented in this paper (see Examples 1, 3, and 4) and the following examples taken from the literature: (i) the *Adjacent* program which checks whether or not two elements have adjacent occurrences in a list [9], (ii) the *Equal Frontier* program which checks whether or not the frontiers of two binary trees are equal [5, 21], (iii) a program for solving the *N*-queens problem [18], (iv) the *In_Correct_Position* program taken from [8], and (v) the program that encodes a liveness property of an *n*-bit shift register [16]. Even in the most complex derivation we carried out, that is, the *Equal Frontier* example taken from [21], consisting of 86 transformation steps, the system checked the total correctness of the transformation within milliseconds. For making that derivation we also had to apply several replacement laws which were proved correct by using the unfold/fold proof method described in Section 5.

Acknowledgements

We thank the anonymous referees for constructive comments.

References

1. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pp. 493–576. Elsevier, 1990.
2. M. Bezem. Characterizing termination of logic programs with level mappings. In *Proc. of NACLP, Cleveland, Ohio (USA)*, pp. 69–80. MIT Press, 1989.
3. A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In *Proceedings ALP '94*, LNCS 850, pp. 269–286, Berlin, 1994. Springer-Verlag.
4. A. Bossi, N. Cocco, and S. Etalle. On safe folding. In *Proceedings of PLILP '92, Leuven, Belgium*, LNCS 631, pp. 172–186. Springer-Verlag, 1992.
5. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

6. J. Cook and J. P. Gallagher. A transformation system for definite programs based on termination analysis. In *Proceedings of LoPSTR'94 and META'94, Pisa, Italy*, LNCS 883, pp. 51–68. Springer-Verlag, 1994.
7. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
8. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. *Proceedings PLILP '94*, LNCS 844, pp. 340–354. Springer-Verlag, 1994.
9. T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. Technical Report 179, ICOT, Tokyo, Japan, 1986.
10. L. Kott. About transformation system: A theoretical study. In *3ème Colloque International sur la Programmation*, pp. 232–247, Paris (France), 1978. Dunod.
11. K.-K. Lau, M. Ornaghi, A. Pettorossi, and M. Proietti. Correctness of logic program transformation based on existential termination. In J. W. Lloyd, editor, *Proceedings of ILPS '95*, pp. 480–494. MIT Press, 1995.
12. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987. 2nd Edition.
13. J. McCarthy. Towards a mathematical science of computation. *Proceedings of IFIP 1962*, pp. 21–28, Amsterdam, 1963. North Holland.
14. A. Pettorossi and M. Proietti. A theory of totally correct logic program transformations. *Proceedings of PEPM '04*, pp. 159–168. ACM Press, 2004.
15. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *Int. Journal on Foundations of Computer Science*, 13(3):387–403, 2002.
16. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26:264–509, 2004.
17. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Toplas*, 18(2):175–234, 1996.
18. T. Sato and H. Tamaki. Examples of logic program transformation and synthesis. Unpublished manuscript, 1985.
19. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. *Proceedings of ICLP '84*, pp. 127–138, Uppsala, Sweden, 1984. Uppsala University.
21. H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

Frame Logic under Answer Set Semantics

Mario Alviano and Francesco Calimeri and Giovambattista Ianni and
Alessandra Martello

Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy.
{alviano,calimeri,ianni,a.martello}@mat.unical.it

Abstract. Frame logic is well known as a useful ontology modeling formalism, mainly appreciated for its object-oriented features and its non-monotonic variant, capable to deal with typical nonmonotonic features such as object oriented inheritance.

This paper presents a preliminary work defining a framework for coping with frame-like syntax and higher order reasoning within an answer set programming environment. Semantics is defined by means of a translation to an ordinary answer set program. A working prototype, together with usage examples, is presented.

The work paves the way to further deeper studies about the usage of frame logic-like constructs under Answer Set Semantics.

1 Introduction

Frame Logic (F-logic) [1, 2] is a knowledge representation and ontology language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modelling capabilities supported by the object oriented data model.

The basic idea behind F-logic is to consider complex data types as in object-oriented databases, combine them with logic and use the result as a programming language. Some of the desirable features of F-logic are:

- it eases the burden of declaring properties of instance data. With respect to the flat relational model (and to its logic counterpart, where tables are modelled as predicates), it is possible to “talk” about all the facts related to a single individual within a single structure, i.e. the *molecule*.
- it eases the burden of reasoning about classes. Higher order reasoning is native in F-logic, so that the border between the notion of class and individual is smooth. With respect to the usual logic programming paradigm, it is then possible to reason about classes with the same ease of use as classic logic programming offers when it is necessary to reason on individuals.

On the other hand, Answer Set Programming (ASP) languages and systems (among the variety of such systems we recall here DLV [3] and GNT/smodels [4]), offer several other desirable features such as declarativity (they are fully compliant with a strongly assessed model-theoretic semantics [5]) and nondeterminism

(possibility to specify, in a declarative way, search spaces, strong and soft constraints [6], and more). Also, ASP shares with F-logic the possibility to reason about ontologies using nonmonotonic constructs, included nonmonotonic inheritance, as it is done in some ASP extensions conceived for modelling ontologies [7].

Nonetheless, ASP misses the useful F-logic syntax and higher order reasoning capabilities. This paper aims at extending answer set programming with some of the F-logic features. In particular, our contributions are:

1. We present the family of Frame Answer Set Programs (FAS programs), allowing usage of frame-like constructs, and of higher order atoms. Interestingly, frames may appear both in the head and in the body of rules and can be nested. Nested frames might be negated, allowing, to some extent, the same liberality in writing programs typical of nested logic programs [8].
2. We provide the semantics of FAS programs in terms of a translation to a higher order answer set program.
3. We show some example describing the ease of use of the language.
4. We present a system (DLT), able to deal with programs in F-logic like syntax, and featuring higher order reasoning. DLT is a front-end, i.e., such programs are translated in the syntax accepted by several solvers, thus enabling F-logic features under Answer Set Semantics.

The remainder of the paper is structured as follows: Section 2 introduces a “running example” that will help the reader understanding our approach. Section 3 introduces the syntax of the language FAS (Frame Answer Set). Section 4 contains a formalization of the semantics of FAS programs, while Section 5 describes how to use frame-like syntax for modeling the running example. Eventually, the system supporting FAS programs is described in Section 6, and conclusions are drawn in Section 7.

2 Running example

In order to make the capabilities of the system and the range of applications clearer, in the rest of this paper we will refer to an example herein introduced. The example refers to a typical *team building* problem. A leader of a given project needs to build a team from a set of employees, everyone having some skills. The team must be selected according to the following specifications:

- s1. The team consists of a given fixed number of employees.
- s2. At least a given set of different skills must be present in the team.
- s3. The sum of the salaries of the employees working in the team must not exceed a given budget.
- s4. The salary of each individual employee is within a specified limit.
- s5. The number of women working in the team has to reach at least a given number.

- s6. Possibly, overlappings on desired skills should be avoided, therefore at most one of employee with some required skills is preferable.
- s7. Possibly, employees with undesired skills should be not selected.

We decided to model employees by instances of the class named *employee*. This class has *properties* corresponding to the actual employee description and skill. Properties are mapped to several binary predicates (*gender, surname, married, skill, salary*). A project is encoded by instances of the *project* class, while specified properties of the project are stored in apposite predicates (*name, budget, numEmployee, maxSalary, numFemale, wantedSkills, nonWantedSkills*).

3 Syntax

We present here the syntax of FAS (Frame Answer Set) programs. We will assume the reader to be familiar with basic notions concerning with Answer Set Programming (ASP) [5].

Let \mathcal{C} be a set of constant and predicate symbols. Let \mathcal{X} be a set of variables. We conventionally denote variables with uppercase first letter (e.g. $X, Project$), while constant with lowercase first letter (e.g. $x, brown, nonWantedSkill$). A Frame Answer Set *program* (FAS program) is a set of *rules*, of the form

$$a_1 \vee, \dots, \vee a_l \leftarrow b_1, \dots, b_k, not\ b_{k+1}, \dots, not\ b_m.$$

where a_1, \dots, a_l and b_1, \dots, b_k are *literals*, $not\ b_{k+1}, \dots, not\ b_m$ are *naf-literals*, and $l \geq 0, k \geq 0, m \geq k$. An atom can be a *standard* or a *frame* atom. A *standard atom* is of the form $t_0(t_1, \dots, t_n)$, where t_0, \dots, t_n are *terms*, and t_0 represents the *predicate name* of the atom. A *term* is either a variable from \mathcal{X} , or a constant symbol from \mathcal{C} . A *literal* is either an atom p , or an expression of the form $\neg p$ (called “strongly negated” atom), where p is an atom. A *naf-literal* is either of the form b , or of the form $not\ b$, where b is a literal.

A *frame atom*, or *molecule*, can be of one of the following three forms:

- i. $obj[a_1, \dots, a_n]$
- ii. $obj : class$
- iii. $obj : class[a_1, \dots, a_n]$

where obj is a term, called the *subject* of the frame and used to define an object, $class$ is a term that defines the *class* which obj belongs to, and a_1, \dots, a_n is a list of *attribute expressions* used to define *properties* of objects.

Informally, a *frame molecule* asserts that the object has some properties as specified by the attribute expressions listed inside the brackets.

An attribute expression defines an association between an *attribute name* and one or multiple *values* than it can take. We use the *auxiliary symbols* “ \rightarrow ” and “ \multimap ” respectively to define the single value and the set-valued mapping. A *positive attribute expression* a can be of one of the following three forms:

- i. *name*
- ii. *name op value* [*op* ∈ {→, →}]
- iii. *name* → *values*

where *name* is a term (or a strongly-negated term) representing the name of the property, *value* is either a molecule or a term, and *values* is a non-empty set of *value*. If more values are given for *multi-valued attributes* (e.g, case (iii)), the values must be enclosed in curly brackets. Note that, when only one element appears inside curly brackets, we may omit these.

A *negative attribute expression* is a negated positive attribute expression. An *attribute expression* is either a positive attribute expression or a negative attribute expression.

A *plain higher order* ASP program contains only standard atoms, while a *plain* ASP program contains only standard atoms $t_0(t_1, \dots, t_n)$ where t_0 is a constant symbol.

Every object name refers to *exactly one* object, although molecules starting with the same *subject* may be combined. Since the value referred to an object attribute can be frames, molecules can be *nested*.

As an example, the following is a frame molecule:

$$\begin{aligned} brown : employee[& surname \rightarrow \text{“Mr. Brown”}, \\ & skill \rightarrow \{java, asp\}, \\ & salary \rightarrow 800, \\ & gender \rightarrow male, \\ & married \rightarrow pink] \end{aligned}$$

This defines membership of the subject *brown* to the *employee* class and asserts some values corresponding to the properties bind to this object. This frame molecule says *brown* is *male* (as expressed by the value of the attribute *gender*), is *married* to another employee identified by the subject *pink*. *brown* knows *java* and *asp* languages, as suggests the values of the *skill* property, while he has a *salary* equal to *800*. We may define a new frame molecule, like this, collecting information about the employee encoded by the subject *pink*, but we can also combine this information nesting frame molecules, as follows:

$$\begin{aligned} brown : employee[& surname \rightarrow \text{“Mr. Brown”}, \\ & skill \rightarrow \{java, asp\}, \\ & salary \rightarrow 800, \\ & gender \rightarrow male, \\ & married \rightarrow pink : employee[& surname \rightarrow \text{“Mrs. Pink”}, \\ & skill \rightarrow \{html, asp, javascript\}, \\ & salary \rightarrow 900, \\ & gender \rightarrow female, \\ & married \rightarrow brown] \\ &] . \end{aligned}$$

The following is an example of logic rule defining the profile a particular employee must have in order to be selected for project $p3$. We encoded this rule using *strong* and *naf* nested negation:

$$E[inProject \rightarrow p3] \vee E[\neg inProject \rightarrow p3] \leftarrow X : employee, \\ E : employee[skill \rightarrow \{c++, perl\}, \\ not\ married \rightarrow X : employee[\\ not\ skill \rightarrow \{c++, perl\}]].$$

This means that candidates to the project team $p3$ are employees knowing $c++$ and *perl* programming languages, but not married to another employee not knowing the same programming languages.

4 Semantics

We provide here the semantics of FAS programs in terms of a translation to a higher order ASP program. Thus we first provide the semantics of plain higher order ASP programs.

4.1 Semantics of plain higher order programs

Semantics of higher order programs is defined in terms of the traditional Gelfond-Lifshitz reduct for a ground disjunctive logic program with classical negation [5]. Given a plain higher order program P , its ground version $grnd(P)$ is given by grounding rules of P by all the possible substitutions that can be obtained using consistently elements of \mathcal{C} . A ground rule thus contains only ground atoms; the set of all possible ground atoms that can be constructed combining predicates and terms occurring in the program is usually referred to as *Herbrand base* (B_P). We remark that the grounding process substitutes also nonground predicates names with symbols from \mathcal{C} (e.g., a valid ground instance of the atom $H(brown, X)$ is $married(brown, pink)$): however, $grnd(P)$ is a standard ASP ground program.

An *interpretation* for P is a set of ground atoms, that is, an interpretation is a subset $I \subseteq B_P$. I is said to be *consistent* if $\forall a \in I$ we have that $\neg a \notin I$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal *not* A is *true* w.r.t. I if A is false w.r.t. I ; otherwise *not* A is false w.r.t. I .

Given a ground rule $r \in grnd(P)$, the head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I . A *model* for P is an interpretation M for P such that every rule $r \in grnd(P)$ is true w.r.t. M . 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 $MM(P)$.

Given a program P and an interpretation I , the *Gelfond-Lifschitz (GL) transformation* of P w.r.t. I , denoted P^I , is the set of positive rules of the form $\{a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_k\}$ such that $\{a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m\}$ is in $\text{grnd}(P)$ and $b_i \notin I$, for all $k < i \leq m$. An interpretation I for a program P is an *answer set* for P if $I \in \text{MM}(P^I)$ (i.e., I is a minimal model for the positive program P^I) [9, 5]. The set of all answer sets for P is denoted by $\text{ans}(P)$.

4.2 From FAS programs to higher order programs

We show how to reduce the frame logic-like formalism embedded in our hybrid framework to ASP, thus allowing to manipulate frames with logic programming techniques. This operational semantics is defined through a suitable algorithm which is able, given a FAS programs containing frame structures, to produce an equivalent plain higher order ASP program.

Roughly, the idea is to introduce new predicate names wrapping properties and classes. Classes are mapped to unary predicates, while properties are mapped to unary or binary predicates. Then, a FAS program is *unfolded* in order to replace frame atoms with their equivalent predicates.

The algorithm providing the semantics is called *Standardize Algorithm (S)*: it takes as input a *FAS program* P containing frame atoms; the output is a plain higher order program R . The Answer Sets of P are defined as to be the answer sets of R . The algorithm is sketched in Figure 1.

In order to better explain how S works, we show how a frame structure is examined and processed. For instance, if we consider the following frame:

$$\begin{aligned}
 E[\text{inProject} \rightarrow p3] \vee E[\text{-inProject} \rightarrow p3] \leftarrow & X : \text{employee}, \\
 & E : \text{employee}[\\
 & \text{skill} \rightarrow \{c + +, \text{perl}\}, \\
 & \text{not married} \rightarrow X].
 \end{aligned}$$

The application of S generates this output:

$$\begin{aligned}
 \text{inProject}(E, p3) \vee \text{-inProject}(E, p3) \leftarrow & \text{employee}(X), \text{skill}(E, c + +), \\
 & \text{skill}(E, \text{perl}), \text{employee}(E), \\
 & \text{not aux_e}(E, X). \\
 \text{aux_e}(E, X) \leftarrow & \text{married}(E, X).
 \end{aligned}$$

```

Standardize (INPUT:  $P$  containing frames OUTPUT:  $R$  without frames)
Let  $R = P$ ;
while  $R$  contains frame literals do
  let  $r \in R$  be a rule containing frame atoms;
  while  $r$  contains frame literals do
    remove frame  $f$  from  $r$ ;
    let  $o$  be the subject,  $L$  the set of attributes, and  $\mathbf{X}$  the set of variables of  $f$ ;
    case  $f$  appeared in the body of  $r$ :
      if  $f$  is positive then
        if  $f$  has class  $c$  then
          add  $c(o)$  to the body of  $r$ ;
        for each attribute expression  $e \in L$  do
          let  $a$  be the name, and  $V$  the set of values of  $e$ ;
          if  $e$  is positive then
            if  $V$  is empty then
              add  $a(o)$  to the body of  $r$ ;
            else
              for each term  $t \in V$  do
                add  $a(o, t)$  to the body of  $r$ ;
              for each molecule  $m \in V$  with subject  $s$  do
                add  $a(o, s)$  and  $m$  to the body of  $r$ ;
          else
            let  $e$  be in the form not  $e'$ ;
            add the frame not  $o[e']$  to the body of  $r$ ;
        else (Let  $f$  in form not  $f'$ )
          add to  $r$  a new fresh literal not  $aux_f(\mathbf{X})$ ;
          add to  $R$  a new rule  $aux_f(\mathbf{X}) \leftarrow f'$ ;
      case  $f$  appeared in the head of  $r$ :
        if  $f$  is in the form  $o : c$  (resp. in the form  $o[a \rightarrow v]$ ) then
          add to the head of  $r$  the literal  $c(o)$  (resp.  $a(o, v)$ );
        else
          add to the head of  $r$  a new atom  $aux_f(\mathbf{X})$ ;
          if  $f$  has class  $c$  then
            add  $c(o) \leftarrow aux_f(\mathbf{X})$  to  $R$ ;
          for each attribute expression  $e \in L$  do
            let  $a$  be the name, and  $V$  the set of values of  $e$ ;
            if  $V$  is empty then
              add  $a(o) \leftarrow aux_f(\mathbf{X})$  to  $R$ ;
            else
              for each term  $t \in V$  do
                add  $a(o, t) \leftarrow aux_f(\mathbf{X})$  to  $R$ ;
              for each molecule  $m \in V$  with subject  $s$  do
                add  $a(o, s) \leftarrow aux_f(\mathbf{X})$  and  $m \leftarrow aux_f(\mathbf{X})$  to  $R$ ;

```

Fig. 1. The Standardize Algorithm.

5 Examples

In this section we will provide several examples aiming at showing how to use the frame-like language.

Higher order reasoning enables the possibility to quantify over predicate names. The rule

$$related(X, Y) \leftarrow X[Z \rightarrow Y].$$

relates all the X and Y for which some property Z holds. Also higher order reasoning enables the possibility to enforce axioms that must hold on predicates,

such as

$$X : C : \text{--subClassOf}(C, D), X : D.$$

this enforces membership of an individual X to the class C when it is known that D is a subclass of C and that X is member of C .

Ease of use of frames can be seen by looking at our running example. Employees might be encoded as:

```
brown : employee[ surname → "Mr. Brown",  
                  skill → {java, asp},  
                  salary → 800,  
                  gender → male,  
                  married → pink].  
  
red : employee[ surname → "Mrs. Red",  
                skill → {java, php, perl, python},  
                salary → 1200,  
                gender → female,  
                married → black].  
  
pink : employee[ surname → "Mrs. Pink",  
                 skill → {html, asp, javascript},  
                 salary → 900,  
                 gender → female,  
                 married → brown].  
  
black : employee[ surname → "Mr. Black",  
                  skill → {c++, asp, perl, php, python},  
                  salary → 1900,  
                  gender → male,  
                  married → red].
```

While projects might be described as:

```
p1 : project[ name → "A System for a Really Nice Web Site",  
              budget → 1800,  
              numEmployee → 2,  
              maxSalary → 1000,  
              numFemale → 1,  
              wantedSkills → {html, java},  
              nonWantedSkills → {c++}].
```

$p2 : project[name \rightarrow \text{“A Semantic-Web-Oriented Extension of DLV”},$
 $budget \rightarrow 3000,$
 $numEmployee \rightarrow 2,$
 $maxSalary \rightarrow 1800,$
 $numFemale \rightarrow 1,$
 $wantedSkills \rightarrow \{c++, html\},$
 $nonWantedSkills \rightarrow \{java\}].$

Membership of a given employee E , in a given project P , can be *guessed* with the following disjunctive rule:

$$E[inProject \rightarrow P] \vee E[-inProject \rightarrow P] \leftarrow E : employee, P : project.$$

Conditions (s1), ..., (s7) can be addressed by the following FAS constraints.

- (s1) $\leftarrow P : project[numEmployee \rightarrow N_{empl},$
 $not \#count\{E : inProject(E, P)\} = N_{empl}.$
- (s2) $\leftarrow P : project[wantedSkills \rightarrow W_{sk},$
 $not \#count\{E : skill(E, W_{sk}), inProject(E, P)\} \geq 1.$
- (s3) $\leftarrow P : project[budget \rightarrow B],$
 $not \#sum\{S, E : salary(E, S), inProject(E, P)\} \leq B.$
- (s4) $\leftarrow P : project[maxSalary \rightarrow M_{sal},$
 $E : employee[inProject \rightarrow P, salary \rightarrow S], not S \leq M_{sal}.$
- (s5) $\leftarrow P : project[numFemale \rightarrow N_{fem},$
 $not \#count\{E : gender(E, female), inProject(E, P)\} \leq N_{fem}.$
- (s6) $\sim E1 : employee[skill \rightarrow S, inProject \rightarrow P[wantedSkill \rightarrow S]],$
 $E2 : employee[skill \rightarrow S, inProject \rightarrow P], E1 \neq E2. [1 :]$
- (s7) $\sim E : employee[skill \rightarrow S,$
 $inProject \rightarrow P[nonWantedSkill \rightarrow S]]. [1 :]$

It is worth noting that in the above set of constraints we take advantage of some peculiar features of the DLV system, namely weak constraints [6] and aggregates [10].

Intuitively, a weak constraint W induces an ordering among the answer sets of a given program, depending on the number of ground instances that violate W (the lesser W is violated in an answer set A , the more A is preferred). The aggregates atoms $\#count$ and $\#sum$, on the other hand, are exploited in order to count the number of values a given conjunction of atoms may have, and for summing up numeric terms appearing in a given conjunction of atoms, respectively. For the sake of simplicity, such constructs were not included in the formal syntax and semantics definition given in Section 3 and 4. We refer the reader to the cited literature for further details.

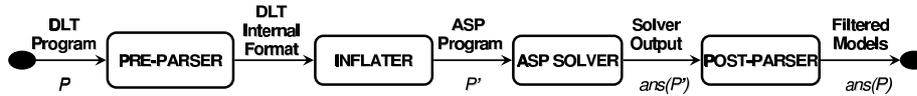


Fig. 2. Architecture of DLT System.

6 System Overview

FAS programs have been implemented within the DLT environment [11]. DLT extends the DLV system (and, to some extent, any other ASP solver) with Template predicates, Frame Logic and Higher Order predicates. The current version of the system is freely available on the DLT Web page¹.

The overall architecture of the system is shown in Figure 2. Roughly, the DLT system works as follows. A FAS program P is sent to a DLT *pre-parser*, which performs *syntactic checks*, converts frame syntax to plain syntax (by applying the algorithm \mathcal{S} defined in Section 4), and builds an internal representation of P . The DLT *Inflater* produces an equivalent program P' by processing and eliminating other special constructs of the language, such as templates; P' is piped towards an answer set solver. The answer sets $ans(P')$ of P' , computed by the solver are then converted in a readable format through the *Post-parser* module, which filters out from $ans(P')$ information about predicates and rules that were internally generated.

Additional features of the system

As a front-end system, DLT features several other constructs that can be used for enriching an answer set solver. Furthermore, although the syntax of produced programs is compliant with DLV, if special constructs of DLV (like disjunction, aggregate atoms, weak constraints) are avoided, DLT is compliant with any other solver supporting the traditional, prolog-like, syntax. Among others, DLT features are:

Template definitions. A DLT program may contain *template atoms*, that allow to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable. This feature provides a succinct and elegant way for quickly introducing new constructs using the DLT language. Syntax and semantics of template atoms are described in [11].

Frame Spaces. A Frame Space directive tells how frames are mapped to regular atoms, and can be used for defining modules where each predicate has local scope within a given frame space. The directive has syntax *@name*. From the point after the directive each frame is interpreted as belonging to the frame space *name*, and local to this. For referring to a predicate or frame belonging to

¹ <http://dlt.gibbi.com>.

a given frame space it is possible to use the syntax *atom@framespace*, like in e.g. *person(gibbi)@local*.

Internally, a frame like

$X[f \rightarrow Y]$ is rewritten as $f(X, Y, name)$.

When the directive “@.” is used, the systems switches to the default frame space, thus triggering the traditional behavior of the system.

Solvers support. DLT can virtually support any solver that accepts inputs in the format generated by DLT. Models produced by the external solver are then parsed back to the DLT syntax. The `-solver=[pathname]` option allows to specify the path of the solver. Compatibility with the systems DLV [3], DLV-EX [12], DLV-HEX [13] is provided; compatibility with S-models is guaranteed within a subset of the language. A detailed compatibility table is available on the DLT web site.

Function Symbols. Besides constant and variable terms, the DLT parser allows also functional terms. Solvers allowing function symbols are thus ready to be coupled with DLT.

7 Conclusions

We have presented a framework that allows to enrich an Answer Set Programming language with frame-like syntax and higher order reasoning. While preliminary, our work paves the way to a more formal investigation of possible semantics for F-logic under stable models semantics.

References

1. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. *Journal of the ACM* **42**(4) (1995) 741–843
2. Yang, G., Kifer, M.: Inheritance in Rule-Based Frame Systems: Semantics and Inference. *Journal on Data Semantics VII* (2006) 79–135
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlvs system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* **7**(3) (2006) 499–562
4. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*. Volume 2923 of *Lecture Notes in AI (LNAI)*., Fort Lauderdale, Florida, USA, Springer (2004) 331–335
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
6. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering* **12**(5) (2000) 845–860
7. Ricca, F., Leone, N., Bonis, V.D., Dell’Armi, T., Galizia, S., Grasso, G.: A dlp system with object-oriented features. In: *LPNMR*. (2005) 432–436
8. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence* **25**(3–4) (1999) 369–389

9. Przymusiński, T.C.: Stable Semantics for Disjunctive Programs. *New Generation Computing* **9** (1991) 401–424
10. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*. (2005) 406–411
11. Calimeri, F., Ianni, G.: Template programs for disjunctive logic programming: An operational semantics. *AI Communications* **19**(3) (2006) 193–206
12. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence* (2007. To appear.)
13. Eiter, T., Ianni, G., Tompits, H., Schindlauer, R.: A uniform integration of higher-order reasoning and external evaluations in answer set programming. In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, UK August 2-5, 2005*. (2005) 90–96

Parallel Instantiation of ASP Programs^{*}

F. Calimeri, S. Perri, and F. Ricca

Dipartimento di Matematica, Università della Calabria, 87036 Rende (CS), Italy
{calimeri, perri, ricca}@mat.unical.it

Abstract. Most of the Answer Set Programming (ASP) systems are endowed with an instantiation module, which generates a new program equivalent to the input one, but not containing variables. The instantiation process may be computationally expensive, especially for solving real-world problems, where large amounts of data have to be processed: this has been confirmed by recent applications of ASP in different emerging areas, such as knowledge management or information extraction/integration, where also scalability has been recognized as a crucial issue.

In this paper we present a new strategy for the parallel instantiation, that allows to improve both performances and scalability of ASP systems by exploiting the power of multiprocessor computers. Indeed, in the last few years, the microprocessors technologies have been moving to multi-core architectures; this makes the real Symmetric MultiProcessing (SMP) finally available even on non-dedicated machines, and paves the way to the development of more scalable softwares.

We have implemented such approach into the ASP system DLV, and carried out an experimental analysis which confirms the validity of the proposed strategy, especially for real-world applications.

1 Introduction

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [1–3]. The main advantage of ASP is its high expressiveness; unfortunately, this comes at the price of a high computational cost, which has made the implementation of efficient ASP systems a difficult task. Several efforts have been spent to this end, and, after some pioneering work [4, 5], a number of modern systems are now available. The most widespread ones are DLV [6], GnT [7], and Cmodels-3 [8]; many other support various fragments of the ASP language. The kernel modules of ASP systems operate on a ground instantiation of the input program, i.e. a program that does not contains any variable, but is semantically equivalent to the original input [9]. Consequently, any given program \mathcal{P} first undergoes the so called instantiation process computing from \mathcal{P} an equivalent ground program \mathcal{P}' . This pre-processing phase is computationally very expensive; thus, having a good and scalable instantiation procedure is, in general, a key feature of ASP systems.

^{*} 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.”

Many optimization techniques have been proposed for this purpose [10–12]; nevertheless, performances of instantiators are still not acceptable, especially in case of real-world problems, where the input data may be huge and also scalability is crucial. Indeed, the recent application of ASP in different emerging areas, such as knowledge management or information extraction/integration [13–15], have confirmed the practical need of more performant and scalable ASP systems.

Besides the other techniques, a technology that might be profitably exploited in the field of Answer Set Programming (ASP) is Symmetric MultiProcessing (SMP). SMP is a computer architecture where two or more identical processors connect to a single shared main memory resource allowing simultaneous multithread execution. In the past, only servers and workstations took advantage of it. However, recently, technology has moved to multi-core/multi-processor architectures also for entry-level systems and PCs; this permitted to enjoy the benefits of parallel processing on a large scale. These benefits include better workload balances, enhanced performances, improved scalability, not only for systems that run many processes simultaneously, but also for single (i.e., multithreaded) applications.

In this paper we present a brand new strategy for the parallel instantiation, allowing to improve both performances and scalability of ASP systems by exploiting the power of multiprocessor computers. The proposed technique exploits some structural properties of the input program in order to detect subprograms of \mathcal{P} that can be evaluated in parallel minimizing the usage of concurrency-control mechanisms, and thus minimizing the “parallel overhead”. We implemented it in an experimental version of the DLV system, and performed several experiments which confirmed the effectiveness of our technique especially in the evaluation of real-world problem instances.

The remainder of the paper is structured as follows: in Section 2 we introduce syntax and semantics of ASP; in Section 3 we briefly describe the instantiation procedures of DLV system; in Section 4 we present our parallel instantiation algorithm; in Section 5 we report and discuss the results of the experiments carried out in order to evaluate the proposed technique; in Section 6, eventually, we look at related works and draw the conclusions.

2 Answer Set Programming

We next provide a formal definition of syntax and semantics of answer set programs.

2.1 Syntax

A variable or a constant is a *term*. An *atom* is $a(t_1, \dots, t_n)$, where a is a *predicate* of arity n and t_1, \dots, t_n are terms. A *literal* is either a *positive literal* p or a *negative literal* $\text{not } p$, where p is an atom.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \vee \dots \vee a_n \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m. \quad (1)$$

where $a_1, \dots, a_n, b_1, \dots, b_m$ are atoms and $n \geq 0, m \geq k \geq 0$. The disjunction $a_1 \vee \dots \vee a_n$ is the *head* of r , while the conjunction $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$ is the *body* of r . A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*.

We denote by $H(r)$ the set $\{a_1, \dots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. A Rule r is *safe* if each variable appearing in r appears also in some positive body literal of r .

An *ASP program* \mathcal{P} is a finite set of safe rules. A *not*-free (resp., \vee -free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it.

Accordingly with the database terminology, a predicate occurring only in *facts* is referred to as an *EDB* predicate, all others as *IDB* predicates.

2.2 Semantics

Let \mathcal{P} be a program. The *Herbrand Universe* and the *Herbrand Base* of \mathcal{P} are defined in the standard way and denoted by $U_{\mathcal{P}}$ and $B_{\mathcal{P}}$, respectively.

Given a rule r occurring in \mathcal{P} , a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the variables occurring in r to constants in $U_{\mathcal{P}}$. We denote by $\text{ground}(\mathcal{P})$ the set of all the ground instances of the rules occurring in \mathcal{P} .

An *interpretation* for \mathcal{P} is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal $\text{not } A$ is *true* w.r.t. I if A is false w.r.t. I ; otherwise $\text{not } A$ is false w.r.t. I .

Let r be a ground rule in $\text{ground}(\mathcal{P})$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I .

A *model* for \mathcal{P} is an interpretation M for \mathcal{P} such that every rule $r \in \text{ground}(\mathcal{P})$ is true w.r.t. M . A model M for \mathcal{P} is *minimal* if no model N for \mathcal{P} exists such that N is a proper subset of M . The set of all minimal models for \mathcal{P} is denoted by $\text{MM}(\mathcal{P})$.

Given a program \mathcal{P} and an interpretation I , the *Gelfond-Lifschitz (GL) transformation* of \mathcal{P} w.r.t. I , denoted \mathcal{P}^I , is the set of positive rules

$$\mathcal{P}^I = \{ a_1 \vee \dots \vee a_n :- b_1, \dots, b_k \mid a_1 \vee \dots \vee a_n :- b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m \\ \text{is in } \text{ground}(\mathcal{P}) \text{ and } b_i \notin I, \text{ for all } k < i \leq m \}$$

Let I be an interpretation for a program \mathcal{P} . I is an *answer set* for \mathcal{P} if $I \in \text{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the positive program \mathcal{P}^I) [16, 1]. The set of all answer sets for \mathcal{P} is denoted by $\text{ANS}(\mathcal{P})$.

3 The DLV Instantiator

In this section we provide a description of the DLV instantiator. Given an input program \mathcal{P} , it efficiently generates a ground instantiation that has the same answer sets as the

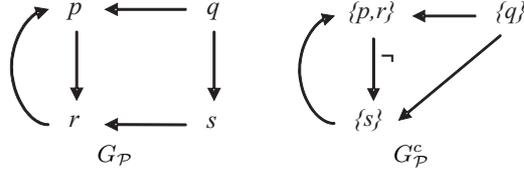


Fig. 1. Dependency and Component Graphs.

full one, but is much smaller in general [6]. Note that the size of the instantiation is a crucial aspect for the efficiency of ASP systems, since the answer set computation takes an exponential time in the size of the ground program received as input (i.e., produced by the instantiator).

In order to generate a small ground program equivalent to \mathcal{P} , the DLV instantiator generates ground instances of rules containing only atoms which can possibly be derived from \mathcal{P} , and thus avoiding the combinatorial explosion which can be obtained by naively considering all the atoms in the Herbrand Base[11]. This is obtained by taking into account some structural information of the input program, concerning the dependencies among IDB predicates.

We give now the definition the *Dependency Graph* of \mathcal{P} , which, intuitively, describes how predicates depend on each other.

Definition 1. Let \mathcal{P} be a program. The *Dependency Graph* of \mathcal{P} is a directed graph $G_{\mathcal{P}} = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of \mathcal{P} , and E contains an arc $e = (p, q)$ if there is a rule r in \mathcal{P} such that q occurs in the head of r and p occurs in a positive literal of the body of r .

The graph $G_{\mathcal{P}}$ naturally induces a partitioning of \mathcal{P} into subprograms (also called *modules*) which allows for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate p if p appears in the head of r . A *module* of \mathcal{P} is the set of rules defining all the predicates contained in a particular maximal strongly connected component (SCC) of $G_{\mathcal{P}}$. Intuitively, a module includes (among others) all rules defining mutually dependent predicates.

Example 1. Consider the following program \mathcal{P} , where a is an EDB predicate:

$$\begin{array}{ll} p(X, Y) \vee s(Y) :- q(X), q(Y), \text{not } r(X, Y) & q(X) :- a(X) \\ p(X, Y) :- q(X), r(X, Y) & r(X, Y) :- p(X, Y), s(Y) \end{array}$$

The graph $G_{\mathcal{P}}$ is illustrated in Figure 1; moreover, the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{q\}$ and $\{p, r\}$. They correspond to the three following modules:

- $\{ p(X, Y) \vee s(Y) :- q(X), q(Y), \text{not } r(X, Y). \}$
- $\{ q(X) :- a(X). \}$
- $\{ p(X, Y) :- q(X), r(X, Y). \quad p(X, Y) \vee s(Y) :- q(X), q(Y), \text{not } r(X, Y). \\ r(X, Y) :- p(X, Y), s(Y). \}$

It is possible to single out an ordered sequence C_1, \dots, C_n of SCC components of $G_{\mathcal{P}}$ (which is not unique, in general) such that the evaluation of the program module

```

Procedure Instantiate ( $\mathcal{P}$ : Program;  $G_{\mathcal{P}}$ : DependencyGraph;
    var  $\Pi$ : GroundProgram; var  $T$ : SetOfAtoms);
begin
    var  $I$ : SetOfAtoms;
    var  $C$ : SetOfPredicates;
     $T := EDB(\mathcal{P}); I = EDB(\mathcal{P}); \Pi := \emptyset;$ 
    while  $G_{\mathcal{P}} \neq \emptyset$  do
        Remove a SCC  $C$  from  $G_{\mathcal{P}}$  without incoming edges;
        InstantiateComponent( $\mathcal{P}, C, T, I, \Pi$ );
    end while
end Procedure;

```

Fig. 2. The DLV Instantiation Procedure.

corresponding to component C_i depends only on the evaluation of the components C_j such that $i < j$ ($1 \leq i < n, 1 < j \leq n$). Basically, this follows from the definition of SCC which corresponds to a maximal subset of mutually dependent predicates. Intuitively, this ordering allows one to evaluate the program one module at a time, so that all data needed for the instantiation of a module C_i have been already generated by the instantiation of the modules preceding C_i .

We sketch now a description of the instantiation process based on this principle, omitting details on how a single module is grounded and providing a general idea of the whole process.

The procedure *Instantiate* shown in Figure 2 takes as input a program \mathcal{P} to be instantiated and the dependency graph $G_{\mathcal{P}}$ and outputs a set of true atoms T and a set of ground rules containing only atoms which can possibly be derived from \mathcal{P} , such that $ANS(T \cup \Pi) = ANS(\mathcal{P})$. As previously pointed out, the input program \mathcal{P} is partitioned in modules corresponding to the maximal strongly connected components of the dependency graph $G_{\mathcal{P}}$. Such modules are evaluated one at a time starting from those that do not depend on other components, according to the ordering induced by the dependency graph.

More in detail, the algorithm initially creates a new set of atoms I that will contain the subset of the Herbrand Base relevant for the instantiation. Initially, $T = EDB(\mathcal{P})$, $I = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, a strongly connected component C , which has no incoming edge, is removed from $G_{\mathcal{P}}$, and the program module corresponding to C is evaluated by invoking *InstantiateComponent* which uses an improved version of the generalized semi-naive technique [17] for the evaluation of (recursive) rules.

Roughly, *InstantiateComponent* takes as input the component C to be instantiated, the sets T and I , and for each atom a belonging to C , and for each rule r defining a , computes the ground instances of r containing only atoms which can possibly be derived from \mathcal{P} . At the same time, it updates both the set T with the newly generated ground atoms already recognized as true, and the set I with the atoms occurring in the heads of the rules of Π . The algorithm runs on until all the components of $G_{\mathcal{P}}$ have been evaluated.

It can be shown that, given a program \mathcal{P} , the ground program $\Pi \cup T$ generated by the algorithm *Instantiate* is such that \mathcal{P} and $\Pi \cup T$ have the same answer sets.

4 The Parallel Instantiation Procedure

In this Section we describe the new instantiation algorithm that computes a ground version of a given program \mathcal{P} by exploiting parallelism. It takes advantage of some structural properties of the input program \mathcal{P} in order to detect the modules that can be evaluated in parallel *without using* “mutexes” in the main data structures.

Roughly, the parallel instantiation of the input program \mathcal{P} is based on a pattern similar to the classical producer-consumers problem. A *manager* thread (acting as a producer) identifies the components of the dependency graph of \mathcal{P} that can be run in parallel, and delegates their instantiation to a number of *instantiator* threads (acting as consumers) that exploit the same *InstantiateComponent* function introduced in Section 3.

Once the general idea has been given, we introduce some formal definition in order to detail the proposed technique. First of all, we define a new graph, called *Component Graph*, whose nodes correspond to the strongly connected components of the Dependency Graph $G_{\mathcal{P}}$. Then, we give the definition of a partial ordering among the nodes of $G_{\mathcal{P}}^c$. Please note as, with a small abuse of notation, we will indifferently refer to components of $G_{\mathcal{P}}$ and corresponding nodes of the Component Graph.

Definition 2. Given a program \mathcal{P} , let $G_{\mathcal{P}}$ be the corresponding dependency graph. The *Component Graph* of \mathcal{P} is a directed labelled graph $G_{\mathcal{P}}^c = \langle N, E, lab \rangle$, where N is a set of nodes, E is a set of arcs, and $lab : E \rightarrow \{+, -\}$ is a function assigning to each arc a label. N contains a node for each (maximal) strongly connected component of $G_{\mathcal{P}}$; E contains an arc $e = (B, A)$ if there is a rule r in \mathcal{P} such that $q \in A$ occurs in the head of r and $p \in B$ occurs in a positive (resp., negative) literal of the body of r ; $lab(e) = “+”$ (resp., $lab(e) = “-”$).

Definition 3. For any pair of nodes A, B of $G_{\mathcal{P}}^c$, A *precedes* B (denoted $A \preceq B$) if there is a *path* in $G_{\mathcal{P}}^c$ from A to B ; and A *strictly precedes* B (denoted $A \prec B$), if $A \preceq B$ and $B \not\prec A$.

Example 2. Consider the program \mathcal{P} of Example 1. The component graph of \mathcal{P} is illustrated in Figure 1. It easy to see that the node $\{p, r\}$ precedes $\{s\}$, while $\{q\}$ strictly precedes $\{s\}$.

Basically, this ordering guarantees that a node A strictly precedes a node B if the program module corresponding to A has to be evaluated before the one corresponding to B .¹

We are now ready to describe the parallel instantiation procedures exploiting this ordering. As previously pointed out, we make use of some threads: a *manager*, and a number of *instantiators* running the procedures *Manager* and *Instantiator* reported in Figure 3, respectively.

¹ Note that the presence of negative arcs in $G_{\mathcal{P}}^c$ only determines a preference among the admissible orderings induced by the dependency graph, thus it does not affect the correctness of the overall instantiation process.

```

Procedure Manager ( $\mathcal{P}$ : Program;  $G_{\mathcal{P}}^c$ : ComponentGraph;
                   var  $T$ : SetOfAtoms; var  $\Pi$ : GroundProgram);
begin
  var  $\mathcal{U}$ :SetOfComponents ; var  $\mathcal{D}$ :SetOfComponents; var  $\mathcal{R}$ :SetOfComponents;
  var  $I$ : SetOfAtoms; var  $C$ : SetOfPredicates;

   $\mathcal{D} = \emptyset$ ;  $\mathcal{R} = \emptyset$ ;  $\mathcal{U} = \text{nodes}(G_{\mathcal{P}}^c)$ 
   $T := \text{EDB}(\mathcal{P})$ ;  $I = \text{EDB}(\mathcal{P})$ ;  $\Pi := \emptyset$ ;
  while ( $\mathcal{U} \neq \emptyset$ )
    for all  $C \in \mathcal{U}$ ;
      if ( $\text{canBeRun}(C, \mathcal{U}, \mathcal{R}, G_{\mathcal{P}}^c)$ )
        begin
           $\mathcal{R} = \mathcal{R} \cup \{C\}$ ;
           $\text{Spawn}(\text{Instantiator}, \mathcal{P}, C, \mathcal{U}, \mathcal{R}, \mathcal{D}, T, I, \Pi)$ ;
        end if
      end
    end

Procedure Instantiator ( $\mathcal{P}$ : Program;  $C$ : Component; var  $\mathcal{U}$ : SetOfComponents;
                        var  $\mathcal{R}$ : SetOfComponents; var  $\mathcal{D}$ : SetOfComponents;
                        var  $T$ : SetOfAtoms; var  $I$ : SetOfAtoms; var  $\Pi$ : GroundProgram);
begin
   $\text{InstantiateComponent}(\mathcal{P}, C, T, I, \Pi)$ ;
   $\mathcal{D} = \mathcal{D} \cup \{C\}$ ;
   $\mathcal{R} = \mathcal{R} - \{C\}$ ;
   $\mathcal{U} = \mathcal{U} - \{C\}$ ;
end

```

Fig. 3. The Parallel Instantiation Procedures.

The *Manager* procedure takes as input both a program \mathcal{P} to be instantiated and its Component Graph $G_{\mathcal{P}}^c$; it outputs both a set T of true atoms and a set of ground rules Π , such that $\text{ANS}(T \cup \Pi) = \text{ANS}(\mathcal{P})$.

First of all, the sets T , I , and Π are initialized like in the standard DLV Instantiator. Moreover, three new sets of components are created: \mathcal{U} (which stands for *Undone*) represents the components of \mathcal{P} that have still to be processed, \mathcal{D} (which stands for *Done*) those that have already been instantiated, and \mathcal{R} (which stands for *Running*) those currently being processed.

Initially, \mathcal{D} and \mathcal{R} are empty, while \mathcal{U} contains all the nodes of $G_{\mathcal{P}}^c$. The manager checks, by means of function *canBeRun* described below, whether components in \mathcal{U} can be instantiated. As soon as some C is processable, it is added to \mathcal{R} , and a new instantiator thread is spawned in order to instantiate C by exploiting the *InstantiateComponent* function defined in Section 3. Once the instantiation of C has been completed, C is moved from \mathcal{R} to \mathcal{D} , and deleted from \mathcal{U} . The manager thread goes on until all the components have been processed (i.e., $\mathcal{U} = \emptyset$).

The function *canBeRun*, as the name suggests, checks whether a component C can be *safely* evaluated (i.e. without requiring “mutexes” in the main data structures) by exploiting the following definition:

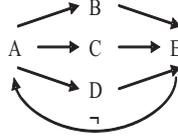


Fig. 4. An example of Component Graph.

Definition 4. Let \mathcal{U} be the set of components which still have to be processed. We say that $C \in \mathcal{U}$ *can be run* if $\forall A \in \mathcal{U}$ at least one of the following conditions holds:

- (i) $C \prec A$;
- (ii) $C \preceq A$ and $\nexists A' \in \mathcal{U}$ s.t. exists an arc $e = (A', C)$ of $G_{\mathcal{P}}^c$ with $lab(e) = "+"$, and $\forall K \in \mathcal{R}$, do not exist arcs e', e'' of $G_{\mathcal{P}}^c$ such that $e' = (R, C)$ and $e'' = (C, R)$.

Basically, this definition ensures that (i) a component C is not evaluated before all the components strictly preceding C (w.r.t. the partial ordering defined above) have been processed; and, (ii) if C appears in a cycle of $G_{\mathcal{P}}^c$ then it is selected only if it has no positive incoming edges and does not directly depend on some currently running component. The two conditions of Definition 4 checked by function *canBeRun* guarantee the correctness, since they respect the dependencies of $G_{\mathcal{P}}$ (as for the standard instantiation algorithm defined in Section 3).

Moreover, *canBeRun* singles out components that can be evaluated in parallel without using “mutex” locks in the data structures that implement the sets T and I ; this allows us to save resources and reduce the time spent in lock-contentions. It is easy to see that two components C_1 and C_2 are “selected” only if any predicate p occurring in the body of some rule of C_1 does not appear in the head of some rule of C_2 , and vice versa. If the data structures implementing the sets T and I properly store the ground atoms in different containers (i.e., one for each predicate name, as in DLV), then no “mutex” lock is needed to protect them: during the evaluation of a rule, an instantiator thread may write in the container of an atom a only when a rule defining a is processed; thus, it will never write in a location being accessed by another instantiator.

Interestingly, condition (ii) of Definition 4 allows one to run in parallel even components appearing in cycles of $G_{\mathcal{P}}^c$ (i.e., components that are, somehow, interdependent). This can be illustrated by the following example:

Example 3. Consider the Component Graph of Figure 4. All the nodes of the graph are involved in a cycle; thus, the evaluation of each component is somehow dependent on the evaluation of each other. However, condition (ii) of Definition 4 allows to select A to be evaluated first. While A is running, no other component can be processed, because both conditions (i) and (ii) are violated for all of them. Once the instantiation of A has ended, component B can be run, because it satisfies condition (ii). At the same time, by virtue of condition (ii), also components C and D can be run. Then, component E can be evaluated only when the instantiations of all B , C and D have been completed.

It is important noting that the actual implementation is more involved, but only because of technical reasons. First of all, the auxiliary control structures (like \mathcal{U} , \mathcal{D} and \mathcal{R}) are properly protected by “mutex” locks, and the busy waiting is properly avoided.

Finally, we also have to deal with additional structures which allow the user to set the maximum number of instantiator threads. We do not believe that these technical issues may help to get a better insight, but they are rather lengthy in description; for this reason, we refrain from discussing them here.

5 Experiments

In order to consistently evaluate the parallel grounding technique described in Section 4, we have implemented it as an extension of the DLV system, and performed some experiments. We took into account several problems belonging to different applications, ranging from classical ASP benchmarks to “real-word” applications.

We have compared the prototype with the official DLV [18] release² on which it is based. In addition, we considered the maximum number of concurrent instantiator threads as a parameter; thus, we deal with the following versions of DLV:

- **dl.release**: the original DLV system release without parallel grounding;
- **dl.th.X**: the modified DLV system with X independent working threads (X ranges from 1 to 4).

All the binaries have been built with GCC 3.4 (the same used to build the original DLV release), statically linking the Posix Thread Library.

Experiments have been performed on a machine equipped with two Intel Xeon HT (single core) CPUs clocked at 3.60GHz with 1 MB of Level 2 Cache and 3GB of RAM, running Debian GNU Linux (kernel 2.4.27-2-686-smp). This machine is capable of *simultaneously* run (i.e., each thread executed on a different processing unit) at most 4 threads; with more, the system performs poorly because of the preemptive thread scheduling overhead. This has been confirmed by the experiments; thus, we decided to omit here the results obtained by allowing more than 4 concurrent instantiator threads. Limiting to 4 concurrent instantiator threads does not eliminate the effects of preemption, but, reasonably, they become negligible; indeed, we ran the tests on an “unloaded” machine, and the operating system always tries to schedule active threads on free CPUs.

Time measurements have been performed by means of the `time` command shipped with the above cited version of Debian GNU Linux. Unfortunately, we could not consider the total CPU times³, because, in case of multi-threaded applications, they result as the sum of the time spent by the process on *each* processing unit (e.g., when a process fully exploits simultaneously two processors for 5 minutes, the total reported CPU time is 10 minutes). We decided to overcome the problem by considering the so called *real* time, based on the system wall-clock time. Obviously, this measure is less accurate than the total CPU time, since it unavoidably includes the time spent by other processes in the system (even by unrelated operating system routines). In order to obtain more reliable information, we have repeated each test three times, and provide here both average and standard deviation of the results.

² Official DLV release, July 14th 2006.

³ The sum of *user* and *system* time; we refer the reader to *time* manual pages for a detailed description of these quantities [19].

In the following, we describe the benchmark problems, and finally report and discuss the results of the experiments.

5.1 Benchmark Programs

We provide here a brief description of the problems considered for the experiments. In order to meet the space constraints, we refrain from showing the encodings (consider that some are automatically generated, and are very long and involved). However, they are available at http://www.mat.unical.it/parallel/cilc_inst.tar.gz.

3-Colorability. This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors.

Ancestor. Given a *parent* relationship over a set of persons, find the genealogy tree of each one. It is a classical deductive database problem exploiting recursive rules.

Knowledge Discovery. Given an ontology and a text document, an ASP program classifies the document w.r.t. the ontology. Basically, the goal is to associate the document contents to one or more concepts in the given ontology: a document is associated to the concepts it deals with. Problems have been provided by the company EXEURA s.r.l. [20].

Player. A data integration problem. Given some tables containing discording data, find a repair where some key constraints are satisfied. The problem was originally defined within the EU project INFOMIX [15].

Hypertree Decomposition. Compute a k-width complete hypertree decomposition [21] of a given query Q in a given predicate P.

ETL Workflow. In general, ETL stands for Extraction Transformation and Loading. Here the goal is to emulate, by means of an ASP program, the execution of a workflow, in which each step constitutes a transformation to be applied to some data (in order to query for and/or extract implicit knowledge). We considered the encoding of three different steps, automatically generated by a software working on some american insurance data. Problems have been provided by the company EXEURA s.r.l. [20].

Cristal. A deductive databases application that involves complex knowledge manipulations. The application was originally developed at CERN [22].

Timetabling. A real timetable problem from the faculty of Science of the University of Calabria. We have considered for the evaluation the programs that the faculty exploited for two different academic years.

The above-mentioned problems can be roughly divided into two classes, with respect to their structure. One contains problems having very “dense” dependency graphs (meaning that there are only few components), like 3-Colorability or Ancestor. The other contains problems featuring several rules belonging to independent components of the dependency graph, like ETLs or Timetablins. Therefore, the parallel instantiation of the first might only moderately be profitable, while the latter should be easier grounded in parallel; having both classes of problems allows one to get a sharpen picture of the behavior of our prototype.

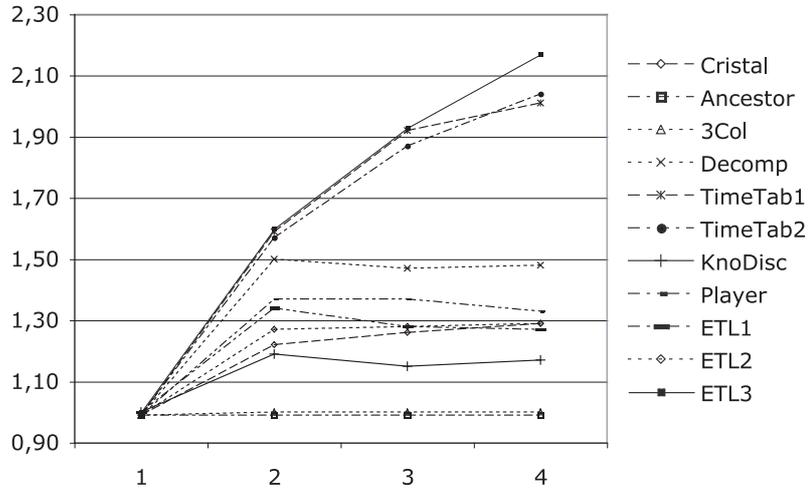


Fig. 5. Average CPU usage.

5.2 Results

The results of the experimental activities are summarized in Table 1 and Table 2, showing average total execution times and average instantiation times, respectively, both in seconds. The first consists of the overall time spent by DLV, from the invocation by the OS to the end of the instantiation (including parsing, output printing, etc.)⁴, while the second takes into account exactly the time spent by the instantiation itself.

We wanted also to outline the amount of parallel computation resources actually exploited; to this extent, in Figure 5 we report the average CPU usages⁵. The graph draws the CPU usage w.r.t. to the number of allowed threads: “1.00” means that the process “took” a single CPU, while higher values mean an higher rate of parallel execution. It is important recalling also CPU usages take into account the *overall* time spent by DLV.

We discuss now the results. At a first glance, it is easy to notice as, apart from few exceptions, as the number of allowed threads increases, the performances get better. However, among all the considered problems, one can observe different behaviors.

First of all, 3col and Ancestor do not enjoy any improvement. This was expected, since their dependency graphs are almost composed by a single huge component, thus preventing the possibility of taking advantage from our parallel techniques. Indeed, they settle at the bottom of the graph of Figure 5. Nevertheless, times are not affected by noticeable overheads, even w.r.t. the official DLV release.

⁴ This means, both the parallel instantiation and the other non-parallel phases. It is worthwhile stating that the the component graph is computed by the old DLV version as well as the new one; in addition, we verified that the time spent in the computation is actually negligible.

⁵ Computed as $((User + System)/Real)$; see [19]. Please note that this does not rate parallelism, but the exploitation of the CPUs, thus giving us an idea about how much the execution has actually been parallelized, even if it is machine-dependent.

It's then possible to identify a set of instances (namely: Knowledge discovery, ETL1, ETL2, Player, Decomp) that, as soon as the number of allowed threads moves from one to two, show an appreciable gain in instantiation time, all above 20% (e.g.: Knowledge Discovery passes from 2.06s to 1.86s; ETL1 from 64.35s to 53.93s). Then, the advantage does not grow, in practice, with the number of allowed threads; as an example, allowing more than three threads for solving Knowledge discovery is useless (1.86s, 1.53s and 1.65s with two, three and four threads, respectively). Almost the same observation can be made for ETL1, ETL2 and Decomp, while Player still exhibits a little performance gain even with four threads. Indeed, looking at the graph of Figure 5, all these instances show an almost flat pattern over 2 allowed threads.

Unfortunately, we note that something strange happens when we look at total average times: for Decomp and Knowledge discovery, all the *dl.thX* executables show a clear degradation in the performances if compared with the DLV release. We have investigated this strange phenomenon, and discovered that it is actually a technological issue, concerning the standard STL [23] multithreaded memory allocator. In fact, the DLV system heavily relies on STL data structures; these exploit a memory allocator function that suffers from a dramatic performance degradation when linked against a multithreaded executable [24]. In our case, this sometimes neutralizes all the benefits provided by parallelism. Fortunately, in case of ETL1 and ETL2, this does not waste all the gain, which still stands on about 23% also in the total execution times (ETL1, for instance, moves from 64.99s to 50.4s). This technological problem can be fixed, as indicated for instance in [24]; since the implementation takes quite some time, we left it as a future work.

Finally, a last set of instances (namely: ETL3, Cristal, Timetabling 1, Timetabling 2), clearly exhibits a performance gain growing as the number of allowed threads increases. Instantiation times improvements go from about 18% for Cristal (which passes from 4.17s with one thread to 3.58s with four) to about 40% for Timetabling 1 (from 10.64s to 6.42s). In the graph of Figure 5 the patterns related to these problems are monotonically increasing (in particular, when four threads are allowed, the exploited cpu usage grows up to a factor of 2). These benefits still survive in the total execution times for all instances, apart from Timetabling 2. The difference with this is due to the same memory allocation drawbacks previously discussed⁶.

Summarizing, best improvements have been observed within the last set of problems; it is worthwhile noting that all of them come from concrete applications, thus confirming that our approach can be profitably exploited to improve performances of ASP while dealing with real world problems.

For the sake of completeness, it is important noting that the Intel Hyper Threading (HT) technology [25] (implemented by the machine exploited for the experiments) works by duplicating certain sections of the processor, but not all the main execution resources. Basically, each processor pretends to be two "logical" processors in front of the host operating system which, thus, can schedule four threads or processes simultaneously (two processes/threads per CPU); however, these will compete for some important execution resources (e.g., the cache). Consequently, there is only an approximation

⁶ Intuitively, these drawbacks increase their weight when memory allocation functions are more frequently invoked.

Problem	dl.release	dl.th1	dl.th2	dl.th3	dl.th4
<i>Cristal</i>	4.04 (0.07)	4.17 (0.08)	4.23 (0.08)	4.41 (0.12)	4.66 (0.36)
<i>Ancestor</i>	51.78 (0.54)	52.15 (0.48)	52.23 (0.48)	52.68 (0.35)	52.16 (0.64)
<i>3Col</i>	15.83 (0.35)	15.71 (0.05)	14.90 (0.30)	15.40 (0.69)	15.32 (0.35)
<i>Decomp</i>	6.91 (0.00)	8.83 (0.01)	7.41 (0.77)	6.81 (0.07)	7.23 (0.25)
<i>TimeTab₁</i>	7.27 (0.10)	12.14 (0.04)	9.56 (0.56)	8.50 (0.06)	7.93 (0.34)
<i>TimeTab₂</i>	12.42 (0.46)	12.85 (0.42)	10.98 (1.99)	8.59 (0.55)	10.78 (3.43)
<i>KnoDisc</i>	3.41 (0.00)	5.29 (0.00)	5.09 (0.00)	4.76 (0.00)	4.87 (0.15)
<i>Player</i>	6.87 (0.02)	7.00 (0.02)	5.52 (0.47)	5.56 (0.46)	5.34 (0.06)
<i>ETL₁</i>	64.72 (0.35)	64.99 (0.61)	54.58 (2.77)	49.61 (0.61)	50.40 (0.11)
<i>ETL₂</i>	64.25 (0.22)	64.26 (0.42)	49.70 (0.56)	49.89 (0.13)	49.92 (0.53)
<i>ETL₃</i>	182.38 (0.38)	186.72 (0.43)	126.82 (10.21)	118.57 (1.53)	120.97 (2.42)

Table 1. Average Real Execution Times (standard deviations within parentheses).

Problem	dl.release	dl.th1	dl.th2	dl.th3	dl.th4
<i>Cristal</i>	3.96 (0.07)	4.03 (0.07)	3.31 (0.08)	3.33 (0.07)	3.44 (0.11)
<i>Ancestor</i>	51.75 (0.54)	52.08 (0.48)	52.16 (0.48)	52.61 (0.35)	52.08 (0.64)
<i>3Col</i>	15.67 (0.35)	15.40 (0.05)	14.59 (0.30)	15.10 (0.69)	15.01 (0.35)
<i>Decomp</i>	6.65 (0.00)	8.33 (0.01)	6.92 (0.77)	6.32 (0.07)	6.73 (0.25)
<i>TimeTab₁</i>	6.51 (0.10)	10.64 (0.04)	8.06 (0.54)	6.99 (0.05)	6.42 (0.34)
<i>TimeTab₂</i>	11.46 (0.46)	10.97 (0.42)	9.08 (1.98)	6.70 (0.55)	8.90 (3.44)
<i>KnoDisc</i>	1.88 (0.00)	2.06 (0.01)	1.86 (0.00)	1.53 (0.01)	1.65 (0.15)
<i>Player</i>	6.76 (0.02)	6.79 (0.01)	5.31 (0.47)	5.36 (0.46)	5.13 (0.07)
<i>ETL₁</i>	64.32 (0.35)	64.35 (0.62)	53.93 (2.77)	48.96 (0.61)	49.75 (0.11)
<i>ETL₂</i>	64.00 (0.22)	63.88 (0.41)	49.31 (0.56)	49.50 (0.13)	49.48 (0.57)
<i>ETL₃</i>	180.83 (0.37)	184.17 (0.44)	124.24 (10.19)	115.96 (1.61)	118.41 (2.42)

Table 2. Average Grounding Times (standard deviations within parentheses).

of the behavior of a true four processor machine. This effect cannot be avoided with the standard linux SMP kernel, since it is not aware of all the HT peculiarities. Thus, with pure multi-processor or multi-core machines, the performances of our technique should be even better.

6 Related Work and Conclusions

In this paper, we proposed a new technique for the parallel computation of the instantiation of ASP programs. It exploits some structural properties of the input program \mathcal{P} in order to detect modules of \mathcal{P} that can be evaluated in parallel.

As a matter of fact, the exploitation of parallel techniques for computing answer sets is not new [26–28]; however, our approach is not comparable with the existing ones, since the latter concern the model generation task, instead of the instantiation. Nonetheless, a lot of work has been done in the fields of logic programming and deductive databases [17, 29–39]; still, the techniques are comparable to a limited extent to the one illustrated here: some of them apply to syntactically restricted classes of programs, and some others requires an heavy usage of concurrency-control mechanisms. The only one comparable to the present is the so-called *stream* parallelism, where all the

rules are evaluated simultaneously: basically, the information resulting from the evaluation of each rule is passed to the ones depending on it, like in a pipeline. However, this scheme suffers from heavy communication overheads, while our approach minimizes the usage of “mutexes” in the main data structures, thus reducing the overhead introduced by the concurrency-control constructs.

We have implemented our strategy producing an experimental version of the DLV system, and performed several experiments on a SMP-based machine. The obtained results confirmed, on the one hand, the effectiveness of our technique, which allows one to save real (wall-clock) time, especially while evaluating real-world problem instances; on the other hand, they outlined some annoying technical issues due to the usage of the standard STL multithreaded memory allocator, which is widely considered performance-wise not optimal [24].

We plan to improve the current implementation by solving the problems concerning STL memory allocation performances; nonetheless, we want to extend our parallel grounding technique as well, in order to exploit parallelism also during the instantiation of a single component.

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. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: *Logic-Based Artificial Intelligence*. Kluwer (2000) 79–103
4. Bell, C., Nerode, A., Ng, R.T., Subrahmanian, V.: Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *JACM* **41** (1994) 1178–1215
5. Subrahmanian, V., Nau, D., Vago, C.: WFS + Branch and Bound = Stable Models. *IEEE TKDE* **7**(3) (1995) 362–377
6. 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
7. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM TOCL* **7**(1) (2006) 1–37
8. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: *LPNMR’05*. LNCS 3662
9. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: A Deductive System for Nonmonotonic Reasoning. In J. Dix and U. Furbach and A. Nerode, ed.: *LPNMR’97*. LNCS 1265
10. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In *DDL’99*, Prolog Association of Japan (1999) 135–139
11. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In: *LPNMR’01*. LNCS 2173
12. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: *NMR 2004*. (2004) 258–266
13. Ruffolo, M., Leone, N., Manna, M., Sacca’, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: *Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation*, Bath, UK (2005)
14. Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: *Discovery Science*. (2004) 380–387

15. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kařka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszki, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
16. Przymusiński, T.C.: Stable Semantics for Disjunctive Programs. *NGC* **9** (1991) 401–424
17. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
18. Faber, W., Pfeifer, G.: DLV homepage (since 1996) <http://www.dlvsystem.com/>.
19. Debian: Manual pp. about using a GNU/Linux system <http://packages.debian.org/stable/doc/manpages/>.
20. Exeura s.r.l., Homepage <http://www.exeura.it/>.
21. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions and Tractable Queries. *JCSS* (2002)
22. CRISTAL project, homepage <http://proj-cristal.web.cern.ch/>.
23. Stepanov, A., Lee, M.: The Standard Template Library. (Part of the evolving ANSI C++ standard. <ftp://butler.hpl.hp.com/stl/>) (1995)
24. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. In: ASPLOS. (2000) 117–128
25. Koufaty, D., Marr, D.T.: Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro* **23**(2) (2003) 56–65
26. Finkel, R.A., Marek, V.W., Moore, N., Truszczynski, M.: Computing stable models in parallel. In: Answer Set Programming. (2001)
27. Gressmann, J., Janhunen, T., Mercer, R.E., Schaub, T., Thiele, S., Tichy, R.: Platypus: A Platform for Distributed Answer Set Solving. In: LPNMR. (2005) 227–239
28. Pontelli, E., El-Khatib, O.: Exploiting Vertical Parallelism from Answer Set Programs. In: Answer Set Programming. (2001)
29. Wolfson, O., Silberschatz, A.: Distributed Processing of Logic Programs. In: SIGMOD Conference. (1988) 329–336
30. Wolfson, O.: Parallel Evaluation of Datalog Programs by Load Sharing. *J. Log. Program.* **12**(3&4) (1992) 369–393
31. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.* **23**(4) (2001) 472–602
32. Clark, K.L., Gregory, S.: Parlog: Parallel Programming in Logic. *ACM Trans. Program. Lang. Syst.* **8**(1) (1986) 1–49
33. Ramakrishnan, R.: Parallelism in Logic Programs. *Ann. Math. Artif. Intell.* **3**(2-4) (1991) 295–330
34. Leone, N., Restuccia, P., Romeo, M., Rullo, P.: Expliciting Parallelism in the Semi-Naive Algorithm for the Bottom-up Evaluation of Datalog Programs. *Database Technology* **4**(4) (1993) 245–158
35. de Kergommeaux, J.C., Codognot, P.: Parallel Logic Programming Systems. *ACM Comput. Surv.* **26**(3) (1994) 295–336
36. Wu, Y., Pontelli, E., Ranjan, D.: Computational Issues in Exploiting Dependent And-Parallelism in Logic Programming: Leftness Detection in Dynamic Search Trees. In: LPAR. (2005) 79–94
37. Inoue, K., Koshimura, M., Hasegawa, R.: Embedding Negation as Failure into a Model Generation Theorem Prover. In: CADE. (1992) 400–415
38. Ramakrishnan, R., Ullman, J.D.: A Survey of Deductive Database Systems. *JLP* **23**(2) (1995) 125–149
39. Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. *IEEE Trans. Knowl. Data Eng.* **7**(1) (1995) 163–176

Implementation and Evaluation of Look-Back Techniques and Heuristics in DLV^{*}

Wolfgang Faber¹, Nicola Leone¹, Marco Maratea^{1,2}, and Francesco Ricca¹

¹ Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{faber, leone, maratea, ricca}@mat.unical.it

² DIST, University of Genova, 16145 Genova, Italy
marco@dist.unige.it

Abstract. Answer Set Programming (ASP) is a programming paradigm based on logic rules. It is purely-declarative and allows for both disjunction in the head of the rules and nonmonotonic negation in the body. ASP is very expressive: any property whose complexity is in the second level of the polynomial hierarchy can be expressed with the language of ASP, thus it is strictly more powerful than propositional logic under standard complexity conjectures. DLV is the state-of-the-art *disjunctive* ASP system, and it is based on an algorithm using backtracking search, like the vast majority of the currently available ASP systems. Despite its efficiency, until recently, DLV did not incorporate any “backjumping” techniques (neither did other disjunctive ASP systems). Related, DLV could not use “look-back” information accumulated for backjumping in its heuristics, which have been shown in related research areas to be crucial on large benchmarks stemming from applications. In this paper, we focus on the experimental evaluation of the look-back algorithms and heuristics that have been implemented in DLV. We have conducted a wide experimental analysis considering both randomly-generated and structured instances of the 2QBF problem (the canonical problem for the complexity classes Σ_2^P and Π_2^P). The results show that the new look-back techniques significantly improve the performance of DLV, being performance-wise competitive even with respect to “native” QBF solvers.

1 Introduction

Answer Set Programming (ASP) [1, 2] is a purely-declarative programming paradigm based on nonmonotonic reasoning and logic programming. The idea of answer set programming is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver to find such solutions [3]. The language of ASP is very expressive, allowing for both disjunction in the head of the rules and nonmonotonic negation in the body, and able to represent every property in the second level of the polynomial hierarchy. Therefore, ASP is strictly more powerful than propositional logic unless $P = NP$.

DLV is the state-of-the-art *disjunctive* ASP system, and it is based on an algorithm relying on backtracking search, like most other competitive ASP systems. Until recently,

^{*} 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.”

DLV did not incorporate any “look-back” techniques, like “backjumping” procedures and “look-back” heuristics. By “backjumping” [4] we refer to an optimized recovery upon inconsistency during the search where, instead of restoring the state of the search up to the previous choice and then “flipping” its value, we try to “jump over” choices that are not relevant for the inconsistency we met. This is done by means of a reason calculus, which records information about the literals (“reasons”) whose truth has caused the truth of other derived literals.

Look-back heuristics [5] further strengthen the potential of backjumping by using the information made explicit by the reasons. The idea of such family of heuristics is to preferably choose atoms which frequently caused inconsistencies, thus focusing on “critical” atoms. This significantly differ from classical ASP heuristics that use information arising from the simplification part (“look-head”) of the algorithm. Such look-back optimization techniques and heuristics have been shown, on other research areas, to be very effective on “big” benchmarks coming from applications, like planning and formal verification.

In this paper, we report on the analysis, implementation and evaluation of the backjumping technique and look-back heuristics in DLV, and ultimately, about their efficiency in the disjunctive ASP setting. Such methods have been already used in other ASP systems which (i) do not allow for disjunction in the head of the rules [6, 7], or (ii) apply such methods only indirectly after a transformation to a propositional satisfiability problem [8]. The resulting system, called DLV^{LB} , is therefore the first implementation of disjunctive ASP featuring backjumping and look-back heuristics. Importantly, our system provides several options regarding the initialization of the heuristics and the truth value to be assigned to an atom chosen by the heuristics. In our experimental analysis, we provide a comprehensive comparison of the impact of these options, and demonstrate how the new components of DLV^{LB} enhances the efficiency of DLV. Moreover, we also provide a comparison to the other competitive disjunctive ASP systems GnT and Cmodels, which are generally outperformed considerably by DLV^{LB} on the considered benchmarks. Finally, we also present a comparison with respect to QBF solvers, which also allow for solving problems within the second level of the polynomial hierarchy.

2 Answer Set Programming Language

A (disjunctive) rule r is a formula

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are function-free atoms and $n \geq 0, m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r , while $b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m$ is the *body*, of which b_1, \cdots, b_k is the *positive body*, and $\text{not } b_{k+1}, \cdots, \text{not } b_m$ is the *negative body* of r .

An (ASP) program \mathcal{P} is a finite set of rules. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Given a program \mathcal{P} , let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in \mathcal{P} and the *Herbrand Base* $B_{\mathcal{P}}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in \mathcal{P} with the constants of $U_{\mathcal{P}}$.

Given a rule r , $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. Similarly, given a program \mathcal{P} , the *ground instantiation* $Ground(\mathcal{P})$ of \mathcal{P} is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program \mathcal{P} , its answer sets are defined using its ground instantiation $Ground(\mathcal{P})$ in two steps: First answer sets of positive programs are defined, then a reduction of general programs to positive ones is given, which is used to define answer sets of general programs. A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\text{not } \ell$ is not contained in L . An interpretation I for \mathcal{P} is a consistent set of ground literals over atoms in $B_{\mathcal{P}}$.³ A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I ; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I . Interpretation I is *total* if, for each atom A in $B_{\mathcal{P}}$, either A or $\text{not } A$ is in I (i.e., no atom in $B_{\mathcal{P}}$ is undefined w.r.t. I). A total interpretation M is a *model* for \mathcal{P} if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M . X is an *answer set* for a positive program \mathcal{P} if it is minimal w.r.t. set inclusion among the models of \mathcal{P} .

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program \mathcal{P} w.r.t. an interpretation X is the positive ground program \mathcal{P}^X , obtained from \mathcal{P} by (i) deleting all rules $r \in \mathcal{P}$ the negative body of which is false w.r.t. X and (ii) deleting the negative body from the remaining rules. An answer set of a general program \mathcal{P} is a model X of \mathcal{P} such that X is an answer set of $Ground(\mathcal{P})^X$.

3 Answer Set Computation Algorithms

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine of the DLV system, which will be used for the experiments, but also other ASP systems, employ a similar procedure. In general, an answer set program \mathcal{P} contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of \mathcal{P} .⁴ The subsequent computations, which constitute the non-deterministic heart of the system, are then performed on $ground(\mathcal{P})$ by the so called Model Generator procedure.

In the following paragraphs, we illustrate the original model generation algorithm of DLV (which is based on chronological backtracking); then, we briefly describe a backjumping technique that has been implemented in the system[10]; and, we detail how the model generation algorithm has been changed to introduce it. Finally, we report a description of all the heuristics, including the new ones based look-back techniques, that have been implemented in the DLV system, so far.

The Standard Model Generator Algorithm. The computation of answer sets is performed by exploiting the Model Generator Algorithm sketched in Figure 1.⁵

³ We represent interpretations as set of literals, since we have to deal with partial interpretations in the next sections.

⁴ Note that $ground(\mathcal{P})$ is usually not the full $Ground(\mathcal{P})$; rather, it is a subset (often much smaller) of it having precisely the same answer sets as \mathcal{P} [9]

⁵ Note that for reasons of presentation, the description here is quite simplified w.r.t. the “real” implementation. A more detailed description can be found in [11].

```

bool ModelGenerator ( Interpretation& I ) {
    I = DetCons ( I );
    if ( I ==  $\mathcal{L}$  ) then
        return false;
    if ( "no atom is undefined in I" ) then return IsAnswerSet(I);
    Select an undefined atom  $A$  using a heuristic;
    if ( ModelGenerator (  $I \cup \{A\}$  ) ) then return true;
    else return ModelGenerator (  $I \cup \{\text{not } A\}$  );
}

```

Fig. 1. Computation of Answer Sets

This function is initially called with parameter I set to the empty interpretation.⁶

If the program \mathcal{P} has an answer set, then the function returns True, setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the DPLL procedure employed by SAT solvers. It first calls a function DetCons, which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals \mathcal{L} upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is called on $I \cup \{A\}$ and on $I \cup \{\text{not } A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a “good” atom A is crucial for the performance of an ASP system. In the following, we will describe some heuristic criteria for the selection of such branching atoms.

If no atom is left for branching, the Model Generator has produced a “candidate” answer set, the stability of which is subsequently verified by *IsAnswerSet(I)*. This function checks whether the given “candidate” I is a minimal model of the program $Ground(\mathcal{P})^I$ obtained by applying the GL-transformation w.r.t. I , and outputs the model, if so. *IsAnswerSet(I)* returns True if the computation should be stopped and False otherwise.

Note that the algorithm described above computes one answer set for simplicity, however it can be straightforwardly modified to compute all or n answer sets.

Backjumping and Reason for Literals. If during the execution of the ModelGenerator function described in previous paragraph a contradiction arises, or the stable model candidate is not a minimal model, ModelGenerator backtracks and modifies the last choice. This kind of backtracking is called chronological backtracking.

We now describe a technique in which the truth value assignments causing a conflict are identified and backtracking is performed “jumping” directly to a point so that at least one of those assignments is modified. This kind of backtracking technique is called non-chronological backtracking or backjumping. To give the intuition on how backjumping is supposed to work, we exploit the following example.

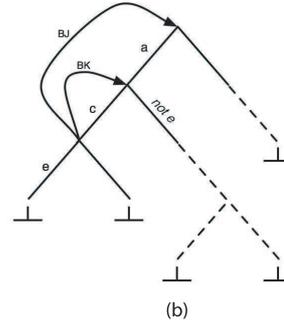
Consider the program of Figure 2(a) and suppose that the search tree is as depicted in Figure 2(b).

⁶ Observe that the interpretations built during the computation are 3-valued, that is, a literal can be True, False or Undefined w.r.t. I .

$$r_1 : a \vee b. \quad r_2 : c \vee d. \quad r_3 : e \vee f.$$

$$r_4 : g :- a, e. \quad r_5 : :- g, a, e.$$

$$r_6 : g :- a, f. \quad r_7 : :- g, a, f.$$



(a)

(b)

Fig. 2. Backtracking vs Backjumping.

According to this tree, we first assume a to be true, deriving b to be false (from r_1 to ensure the minimality of answer sets). Then we assume c to be true, deriving d to be false (from r_2 for minimality). Third, we assume e to be true and derive f to be false (from r_3 for minimality) and g to be true (from r_4 by forward inference). This truth assignment violates constraint r_5 (where g must be false), yielding an inconsistency. We continue the search by inverting the last choice, that is, we assume e to be false and we derive f to be true (again from r_3 to preserve minimality) and g to be true (from r_6 by forward inference), but obtain another inconsistency (because constraint r_7 is violated, here g must also be false).

At this point, ModelGenerator goes back to the previous choice point, in this case inverting the truth value of c (cf. the arc labelled BK in Fig. 2(b)).

Now it is important to note that the inconsistencies obtained are independent of the choice of c , and only the truth value of a and e are the “reasons” for the encountered inconsistencies. In fact, no matter what the truth value of c is, if a is true then any truth assignment for e will lead to an inconsistency. Looking at Fig. 2(b), this means that in the whole subtree below the arc labelled a no stable model can be found. It is therefore obvious that the chronological backtracking search explores branches of the search tree that cannot contain a stable model, performing a lot of useless work.

A better policy would be to go back directly to the point at which we assumed a to be true (see the arc labelled BJ in Fig. 2(b)). In other words, if we know the “reasons” of an inconsistency, we can backjump directly to the closest choice that caused the inconsistent subtree. In practice, once a literal has been assigned a truth value during the computation, we can associate a reason for that fact with the literal. For instance, given a rule $a :- b, c, \text{not } d.$, if b and c are true and d is false in the current partial interpretation, then a will be derived as true (by Forward Propagation). In this case, we can say that a is true “because” b and c are true and d is false. A special case are *chosen* literals, as their only reason is the fact that they have been chosen. The chosen literals can therefore be seen as being their own reason, and we may refer to them as elementary reasons. All other reasons are consequences of elementary reasons, and hence aggregations of elementary reasons. Each literal l derived during the propagation (i.e., DetCons) will have an associated set of positive integers $R(l)$ representing the reason of l , which are essentially the recursion levels of the chosen literals which entail l . Therefore, for any chosen

literal c , $|R(c)| = 1$ holds. For instance, if $R(l) = \{1, 3, 4\}$, then the literals chosen at recursion levels 1, 3 and 4 entail l . If $R(l) = \emptyset$, then l is true in all answer sets.

The process of defining reasons for derived (non-chosen) literals is called *reason calculus*. The reason calculus we employ defines the auxiliary concepts of satisfying literals and orderings among satisfying literals for a given rule. It also has special definitions for literals derived by the well-founded operator. Here, for lack of space, we do not report details of this calculus, and refer to [10] for a detailed definition.

When an inconsistency is determined, we use reason information in order to understand which chosen literals have to be undone in order to avoid the found inconsistency. Implicitly this also means that all choices which are not in the reason do not have any influence on the inconsistency. We can isolate two main types of inconsistencies: (i) Deriving conflicting literals, and (ii) failing stability checks. Of these two, the second one is a peculiarity of disjunctive ASP.

Deriving conflicting literals means, in our setting, that DetCons determines that an atom a and its negation $\text{not } a$ should both hold. In this case, the reason of the inconsistency is – rather straightforward – the combination of the reasons for a and $\text{not } a$: $R(a) \cup R(\text{not } .a)$.

Inconsistencies from failing stability checks are a peculiarity of disjunctive ASP, as non-disjunctive ASP systems usually do not employ a stability check. This situation occurs if the function `IsAnswerSet(I)` of `ModelGenerator` returns false, hence if the checked interpretation (which is guaranteed to be a model) is not stable. The reason for such an inconsistency is always based on an unfounded set, which has been determined inside `IsAnswerSet(I)` as a side-effect. Using this unfounded set, the reason for the inconsistency is composed of the reasons of literals which satisfy rules which contain unfounded atoms in their head (the cancelling assignments of these rules). The information on reasons for inconsistencies can be exploited for backjumping by going back to the closest choice which is a reason for the inconsistency, rather than always to the immediately preceding choice.

In the next paragraph, we will describe a modified version of the `ModelGenerator` algorithm which implements the above-sketched backjumping technique.

The Model Generation Algorithm with Backjumping. In this paragraph we describe `ModelGeneratoBJ` (shown in Fig. 3) a modification of the `ModelGenerator` function, which is able to perform non-chronological backtracking.

It extends `ModelGenerator` by introducing additional parameters and data structures, in order to keep track of reasons and to control backtracking and backjumping. In particular, two new parameters IR and bj_level are introduced, which hold the reason of the inconsistency encountered in the subtrees of which the current recursion is the root, and the recursion level to backtrack or backjump to. When going forward in recursion, bj_level is also used to hold the current level.

The variables $curr_level$, $posIR$, and $negIR$ are local to `ModelGeneratoBJ` and used for holding the current recursion level, and the reasons for the positive and negative recursive branch, respectively.

Initially, the `ModelGeneratorBJ` function is invoked with I set to the empty interpretation, IR set to the empty reason, and bj_level set to -1 (but it will become 0 immediately). Like the `ModelGenerator` function, if the program \mathcal{P} has an answer set, then the function returns true and sets I to the computed answer set; otherwise it returns

```

bool ModelGeneratorBJ (Interpretation& I, Reason& IR,
                        int& bj_level) {
    bj_level ++;
    int curr_level = bj_level;
    I = DetConsBJ ( I, IR );
    if ( I ==  $\mathcal{L}$  ) return false;
    if ( "no atom is undefined in  $\Gamma$ "
        if IsAnswerSetBJ( I, IR ); return true;
        else
            bj_level = MAX ( IR );
            return false;
    Reason posIR, negIR;
    Select an undefined atom  $A$  using a heuristic;
     $R(A) = \{ \text{curr\_level} \}$ ;
    if ( ModelGeneratorBJ( I  $\cup$  {  $A$  }, posIR, bj_level ) return true;
    if ( bj_level < curr_level )
        IR = posIR; return false;
    bj_level = curr_level;
     $R(\text{not } A) = \{ \text{curr\_level} \}$ ;
    if ( ModelGeneratorBJ( I  $\cup$  {  $\text{not } A$  }, negIR, bj_level ) return true;
    if ( bj_level < curr_level )
        IR = negIR; return false;
    IR = trim( curr_level, Union ( posIR, negIR ) );
    bj_level = MAX ( IR );
    return false;
};

```

Fig. 3. Computation of Answer Sets with Backjumping

false. Again, it is straightforward to modify this procedure in order to obtain all or up to n answer sets. Since these modification gives no additional insight, but rather obfuscates the main technique, we refrain from presenting it here.

ModelGeneratorBJ first calls DetConsBJ, an enhanced version of the DetCons procedure. In addition to DetCons, DetConsBJ computes the reasons of the inferred literals, as pointed out in the paragraph for reasons. Moreover, if at some point an inconsistency is detected (i.e. the complement of a true literal is inferred to be true), DetConsBJ (returns the set of all literals \mathcal{L} , and) builds the reason of this inconsistency and stores it in its new, second parameter IR . If an inconsistency is encountered, ModelGeneratorBJ immediately returns false and no backjumping is done. This is an optimization, because it is known that the inconsistency reason will contain the previous recursion level. There is therefore no need to analyze the levels.

If no undefined atom is left, ModelGeneratorBJ invokes IsAnswerSetBJ, an enhanced version of IsAnswerSet. In addition to IsAnswerSet, IsAnswerSetBJ computes the inconsistency reason in case of a stability checking failure, and sets the second parameter IR accordingly. If this happens, it might be possible to backjump, and we set bj_level to the maximal level of the inconsistency reason (or 0 if it is the empty set) before returning from this instance of ModelGeneratorBJ. Indeed, the maximum level in IR corresponds to the nearest (chronologically) choice causing the failure. If the stability check succeeded, we just return true.

Otherwise, an atom A is selected according to a heuristic criterion. We set the reason of A to be the current recursion level and invoke ModelGeneratorBJ recursively, using

$posIR$ and bj_level to be filled in case of an inconsistency. If the recursive call returned true, ModelGeneratorBJ just returns true as well. If it returned false, the corresponding branch is inconsistent, $posIR$ holds the inconsistency reason and bj_level the recursion level to backtrack or backjump to.

Now, if bj_level is less than the current level, this indicates a backjump, and we return from the procedure, setting the inconsistency reason appropriately before. If not, then we have reached the level to go to. We set the reason for not A , and enter the second recursive invocation, this time using $negIR$ and reusing bj_level (which is reinitialized before). As before, if the recursive call returns true, ModelGeneratorBJ immediately returns true also, while if it returned false, we check whether we backjump, setting IR and immediately returning false. If no backjump is done, this instance of ModelGeneratorBJ is the root of an inconsistent subtree, and we set its inconsistency reason IR to the union of $posIR$ and $negIR$, deleting all (irrelevant) integers which are greater or equal than the current recursion level (this is done by the function `trim`). We finally set bj_level to the maximum of the obtained inconsistency reason (or 0 if the set is empty) and return false.

The actual implementation in DLV is slightly more involved, but only due to technical details. Since we do not believe that these technical issues give any particular insight, but are instead rather lengthy in description, we have opted to not include them.

The information provided by reasons can be further exploited in a backjumping-based solver. In particular, in the following paragraph we describe how reasons for inconsistencies can be exploited for defining look-back heuristics.

Heuristics. In this paragraph we will first describe the two main heuristics for DLV (based on look-ahead), and subsequently define several new heuristics based on reasons (or based on look-back), which are computed as side-effects of the backjumping technique. We assume that a ground ASP program \mathcal{P} and an interpretation I have been fixed. We first recall the “standard” DLV heuristic h_{UT} [12], which has recently been refined to yield the heuristic h_{DS} [13], which is more “specialized” for hard disjunctive programs (like 2QBF). These are look-ahead heuristics, that is, the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q , $ext(Q)$ will denote the interpretation resulting from the application of DetCons on $I \cup \{Q\}$; w.l.o.g., we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

Standard Heuristic of DLV (h_{UT}). This heuristic, which is the default in the DLV distribution, has been proposed in [12], where it was shown to be very effective on many relevant problems. It exploits a peculiar property of ASP, namely *supportedness*: For each true atom A of an answer set I , there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r . Since an ASP system must eventually converge to a supported interpretation, h_{DS} is geared towards choosing those literals which minimize the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule. The heuristic h_{UT} is “balanced”, that is, the heuristic values of an atom Q depends on both the effect of taking Q and not Q , the decision between Q and not Q is based on the same criteria involving UT atoms.

Enhanced Heuristic of DLV (h_{DS}). The heuristic h_{DS} , proposed in [14] is based on h_{UT} , and is different from h_{UT} only for pairs of literals which are not ordered by h_{UT} . The idea of the additional criterion is that interpretations having a “higher degree of supportedness” are preferred, where the degree of supportedness is the average number of supporting rules for the true atoms. Intuitively, if all true atoms have many supporting rules in a model M , then the elimination of a true atom from the interpretation would violate many rules, and it becomes less likely finding a subset of M which is a model of \mathcal{P}^M (which would disprove that M is an answer set). Interpretations with a higher degree of supportedness are therefore more likely to be answer sets. Just like h_{UT} , h_{DS} is “balanced”.

The Look-back Heuristics (h_{LB}). We next describe a family of new look-back heuristics h_{LB} . Different to h_{UT} and h_{DS} , which provide a partial order on potential choices, h_{LB} assigns a number ($V(L)$) to each literal L (thereby inducing an implicit order). This number is periodically updated using the inconsistencies that occurred after the most recent update. Whenever a literal is to be selected, the literal with the largest $V(L)$ will be chosen. If several literals have the same $V(L)$, then negative literals are preferred over positive ones, but among negative and positive literals having the same $V(L)$, the ordering will be random. In more detail, for each literal L , two values are stored: $V(L)$, the current heuristic value, and $I(L)$, the number of inconsistencies L has been a reason for since the most recent heuristic value update. After having chosen k literals, $V(L)$ is updated for each L as follows: $V(L) := V(L)/2 + I(L)$. The motivation for the division (which is assumed to be defined on integers by rounding the result) is to give more impact to more recent values. Note that $I(L) \neq 0$ can hold only for literals that have been chosen earlier during the computation.

A crucial point left unspecified by the definition so far are the initial values of $V(L)$. Given that initially no information about inconsistencies is available, it is not obvious how to define this initialization. On the other hand, initializing these values seems to be crucial, as making poor choices in the beginning of the computation can be fatal for efficiency. Here, we present two alternative initializations: The first, denoted by h_{LB}^{MF} , is done by initializing $V(L)$ by the number of occurrences of L in the program rules. The other, denoted by h_{LB}^{LF} , involves ordering the atoms with respect to h_{DS} , and initializing $V(L)$ by the rank in this ordering. The motivation for h_{LB}^{MF} is that it is fast to compute and stays with the “no look-ahead” paradigm of h_{LB} . The motivation for h_{LB}^{LF} is to try to use a lot of information initially, as the first choices are often critical for the size of the subsequent computation tree. We introduce yet another option for h_{LB} , motivated by the fact that answer sets for disjunctive programs must be minimal with respect to atoms interpreted as true, and the fact that the checks for minimality are costly: If we preferably choose false literals, then the computed answer set candidates may have a better chance to be already minimal. Thus even if the literal, which is optimal according to the heuristic, is positive, we will choose the corresponding negative literal first. If we employ this option in the heuristic, we denote it by adding AF to the superscript, arriving at $h_{LB}^{MF,AF}$ and $h_{LB}^{LF,AF}$ respectively.

4 Experiments

We have implemented the above-mentioned look-back techniques and heuristics in DLV; in this section, we report on their experimental evaluation.

Compared Methods. For our experiments, we have compared several versions of DLV [15], which differ on the employed heuristics and the use of backjumping. For having a broader picture, we have also compared our implementations to the competing systems GnT and CModels3, and with the QBF solver Ssolve. The considered systems are:

- **dlv.ut**: the standard DLV system employing h_{UT} (based on look-ahead).
- **dlv.ds**: DLV with h_{DS} , the look-ahead based heuristic specialized for Σ_2^P/Π_2^P hard disjunctive programs.
- **dlv.ds.bj**: DLV with h_{DS} and backjumping.
- **dlv.mf**: DLV with h_{LB}^{MF} .⁷
- **dlv.mf.af**: DLV with $h_{LB}^{MF,AF}$.
- **dlv.lf**: DLV with h_{LB}^{LF} .
- **dlv.lf.af**: DLV with $h_{LB}^{LF,AF}$.
- **gnt** [16]: The solver GnT, based on the Smodels system, can deal with disjunctive ASP. One instance of Smodels generates candidate models, while another instance tests if a candidate model is stable.
- **cm3** [8]: CModels3, a solver based on the definition of completion for disjunctive programs and the extension of loop formulas to the disjunctive case. CModels3 uses two SAT solvers in an interleaved way, the first for finding answer set candidates using the completion of the input program and loop formulas obtained during the computation, the second for verifying if the candidate model is indeed an answer set.
- **ssolve** [17]: is a search based native QBF solver that won the QBF Evaluation in 2004 on random (or probabilistic) benchmarks (performing very well also on non-random, or fixed, benchmarks), and performed globally (i.e., both on fixed and probabilistic benchmarks) well in the last two editions.

Note that we have not taken into account other solvers like Smodels_{cc} [6] or Clasp [7] because our focus is on disjunctive ASP.

Benchmark Programs and Data. The proposed heuristic aims at improving the performance of DLV on disjunctive ASP programs. Therefore we focus on hard programs in this class, which is known to be able to express each problem of the complexity class Σ_2^P/Π_2^P . All of the instances that we have considered in our benchmark analysis have been derived from instances for 2QBF, the canonical problem for the second level of the polynomial hierarchy. This choice is motivated by the fact that many real-world, structured (i.e., fixed) instances in this complexity class are available for 2QBF on QBFLIB [18], and moreover, studies on the location of hard instances for randomly generated 2QBFs have been reported in [19–21].

The problem 2QBF is to decide whether a quantified Boolean formula (QBF) $\Phi = \forall X \exists Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = D_1 \wedge \dots \wedge D_k$ is a CNF formula over $X \cup Y$, is valid.

⁷ Note that all systems with h_{LB} heuristics exploit backjumping.

The transformation from 2QBF to disjunctive logic programming is a slightly altered form of a reduction used in [22]. The propositional disjunctive logic program \mathcal{P}_ϕ produced by the transformation requires $2 * (|X| + |Y|) + 1$ propositional predicates (with one dedicated predicate w), and consists of the following rules. Rules of the form $v \vee \bar{v}$. for each variable $v \in X \cup Y$. Rules of the form $y \leftarrow w$. $\bar{y} \leftarrow w$. for each $y \in Y$. Rules of the form $w \leftarrow \bar{v}_1, \dots, \bar{v}_m, v_{m+1}, \dots, v_n$. for each disjunction $v_1 \vee \dots \vee v_m \vee \neg v_{m+1} \vee \dots \vee \neg v_n$ in ϕ . The rule $\leftarrow \text{not } w$. The 2QBF formula Φ is valid iff \mathcal{P}_Φ has no answer set [22].

We have selected both random and structured QBF instances. The random 2QBF instances have been generated following recent phase transition results for QBFs [19–21]. In particular, the generation method described in [21] has been employed and the generation parameters have been chosen according to the experimental results reported in the same paper. First, we have generated 10 different sets of instances, each of which is labelled with an indication of the employed generation parameters. In particular, the label “*A-E-C-ρ*” indicates the set of instances in which each clause has *A* universally-quantified variables and *E* existentially-quantified variables randomly chosen from a set containing *C* variables, such that the ratio between universal and existential variables is ρ . For example, the instances in the set “3-3-70-0.8” are 6CNF formulas (each clause having exactly 3 universally-quantified variables and 3 existentially-quantified variables) whose variables are randomly chosen from a set of 70 containing 31 universal and 39 existential variables, respectively. In order to compare the performance of the systems in the vicinity of the phase transition, each set of generated formulas has an increasing ratio of clauses over existential variables (from 1 to $\max r$). Following the results presented in [21], $\max r$ has been set to 21 for each of the sets 3-3-70-*, and 12 for each of the 2-3-80-*. We have generated 10 instances for each ratio, thus obtaining, in total, 210 and 120 instances per set, respectively. Then, because such instances do not provide information about the scalability of the systems w.r.t. the total number of variables, we generated other sets. We took the “2-3-80-0.8” and “3-3-70-1.0” sets, we fixed the ratio of clauses over existential variables to the “harder” value for the DLV versions and vary the number of variables *C* (from 5 to $\max C$, step 5), where $\max C$ is 80 and 70, respectively. We have generated 10 instances for each point, thus obtaining, in total, 160 and 140 instances per set, respectively.

About the structured instances, we have analyzed:

- **Narizzano-Robot** - These are real-word instances encoding the robot navigation problems presented in [23], as used in the QBF Evaluation 2004 and 2005.
- **Ayari-MutexP** - These QBFs encode instances to problems related to the formal equivalence checking of partial implementations of circuits, as presented in [24].
- **Letz-Tree** - These instances consist of simple variable-independent subprograms generated according to the pattern: $\forall x_1 x_3 \dots x_{n-1} \exists x_2 x_4 \dots x_n (c_1 \wedge \dots \wedge c_{n-2})$ where $c_i = x_i \vee x_{i+2} \vee x_{i+3}$, $c_{i+1} = \neg x_i \vee \neg x_{i+2} \vee \neg x_{i+3}$, $i = 1, 3, \dots, n - 3$.

The benchmark instances belonging to Letz-tree, Narizzano-Robot, Ayari-MutexP have been obtained from QBFLIB [18], including the 32 (resp. 40) Narizzano-Robot instances used in the QBF Evaluation 2004 (resp. 2005), and all the $\forall\exists$ instances from Letz-tree and Ayari-MutexP.

Results. All the experiments were performed on a 3GHz PentiumIV equipped with 1GB of RAM, 2MB of level 2 cache running Debian GNU/Linux. Time measurements have been done using the `time` command shipped with the system, counting total CPU time for the respective process.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve
2-3-80-0.4	119	120	120	120	120	120	120	3	57	120
2-3-80-0.6	91	102	99	103	83	101	96	4	62	120
2-3-80-0.8	88	99	99	99	79	97	92	5	73	120
2-3-80-1.0	81	95	96	106	80	95	95	10	81	120
2-3-80-1.2	84	99	101	109	85	101	102	6	93	120
3-3-70-0.6	159	174	168	172	157	164	166	4	76	210
3-3-70-0.8	128	138	135	150	123	132	140	2	82	210
3-3-70-1.0	114	128	127	149	112	128	125	7	96	205
3-3-70-1.2	123	131	133	156	115	129	140	9	117	209
3-3-70-1.4	124	139	142	161	117	142	141	9	131	210
#Total	1111	1225	1220	1325	1071	1209	1217	59	868	1644

Table 1. Number of solved instances within timeout for Random 2QBF.

We start with the results of the experiments with random 2QBF formulas. For every instance, we have allowed a maximum running time of 20 minutes. In Table 1 we report, for each system, the number of instances solved in each set within the time limit. Looking at the table, it is clear that the new look-back heuristic combined with the "mf" initialization (corresponding to the system `dlv.mf`) performed very well on these domains, being the version which was able to solve most instances in most settings, particularly on the 3-3-70-* sets. Also `dlv.lf`, in particular when combined with the "af" option, performed quite well, while the other variants do not seem to be very effective. Considering the look-ahead versions of DLV, `dlv.ds` performed reasonably well. Considering GnT and CModels3, we can note that they could solve few instances, while it is clear that Ssolve is very efficient, being able to solving almost all instances.

Figures 4 (resp. 5) show the results for the "2-3-80-0.8" (resp. "3-3-70-1.0") set, regarding scalability. For sake of readability, only the instances with an high number of variables are presented: GnT, Cmodels3, Ssolve and all the DLV versions solve all instances not reported. The left (resp. right) plot of each Figure contains the cumulative number of solved instances about all the DLV versions (resp. GnT, CModels3, Ssolve and the best version of DLV). Overall, on these particular sets, we can see that all the "look-back" versions of DLV scaled much better than GnT and CModels3, with very similar results among them (`dlv.lf.af` just solve one more instance (Fig. 5 left)). Ssolve managed to solve all instances, and in less time (not reported).

In Tables 2, 3 and 4, we report the results, in terms of execution time for finding one answer set, and/or number of instances solved within 20 minutes, about the groups: Narizzano-Robot, Ayari-MutexP and Letz-Tree, respectively. The last columns (AS?) indicate if the instance has an answer set (Y), or not (N), but for Table 2 where it indicates how many instances have answer sets. A "-" in these tables indicates a timeout. For h_{LB} heuristics, we experimented a few different values for "k", and we obtained the

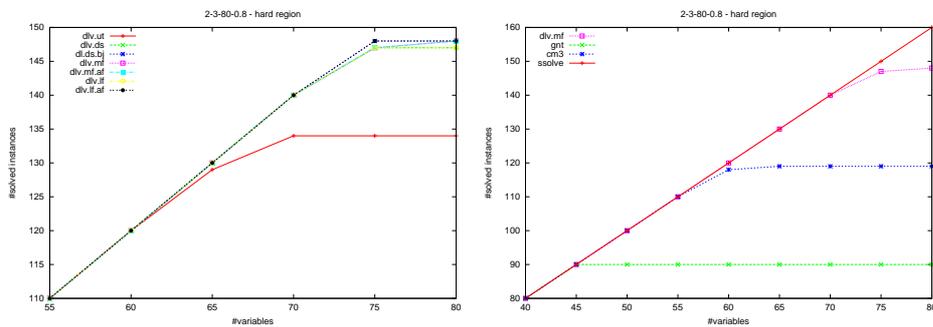


Fig. 4. Left: Number of solved instances by all DLV versions. Right: Number of solved instances by dlv.mf, GnT, CModels3 and Ssolve.

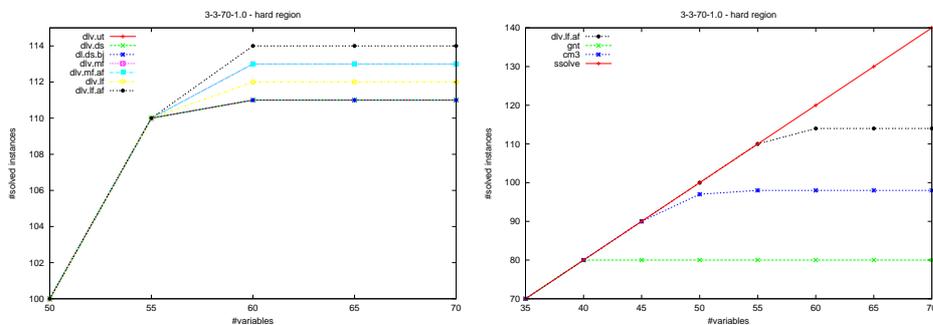


Fig. 5. Left: Number of solved instances by all DLV versions. Right: Number of solved instances by dlv.If.af, GnT, CModels3 and Ssolve.

best results for $k=100$. However, it would be interesting to analyze more thoroughly the effect of the factor k .

In Table 2 we report only the instances from the QBF Evaluation 2004 and 2005, respectively, which were solved within the time limit by at least one of the compared methods. In Table 2, dlv.mf was the only ASP and QBF solver able to solve all the reported 63 (23 for QBF Evaluation 2004 and 40 for QBF Evaluation 2005) instances, followed by Ssolve (60), CModels3 (58) and dlv.If (47). Moreover, dlv.mf was always the fastest ASP system on each instance (sometimes dramatically, even if for lack of space we consider the instances on which it took more than 1 sec, and often faster than Ssolve, especially on the QBF Evaluation 2004 instances. On the QBF Evaluation 2005 instances, dlv.mf, Cmodels3 and Ssolve solved all of them, with a mean execution time of 228.07s, 189.74s and 76.91s, respectively. The “traditional” DLV versions could solve 10 instances, while dlv.ds.bj solved 21 instances, and took less execution time. This indicates the advantages of using a backjumping technique on these robot instances.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve	AS?
QBF Eval. 2004	10	10	11	23	12	15	12	5	18	20	5
QBF Eval. 2005	0	0	10	40	34	32	22	0	40	40	0
#Total	10	10	21	63	46	47	34	5	58	60	5

Table 2. Number of solved instances on Narizzano-Robot instances as selected in the QBF Evaluation 2004 and 2005. The last column indicates how many instances have answer sets.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve	AS?
mutex-2-s	0.01	0.01	0.01	0.01	0.01	0.01	0.01	1.89	0.65	0.03	N
mutex-4-s	0.05	0.05	0.05	0.06	0.05	0.06	0.05	–	–	0.04	N
mutex-8-s	0.21	0.2	0.23	0.21	0.21	0.23	0.21	–	–	0.07	N
mutex-16-s	0.89	0.89	0.98	0.89	0.89	1.01	0.9	–	–	0.13	N
mutex-32-s	3.67	3.72	4.06	3.63	3.64	4.16	3.79	–	–	0.3	N
mutex-64-s	15.38	16.08	17.64	14.97	15.04	18.08	16.97	–	–	0.81	N
mutex-128-s	69.07	79.39	90.92	62.97	62.97	92.92	93.05	–	–	2.83	N
#Solved	7	7	7	7	7	7	7	1	1	7	

Table 3. Execution time (seconds) and number of solved instances on Ayari-MutexP instances.

In Table 3, we then report the results for Ayari-MutexP. In that domain all the versions of DLV and Ssolve were able to solve all 7 instances, outperforming both CModels3 and GnT which solved only one instance. Comparing the execution times required by all the variants of dlv we note that, also in this case, dlv.mf is the best-performing version, while Ssolve scaled up much better. About the Letz-Tree domain, the DLV versions equipped with look-back heuristics solved a higher number of instances and required less CPU time (up to two orders of magnitude less) than all ASP competitors.

In particular, the look-ahead based versions of DLV, GnT and CModels3 could solve only 3 instances, while dlv.mf and dlv.lf solved 4 and 5 instances, respectively. Interestingly, here the "lf" variant is very effective in particular when combined with the "af" option, like in the random instances for testing scalability. It could solve the same number of instances as Ssolve, with Ssolve having better scaling capabilities.

Summarizing, dlv.ds.bj showed (especially on same sets of the random programs, and on the Narizzano-Robot instances) improvements w.r.t. the "traditional" DLV versions. Moreover, if equipped with look-back heuristics, DLV showed very positive performance, further strengthening the potential of look-back techniques. In all of the test cases presented, both random and structured, DLV equipped with look-back heuristics obtained good results both in terms of number of solved instances and execution time compared to traditional DLV, GnT and CModels3. dlv.mf, the "classic" look-back heuristic, performed best in most cases, but good performance was obtained also by dlv.lf. The results of dlv.lf.af on the some random and Letz-Tree instances show that this option can be fruitfully exploited in some particular domains. We also included in the picture the QBF solver Ssolve: while often it showed very good results, on same domains, i.e., the Narizzano-Robot, dlv.mf performed better than Ssolve, both in terms of number of instances solved and CPU execution time. It should be also noted that the

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve	AS?
exa10-10	0.18	0.17	0.17	0.04	0.1	0.06	0.06	0.12	0.03	0.01	N
exa10-15	7.49	7.09	7.31	0.34	0.71	0.48	0.38	6.46	0.73	0.01	N
exa10-20	278.01	264.53	275.1	12.31	17.24	5.43	2.86	325.26	67.56	0.02	N
exa10-25	–	–	–	303.67	432.32	44.13	19.15	–	–	0.02	N
exa10-30	–	–	–	–	–	166.93	129.54	–	–	0.05	N
#Solved	3	3	3	4	4	5	5	3	3	5	

Table 4. Execution time (seconds) and number of solved instances on Letz-Tree instances.

vast majority of the structured instances presented do not have answer sets, while the bigger advantages of dlv.mf over Ssolve on the Narizzano-Robot instances are obtained on the instances with answer sets.

5 Conclusions

We have described a general framework for employing look-back techniques in disjunctive ASP. In particular, we have designed a number of look-back based heuristics, addressing some key issues arising in this framework. We have implemented all proposed techniques in the DLV system, and carried out a broad experimental analysis on hard instances encoding 2QBFs, comprising both randomly generated instances and structured instances. It turned out that the proposed heuristics outperform the traditional (disjunctive) ASP systems DLV, GnT and CModels3 in most cases, and a rather simple approach (“dlv.mf”) works particularly well, being performance-wise competitive with respect to “native” QBF solvers. A possible topic for future research is to further expand the range of look-back techniques in DLV by employing, e.g., *learning*, i.e., the ability to record reasons in order to avoid exploration of useless parts of the search tree which would lead to an inconsistency already encountered.

References

1. 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
2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
3. 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 (1999) 23–37
4. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* **9** (1993) 268–299
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001, ACM (2001) 530–535
6. Ward, J., Schlipf, J.S.: Answer Set Programming with Clause Learning. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). Volume 2923 of LNAL, Springer (2004) 302–313

7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Morgan Kaufmann Publishers (2007) 386–392
8. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR’05, Diamante, Italy, September 2005, Proceedings. Volume 3662 of Lecture Notes in Computer Science., Springer Verlag (2005) 447–451
9. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, ed.: Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDL’99), Prolog Association of Japan (1999) 135–139
10. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. *AI Communications – The European Journal on Artificial Intelligence* **19**(2) (2006) 155–172
11. Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, Institut für Informationssysteme, Technische Universität Wien (2002)
12. Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001, Seattle, WA, USA, Morgan Kaufmann Publishers (2001) 635–640
13. Faber, W., Ricca, F.: Solving Hard ASP Programs Efficiently. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR’05, Diamante, Italy, September 2005, Proceedings. Volume 3662 of Lecture Notes in Computer Science., Springer Verlag (2005) 240–252
14. Faber, W., Leone, N., Ricca, F.: Solving Hard Problems for the Second Level of the Polynomial Hierarchy: Heuristics and Benchmarks. *Intelligenza Artificiale* **2**(3) (2005) 21–28
15. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* **7**(3) (2006) 499–562
16. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). Volume 2923 of LNAI., Springer (2004) 331–335
17. Feldmann, R., Monien, B., Schamberger, S.: A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In: Proceedings National Conference on AI (AAAI’00), Austin, Texas, AAAI Press (2000) 285–290
18. Narizzano, M., Tacchella, A.: QBF Solvers Evaluation page (2002) <http://www.qbflib.org/qbfeval/index.html/>.
19. Cadoli, M., Giovanardi, A., Schaerf, M.: Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In: Proceedings of the 5th Congress: Advances in Artificial Intelligence of the Italian Association for Artificial Intelligence, AI*IA 97. Lecture Notes in Computer Science, Rome, Italy, Springer Verlag (1997) 207–218
20. Gent, I., Walsh, T.: The QSAT Phase Transition. In: Proceedings of the 16th AAAI. (1999)
21. Chen, H., Interian, Y.: A model for generating random quantified boolean formulas. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05), Professional Book Center (2005) 66–71
22. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence* **15**(3/4) (1995) 289–323
23. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence* **147**(1/2) (2003) 85–117
24. Ayari, A., Basin, D.A.: Bounded Model Construction for Monadic Second-Order Logics. In: Proceedings of Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Chicago, IL, USA (2000)

Towards Proof Theoretic Model Generation

Camillo Fiorentini, Alberto Momigliano, Mario Ornaghi

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39, 20135 Milano–Italy
{fiorenti,momiglia,ornaghi}@dsi.unimi.it

Abstract In recent years model generation tools have been developed in different areas, such as DLV or Smodels in the context of Answer Set Programming, or USE in the context of the UML modeling language. In the latter case those models are known as *snapshots* and represent possible system states. We are developing COOML (Constructive Object Oriented Modeling Language), an OO modeling language in the spirit of the UML, but based on a constructive semantics, in particular the BHK explanation of the logical connectives. We introduce a general notion of snapshot based on populations of objects and structured pieces of information, based on which snapshot generation algorithms can be designed. COOML's underlying semantics allows the user to tune the appropriate amount of information to the problem domain. In this paper we investigate the language suitability to represent general combinatorial problems and to solve them by snapshot generation.

1 Introduction

The Constructive Object Oriented Modeling Language COOML has been developed as a specification language for OO systems. It is similar to UML, although it uses a constructive semantics that allows the user to specify in a selective way the pieces of information he/she wants. This is obtained through a semantics related to the BHK explanation of logical connectives. In our formalisation, a piece of information is a pair $i : F$, where F is a formula and i an explanation of it. According to the BHK, an explanation of $A \wedge B$ contains an explanation of A and one of B , an explanation of $A \vee B$ contains an explanation of A or one of B , and so on. However, while in the BHK an explanation of F should give evidence for the validity of F , in our case $i : F$ represents a piece of information which may be *true* or *false* in a world w . Thus, we have notion of *model* of a piece of information based on classical logic. In particular, we use $true : F$ to indicate the truth of F ; in fact, $true$ does not contain evidence for F , but represents a piece of information true in all the models of F . This allows us to choose different levels of detail. For example, the piece of information $[1, i_A] : A \vee B$ indicates that A is true according to $i_A : A$, while $true : A \vee B$ simply means that $A \vee B$ is true, but nothing is said about A or B . For more details on the logical aspects, we refer to [12].

A system state, or *snapshot*, of an OO system is characterized by the populations of the currently live objects and their information content. COOML uses pieces of information to model snapshots. In fact, the former provide a general notion of snapshot and snapshot generation algorithms can be designed. Snapshot generation is similar to:

- the analogous notion in the UML, as implemented e.g. in USE [7];
- model generation in SAT [11] and in Answer Set Programming such as Smodels [13] and DLV [10].

Snapshot generation is interesting for at least two different reasons:

- a) It is important for understanding and validating OO specification. In the UML, tools based on the OCL have been developed [8]. In [14] we have given a first sketch of COOML model generation in this direction.
- b) Models have been proposed for the representation of combinatorial problems through propositional theories; thus model generation yields their solution [1]. Interesting experiences have been developed, for example, in the area of planning [2,16,5].

The latter point suggests to exploit COOML as a language for representing combinatorial problems and solving them by snapshot generation. Note that UML snapshots, as well as models in answer set programming are defined on a model theoretical ground. COOML snapshots are still based on the classical notion of model, but they have also a proof theoretic flavor, which enables selective information. The possibility of treating information in a more selective way, with respect to a purely model-theoretic approaches, has effects on the possibility of *selecting only the relevant information*, and efficiency of the *grounding procedure*.

In this paper, we present a first prototype for COOML snapshot generation. For our discussion we have chosen the example of planning, since it is significant and is one of the examples where model generation has been intensively investigated (see [6,3,2] to name just a few). In the conclusions, we briefly discuss the possibility of integrating our approach with model-theoretic generation methods such as the one exploited by DLV.

2 COOML Specifications

The link between the data stored in a software system and their *meaning* in the “real world” is the result of the abstractions performed in the *analysis phase*. Typically (see e.g., [9]), the analysis has to produce a *dictionary* containing the abstract concepts used in specifications and choose the *data types* (possibly depending on the implementation language). The analysis phase should result in a language to talk about the *world* and its *states*. We call *problem formulas* (PF) the sentences of this language. In this paper worlds are formalized as classical interpretations, but other problem logics can be used, and even informal interpretations are allowed. We only require that problem formulas can be understood by the final user as properties that may or may not *hold in a world* w ; we write $w \models S$ to denote that the PF formula S holds in w .

Thus, COOML specifications distinguish two different layers: the *COOML layer*, concerning the properties of the information stored in the system, and the *problem domain (PD) layer*, concerning the problem domain at hand. The logic of the COOML layer is a fragment of the constructive logic E^* [12], while the choice of the problem domain signature and logic is left open. In this section we explain the COOML layer, leaving the problem domain layer as understood, provided that it satisfies the above assumptions.

In the OO COOML syntax, a class specification has the following general form:

Class C { (*)
 ENV { $\underline{T} \underline{e} : Constr_C(\text{this}, \underline{e})$ }
 $\text{ptyName} : Pty_C(\text{this}, \underline{e})$
Methods }

where C is the name of the class and $Pty_C(\text{this}, \underline{e})$ is its *class property*, specifying the structure and the meaning in the problem domain of the data stored by its objects. The class property depends on the self-reference variable `this` and on the possible environment variables \underline{e} . The latter are a distinguished feature of COOML: they refer to values that are defined outside of C , but are used in its specification. We call $Constr_C(\text{this}, \underline{e})$ the *environment constraint* of C . Environment variables, as well as properties and constraints, are illustrated in Example 1.

Sub-classing is defined in the usual way, together with the notions of inheritance and overriding. Moreover, there are *abstract* classes, i.e., such that cannot be used to create objects. Non abstract classes are called *concrete* classes. A COOML specification consists in a set of class specifications.

The snapshot generation algorithm SG considered in this paper requires a compilation from a COOML specification into a suitable normal form. For lack of space we do not introduce the OO COOML syntax, but we express COOML specifications and their semantics directly in the SG-representation (which turns out here to be Prolog). This section is organized as follows: firstly we give the basic COOML assumptions on classes and objects and we introduce constraints; then we explain COOML specifications and their semantics; finally, we define COOML populations of objects, representing system states.

Classes and objects In a COOML system state there is a finite set of *live objects*. Each live object is uniquely identified by a ground term o that we call the *object identifier*. For object identifiers, we assume Clark's equality theory [4]. By the SG-predicate $class(c, o, \underline{e})$ we denote that o is a live object with implementation class c (i.e. the class used to create it) and environment \underline{e} . To take into account sub-classing, we also have the SG-predicate $concreteSubclass(c', \underline{e}', c, \underline{e})$, meaning that c' with environment \underline{e}' is a concrete subclass of the (concrete or abstract) class c with environment \underline{e} , and the SG-predicate $objType(c, o, \underline{e})$, *explicitly defined* as follows:

$$\forall c, o, \underline{e}. objType(c, o, \underline{e}) \leftrightarrow \exists c', \underline{e}'. concreteSubclass(c', \underline{e}', c, \underline{e}) \wedge class(c', o, \underline{e}') \quad (1)$$

Following the OO view, classes and objects are abstractions of the modelled world. Thus, we treat both the above SG-predicates as part of the problem domain signature and classes as first class citizens, i.e., individuals of the domain of a world w . In this context, the class hierarchy is represented by *subclass axioms* of the form

$$\forall \underline{e}. concreteSubclass(c, \underline{e}, c', \underline{t}(\underline{e})) \quad (2)$$

where $\underline{t}(\underline{e})$ are terms mapping the environment of c into the one of c' . We consider only concrete subclasses, because abstract classes are not translated into a SG-representation. Abstract classes can be mentioned by the SG-predicate $objType$, interpreted according to its definition (1). Inherited properties and methods are inserted in the concrete

(sub)classes by the compilation phase, which also takes into account overriding. In this way, the SG algorithm directly finds the property and methods of an object in its creation class.

Constraints (CON) Constraints are the SG-representation of problem formulas. They are built on top of the atomic problem formulas, using the conjunction ‘&’, interpreted as usual: $w \models C_1 \& C_2$ iff $w \models C_1$ and $w \models C_2$. Open constraints are understood to be universally quantified. Other logical connectives are possible, depending on the problem domain logic.

Class specifications and COOML formulas (CF) A specification of a class C is associated with the environment constraint and the property of C , as defined in (*):

$$\begin{aligned} & envConstr(class(c, this, \underline{e}), Constr_c(this, \underline{e})). \\ & pty(class(c, this, \underline{e}), Pty_c(this, \underline{e})). \end{aligned}$$

The environment constraint is the SG-representation of the formula $\forall x, \underline{e}. class(c, x, \underline{e}) \rightarrow Constr_c(x, \underline{e})$. The property, $Pty_c(this, \underline{e})$ is a COOML formula CF. The syntax of CF’s is as follows:

$$CF ::= CON \quad | \quad CF \text{ and } CF \quad | \quad CF \text{ or } CF \quad | \quad exi(\underline{x}, CF) \quad | \quad for(\underline{u}: G(\underline{u}), CF)$$

where *and*, *or* are binary operators, \underline{x} , \underline{u} are lists of variables, and $G(\underline{u})$ is a *generator* for the universal variables \underline{u} . More precisely, a generator for \underline{u} is a particular constraint $G(\underline{u}, \underline{y})$ such that, for every ground \underline{t} and every world w , the domain

$$Dom(G(\underline{u}, \underline{t}), w) = \{ \underline{d} \mid w \models G(\underline{d}, \underline{t}) \}$$

is *finite*. We distinguish between *data* and *object generators*. A data generator $G(\underline{u}, \underline{y})$ for \underline{u} must have an associated procedure to generate $Dom(G(\underline{u}, \underline{t}), w)$ and the domain does not depend on w . In Example 1 we use the data generator $in(u, t_1, t_2)$, where u is an integer variable and t_1, t_2 are integer terms that do not contain u : for every instance of t_1, t_2 , $in(u, t_1, t_2)$ generates the sequence of the integers u such that $t_1 \leq u \leq t_2$. An object generator has form $class(c, u, \underline{t})$ and depends on the current world w . Since in every world w there are only finitely many objects of class c , we can enumerate all the o such that $class(c, o, \underline{t})$ holds in w .

A CF defines a set of possible *information values*, represented in SG by Prolog lists. By $IT(F)$ we indicate the set of the possible information values associated with a property F . It is defined as follows:

$$\begin{aligned} IT(CON) &= \{ true \} \\ IT(F_1 \text{ and } F_2) &= \{ [i_1, i_2] \mid i_1 \in IT(F_1) \text{ and } i_2 \in IT(F_2) \} \\ IT(F_1 \text{ or } F_2) &= \{ [k, i] \mid 1 \leq k \leq 2 \text{ and } i \in IT(F_k) \} \\ IT(exi(\underline{x}, F)) &= \{ [exi(\underline{t}), i] \mid \underline{t} \text{ are ground terms and } i \in IT(F) \} \\ IT(for(\underline{u}: G(\underline{u}), F)) &= \{ [[for(\underline{t}_1), i_1], \dots, [for(\underline{t}_n), i_n]] \mid \\ & \quad n \geq 0 \text{ and, for all } j \in 1..n, \underline{t}_j \text{ are ground terms and } i_j \in IT(F) \} \end{aligned}$$

A *piece of information* is a pair $i : F$, with $i \in \text{IT}(F)$. For every ground F , the *meaning* of $i : F$ in a world w is formalized by the relation $w \models i : F$:

$w \models \text{true} : \text{CON}$	$\text{iff } w \models \text{CON}$
$w \models [i_1, i_2] : F_1 \text{ and } F_2$	$\text{iff } w \models i_1 : F_1 \text{ and } w \models i_2 : F_2$
$w \models [k, i] : F_1 \text{ or } F_2$	$\text{iff } w \models i : F_k$
$w \models [\text{exi}(\underline{t}), i] : \text{exi}(\underline{x}, F(\underline{x}))$	$\text{iff } w \models i : F(\underline{t})$
$w \models [[\text{for}(\underline{t}_1), i_1], \dots, [\text{for}(\underline{t}_n), i_n]] : \text{for}(\underline{u} : G(\underline{u}), F(\underline{u}))$	$\text{iff } \text{Dom}(G(\underline{u}), w) = \{\underline{t}_1, \dots, \underline{t}_n\} \text{ and } w \models i_j : F(\underline{t}_j) \text{ for all } j \in 1..n$

Example 1. Let us specify words of parametric length N . We need a class word with environment \mathbf{N} which collects the N bits of the word. Since \mathbf{N} is an environment variable, it is fixed externally and cannot be updated by the methods of the class word. Each bit is an object of class `bitInAWord`, a subclass of the abstract class `bit`. Since a bit b occurs in a position Pos of a word \mathbf{W} , \mathbf{W} and Pos are in the environment of `bitInAWord` (whereas `bit` has empty environment). We also use the problem formulas `len(Word, N)` (the length of `Word` is N), `contains(Word, Bit, Pos)` (`Bit` is the bit of `Word` in position Pos), `low(Bit)` (bit `Bit` has low value 0) and `high(Bit)` (`Bit` has high value 1). Using the Prolog SG-representation, the specification is:

```
concreteSubclass(bitInAWord, [W,Pos], bit, []).
envConstr(class(bitInAWord, This, [W,Pos]),
          class(word, W, [N]) & contains(W, This, Pos)).
envConstr(class(word, This, [N]), len(This, N)).
pty(class(bit, This, []), low(This) or high(This)).
pty(class(bitInAWord, This, [W,Pos]), low(This) or high(This)).
pty(class(word, This, [N]),
     for(Pos:in(Pos, 0, N-1),
        exi(Bit, class(bitInAWord, Bit, [This, Pos])))).
```

The property `low(This) or high(This)` of the class `bit` admits as information value `[1, true]`, corresponding to a low bit, or `[2, true]`, corresponding to a high bit. Such a property is explicitly shown also in the subclass `bitInAWord`, as this is the result of the compilation phase. In the property of class `word`, the `for` construct generates the bits of the word, where each bit has a position Pos in $0..N - 1$. For conciseness, we have omitted further constraints, e.g., the fact that a bit cannot occur in two distinct positions.

COOML systems and their populations A COOML system specification contains a set of *subclass axioms*, representing the class hierarchy, and a set of *class specifications*, where each class is specified by its environment constraint and its class property definition. Subclass axioms, environment constraints and their semantics have been explained in the previous paragraphs. A class property $\text{pty}(\text{class}(c, \text{this}, \underline{e}), \text{Pty}_c(\text{this}, \underline{e}))$ is interpreted as the CF:

$$\text{for}(x, \underline{e} : \text{class}(c, x, \underline{e}), \text{Pty}_c(x, \underline{e})) \quad (3)$$

Thus, in a piece of information such as $i : (3)$, i has the form

$$[[\text{for}(o_1, \underline{e}_1), i_1], \dots, [\text{for}(o_n, \underline{e}_n), i_n]]$$

where o_1, \dots, o_n are the live objects with class c and each $i_j \in \text{IT}(\text{Pty}_c(x, \underline{e}))$. In COOML we represent the above piece of information as the set of the live objects of class c , together with their information content:

$$\{i_1 : \text{class}(c, o_1, \underline{e}_1), \dots, i_n : \text{class}(c, o_n, \underline{e}_n)\}. \quad (4)$$

We write $w \models i_j : \text{class}(c, o_j, \underline{e}_j)$ to indicate that $w \models i_j : \text{Pty}_c(o_j, \underline{e}_j)$. Clearly, (3) is true in a world w iff $w \models i_j : \text{class}(c, o_j, \underline{e}_j)$ for every $1 \leq j \leq n$. A set such as (4) is called a *population of class c* . The *population of a system S* is obtained as the union of the population of its classes. The above discussion justifies the following definition:

Definition 1. *Let Spec be a COOML system specification in a problem domain PD , \mathcal{P} be a population of Spec , and w be a world for PD . Then we say that \mathcal{P} holds in w , and we write $w \models \mathcal{P}$, iff:*

1. *for every subclass axiom SubC of Spec , $w \models \text{true} : \text{SubC}$,*
2. *for every environment constraint CON of Spec , $w \models \text{true} : \text{CON}$, and*
3. *for every $i : \text{class}(c, o, \underline{e}) \in \mathcal{P}$, $w \models i : \text{class}(c, o, \underline{e})$.*

If $w \models \mathcal{P}$, we also say that w is a *model* of \mathcal{P} .

Example 2. Let us consider the following population (or snapshot) \mathcal{P}_{w_2} of the specification in Example 1:

```
[2,true]: class(bitInAWord,b0,[w2,0]),
[1,true]: class(bitInAWord,b1,[w2,1]),
[[for(0),[exi(b0),true]], [for(1),[exi(b1),true]]]: class(word,w2,[2])
```

Let w be a model of \mathcal{P}_{w_2} . In w , w_2 is an object of class `word` that represents the word 10 of length 2 (the value of the environment variable `N`). Indeed, the information value associated with `class(word,w2,[2])` establishes that the bits of w_2 are `b0` (in position 0) and `b1` (in position 1). Moreover, `b0` is high (the information value of the corresponding class property is `[2,true]`), while `b1` is low. We remark that `b0` and `b1` are objects of class `bitInAWord` and, by the subclass axiom, `objType(bit,b0,[])` and `objType(bit,b1,[])` hold in w .

3 Snapshot Generation

Once a COOML specification Spec has been translated into its SG-representation, the user has to specify a set of *generation requirements* and a *generation goal* (g-goal). The generation requirements are first-order formulae in the constraint language of SG and must be satisfied by the generated population:

Definition 2. *Let Spec be a COOML specification, \mathcal{P} be a population of Spec , and GR be a set of generation requirements. Then \mathcal{P} satisfies GR iff, for every model w of \mathcal{P} , $w \models GR$.*

A g-goal is represented by a list $[T_1 : \text{class}(c_1, o_1, \underline{e}_1), \dots, T_n : \text{class}(c_n, o_n, \underline{e}_n)]$, where every $\text{class}(c_j, o_j, \underline{e}_j)$ is ground. In general, the terms T_j in a g-goal are open. Ground g-goals coincide with populations. To define the solutions of a g-goal in a declarative way, we introduce the following partial ordering on g-goals GG_k , where hereafter we use set-theoretic operations for the sets underlying lists of populations:

$$GG_1 \preceq GG_2 \text{ iff there is a substitution } \sigma \text{ s.t. } GG_1 \sigma \subseteq GG_2 \quad (5)$$

Definition 3. Let *Spec* be a COOML specification, *GR* be a set of generation requirements, and *GG* a g-goal. A solution of *GG* is a population \mathcal{P} of *Spec* such that \mathcal{P} satisfies *GR* and $GG \preceq \mathcal{P}$.

We are interested in the *minimal solutions* with respect to \preceq . A minimal solution is representative of all the larger solutions in the following sense: if $GG_1 \preceq GG_2$, then every model w of GG_2 is also a model of GG_1 . If the generation requirements are sufficiently strong, we may obtain finitely many minimal solutions.

Example 3. Let us consider the specification of Example 1 and the following generation requirements:

$$\begin{aligned} \text{class}(\text{word}, W, [N]) &\rightarrow W = w(N) \wedge \text{member}(N, [2, 3]) \\ \text{class}(\text{bitInAWord}, B, [W, \text{Pos}]) &\rightarrow \exists N (\text{class}(\text{word}, W, [N]) \wedge B = b(W, \text{Pos}) \wedge 0 \leq \text{Pos} \leq N - 1) \end{aligned}$$

This implies that the possible objects of class `word` are $w(2)$ (of length 2) and $w(3)$ (of length 3) and the objects of class `bitInAWord` have the form $b(W, \text{Pos})$, where W is $w(2)$ or $w(3)$ and $0 \leq \text{Pos} \leq \text{length}(W) - 1$. The g-goal $[\text{InfoW}: \text{class}(\text{word}, w(2), [2])]$ has 4 minimal solutions, corresponding to the words 00, 01, 10, 11. For instance, the population of the second solution is:

$$\begin{aligned} [1, \text{true}]: & \text{class}(\text{bitInAWord}, b(w(2), 0), [w(2), 0]) \\ [2, \text{true}]: & \text{class}(\text{bitInAWord}, b(w(2), 1), [w(2), 1]) \\ [[\text{for}(0), [\text{exi}(b(w(2), 0)), \text{true}]], [\text{for}(1), \dots]]: & \text{class}(\text{word}, w2, [2]) \end{aligned}$$

On the other hand, the g-goal

$$[[1, \text{true}]: \text{class}(\text{bitInAWord}, B, [w(2), 1]), \text{InfoW}: \text{class}(\text{word}, w(2), [2])]$$

has two minimal solutions corresponding to 00 and 10. As a matter of fact, any solution has to satisfy $[1, \text{true}]: \text{class}(\text{bitInAWord}, B, [w(2), 1])$, which forces the bit $b(w(2), 1)$ (the second bit of $w2$) to 0.

The purpose of the SG algorithm is to generate all the minimal solutions of a g-goal *GG*. The nodes of the search space are generation-states (g-states) of the form $S = \langle \text{Pops}, \text{ToDos}, \text{ClosedS} \rangle$. The list *Pops* contains the currently generated population, *ToDos* is the g-goal to be solved, while *ClosedS* is a list of atoms of the form $\text{closed}(c, \underline{e})$. An atom $\text{closed}(c, \underline{e})$ is inserted in *ClosedS* as soon as SG generates a complete piece of information $\text{Info}: \text{for}(x: \text{class}(c, x, \underline{e}), F)$ for a universally bounded CF. In this case, the domain of $\text{class}(c, x, \underline{e})$ cannot be modified in subsequent search steps, because it is fixed by *Info*. We denote the domain of $\text{class}(c, x, \underline{e})$ in a g-state *S* by

$$\text{Doms}(c, \underline{e}) = \{o \mid \exists T (T : \text{class}(c, o, \underline{e}) \in \text{Pops} \cup \text{ToDos})\}$$

The meaning of the predicate $closed(c, \underline{e})$ in S is defined by the axiom

$$ClosureAx(S, c, \underline{e}) = \forall x (class(c, x, \underline{e}) \leftrightarrow x = o_1 \vee \dots \vee x = o_m)$$

where o_1, \dots, o_m are all the elements of $Dom_S(c, \underline{e})$. By $ClosureAx(S)$ we denote the conjunction of all the axioms $ClosureAx(S, c, \underline{e})$ such that $closed(c, \underline{e}) \in Closed_S$.

According to the above discussion, we extend the relation \preceq and the definition of solution to g-states, as follows. Let $Spec$ be a COOML specification. Then we say:

- $\langle Pop_{S_1}, ToDos_1, Closed_{S_1} \rangle \preceq \langle Pop_{S_2}, ToDos_2, Closed_{S_2} \rangle$ iff:
 1. $(Pop_{S_1} \cup ToDos_1) \preceq (Pop_{S_2} \cup ToDos_2)$,
 2. $Pop_{S_1} \subseteq Pop_{S_2}$ and $Closed_{S_1} \subseteq Closed_{S_2}$ and
 3. for every $closed(c, \underline{e}) \in Closed_{S_1}$, $Dom_{S_1}(c, \underline{e}) = Dom_{S_2}(c, \underline{e})$.
- $\langle Pop_S, ToDos_S, Closed_S \rangle$ is in solved form iff $ToDos_S$ is the empty list $[\]$.
- $\langle Pop_S, [\], Closed_S \rangle$ is a solution of $Spec$ with generation requirements GR iff Pop_S is a population of $Spec$ and Pop_S satisfies $GR \cup \{ClosureAx(S)\}$.

We now outline a non-deterministic generation algorithm NDGA.

NON-DETERMINISTIC GENERATION ALGORITHM

Input: A specification $Spec$, a g-goal GG , a set GR of generation requirements

var S : g-states;

$S := \langle [\], GG, [\] \rangle$;

Repeat

1. Choose (non deterministically) $T : class(c, o, \underline{e}) \in ToDos_S$;
2. STEP($T : class(c, o, \underline{e})$, S , S);

until $S = \langle Pop_S, ToDos_S, Closed_S \rangle$ is in solved form.

Output: Pop_S

STEP($+T : class(c, o, \underline{e})$, $+S$, $-S'$)

build a S' such that:

1. $S \preceq S'$;
2. $Pop_{S'} = Pop_S \cup \{T\sigma : class(c, o, \underline{e})\}$, with σ a ground substitution;
3. $Pop_{S'}$ is a population of $Spec$ satisfying $GR \cup \{ClosureAx(S')\}$;
4. For every minimal g-state S^* such that $S \preceq S^*$, there is S' such that $S \preceq S' \preceq S^*$.

The STEP procedure has to generate a g-state S' starting from S and a piece of information $T : class(c, o, \underline{e})$ (the class to be solved). We have formalized it as a non-deterministic procedure and we have only fixed the minimal requirements needed to get the following:

Theorem 1. *Let $Spec$ be a COOML specification, GR be a set of generation requirements, and GG be a g-goal. We have:*

- (Correctness). *If the SG algorithm halts with a solution Pop_S , then Pop_S is a minimal solution of GG .*
- (Completeness). *If \mathcal{P} is a finite minimal solution of GG , then there is a computation which outputs \mathcal{P} and requires n calls to the procedure STEP, where n are the elements of \mathcal{P} .*

Such requirements are a guide to design correct and complete deterministic implementations, which may employ different *generation strategies*, i.e., ways of implementing the generation step and of choosing $T : class(c, o, \underline{e}) \in ToDo_S$.

3.1 A Prototype SG Algorithm and its Generation Requirements

We have implemented a prototype of the NDGA algorithm. It is built on top of three main procedures:

1. `addConstrList` checks the environment constraints and adds, when necessary, new goals to the `ToDo` list;
2. `info` generates an information value $T\sigma \in IT(Pty_c(o, \underline{e}))$, where $T : class(c, o, \underline{e})$ is the selected goal;
3. `generateDomain` is used by `info` when a CF $for(\underline{u} : G(\underline{u}), F)$ is encountered, to generate the domain of $G(\underline{u})$ and update the `Closed` list.

All the procedures use the generation requirements to check the consistency of the generation hypotheses and to instantiate existential goals. So far, we have not considered efficiency aspects. The effort has been on correctness and completeness and on a kernel language to express computable generation requirements. We do not explain here the details of our implementation, as not relevant for the purposes of this paper. Rather, we prefer to explain the supported generation requirement language. It is a first rough kernel, which can be improved and expanded. However, it is already sufficient to express non trivial problems, as shown in the next section. We have three types of requirements: information type requirements, errors, and instantiation requirements.

An *information-requirement* has the form

$$\langle name \rangle (This : class(c, o, \underline{e}), T_1 : class(c_1, o_1, \underline{e}_1), \dots, T_n : class(c_n, o_n, \underline{e}_n))$$

It can occur in a constraint $Constr_c$ of an $envConstr(class(c, This, \underline{e}), Constr_c)$. Its meaning is that the next g-state S' has to satisfy the requirement

$$Pop_S \cup ToDo_S \cup \{T_1 : class(c_1, o_1, \underline{e}_1), \dots, T_n : class(c_n, o_n, \underline{e}_n)\} \preceq S'$$

That is, whenever an object *This* with class c and environment \underline{e} is created, then the population is expected to contain instances of $T_1 : class(c_1, o_1, \underline{e}_1), \dots, T_n : class(c_n, o_n, \underline{e}_n)$.

We have two kinds of *errors*: `localErr(A, S)` and `globalErr(S)`. The former is checked whenever the SG algorithm in current g-state S encounters an existential goal $\exists(A)$ and looks for an instance $A\sigma$. If `localErr(A, S)` succeeds, A is treated as inconsistent and refused. Otherwise, it is accepted as consistent. Instead, `globalErr(S)` is checked only when a g-state in solved form is reached: S is accepted as a solution only if `globalErr(S)` fails. The user can define `localErr(A, S)` and `globalErr(S)` using the predicate `holds(K, S)` implemented (and used) by SG, where K is a constraint of the PD and S the current g-state. The semantics of `hold` is defined as follows:

- For ground atomic constraints on live objects:
 - $class(c, o, \underline{e})$ holds (in S) iff there is a T s.t. $T : class(c, o, \underline{e}) \in Pop_S \cup ToDo_S$;
 - $objType(c, o, \underline{e})$ holds iff $class(c', o, \underline{t})$ holds, for some concrete subclass c' of c ;
 - $closed(c, \underline{e})$ holds iff it belongs to $Closed_S$.

- For the other ground atomic constraints K :
 K holds in S iff it is true in every model of the *already generated* population $Pops$.

Concerning non atomic constraints, so far we have implemented the conjunction ‘&’. The existential quantification is implicit in the fact that $holds(K, S)$ can be invoked on an open K and yields a computed answer substitution. We have the following monotonicity and non-monotonicity result: $holds(K, S)$ and $S \preceq S'$ entails $holds(K, S')$, while $not(holds(K, S))$ obtained by finite failure does not entail $not(holds(K, S'))$. Thus, when programming `localErr`, the user should not use negation on `holds` atoms. However, there are cases where one can define a monotonic $holds(not(K), S)$. In particular:

```
holds(not(class(C,O,E),S) :-
    holds(closed(C,E),S), not(holds(class(C,O,E),S)).
```

Of course, the excluded-middle law $holds(K, S)$ or $holds(not(K), S)$ may not be appropriate in general. The use of `holds` in the definition of `localErr` will be exemplified in the next section.

Finally, we have *instantiate constraints*, which are used for grounding purposes. The SG algorithm implements the following constraints:

$$\begin{aligned} class(c, o, \underline{e}) &\rightarrow id(c, o, \underline{e}) \\ objType(c, o, \underline{e}) &\rightarrow \exists c', \underline{t} (concreteSubclass(c', \underline{t}, c, \underline{e}) \wedge id(c, o, \underline{e})) \\ K &\rightarrow cInstance(K, S) \quad \text{for the other atomic constraints } K \end{aligned}$$

The user has to define the predicates $id(c, o, \underline{e})$ to generate the ground terms to be used as *live objects* when `class` and `objType` predicates are instantiated, and $cInstance(K, S)$ to generate the instances of the other atoms. To exemplify, the generation requirements in Example 3 can be expressed as follows:

```
id(word,W,N) :- W=w(N), member(N, [2,3]).
id(bitInAWord,B,[W,Pos]) :- id(word,W,N), B=b(W,Pos), 0<=Pos, Pos<N.
```

The use of instantiation constraints is also illustrated in the next section.

4 Representing Combinatorial Problems: the Planning Example

In this section we illustrate the use of COOML specifications to represent combinatorial problems and of the generation algorithm to solve them. We represent a combinatorial problem P at three levels:

1. We specify a set of COOML classes such that snapshots represent the possible solutions of P .
2. We specify the general properties of P , in the form of generation requirements that must be satisfied independently from the specific problem instance to be solved.
3. A specific instance I of P to be solved is expressed by means of specific generation requirements; the snapshots generated represent the solutions of I .

To illustrate the above levels in the specification of a combinatorial problem, we have chosen planning. At level 1 we use abstract classes to represent general features that have to be specialized by concrete classes in the lower levels. Abstract classes are not translated and do not occur in the SG representation. However, we need to show them to explain this level. We denote (possible) super-types, constraints and properties of an abstract class c as follows:

$$\begin{aligned} & subType(c, \underline{e}, c', \underline{t}) \\ & envConstr(objType(c, this, \underline{e}), Constr_c(this, \underline{e}) [\&SubClassConstr]) \\ & pty(objType(c, this, \underline{e}), Pty_c(this, \underline{e}) [and SubClassPty]) \end{aligned}$$

SubClassConstr and *SubClassPty* stand for (possible) generic sub-constraints and sub-properties, to be instantiated by the concrete subclasses.

Level 1. Representing conditional plans We model a plan as a set of plan states (called *p-states*, to distinguish them from the *g-states* of the SG algorithm) linked by actions. The plan begins with a start action B , leading into an initial p-state I , and ends with a stop action E , exiting from a goal p-state G . Each p-state has at least one entry action *Aprec* and at least one exit action *Anext*. Entry and exit actions are formalized as environments.

```
pty(class(plan, This, []),
    exi(B,
        exi(E,
            exi(I,
                exi(G,
                    objType(start, B, [I]) & objType(stop, E, [G]))
                ))))
    ))).

pty(objType(pstate, This, [Aprec,Anext]),
    PstateSubPty
    ))).
```

Actions can be modelled in different ways, depending on the kind of planning problems at hand. Here we consider conditional plans. According to [16], we distinguish sensing and non-sensing actions. In this formalisation, a non-sensing action corresponds to a unique arc exiting from a p-state, while a sensing action gives rise to $n \geq 2$ exiting arcs, corresponding to n possible outcomes of the sensing action. An example with a start action *beg*, a stop action *end*, a sensing action *sng* and a non-sensing action *mv* is shown in Fig.1.

In our COOML formalisation, each action links the p-states as environment. For simplicity, we assume that sensing actions have two outcomes, and we distinguish the following types of (abstract) actions:

```
subType(move, [From, To], action, []).
subType(sensing, [From, To1, To2], action, []).
subType(start, [To], action, []).
subType(stop, [From], action, []).
```

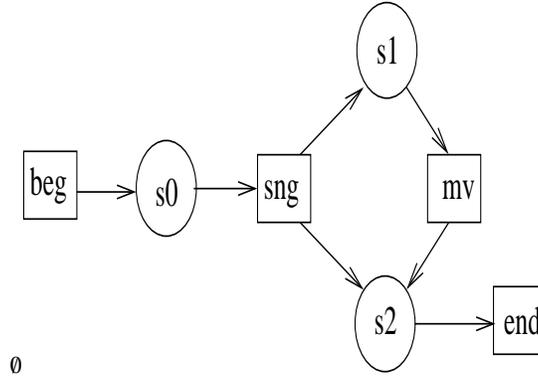


Figure1. A conditional plan

We explain only the move and sensing actions. Their properties and environment constraints are

```

pty(objType(move, This, [From, To]),
    out(This,From) & in(This,To)).
pty(objType(sensing, This, [From, To1, To2]),
    out(This,From) & in(This,To1) & in(This,To2)).

envConstr(objType(move, This, [From, To]),
    next(This:C, InfoFrom:class(pstate,From,[Aprec,This]),
        InfoTo:class(pstate,To,[This,Anext]))).

envConstr(objType(sensing, This, [From, To1, To2]),
    branch(This:C, InfoFrom:class(pstate,From,[Aprec,This]),
        InfoTo1:class(pstate,To1,[This,Anext1]),
        InfoTo2:class(pstate,To2,[This,Anext2]))).
  
```

where `next` and `branch` are information-requirements (see Section 3.1). `InfoFrom`, `InfoTo`, *etc.*, have to be defined in the concrete subclasses of `pstate`. The definition of the GR pertains at level 2. Here, it suffices that all the plans we are interested in can be represented by a population. For example, the conditional plan in Fig. 1 is represented by the following snapshot.

```

[[exi(beg),[exi(end),[exi(s0),[exi(s2),true]]]]]:class(plan,p,[]),
true:class(start,beg,[s0]), true:class(stop,end,[s2]),
true:class(sensing,sng,[s0,s1,s2]), true:class(move,mv,[s1,s2]),
true:class(pstate,s0,[beg,sng]), true:class(pstate,s1,[sng,mv]),
true:class(pstate,s2,[sng,end]), true:class(pstate,s2,[mv,end])
  
```

Level 2. Establishing general problem domain properties In this section we consider the general problem domain properties, to hold in any instance, by means of the generation requirements. We have three main steps, concerning the information-requirements used in the environment constraints of level 1, the instantiation constraints, and possible error

constraints. In our example, this is illustrated as follows. The information-requirements of the action-classes `start`, `stop`, `move` and `sensing` define an action by means of its effect on the information values of the p-states in its environment. In our example this refers to the concrete actions and will be done at level 3.

Concerning the instantiation constraints, we have to define the predicates `id(C,O,E)` and `cInstance(K,S)` (see Section 3.1). At level 2 we partially specify `id` as follows:

```
id(plan, plan, []).
id(start, start, [I]):- init(I).
id(stop, stop, [G]):- goal(G).
id(move,m(A,B),[A,B]) :- lt(A,B).
id(sensing, s(A,B,C),[A,B,C]) :-
    lt(A,B), lt(A,C), not(B=C).
id(pstate, This, [Aprec,Anext]) :-
    init(This); inState(This); goal(This).
```

The predicates `lt(A,B)` (A is less than B), `init(I)` (I is the initial p-state), `goal(G)` (G is the goal p-state) and `inState(This)` (This is an intermediate p-state) have to be fixed when a specific generation problem is established. We have a unique `start` action and a unique `stop` action. If the `lt` predicate is a well-founded order, then no cyclic path can be generated. Indeed, a cycle such as, for example, `m(s0,s1),m(s1,s0)`, would require `lt(s0,s1)` and `lt(s1,s0)`.

Finally, by `cInstance` we constrain the instantiation of atoms with predicates different from `class` and `objType`. We show only the following instantiation constraint for the atom `out`. It imposes that the action exiting from a goal must be `stop`, while an action exiting from a non-goal p-state must be either a `sensing` or a `move` action.

```
cInstance(out(Anext,From),State) :-
    goal(From) -> Anext=stop;
    ( id(sensing,Anext,[From,To1To2]);
      id(move,Anext,[From,To]) ).
```

Level 3. Formalizing and solving a problem instance Now we show how, using the above general formalisation of conditional plans, one can design a specific planning problem. The information content of the specific p-states is specified by a subclass of `pstate`. Then the effect of the actions is specified by implementing the information-requirements of `start`, `stop`, `move` and `sensing`. Finally, by means of `lt`, one fixes a finite set of p-states and a (well-founded) ordering on them.

As an example, let us consider an instance of the *medicate* problem [15]: “A patient is infected. He can take the medicine and get cured if he were hydrated; otherwise, the patient will be dead. To become hydrated, the patient can drink. The check action allows us to determine if the patient is hydrated or not”. We first specify the information content of p-states by the concrete subclass:

```
concreteSubclass(patientstate,[Aprec,Anext],pstate,[Aprec,Anext]).
pty(class(patientstate,This,[Aprec,Anext]),
    ( dead(This) or nonDead(This) ) and
    ( infected(This) or nonInfected(This) ) and
    ( hydrated(This) or true ) )
```

An information value could be, for example, `[[2,true],[1,true],[2,true]]`, indicating that the patient is not dead and infected, but we do not know whether it is hydrated or not (the piece of information `true:true` holds in any state, i.e., it represents complete ignorance).

Then we specify the moves and the sensing actions. We have the sensing action `check` (checking whether the patient is hydrated or not), the move `drink` (the patient drinks and becomes hydrated), and the move `medicate` (the patient is medicated and either becomes non-infected or dead). Here we show the specification of `drink`. We do not have to specify `pty` and `envConstr`, because they are inherited. We have only to specify the abstract method `next`

```
concreteSubclass(drink,[From,To],move,[From,To]).
next(This:class(drink,This,[From,To]),
      [DeadInfo, InfectedInfo, HydratedInfo]:class(patientstate,From,[Aprec,This]),
      [DeadInfo, InfectedInfo, [1,true]]:class(patientstate,To,[This,Anext]))
      :- DeadInfo = [2,true].
```

that is, drinking makes the patient hydrated (the information value `[1,true]` is associated with `hydrated(This)` or `true`), while leaving `DeadInfo` and `InfectedInfo` (associated with the formulas `dead(This)` or `nonDead(This)` and `infected(This)` or `nonInfected(This)` respectively) unchanged. This example also shows how we deal with the frame problem. By `DeadInfo=[2,true]` we express the enabling condition of the action, i.e., that the patient is not dead.

The initial and final states are specified by the concrete subclasses of `start` and `stop`, implementing their information-constraints. Those affect `initially`, and implement `finally`. A possible implementation is:

```
initially([[2,true],[1,true],[2,true]],State).
finally([[2,true],[2,true],X], State).
```

that is, initially the patient is non-dead and infected and we do not know whether he is hydrated or not. Finally, we aim for non-dead and non-infected.

Concerning the states, the relation `lt` is axiomatized by the usual order $0 < 1 < 2 < 3$. With this instantiation we have obtained both the conformant plan where the patient drinks and then is medicated, and the conditional plan where the first action is the sensing action `check` and, according to the cases, either the patient is medicated (if hydrated), or drinks and is medicated.

5 Conclusion

We have presented the semantics of the OO modelling language COOML, a language in the spirit of the UML, but based on a constructive semantics, in particular the BHK explanation of the logical connectives. We have introduced a general notion of snapshot based on populations of objects and structured pieces of information, from which snapshot generation algorithms can be designed. More technically, we have introduced generation goals and the notion of minimal solution of such a goal in the setting of a COOML specification, and we have outlined a non-deterministic generation algorithm

NDGA, showing that finite minimal solutions can be, in principle, generated. As far as implementation is concerned, one needs a decidable generation constraint language GC in order to specify the general properties of the problem domain, as well as the instantiation constraints.

Based on NDGA, we have developed a first prototype SG. At the moment, it has a minimal and low level GC. In particular, for information constraints, we use directly the internal representation of information values. We are designing a higher level language, where one specifies the required information at a logical level, and the corresponding information constraint can be automatically generated. We have done few experiments with our SG algorithm, including some planning examples: the one considered here, the blocks world and the bomb in a toilet problem. Although the GR language is somewhat low level, it was possible to express easily aspects related to instantiation (grounding) and to different kinds of complete or incomplete information. There are similarities with respect to the approaches based on propositional model generation, in particular those based on Answer Set Programming, in so much as an information value can be seen as a kind of answer set. The main difference is that information values are oriented to the kind of information one looks for. Moreover, to generate snapshots we need instantiation constraints, which essentially play the role of grounding in ASP.

The research presented here is a work in progress. We are improving and extending our prototype. Future work will mainly explore two directions.

- First, we want to study in a deeper way the relationship with generation methods based on Answer Set Programming, and the possibility of integrating the two approaches.
- Secondly, we will exploit the possibility offered by CLP's, in particular open constraints.

In particular, we will consider the possibility of using DLV for performing the propositional level of snapshot generation. Roughly, the idea is to adapt our snapshot generation algorithm for universal and existential quantifiers to get a kind of “grounding procedure” which prepares the input for the DLV solver.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. C. Baral, T. Eiter, and J. Zhao. Using SAT and logic programming to design polynomial-time algorithms for planning in non-deterministic domains. In M. M. Veloso and S. Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 578–583. AAAI Press, 2005.
3. C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artif. Intell.*, 147(1-2):85–117, 2003.
4. K. L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 311–325. Morgan Kaufmann, Los Altos, California, 1987.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV^K system. *Artif. Intell.*, 144(1-2):157–211, 2003.

6. M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
7. M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL models by automatic snapshot generation. In *UML*, pages 265–279, 2003.
8. M. Gogolla, M. Richters, and J. Bohling. Tool support for validating UML and OCL models through automatic snapshot generation. In *SAICSIT '03*, pages 248–257, 2003.
9. C. Larman. *Applying UML and Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, 1998.
10. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
11. Y. S. Mahajan, Z. Fu, S. Malik, Zchaff2004: An Efficient SAT Solver. *LNCS 3542*, Special Volume, pages 360–375, Springer Verlag 2004.
12. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
13. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In *LPNMR*, pages 421–430, 1997.
14. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 153(1):67–90, 2006.
15. T. C. Son, P. Tu, and X. Zhang. Reasoning about sensing actions in domains with multi-valued fluents. *Studia Logica*, 79(1):135–160, 2005.
16. T. C. Son, P. H. Tu, and C. Baral. Planning with sensing actions and incomplete information using logic programming. In *Logic programming and nonmonotonic reasoning*, volume 2923 of *Lecture Notes in Comput. Sci.*, pages 261–274. Springer, Berlin, 2004.

Web Site Verification: an Abductive Logic Programming Tool

P. Mancarella¹, G. Terreni¹, and F. Toni²

¹ Dipartimento di Informatica, Università di Pisa, Italy
Email: {paolo,terreni}@di.unipi.it

² Department of Computing, Imperial College London, UK
Email: ft@doc.ic.ac.uk

Abstract. We present the CIFFWEB system, an innovative tool for the verification of web sites, relying upon abductive logic programming. The system allows the user to define web site requirements, i.e. rules that a web site should fulfill. Their fulfillment is checked through abductive reasoning, returning the parts of the web sites responsible for the violation. The rules are expressed by using (a fragment of) the rule-based semi-structured query language Xcerpt. Then, we give a mapping from Xcerpt rules into abductive logic programs, that can be executed by means of the general-purpose CIFF proof procedure for abductive logic programming with constraints. Thus, the resulting CIFFWEB system is a prototype system composed of (1) the CIFF System 4.0 and (2) a translator which, given rules expressed in Xcerpt and XML/XHTML pages of a web site, computes an abductive logic program with constraints that can be run under CIFF 4.0 to identify violations of the rules.

1 Introduction

The exponential growth of the WWW raises the question of maintaining automatically web sites, in particular when the designers of these sites require them to exhibit certain properties at both structural and data level. The capability of maintaining web sites is also important to ensure the success of the Semantic Web vision. As the Semantic Web relies upon the definition and the maintenance of consistent data schemas (XML/XMLSchema, RDF/RDFSchemas, OWL and so on [2]), tools for reasoning over such schemas (and possibly extending the reasoning to multiple web pages) show great promise.

This paper introduces a tool, referred to as CIFFWEB, for verifying web sites against sets of requirements which have to be fulfilled by a web site instance through abductive reasoning.

We define an expressive characterization of rules for checking web sites' errors by using (a fragment of) the well-known semi-structured data query language Xcerpt [5]. With respect to other semi-structured query languages (like XQuery [4] and XPath [8]) which all propose a path-oriented approach for querying semi-structured data, Xcerpt is a rule-based language which relies upon a (partial) pattern matching mechanism allowing one to easily express complex queries

in a natural and human-tailored syntax. Xcerpt shares many features of logic programming, for example its use of variables as place-holders and unification. However, to the best of our knowledge, it lacks (1) a clear semantics for negation constructs and (2) an implemented tool for running Xcerpt programs/evaluating Xcerpt queries. A by-product of this paper is the provision of both (1) and (2) for a fragment of Xcerpt, namely the subset of this language that we adopt for expressing web checking rules.

We map formally the chosen fragment of Xcerpt for expressing checking rules into abductive logic programs with constraints that can be fed as input to the general-purpose CIFF proof procedure [10]. This proof procedure is an extension of the IFF abductive proof procedure [12] allowing a more sophisticated handling of variables and integrating a constraint solver for arithmetical operations. CIFF is proven sound with respect to the three valued completion semantics [13] and has been implemented in the CIFF System 4.0 [14]. By mapping web checking rules onto abductive logic programs with constraints and deploying CIFF for determining fulfillment (or identify violation) of the rules, we inherit the soundness properties of CIFF thus obtaining a sound concrete tool for web checking.

CIFF is a general-purpose logic programming procedure and it is not able to handle directly semi-structured data of the kind Xcerpt does. Thus, we also define, as part of the CIFFWEB tool, a simple XML/XHTML translation into a representation of the web pages suitable for CIFF, and we rely upon that representation in defining the translation function for the web checking rules.

At the end of the translation process the CIFF System 4.0 can be successfully used to reason upon the (translation of the) web checking rules finding those XML/XHTML instances not fulfilling the rules, and representing errors as abducibles in abductive logic programs.

2 Motivation

When searching the web, it is easy to encounter web pages containing errors in their structure and/or their data. The main goal of this section is to identify some typical errors and define a rule pattern for checking these errors.

We argue that, in most cases, considering an XML/XHTML web site instance, the errors can be divided into two main categories: structural errors and content-related (data) errors. Structural errors are those errors concerning the presence and/or absence of tag elements and relations amongst tag elements in the pages. For example, if a tag *tag1* is intended to be a child of a tag *tag2*, the occurrence in the web site of a *tag1* instance outside the scope of a *tag2* instance is a structural error. Data errors, instead, are about the in-tag data content of tag elements. For example a *tag3* could be imposed to hold a number greater than 100.

To better exemplify the types of error we consider, we present here a very simple XML web site instance of a theater company. The site is composed of two pages representing respectively a list of shows produced by the company and the list of directors of the company, given below

```
%% showindex.xml
```

```
%% directorindex.xml
```

```

<showlist>
  <show>
    <showname>Epiloghi</showname>
    <dir_by>dir1</dir_by>
  </show>
  <show>
    <showname>Black Comedy</showname>
  </show>
  <show>
    <showname>Confusioni</showname>
    <dir_by>dir3</dir_by>
  </show>
</showlist>

```

```

<directorlist>
  <director>
    dir1</director>
  <director>
    dir2</director>
  <director>
    dir3</director>
  <director>
    dir2</director>
</directorlist>

```

We can specify a number of rules which any such web site instance should fulfill. For example, we could specify that the right *structure* of a `show` tag in the first page must contain both a `showname` tag element and a `dir_by` tag element as its child. In this case we have a *structural error*, due to the lack of a `dir_by` tag element in the second show of the list. Moreover we could specify that two `director` tags in the second page must not contain the same data. In this case we have a *data error* due to the double occurrence of the `dir2` data.

Requirements (and thus errors) can involve more than one web page. For example, a possible requirement for the theater company specification may be that

each director must direct at least one show

The above requirement can lead to content-related errors which involve both pages. There is a simple way to check this requirement: for each `director` tag data in the `directorindex` page, if there does not exist a matching data of a `dir_by` tag in the `showindex` page then a data-content error is detected. This is the case for `dir2` in the earlier example.

In all examples in this section, we have assumed that an error instance is fired by a piece of XML/XHTML data which does not fulfill a certain requirement (or specification). We will first formalize any such requirement as a *web checking rule*. For each such rule we need both a *condition part* and an *error part*. For each instance of the considered data such that the condition part is matched, an error needs to be returned.

3 Web checking rules

In order to formalize and characterize requirements, such as the ones expressed in natural language in the earlier section, as web checking rules we first need a formal language. Our choice is to use the Xcerpt [5] language: a deductive, rule-based query language for semi-structured data which allows for direct access to XML data in a very natural way. Our characterization of web checking rules can be accommodated straightforwardly in a fragment of the Xcerpt language. Here, we give some background notions about the Xcerpt fragment we use ³.

³ The full Xcerpt language is much more expressive than the fragment we adopt here for expressing web checking rules. For further information about Xcerpt see [5].

An Xcerpt program is composed of a GOAL part and a FROM part. The FROM part provides access to the sources (XML files or other sources) via (partial) pattern matching among terms, while the GOAL part reassembles the results of the query into new terms. Variables can be used within either parts and act as placeholders (as in logic programming).

As an example, the requirement, in the context of our earlier example pages, that *each director must appear at most once in the director list [Rule1]* can be expressed in Xcerpt as follows:

```
GOAL
  all err [var Dir1, var Dir2, "double director occurrence"]
FROM
  in { resource {"file:directorindex.xml"},
      directorlist {{
        director {{var Dir1}},
        director {{var Dir2}}
      }}
  } where {Dir1 = Dir2}
END
```

The main Xcerpt statement we use in the GOAL part is the `all t` statement (where `t` is a term, `err` in our example), indicating that each possible instance of `t` satisfying the FROM part gives rise to a new instance of `t` returned by the GOAL part. In our methodology for writing web checking rules, `all t` will always be `all err`, where `err` stands for “error”.

In the FROM part, an access to a `resource` is wrapped within an `in` statement. Multiple accesses must be connected by `and` indicating that all subqueries have to succeed in order to make the whole query succeed. The main Xcerpt query terms `t` which we use in our work are: (1) double curly brackets, i.e. `t{{ }}`, denoting *partial term specification* of `t`; the order of the subterms of `t` within the curly brackets is irrelevant; (2) variables, expressed by `var` followed by an identifier (variable name); values for variables can be strings and numeric values; (3) a `where` statement for expressing constraints through standard operators like `=`, `\=`, `<`, `>`, `<=`; (4) subterms of the form `without t` denoting *subterm negation*, illustrated within the following formulation of the requirement *each director directs at least one show [Rule2]*:

```
GOAL
  all err [var DirName, "director w/out show"]
FROM
  and {
    in { resource {"file:directorindex.xml"},
        director {{ var DirName }},
    in { resource {"file:showindex.xml"},
        showlist {{
          without show {{ dir_by {{ var DirName}} }}
        }}
    }
  }
}
END
```

`without` is only applicable to subterms t that do not occur at *root level* in the underlying web pages (in our example, `showlist` and `directorlist` occur at root level); all variables that occur within a `without` have to appear elsewhere outside the `without` and finally a `without` subterm cannot occur nested.

4 A Xcerpt-like grammar for positive web checking rules

To simplify the presentation, we first focus our attention on positive web checking rules, i.e. rules in which negation (the `without` statement in Xcerpt syntax) does not occur.

In defining the syntax, we distinguish between *basic* syntactic categories and *structural* syntactic categories. The basic categories are used to define the basic components of the rules, namely variables, constants (strings and numbers), XML tags and constraints (e.g. $X > Y$). The structural categories, instead, are the main parts of the Xcerpt rules, as we have seen in the examples before: the query part, the error part, the `in` construct and so on.

For each syntactic category, we avoid stating explicitly the syntactic rules for sequences of elements derived from it. We use instead the following notational convention. Given a syntactic category C , C^* denotes the syntactic category for elements derived from C . The metarule for C^* is the following:

$$C^* ::= C, C^* \mid \epsilon \quad (\epsilon \text{ denotes the empty string})$$

The grammar for the basic categories is the following:

$Const$	$::= String \mid Number$	$VarOrConst$	$::= Var \mid Const$
Tag	$::= String$	Rel	$::= < \mid > \mid \leq \mid \geq \mid = \mid \backslash =$
$VarName$	$::= String$	$Constraint$	$::= Var Rel VarOrConst$
Var	$::= \text{var } VarName$		

Notice that the Tag and the $VarName$ categories generate simply strings. However we keep them distinct categories to improve readability of the syntactic rules for the structural categories.

In the second part of the grammar we define effectively the interesting constructs of the web checking rules. Recall that a web checking rule is composed of two main parts: a query part and an error part. The error part gives the error specification, while the query part is a conjunction of queries represented by an `and` wrapping a list of `in` construct. Each element of the conjunction (i.e. each `in` construct) expresses a query involving a specific resource. At the end of the whole conjunction a set of constraints can be expressed by the `where` construct.

$CheckRule$	$::= \text{GOAL } Error \text{ FROM } Query \text{ END}$
$Error$	$::= \text{all err } [VarOrConst^*, String]$
$Query$	$::= InPart \text{ Where}$
$InPart$	$::= In \mid \text{and } In^*$
In	$::= \text{in } \{ \{ Resource \} Term \}$
$Resource$	$::= \text{resource } \{ \text{file: } String \}$
$Term$	$::= Tag \{ \{ VarOrConst^* Term^* \} \}$
$Where$	$::= \text{where } \{ Constraint^* \} \mid \epsilon$

We denote by $\alpha \in C$ that α is a string derived from the syntactic category C .

5 The (positive) translation process

In this section we show formally how to translate positive web checking rules into abductive logic programs suitable for the CIFF proof procedure. This section is organized as follows: first we give a brief background about abductive logic programs, then we show how we represent the XML/XHTML pages in a useful way for CIFF, then we show the formal translation function for the rules. Finally we give some examples of translations.

5.1 Abductive logic programming with constraints

An *abductive logic program with constraints* (ALPC) consists of (1) a constraint logic program P , referred to as the *theory*, namely a set of clauses of the form $A \leftarrow L_1 \wedge \dots \wedge L_m$, where the L_i s are literals (ordinary or abducible atoms, their negation, or constraint atoms in some underlying language for constraints), A is an ordinary atom and whose variables are all implicitly universally quantified from the outside; (2) a finite set of *abducible predicates*, that do not occur in any conclusion A of any clause in the theory, and (3) a finite set of *integrity constraints* (ICs), namely implications of the form $L_1 \wedge \dots \wedge L_m \rightarrow A_1 \vee \dots \vee A_n$ where the L_i s are literals and the A_j s are (ordinary, abducible, constraint or *false*) atoms, and whose variables are all implicitly universally quantified from the outside. The theory provides definitions for ordinary predicates, while constraint atoms are evaluated within an underlying structure \mathfrak{R} , as in conventional constraint logic programming.

A *query* is a conjunction of literals (whose variables are implicitly existentially quantified). An *answer* to a query specifies which instances of the abducible predicates should hold so that both (1) (some instance of) the query is entailed by the constraint logic program extended with the abducibles and (2) the ICs are satisfied [10]. More precisely, chosen the three-valued completion semantics [13] as the underlying logic programming semantics, let $\models_{3(\mathfrak{R})}$ denote the notion of entailment in the 3-valued completion appropriately augmented, à-la-constraint logic programming, with a notion of satisfiability for the underlying structure \mathfrak{R} for interpreting the constraint atoms in the abductive logic program. Then, an answer for a query Q is a pair (Δ, C) such that, for each substitution σ for the variables in $\Delta \cup C \cup Q$ such that $\sigma \models_{3(\mathfrak{R})} C$, it holds that (1) $P \cup \Delta \sigma \models_{3(\mathfrak{R})} Q \sigma$ and (2) $P \cup \Delta \sigma \models_{3(\mathfrak{R})} ICs$.

The CIFF procedure computes such answers. It operates with a presentation of the theory as a set of *iff-definitions*, which are obtained by the (selective) *completion* of all predicates defined in the theory except for the abducible and the constraint predicates. CIFF returns three possible outputs: (1) an abductive answer to the query (a set of possibly non-ground abducible atoms and a set of constraints on the variables of the query and of the abducible atoms); (2) a failure, indicating that there is no answer and finally (3) an *undefined* answer, indicating that a critical part of the input is not *allowed*. Allowedness relates to input formulae with certain quantification patterns for which the concept of a (finite) answer cannot be defined [10].

A CIFF computation starts from a *node* composed of the query plus the ICs. Then the CIFF procedure repeatedly applies a set of *rewrite rules* (see [10]) to the nodes in order to update them and to construct a proof search tree. A node containing a goal *false* is called a *failure node*. If all branches in a derivation terminate with failure nodes, then the procedure is said to fail (no answer to the query). A non-failure (allowed) node to which no more proof rules apply can be used to extract an abductive answer.

The CIFF System 4.0 is a Prolog implementation of CIFF, available at www.di.unipi.it/~terreni/research.php.

The system takes as input a list of files representing an ALPC. The abducible predicates `Pred` are declared explicitly in the form `abducible(Pred)` while the operators representing clauses and ICs are respectively `:-` (Prolog-like syntax) and `implies`. Within clauses and integrity constraints, negative literals are represented in the form `not(Atom)`, constraint atoms are represented in the form `#>`, `#<`, `#=`, `#\=...`, and disequality atoms are represented as `A\=B`.

5.2 XML representation

To pave the way to writing, in a format suitable for the CIFF system, specification rules for properties of web sites, we first provide a suitable representation of web sites data. Obviously CIFF is not able to handle directly XML data and XML structure, hence we propose a translation whereby for each tag element of the original XML file, an atom `pg_el` is generated having the following form:

```
pg_el(ID,TagName,IDFather)
```

where `TagName` is the name of the tag element, whereas `ID` and `IDFather` represent the unique identifiers for the tag element and its father. They are needed for keeping information about the structure of the XML page.

Furthermore a data inside a tag element is represented by an atom of the form:

```
data_el(ID,Data,IDFather)
```

The following is the translation of the XML pages seen in section 2

```
xml_pg('showindex.xml',0).      data_el(12,'Confusioni',11).
pg_el(1,showlist,0).          pg_el(13,dir_by,10).
pg_el(2,show,1).              data_el(14,'dir3',13).
pg_el(3,showname,2).
data_el(4,'Epiloghi',3).      xml_pg('directorindex.xml',15).
pg_el(5,dir_by,2).            pg_el(16,directorlist,15).
data_el(6,'dir1',5).          pg_el(17,director,16).
pg_el(7,show,1).              data_el(18,'dir1',17).
pg_el(8,showname,7).          pg_el(19,director,16).
data_el(9,'Black Comedy',8).  data_el(20,'dir2',19).
pg_el(10,show,1).             pg_el(21,director,16).
pg_el(11,showname,10).        data_el(22,'dir3',21).
```

For each page, we store also a fact of the form `xml_pg(FileName,ID)`, holding the name of the file and the unique identifier of the page

5.3 Positive translation function

We are now ready to show the formal translation function for positive web checking rules mapping them from Xcerpt-like syntax to abductive logic programs. We define inductively two functions, namely $\mathcal{T}[\mathbb{C}]$ and $\mathcal{T}'[\alpha](id)$ which, given a string α derivable from one of the syntactic categories defined in the previous section, return a string, representing a part of an abductive logic program⁴. The abductive logic program corresponding to a whole web checking rule is obtained by applying the function $\mathcal{T}[\mathbb{R}]$ to a rule $\mathbb{R} \in \text{CheckRule}$. This abductive logic program with constraints is then suitable as input for CIFF. In particular the translation of a *CheckRule* gives rise to a single CIFF integrity constraint.

The two translation functions differ in that $\mathcal{T}'[\alpha](id)$ has one extra argument, id , representing a number. Intuitively, a call of the form $\mathcal{T}'[\alpha](id)$ amounts at translating the element α which is a component of another element, uniquely identified by id . This identifier id is needed in the translation of α to keep the correspondence between α and the element it is part of. In what follows we assume that the auxiliary function

$newID()$

generates a fresh variable whenever called. This function will be used when a new identifier id will be needed to uniquely identify the element being translated.

For brevity we omit the specification of the translation of the basic syntactic elements, since they basically remain unchanged (e.g. the translation of a string α representing a variable name is α itself).

In Xcerpt, variables are used as in logic programming, that is they can be shared between distinct parts of a rule by using the same name. In the translation, variable names are simply left unchanged, so that sharing is maintained.

We use a top-down approach in defining the translation functions. Hence, the first rule is for the *CheckRule* category.

CheckRule

```

 $\mathcal{T}[\text{GOAL Error FROM Query END}] =$ 
  let   Head =  $\mathcal{T}[\text{Error}]$ 
    and Body =  $\mathcal{T}[\text{Query}]$ 
  in
    Body implies Head.

```

As we can see the result is a string representing a CIFF integrity constraint: the body is the translation of the query part and the head is the translation of the error part. Intuitively, each instance of the XML data matching the body will fire an instance of the head representing the corresponding error. We will see that the predicate used in Head of such constraint is abducible, hence firing the constraint will amount to abducting its head, constructed as follows:

Error

```

 $\mathcal{T}[\text{all err[Vars,Msg]}] = [\text{abd_err}([\text{Vars}],\text{Msg})]$ 

```

⁴ As usual, we use $[\]$ to highlight the syntactic arguments of the translation functions

The predicate `abd_err/2` is defined as abducible. The two arguments are (1) the variable list occurring in the error part of the rule and (2) the error message. For the query component, the translation produces the conjunction of the translations of the *InPart* component and of the *Where* component of the query.

Query

```

T[[InPart Where]] =
  let      Conjunction = T[[InPart]]
    and    Constraints = T[[Where]]
  in
    [Conjunction, Constraints]

```

The constraints in the *Where* part of a query are simply translated into the same conjunction of CIFF constraints: we omit this part for space reasons. The translation of the *InPart* of a query results in the conjunction of the translations of the single components.

In-1

```

T[[in{{resource{file : Res} Term}}]] = T[[Term]]

```

In-2

```

T[[In and InRest]] =
  let      Conjunct = T[[In]]
    and    OtherConjuncts = T[[InRest]]
  in
    Conjunct, OtherConjuncts

```

The following rules translate terms in *Term*, and produce the conjuncts in the conjunction constituting the body of the integrity constraint corresponding to a checking rule. Each term specifies a tag and hence the translation amounts at producing an instance of the `pg_e1` predicate, possibly in conjunction with other conjuncts corresponding to the translation of the subtags. A fresh variable is generated through the `newID()` function, to uniquely represent the tag being translated. Note the usage of the function $\mathcal{T}'\cdot$ for the translation of the data elements (variables and constants) and of the subtags of the tag being translated.

Term-1

```

T[[tagName {{VarOrConstSeq, TermSeq}}]] =
  let      Id = newID()
    and    DataElements = T'[[VarOrConstSeq]](Id)
    and    SubTerms = T'[[TermSeq]](Id)
    and    Ids = IdVars(DataElements) ∪ IdVars(SubTerms)
    and    Constraints = all_distinct(Id)
  in
    pg_e1(Id, tagName, ...), DataElements, SubTerms, Constraints

```

In the above rule, we have introduced two new auxiliary functions:

- `IdVars(X)` which is assumed to return the set of all *Ids*, generated by the `newID()` function, occurring in its argument *X*;
- `all_distinct(Vars)` which is assumed to generate the constraint

$$\bigwedge_{\substack{x,y \in Vars \\ x \neq y}} x \neq y$$

Term-2.1

$$\begin{aligned}
T'[\text{var VarName}](Id) = & \\
\text{let } Id' = \text{newID}() & \\
\text{in} & \\
\text{data_el}(Id', \text{VarName}, Id) &
\end{aligned}$$

Term-2.2 (A similar rule for constants which we omit for space reasons)

Term-3

$$\begin{aligned}
T'[\text{subTagName } \{\{\text{VarOrConstSeq}, \text{TermSeq}\}\}](Id) = & \\
\text{let } Id' = \text{newID}() & \\
\text{and DataElements} = T'[\text{VarOrConstSeq}](Id') & \\
\text{and SubTerms} = T'[\text{TermSeq}](Id') & \\
\text{in} & \\
\text{pg_el}(Id', \text{subTagName}, Id), \text{DataElements}, \text{SubTerms} &
\end{aligned}$$
Term-4

$$\begin{aligned}
T'[\text{X}, \text{Xs}](Id) = & \\
\text{let } \text{Conjunct} = T'[\text{X}](Id) & \\
\text{and OtherConjuncts} = T'[\text{Xs}](Id) & \\
\text{in} & \\
\text{Conjunct}, \text{OtherConjuncts} &
\end{aligned}$$

Cases **Term-2.1** and **Term-2.2** produce instances of the `data_el/3` predicate. Notice that the translation amounts at producing an instance `data_el(Id', El, Id)`, where `El` is the actual data element, `Id'` is the unique identifier assigned to it and `Id` is the unique identifier of the term the data element is part of. In the case **Term-3** we have a translation similar to case **Term-1**, the only difference being that in the case **Term-1** the third argument of `pg_el/3` is the anonymous variable, whereas in case **Term-3** is the unique identifier of the term containing the subterm being translated. Finally, the case **Term-4** corresponds to the translation of a sequence `X, Xs` in *VarOrConst** or in *Term**.

Example 1. Let us consider the web checking rule [*Rule1*] for the theater company web site, seen in section 3. To simplify the presentation of the translation results, let us identify the main syntactic components of the rule:

```

err1    = all_err [ var Dir1, var Dir2, "double director occurrence" ]
query1  = in1 , where1
in1     = in {resource {"file:directorindex.xml"} root1 }
root1   = directorlist { {
            director { {var Dir1} }, director { {var Dir2} } } }
where1  = where {Dir1 = Dir2}

```

Let us apply the translation function \mathcal{T} to *Rule1*.

$$\begin{aligned}
\mathcal{T}[\text{Rule1}] &= \mathcal{T}[\text{query}_1] \text{ implies } \mathcal{T}[\text{err}_1]. \\
\mathcal{T}[\text{err}_1] &= \text{abd_err}([\text{Dir1}, \text{Dir2}], \text{"double director occurrence"}) \\
\mathcal{T}[\text{query}_1] &= [\mathcal{T}[\text{in}_1] \ \mathcal{T}[\text{where}_1]] \\
\mathcal{T}[\text{in}_1] &= \mathcal{T}[\text{root}_1] \\
\mathcal{T}[\text{where}_1] &= \text{Dir1} = \text{Dir2} \\
\mathcal{T}[\text{root}_1] &= \text{pg_el}(\text{ID1}, \text{directorlist}, _), \text{ID2} \# \neq \text{ID4}, \\
&\quad \text{pg_el}(\text{ID2}, \text{director}, \text{ID1}), \text{data_el}(\text{ID3}, \text{Dir1}, \text{ID2}), \\
&\quad \text{pg_el}(\text{ID4}, \text{director}, \text{ID1}), \text{data_el}(\text{ID5}, \text{Dir2}, \text{ID4})
\end{aligned}$$

Summarizing we have the following translation:

```

[pg_el(ID1,directorlist,_), ID2 #\= ID4
 pg_el(ID2,director,ID1), data_el(ID3,Dir1,ID2)
 pg_el(ID4,director,ID1), data_el(ID5,Dir2,ID4), Dir1 = Dir2]
 implies
 [abd_err([Dir1,Dir2], "double director occurrence")].

```

6 Adding negation to the translation process

The mapping of web checking rules considering also negative parts, i.e. with **without** statements in Xcerpt syntax, is a bit more complicated. The resulting abductive logic program is no longer a single integrity constraint but it includes also a set of clauses which defines new (fresh) predicates needed for handling correctly the negations. Also the grammar must be extended for covering the **without** construct on which we impose the following restrictions: (1) a **without** could not appear at the top level of an **in** statement, (2) **without** statements could not appear nested, and (3) variables appearing at any deep level in the scope of a **without** appear elsewhere outside a **without** in the *query part*. These limitations are directly borrowed from the Xcerpt specification.

The modifications/extensions to the grammar are the following:

$ \begin{aligned} Term & ::= Tag \{ \{ VarOrConst^* Term^* Without^* \} \} \\ Without & ::= without PosTerm without VarOrConst \\ PosTerm & ::= Tag \{ \{ VarOrConst^* PosTerm^* \} \} \end{aligned} $
--

Due to lack of space we omit the definition of the modified translation function but we show only how it works by means of a translation example. A complete definition can be found at www.di.unipi.it/~terreni/research.php.

Let us consider again the theater company web site. We want to express that *each show has at least a director* [Rule3]. The Xcerpt representation is as follows:

```

GOAL all err [ var ShowName, "show without a director" ]
FROM in {resource {"file:showindex.xml"},
 show {{ showname {{var ShowName}},
 without dir_by {{ }} }} } END

```

We show only the detailed translation of the interesting components:

```

root3 = show {{ showname {{var ShowName}} wout3 }}
wout3 = without dir_by {{ }}

```

The $root_3$ element is translated as follows (note that now the output is composed of two parts: the conjuncts part and the clauses part resulting from $wout_3$).

```

T[[root3]] =
let < wout_conjs ; wout_clauses > = T'[[wout3]](ID1)
in < pg_el(ID1,show,_), pg_el(ID2,showname,ID1),
 data_el(ID3,ShowName,ID2),wout_conjs ; wout_clauses >

```

Finally, we translate the $wout_3$ part adding a negated atom (of a fresh predicate pred_1) into the body of the integrity constraint, and we define pred_1 with the conjuncts derived from the term specification inside the **without** statement. The variables used as arguments for pred_1 are for keeping the right bindings outside the integrity constraint.

$$\mathcal{T}'[\llbracket wout_3 \rrbracket](ID1) = \langle \text{not}(\text{pred}_1(ID1)) ; \text{pred}_1(ID1) :- \text{pg_el}(ID4, \text{dir_by}, ID1) \rangle$$

Roughly speaking, the definition of $\text{pred}_1(ID1)$ gives the following meaning to $\text{not}(\text{pred}_1(ID1))$: *there exists no ID4 dir_by tag that is a child of ID1.*

7 Analysis

In this section we outline how our translation provides a semantics for the fragment of Xcerpt we have adopted for defining web checking rules, and CIFFWEB a sound mechanism for performing the checking, by virtue of the soundness of CIFF for abductive logic programming with constraints. Given a web site W , let $\mathcal{X}(W)$ be the result of applying the translation given in section 5.2 to W . $\mathcal{X}(W)$ is a set of ground unit clauses. Given a set of web checking rules R , let $\mathcal{T}[\llbracket R \rrbracket] = \{\mathcal{T}[\llbracket r \rrbracket] \mid r \in R\}$.

$\mathcal{T}[\llbracket R \rrbracket]$ is an abductive logic program with constraints $\langle P, A, I \rangle$ such that A consists of all atoms in the predicate abd_err and P and I are given as in section 6.⁵ Trivially, $\langle P \cup \mathcal{X}(W), A, I \rangle$ is an ALPC. Note that the abducible predicate abd_err only occurs in the conclusions of integrity constraints in I .

We say that the rules in R are fulfilled in W if and only if $P \cup \mathcal{X}(W) \models I$, where \models stands for consequence wrt the 3-value completion [13]. We also say that the rules in R are violated in W if and only if $P \cup \mathcal{X}(W) \not\models I$ and there exists $\Delta \subseteq A$ such that $P \cup \mathcal{X}(W) \cup \Delta \models I$.

Intuitively, due to the fact that the abducible atoms can appear only in the head of a rule, the need of abducible atoms (i.e. a non-empty Δ) for satisfying the integrity constraints means that some web checking rule has been violated. Furthermore the abducible atoms in Δ represent those violations. By soundness of CIFF [10] we obtain that, with an empty query, if CIFFWEB succeeds returning an empty answer then all rules in R are satisfied in W ; if CIFFWEB succeeds returning a non-empty answer then some rule in R is violated.

8 Running the System

In order to run the CIFFWEB system the web checking rules in Xcerpt syntax and the XML/XHTML pages are needed. The system will compile all the rules and the sources giving rise to the corresponding abductive logic program with constraints. That ALPC can be used directly as input for the CIFF System 4.0. Detailed information on how to run the CIFFWEB System can be found at www.di.unipi.it/~terreni/research.php.

⁵ \mathcal{T} actually returns the representation of the ALPC in the format required by CIFF System 4.0. We assume here the logical - rather than system - version of this ALPC.

Running the system with the three rules and the two XML pages seen above, the following abductive answer is produced:

```
[abd_err(['dir2','dir2'],'double director occurrence'),
 abd_err(['Black Comedy'],'show without a director'),
 abd_err(['dir2'],'director w/out a show'].
```

representing correctly (1) the fact that 'dir2' appears twice in the director list, (2) that the show 'Black Comedy' has been inserted without a director associated with it and finally (3) that 'dir2' does not direct any show. As the results show, the CIFFWEB system conjoins all the web checking rules and the abducibles in the answer correspond to all the error instances of all the rules.

9 Related Work, Future Work and Conclusions

In this paper we have illustrated the CIFFWEB system for verifying web sites by using abductive logic programming and in particular the CIFF [10] proof procedure as computational counterpart. Despite the exponential WWW growth and the success of the Semantic Web, there is limited work on web sites verification. Notable exceptions are [15] (which mainly inspired our work) [9] and [11]; the XLINKIT framework [7] and the GVERDI-R system [1, 3].

The work more closely related to ours is the GVERDI-R system [1, 3] which verifies web sites against rules written in an ad-hoc language which allows the user to specify both correctness and completeness rules to be fulfilled. The GVERDI-R web rule language is an ad-hoc language which relies upon a (partial) pattern-matching mechanism very similar to the Xcerpt one and its expressiveness is comparable to our Xcerpt fragment. However we argue that our use of negation constructs in the query part of the rules allows for a bit more expressiveness. Furthermore, it seems that queries like *[Rule1]* seen in section 3 are difficult to express in the GVERDI-R language due to the ordering imposed by its language specification in a subterm specification of a query.

Another key difference is that the GVERDI-R system relies upon an ad-hoc computational counterpart for its framework, while our system relies upon the general purpose CIFF abductive proof procedure. This is an important issue because while the GVERDI-R system had to define both the language and its formal properties, the CIFFWEB systems inherits all from the CIFF proof procedure which has a well-defined syntax and has been proved sound.

Conversely, the GVERDI-R system allows for the use of functions for managing strings and data in a non-straightforward way, e.g. by matching strings to regular expressions or using arithmetic functions on numbers. While the CIFFWEB system deals with arithmetic functions thanks to the underlying integrated constraint solver, it lacks the use of other types of functions. As pointed out in [1], this is an important feature for a web verification tool.

We chose Xcerpt [6, 5] as our specification language for web checking rules because it is a well-recognized query language for semi-structured data as XML and its human-tailored syntax allows us to express complex queries in a simpler way than other well-known query languages as, e.g., XPath [8] and XQuery [4].

Moreover, the possibility of expressing negation (through the `without` construct) makes Xcerpt a very expressive language and its similarities to logic programming, e.g. the use of variables as placeholders and its declarative and operational semantics based on the concept of unification, make Xcerpt very suitable to have a computational counterpart based itself on (abductive) logic programming. We remark that there are a number of interesting features of Xcerpt which could be integrated in our framework, e.g. the possibility of representing aggregated data, which are typical contents of web pages.

Although we benefit from the choice of Xcerpt for representing web checking rules for the reasons given above, note that our approach could be applied to any other formulation of web checking rules that can be mapped onto ALPCs. Finally, we give a hint of the main line of our ongoing work. We believe that abductive reasoning could be exploited not only for checking errors of a web site as we have shown in this paper, but also for repairing such errors. The idea is to modify checking rules by adding new clauses and integrity constraints which could abductively suggest how to repair the site instead of simply detecting errors. These suggestions can then be used by a human expert to perform repairs. In practice, the system should identify actions for repairing the web sites in the form of new elements to be integrated in the data. Obviously new problems arise as the introduced data could interfere with the other checking/repairing rules. This means that a simple Xcerpt interpreter is not sufficient to evaluate web repair rules: we need a computational tool which takes into account all the possible rules interconnections. We give here a sketch of how our framework could be extended for embracing repairing rules. The first idea is to introduce two new abducible predicates, namely `abd_pg_el` and `abd_data_el` which intuitively represent the "abducible versions" of the `pg_el` and `data_el` predicates. The key difference is that atoms of the abducible versions can be dynamically introduced into the framework giving a way for fixing the errors into the web pages.

Let us consider again the requirement that *each show has at least a director [Rule3]*. The ALPC obtained after the translation process (as described in this paper) is the following:

```
pred_3(ID0) :- pg_el(ID3,dir_by,ID0).

[pg_el(ID0,show,_), pg_el(ID1,showname,ID0),
 data_el(ID2,ShowName,ID1), not(pred_3(ID0))]
implies
[abd_err([ShowName], 'show without a director')].
```

Instead of returning an error by means of an `abd_err` atom, we could add a new director as a child of the interested `show` tag. This could be done by replacing the head of the integrity constraint with `[repair_3(ID0)]` where:

```
repair_3(ID0) :- abd_pg_el(_,dir_by,ID0).
```

In this way the abductive process tries to add a `dir_by` element as a child of the show `ID0` instead of displaying an error. This example can be run directly as an ALPC on the CIFF System (although CIFFWEB does not handle repair rules directly yet) which returns the answer: `abd_pg_el(_A,dir_by,18)`. This answer represents a new element intended to be a child of the `showname` element whose

id number given by the CIFFWEB translation is 18. Being a new element, the system cannot instantiate an id number for the abduced *dir_by* element and it models that situation through the unbound variable $_A$. This is an example of not-straightforward, non-ground abduction which can be handled by CIFF.

The above is a very simple example giving an idea of the possibilities of the abductive reasoning for repairing purposes. A characterization of web repair rules and a formal methodology for obtaining them from web checking rules is still work in progress. It is also not totally clear if an Xcerpt-like language could accommodate straightforwardly web repair rules.

Indeed, there are many issues to take into account when new atoms are abduced and added to the framework. In particular web checking rules must be aware of the fact that tag elements or data elements could be added to the framework in order to check if those additions imply further violations to them.

Also the GVERDI-R system seems to make some steps in this direction as pointed out in [3], but a repairing specification and a concrete tool are still work in progress. Also the XLINKIT system seems to be capable of some form of repairing actions but they are limited to broken links in web sites.

References

1. M. Alpuente, D. Ballis, and M. Falaschi. A rewriting-based framework for web sites verification. *Elect. Notes in Theoretical Computer Science*, 124:41–61, 2005.
2. G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. MIT Press, 2004.
3. D. Ballis and D. Romero. Fixing web sites using correction strategies. In *Proc. WWW'06*, 2006.
4. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML query language, 2007.
5. F. Bry, T. Furche, and B. Linse. Data model and query constructs for versatile web query languages: State-of-the-art and challenges for Xcerpt. In *Proc. PPSWR06*, pages 90–104, 2006.
6. F. Bry and S. Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics, 2002.
7. L. Capra, W. Emmerich, A. Finkelstein, and C. Nentwich. Xlinkit: a consistency checking and smart link generation service. *ACM Transac. on IT*, 2:151–185, 2002.
8. J. Clark and S. DeRose. XML Path language (XPath) version 1.0, 1999.
9. T. Despeyroux and B. Trousse. Semantic verification of web sites using natural semantics. In *Proc. CC-BMIA'00*, 2000.
10. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In *Proc. JELIA 2004*, 2004.
11. M. Fernandez, D. Florescu, A. Levy, and D. Suci. Verifying integrity constraints on web site. In *Proc. IJCAI 1999*, 1999.
12. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.
13. K. Kunen. Negation in logic programming. *Journal of LP*, 4:231–245, 1987.
14. P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Programming applications in CIFF. In *Proc. LPNMR 2007, to appear*, 2007.
15. F. Toni. Automated information management via abductive logic agents. *Journal of Telematics and Informatics*, 18(1):89–104, 2001.

PROGRAMMAZIONE LOGICA MULTI-VALUED, SINONIMIA, CONTROLLO

Daniele Genito

Dipartimento di Matematica ed Informatica Università di Salerno
via Ponte don Melillo - 84084 - Fisciano (SA)
danielegenito@libero.it

Abstract. In questo lavoro si espongono alcuni tratti di un progetto di ricerca relativo alla possibilità di utilizzare la programmazione logica classica (e in particolare il Prolog) per implementare alcune forme di argomentazione tipiche della logica fuzzy.

1. Introduzione

All'interno della logica fuzzy [4], [7] si è sviluppato un interessante capitolo riguardante la possibilità di estendere la programmazione logica in modo da coinvolgere predicati vaghi [6], [9]. In questo lavoro viene esplorata l'idea per cui la programmazione logica classica possa essere una metalogica con cui si descrive e si definisce la programmazione logica fuzzy. Un approccio interessante riguardante le metaregole viene considerato in [1] in cui si definiscono le nozioni di valori di *support* e *confidence* e li si associano ad ogni regola, trasformandola in una sorta di metaregola.

In [8] viene analizzato un aspetto della programmazione logica relativo a come combinare il concetto di *Answer Set Programming* (ASP) con la logica fuzzy nel contesto della *Fuzzy Answer Set Programming* (FASP), rendendo contemporaneamente più flessibile ed elegante tutto l'apparato.

L'innovatività del nostro approccio consiste però nella possibilità di definire predicati che parlano di altri predicati (una sorta di metapredicati) siano essi classici o vaghi e di introdurre relazioni tra essi. Ciò permette di formalizzare forme di argomentazioni che utilizzano la nozione di sinonimia (intesa come relazione fuzzy), un tipico esempio di metapredicato di tipo fuzzy (si vedano ad esempio [2], [3]).

2. Il Prolog come metalogica della logica classica

Consideriamo il seguente programma che coinvolge una relazione binaria r e la sua estensione simmetrica esr :

$r(a,b).$
 $r(b,c).$
 $r(c,d).$
 $r(s,s).$

$esr(X,Y):- r(X,Y);r(Y,X).$

L'operazione che faremo è considerare i nomi di predicati r e esr come costanti accettando la possibilità che esistano predicati che parlano di predicati. Osserviamo che l'asserzione del fatto $r(a,b)$ a livello metalinguistico si esprimerebbe nel modo seguente: "la relazione r riferita alle costanti a e b definisce un'asserzione il cui grado di verità è 1". Appare allora naturale introdurre un predicato $val2$ in modo che la formula $val2(R,X,Y,V)$ significhi "il predicato binario R è verificato da X ed Y con grado V " per poi considerare il fatto $val2(r,a,b,1)$. Così i primi quattro fatti del nostro programma saranno trasformati in:

$val2(r,a,b,1). \quad val2(r,b,c,1). \quad val2(r,c,d,1). \quad val2(r,s,s,1).$

Per quanto riguarda la regola è possibile tradurla nella metaregola:

$val2(Rel2,X,Y,1):-estende_simm(Rel2,Rel1),(val2(Rel1,X,Y,1);val2(Rel1,Y,X,1)).$

Questa regola è generale e indica il comportamento di una qualunque relazione $Rel2$ che estende simmetricamente una data relazione $Rel1$. A tale regola è necessario poi aggiungere il fatto che la relazione esr estende la relazione r :

$estende_simm(esr,r).$

Allo stesso modo possono essere considerate altri tipi di estensioni di una data relazione. Ad esempio, se al dato programma si volesse aggiungere la definizione dell'estensione riflessiva err di r , normalmente si dovrebbe aggiungere la regola

$err(X,Y) :- X = Y ; r(X,Y).$

Equivalentemente possiamo considerare la metaregola generale:

$val2(Rel2,X,Y,1):- estensione_rifl(Rel2,Rel1),(X=Y;val2(Rel1,X,Y,1)).$

a cui è necessario aggiungere il fatto

$estensione_rifl(err,r).$

Con considerazioni analoghe si può definire anche l'estensione transitiva.

3. Il prolog come metalogica della logica fuzzy

Vediamo ora come si possono estendere le tecniche del paragrafo precedente per implementare la logica fuzzy in Prolog. Ricordiamo che nella logica fuzzy usualmente il connettivo logico della congiunzione viene interpretato tramite una norma triangolare \otimes . Inoltre le regole di inferenza vengono opportunamente "fuzzyficate". Ad esempio, il Modus Ponens, la regola di particolarezzazione e la \wedge -introduzione vengono estese nel modo seguente

$$\frac{\alpha, \alpha \rightarrow \beta}{\beta} \quad \frac{\lambda, \mu}{\lambda \otimes \mu} ; \quad \frac{\forall x \alpha \quad \lambda}{\alpha(t) \quad \lambda} ; \quad \frac{\alpha, \beta}{\alpha \wedge \beta} \quad \frac{\lambda, \mu}{\lambda \otimes \mu}$$

Il significato di tali regole è ovvio. Ad esempio il Modus Ponens esteso assicura che se è stato provato α con grado λ ed è stato provato $\alpha \rightarrow \beta$ con grado μ , allora è possibile concludere β con grado $\lambda \otimes \mu$. Una buona formalizzazione sull'uso di varie norme per associare valori di verità di tipo fuzzy (plausibilità) a regole di programmi si può trovare in [5], con un'attenta analisi di come si propaga l'incertezza dell'informazione utilizzando regole di programma.

Applichiamo tale modo di procedere ad un programma fuzzy cioè ad un fuzzy insieme di clausole di programma. Consideriamo il seguente esempio dove il valore scritto tra parentesi quadre rappresenta il grado con cui si accetta la relativa clausola.

$ama(carlo,luisa). \quad [0.3]$

ama(carlo,X)←giovane(X)∧bella(X). [0.9]
bella(maria). [0.8]
giovane(maria). [0.7]
giovane(elena). [0.7]

Data la domanda ama(carlo,maria), per particolarizzazione si ottiene la formula

ama(carlo,maria) ← giovane(maria)∧bella(maria)

con grado 0.9. D'altra parte, per la ∧-introduzione si ottiene la formula

giovane(maria)∧bella(maria)

con grado uguale a $0.8 \otimes 0.7$. Successivamente tramite il Modus Ponens si ottiene

ama(carlo,maria)

con grado $0.9 \otimes 0.8 \otimes 0.7$. Per implementare in Prolog una tale procedura si può agire in questo modo. Si trasformano tutti i predicati presenti nel programma in costanti, raggruppandoli a seconda della loro arità. Si definisce poi un nuovo predicato per ogni arità. Vediamo come si potrebbe trasformare il programma precedente:

teo2(ama,carlo,luisa,0.3).
teo2(ama,carlo,X,V) :- teo1(giovane,X,V1),teo1(bella,X,V2), V is V1*V2*0.9 .
teo1(bella,maria,0.8).
teo1(giovane,maria,0.7).
teo1(giovane,elena,0.7).

dove come norma si è usata la norma del prodotto. Osserviamo che i vecchi predicati sono stati trasformati in costanti e così facendo, si è passati ad un livello superiore, in cui gli oggetti sono i predicati stessi, sui quali si possono introdurre nuove relazioni. Inoltre si è implementato il metapredicato “essere un teorema” che per motivi computazionale si è diviso in due predicati teo2 e teo1 relativi ad arità di tipo due o di tipo uno dei predicati coinvolti. Tuttavia se si effettua l'interrogazione teo2(ama,carlo,elena,V) la risposta non sarà $V = 0$ ma NO, espressione di un fallimento. Aggiungiamo la regola per default per cui ogni asserzione è almeno un teorema con grado 0

teo1(R,X,0).

Dopo ciò invece la risposta sarà $V = 0$. Così si è esaminato il caso di regole che siano congiunzione di letterali positivi. E' possibile anche coinvolgere la disgiunzione considerando la regola:

ama(carlo,X)←giovane(X)∨bella(X). [0.9]

che verrà tradotta in

teo2(ama,carlo,X,V) :- teo1(giovane,X,V1),teo1(bella,X,V2), V is 0.9*max(V1,V2).

In tale caso la regola per default è necessaria. Infatti, se si prova a chiedere teo2(ama,carlo,elena,V) la risposta sarà NO, cosa completamente contraria all'intuizione. Naturalmente in programmazione logica la disgiunzione può essere evitata spezzando ogni regola in più regole, nel nostro caso si avrebbe:

teo2(ama,carlo,X,V) :- teo1(giovane,X,V1), V is V1*0.9 .
teo2(ama,carlo,X,V) :- teo1(bella,X,V1), V is V1*0.9 .

Così facendo si evita di inserire la regola per default nel caso della disgiunzione.

Osserviamo che in questo modo si risolvono i possibili problemi dovuti relativi alla regola per default per tutte le regole che sono riconducibili a congiunzione o disgiunzione di letterali positivi (cioè quelle che di solito sono utilizzabili in Prolog).

Ovviamente teo2 e teo1 potrebbero non fornire un unico valore V. Infatti possono esistere diverse dimostrazioni della stessa formula ciascuna con un grado diverso.

Ciò è coerente con le idee della logica fuzzy che prevedono la “fusione” dei contributi forniti dalle diverse dimostrazioni. Questo fatto comporta l’esigenza di raccogliere in una lista tutti i valori forniti dalle diverse dimostrazioni per poi calcolare il massimo dei valori. Questo si può ottenere, ad esempio tramite la regola:

$$\text{teorema}(R,X,Y,V):-\text{findall}(V1,\text{teo2}(R,X,Y,V1),L),\text{massimo}(L,V).$$

4. Sinonimia

Supponiamo che un commesso di una libreria abbia la richiesta, da parte di una cliente, di un libro d’avventura che sia anche economico. Supponiamo inoltre che non trovi disponibile nessun libro che verifichi le proprietà di essere avventuroso ed economico, allora sarà portato a consigliarne altri che si avvicinino alle esigenze della cliente. Ad esempio potrebbe proporre un libro di fantascienza che non costi troppo. Questo atteggiamento è un tipico esempio d’uso della sinonimia che è applicato nella vita di tutti i giorni: il commesso sa che “libro di fantascienza” può essere considerato come sinonimo di “libro d’avventura” e quindi anche se il libro che è in grado di proporre non sarà esattamente (con grado 1) ciò che la cliente cercava, è comunque ragionevole proporlo. L’argomentazione che si vuole implementare è:

<i>Regole e fatti</i>	
Buono (X) ← Avventuroso (X) ∧ Economico (X)	[1]
Fantascienza(Io Robot)	[1]
Non caro(Io Robot)	[1]
<i>Relazione di sinonimia</i>	
sinonimo(Fantascienza,Avventuroso)	[0.8]
sinonimo(Economico,Non caro)	[0.7]
<i>Conclusione</i>	
Buono (Io Robot)	$[1 \otimes 1 \otimes 1 \otimes 0.7 \otimes 0.8]$

N.b.: il simbolo \otimes rappresenta una qualunque norma triangolare.

La spiegazione dello schema di sopra è che il libro X è buono con grado 1 per la cliente se è avventuroso ed economico, ma poiché essere di fantascienza è sinonimo di essere avventuroso con grado ad esempio 0.8, e non caro è sinonimo di economico con grado 0.7, un libro non caro e di fantascienza sarà buono per la cliente con grado $[1 \otimes 1 \otimes 1 \otimes 0.7 \otimes 0.8]$. I numeri 0.8 e 0.7 possono allora essere visti come il prezzo da pagare per forzare un’unificazione che altrimenti non sarebbe possibile. Supponiamo di voler introdurre la relazione di sinonimia tra due predicati qualsiasi, ad esempio per definire il predicato “economico” come sinonimo del predicato “non caro”, vediamo come poterlo fare:

$$\text{pred1}(A,X,V):-\text{sinonimo}(A,B,V1),\text{pred1}(B,X,V2),V \text{ is } V1*V2 .$$

$$\text{sinonimo}(\text{economico},\text{noncaro},0.8).$$

La prima riga è di carattere universale: essa stabilisce che se un qualsiasi predicato A è sinonimo del predicato B con grado di sinonimia V1, e il predicato B vale su X con grado V2, allora si avrà che anche il predicato A vale su X con un grado funzione di V1 e V2 (in questo caso particolare è stato utilizzato il prodotto). Notiamo che in

questo caso i predicati sono indicati come variabili, in modo che questa riga definisca il concetto di sinonimia tra due predicati generici. La seconda riga invece caratterizza semplicemente un possibile valore di sinonimia tra i due particolari predicati “economico” e “non caro”. Un altro esempio è nella definizione di un predicato che ne neghi un altro già presente nel programma; vediamo come ottenerlo:

```
not_pred1(P,X,V):-pred1(P,X,V1),V is 1-V1.  
pred1(brutta,X,V):-not_pred1(bella,X,V).
```

Nella prima riga, è definita la generica negazione di un predicato P: se il predicato P vale su X col valore V1, allora il predicato not p varrà su X con il valore 1-V1; anche questa riga è del tutto generale e può essere utilizzata per un qualsiasi predicato di arità 1 (nella seconda riga c'è un esempio di utilizzo per la definizione del predicato “brutta” come negazione del predicato “bella”).

È immediato il vantaggio che si è conseguito con questo approccio: in precedenza la relazione di sinonimia tra predicati non era facilmente implementabile, invece ora è stata ottenuta modificando i predicati in veri e propri oggetti del programma.

Il Prolog, con questo tipo di approccio, diventa una sorta di metalinguaggio, questo significa che i nomi di predicati sono considerati come costanti e quindi che si possono definire predicati che parlano di predicati. È come se si trattasse di logica del secondo ordine in cui si può operare sui predicati come se fossero degli oggetti su cui stabilire nuove relazioni e operazioni; in altre parole siamo saliti di un livello.

È importante osservare la grande duttilità di quest'approccio: tutto ciò che si poteva affermare prima nel Prolog standard, lo si può ottenere anche adesso, ma si sono potute introdurre nuove relazioni sui predicati in maniera efficiente e pulita, ampliando le possibilità e la potenza del programma.

Un'ulteriore applicazione potrebbe essere un'estensione di quanto visto finora, al caso in cui i valori di verità dei predicati siano considerati in un bireticolo prodotto (ad esempio in $[0,1] \times [0,1]$) invece che nel reticolo $[0,1]$.

Bibliografia

1. Ben-Eliyahu-Zohary R, Gudes E., Ianni G., Metaqueries: Semantics, complexity, and efficient algorithms, *Artificial Intelligence*, 149 (2003), pp. 61-87.
2. Fontana A., Formato F., Gerla G., Fuzzy unification as a foundation of fuzzy logic programming. In *Logic Programming and Soft Computing*, (T. P. Martin and F. Arcelli Fontana Eds.). Research Studies Press, 1998, pp. 51-68.
3. Formato F., Gerla G., Sessa M., Similarity-based unification, *Fundamenta Informaticae*, 41 (2000) 393-414.
4. Goguen J. A., The logic of inexact concepts, *Synthese*, 19 (1968/69) pp. 325-373.
5. Mateis C., Extending Disjunctive Logic Programming by T-norms, *LPNMR 1999* pp. 290-304.
6. Medina J., Ojeda-Arciego M., Vojtas P., Multi-adjoint logic programming with continuous semantics. In *Proc. LPNMR'01*, 2001, to appear.
7. Pavelka J., On fuzzy logic I: Many-valued rules of inference, *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 25 (1979) pp. 45-52.
8. Van Nieuwenborgh D., De Cock M., Vermeir D., Fuzzy Answer Set Programming. In *Proc. JELIA 2006*, pp. 359-372.
9. Vojtas P., Fuzzy logic programming, *Fuzzy Sets and Systems*, 124 (2001) pp. 361-370.

Modeling CNF-formulae and formally proving the correctness of DPLL by means of Referee

Eugenio G. Omodeo¹, Alexandru I. Tomescu²

¹University of Trieste, Dipartimento di Matematica e Informatica
eomodeo@units.it

²University of Bucharest, Faculty of Mathematics and Computer Science
alexandru.tomescu@gmail.com

Abstract. This paper reports on using the *Referee* proof-verification system to formalize issues regarding the satisfiability of CNF-formulae of propositional logic. We specify an “archetype” version of the Davis-Putnam-Logemann-Loveland algorithm through the THEORY of recursive functions based on a well-founded relation, and prove it to be correct. Within the same framework, and by resorting to the Zorn lemma, we develop a straightforward proof of the compactness theorem.

Key words. Proof checking, program-correctness verification, set theory, satisfiability decision procedures, proof modularization.

1 Introduction

Referee, or *Ref* for short (see [7, 5, 6]), is a broad gauge proof-verification system based on set theory; its design and implementation evolved hand-in-hand with the development of a fully formalized proof scenario which will culminate in a proof of the Cauchy integral theorem on analytic functions. One virtue of *Ref* is that the syntax of proofs is very close to natural mathematical notation: this ensures readable and reusable proofware.¹

Formal methods of algorithm verification are a natural yield of proof checking and automated deduction. Algorithms have often been examined formally with systems such as Isabelle [2] or Coq [1], but the experience on which we will report is the first verification of an algorithm carried out with *Referee* (ours could even be—as far as we know—the first fully formalized proof of correctness of the Davis-Putnam-Logemann-Loveland procedure, DPLL). Although *Ref* does not, up until today, encompass specific programming notation, this paper suggests that an integration of such notation with *Ref*'s logical notation is easy to conceive.

We will formalize the notion of model for a formula in conjunctive normal form, and will introduce an “archetype” version of DPLL. To encapsulate the relevant concepts in the *Referee* system, we will use the THEORY mechanism [7]. Like procedures in a programming language, theories have lists of formal

¹ *Proofware* is the peculiar scripting code used to specify in absolute rigor proofs and proof schemes. Its primary application is not to describe algorithms.

parameters. Each theory requires its parameters to meet a set of assumptions. When applied to a list of actual parameters that have been shown to meet the assumptions, a theory will instantiate several additional output set, predicate, and function symbols, and then supply a list of theorems initially proved explicitly (relative to the formal parameters) by the user inside the theory itself. These theorems will generally involve the new symbols. Roughly speaking, the assumptions met by the formal parameters can be regarded as preconditions, while the theorems supplied in the “theory” are postconditions. By the term *interface* we refer to the list of formal parameters of a THEORY, the assumptions they meet, and the statements of the theorems proved within the THEORY.

M. Davis and H. Putnam [3], and later M. Davis, G. Logemann, and D. Loveland [4], proposed an algorithm for determining whether a propositional formula given in conjunctive normal form is satisfiable or not. A vast literature has evolved from this seminal paper [4] — before it, researchers in the Automated Deduction field thought they had to reduce ground sentences to equivalent *disjunctive* normal form to be tested for validity [9], whereas nowadays reduction to conjunctive normal form and satisfiability test have become rather standard steps.

The version of DPLL underlying our work is best specified in rather conventional imperative terms, like those seen in Figure 1 where the programming language used is SETL [10]. But, in order to comply with the Ref notation, we must formalize DPLL in purely logical, set-theoretical terms, as will be shown and explained at due time.

This paper is organized as follows. In Section 2 we discuss how to create, out of a given set of so-called “atoms”, a set of affirmative and negative *literals* (i.e., sets which can act conveniently as pure symbols) of which the affirmative ones are equi-numerous with the atoms. Section 3 introduces *CNF-formulae* along with the pertaining notions of *model* and *satisfiability*. It also states lemmas on how to *reduce* a CNF formula S , relative to a literal X occurring within it: this reduction either produces a single CNF S' , simpler but equi-satisfiable with S , or it produces two simpler CNFs, S' and S'' , such that S is satisfiable if and only if either S' or S'' is satisfiable. In Section 4, the algorithm *DPLL* is specified and shown to be correct. The specification is provided in such terms as to ensure termination, but grossly, in the sense that no criterion is indicated for selecting the literal X relative to which the CNF-formula S will be reduced (such a criterion, in fact, would have a bearing on efficiency but is immaterial as far as correctness is concerned). The variant of *DPLL* is also shown to be correct which discards, at the outset, all tautological clauses from the input formula. In Section 5 we briefly report on how the compactness theorem about propositional logic was proved with Ref.

The issues discussed in Sections 2, 3, 4 reflect into three Ref theories. As these consist of approximately 2000 lines of code altogether, we cannot afford discussing in full the various constructions and proofs which support our mathematical discourse: only the interfaces of the theories will be shown, in a dedicated

Appendix. Notwithstanding, we will provide examples and stress sensible points of the proof of correctness.

The interested reader should turn to [6], [7] and [11] for syntax, basic terminology and features of Ref.

2 The signedSymbols theory

The `signedSymbols` theory, whose interface is shown in Appendix A, receives via its formal parameter a set `atms` not subject to any assumption. Out of this set, it creates a set `litsθ` whose elements shall be regarded as *literals*. Internally, this theory defines an injective affirmation operation, e.g.

$$\text{aff}_\theta(\mathbf{X}) =_{\text{Def}} \{\{\mathbf{X}\}\},$$

converting every element `x` of `atms` into a positive literal. A negation operation, e.g.

$$\text{neg}_\theta(\mathbf{X}) =_{\text{Def}} (\mathbf{X} \setminus \{\emptyset\}) \cup (\{\emptyset\} \setminus \mathbf{X}),$$

is also defined internally. Thus, if we regard each `negθ(affθ(x))` as a negative literal and stick to the above definitions, the collection `litsθ` of all positive and negative literals constructed inside the `signedSymbols` theory turns out to be

$$\text{lits}_\theta =_{\text{Def}} \{b \cup \{x\} : b \subseteq \{\emptyset\}, x \in \text{atms}\}.$$

Along with `affθ(-)`, `negθ(-)`, and `litsθ`, the `signedSymbols` theory returns a designation for falsehood, e.g.

$$\text{false}_\theta =_{\text{Def}} \emptyset,$$

such that the pair `falseθ`, `negθ(falseθ)` of complementary *truth values* does not intersect `litsθ`. Note that the theorems externalized by this theory include

$$\langle \forall x \mid \text{neg}_\theta(\text{neg}_\theta(x)) = x \ \& \ \text{neg}_\theta(x) \neq x \rangle,$$

stating that the global function `negθ` is a GALOIS CORRESPONDENCE.

We remark in passing that the above-shown precise definitions of `affθ(-)`, `negθ(-)`, `litsθ`, and `falseθ`, are devoid of interest outside the theory we are considering and, as such, do not deserve appearing explicitly on the theory interface. Speaking in general, a theory should be designed in such a way that reworking of its internals does not propagate outside the theory in question, to other theories that make use of it.

Actually, a second-release implementation of the `signedSymbols` theory—motivated by the author’s desire to make `signedSymbols` more widely usable, e.g. for the development of a theory of freely generated groups—insists on the fact that all of the syntactic entities in the collection

$$\{x : x \in \text{lits}_\theta\} \cup \{\text{false}_\theta, \text{neg}_\theta(\text{false}_\theta)\}$$

Fig. 1. A procedural specification of the “archetype” DPLL algorithm

```

procedure dp0(s)
  -- s is a finite (conjunctive) set of disjunctive clauses of literals

  t := {c: c in s, h in c | neg(h) in c};
  -- tautological clauses within input conjunction
  m := dp1(s - t);
  -- analyze the rest; if unsatisfiable, propagate {fail},
  -- else enlarge its model
  return
    if fail in m then {fail}
    else m + {arb({h,neg(h)}): c in s, h in (c - m) | neg(h) in (c - m)} end if;

procedure dp1(s);
  -- s is a finite set of disjunctive non-tautological clauses
  -- the model of s under construction will be
  -- enlarged repeatedly, and s will be simplified
  -- until s is either satisfied or blatantly false

  if s = {} then return {} end if; -- obvious
  if {} in s then return {fail}; end if; -- absurd

  h := selectLiteral(s);
  -- if the selected literal appears in a singleton clause,
  -- apply the unit literal rule

  if {h} in s then
    -- the literal h was chosen within a unit clause
    return dp1({c - {neg(h)}: c in s | h notin c}) + {h};
  end if;

  -- otherwise, proceed either to pure literal rule
  -- or to splitting rule

  if (FORALL d in s | neg(h) notin d) then
    -- include the pure literal h in the model
    return dp1({c in s | h notin c}) + {h};
  end if;

  -- split
  m1 := dp1({c - {neg(h)}: c in s | h notin c}) + {h};
  return
    if fail notin m1 then m1
    else dp1({c - {h}: c in s | neg(h) notin c}) + {neg(h)} end if;
end dp1;

procedure selectLiteral(s);
  -- Here we are assuming that s is a non-null set of non-null clauses
  -- we pick a literal in one of the clauses of s, and return it
  return arb(arb(s)); -- refined selection criteria can enhance efficiency
end selectLiteral;

end dp0;

```

have the same, transfinite, set-theoretic rank (see Figure 2).

It may be worth recalling here that the global rank function (intuitively speaking, a measure of how deeply nested every set is) has the recursive definition

$$\text{rk}(X) \stackrel{\text{Def}}{=} \{\text{rk}(y) \cup \{\text{rk}(y)\} : y \in X\}.$$

The new constraint about rank (entailing, among others, that false_θ can no longer equal \emptyset), as well as the modest changes needed inside the theory to meet this constraint, have no bearing whatsoever on the exploitation of `signedSymbols` discussed in the ongoing.

$$\begin{aligned} & \{\text{rk}(x) : x \in \text{lits}_\theta\} \subseteq \{\text{rk}(\text{false}_\theta)\} \\ & \langle \forall n \in \mathbb{N} \cup \{\mathbb{N}\} \mid n \in \text{rk}(\text{false}_\theta) \rangle \\ & \langle \forall x \mid \text{rk}(x) \notin \text{rk}(\text{false}_\theta) \rightarrow \text{rk}(\text{neg}_\theta(x)) = \text{rk}(x) \rangle \end{aligned}$$

Fig. 2. Addendum to the interface of the `signedSymbols` theory

3 The `cnfModels` theory

Any CNF-formula

$$S = c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

of propositional calculus, where each clause c_i is a disjunction

$$c_i = h_{i,1} \vee h_{i,2} \vee \cdots \vee h_{i,\#c_i} \quad (\#c \text{ being the cardinality of } c)$$

of distinct literals, is conveniently represented as the set

$$S = \{\{h_{1,1}, h_{1,2}, \dots, h_{1,\#c_1}\}, \dots, \{h_{n,1}, h_{n,2}, \dots, h_{n,\#c_n}\}\}.$$

In what follows, as we think of a ‘‘CNF-formula’’ in these set-theoretical terms, we shall say that the set M is a `MODEL` of S if and only if:

- i*) M is not contradictory (i.e., M contains no complementary literals $h, \neg h$),
- ii*) every set in S shares at least one literal with M .

We say that a formula S is `SATISFIABLE` iff there exists a model of S .

The `cnfModels` theory (see Appendix B) receives a global Galois correspondence $h \mapsto \neg h$, and introduces the notions of model and satisfiability of a CNF-formula as just explained:

$$\text{Is_cnfModel}_\theta(M) \stackrel{\text{Def}}{\leftrightarrow} \{h \in M \mid \neg h \in M\} = \emptyset;$$

$$\text{Has_cnfModel}_\theta(S, M) \stackrel{\text{Def}}{\leftrightarrow} \text{Is_cnfModel}_\theta(M) \ \& \ \{c \in S \mid M \cap c = \emptyset\} = \emptyset;$$

$$\text{Is_cnfSat}_\theta(S) \stackrel{\text{Def}}{\leftrightarrow} \langle \exists m \mid \text{Has_cnfModel}_\theta(S, m) \rangle.$$

The main purpose of this theory is to prove various lemmas enabling the reduction of an instance of the satisfiability/modeling problem for conjunctive normal forms to a simpler instance of the same problem.

Relative to a CNF-formula S , when c is a clause in S , and h is a literal in c , we shall say that

- c is a TAUTOLOGICAL CLAUSE iff

$$\langle \exists h' \in c \mid \neg h' \in c \rangle \quad (\text{tautological clause})$$

- c is a UNIT CLAUSE and h is a UNIT LITERAL iff

$$c = \{h\} \quad (\text{unit clause, unit literal})$$

- h is a PURE LITERAL iff

$$\langle \forall c' \in S \mid \neg h \notin c' \rangle \quad (\text{pure literal})$$

Let us now examine the algorithm in Figure 1. To support the claims which we will provide as explanations during the analysis that follows, we will produce the statements of various theorems whose proofs have been written and verified with Ref; note that variables written in upper case within such statements are, by convention, universally quantified. To be short, we denote by $\bigcup S$ the union-set $\{x : y \in S, x \in y\}$, and by $\mathbf{arb}(X)$, where X is any set, an arbitrary member of X (conventionally, $\mathbf{arb}(\emptyset) = \emptyset$).² Relative to S , c , and h as above, we must consider the following cases:

(i) Both h and $\neg h$ belong to c , and hence c is a tautological clause. In this case, if M is a model for $S \setminus c$ then if M contains h or $\neg h$, it is a model for S as well. If it does not contain h or $\neg h$, then adding either of $h, \neg h$ will make M into a model for S . To state this, one can introduce the following theorem:

THEOREM. $\text{Has_cnfModel}_\emptyset(S, M) \rightarrow \text{Has_cnfModel}_\emptyset(S \cup \{c : c \in S_0, h \in c \mid \neg h \in c\}, M \cup \{\mathbf{arb}(\{h, \neg h\}) : c \in S_0, h \in c \setminus M \mid \neg h \in c \setminus M\})$.

(ii) h is a unit literal. Using the definition of $\text{Has_cnfModel}_\emptyset$, we have that h belongs to any model M of S . Therefore, any model of the simplified formula $\{\{h\}\} \cup \{c \setminus \{\neg h\} : c \in S \mid h \notin c\}$ will also be a model of S .

THEOREM. $\{H\} \in S \ \& \ \text{Has_cnfModel}_\emptyset(S, M) \rightarrow H \in M \ \& \ \neg H \notin M$;
 THEOREM. $\text{Has_cnfModel}_\emptyset(\{\{H\}\} \cup \{c \setminus \{\neg H\} : c \in S \mid H \notin c\}, M) \rightarrow \text{Has_cnfModel}_\emptyset(S, M)$.

(iii) h is a pure literal. By the theorem below, establishing the satisfiability of S amounts to establishing the satisfiability of the reduced formula

² One can think that $\mathbf{arb}(_)$ originates from Skolemization of the *regularity* axiom $\langle \forall x \exists y \forall v \mid v \in x \rightarrow (v \notin y \ \& \ y \in x) \rangle$ of the Zermelo-Fraenkel set theory.

$\{\{h\}\} \cup \{c \in S \mid h \notin c\}$. This formula, stripped of all clauses containing h , requires h to belong to its model, as discussed above.

THEOREM. $\neg H \notin \bigcup S \rightarrow (\text{Is_cnfSat}_\theta(S) \leftrightarrow \text{Is_cnfSat}_\theta(\{\{H\}\} \cup \{c \in S \mid H \notin c\}))$.

(iv) h is neither a unit literal nor a pure literal. Then $\neg h \in S$. Splitting the formula S on h means checking whether either of the two formulae

$$\{\{h\}\} \cup \{c \setminus \{\neg h\} : c \in S \mid h \notin c\}, \quad \{\{\neg h\}\} \cup \{c \setminus \{h\} : c \in S \mid \neg h \notin c\}$$

has a model. As h or $\neg h$ are unit literals here, this amounts to checking whether h or $\neg h$ can be put in a model of S .

THEOREM. $\text{Is_cnfSat}_\theta(S) \leftrightarrow \text{Is_cnfSat}_\theta(\{\{H\}\} \cup \{c \setminus \{\neg H\} : c \in S \mid H \notin c\}) \vee \text{Is_cnfSat}_\theta(\{\{\neg H\}\} \cup \{c \setminus \{H\} : c \in S \mid \neg H \notin c\})$.

To exemplify the Ref proof-checking mechanism and its inference steps, we illustrate below how to prove the above splitting principle by resorting to the following theorems, which must have been proved already:

THEOREM cnfModels_0 . $\neg\neg H = H \ \& \ \neg H \neq H$;

THEOREM cnfModels_{18} . $\text{Has_cnfModel}_\theta(S, M) \rightarrow$

$$\text{Has_cnfModel}_\theta(\{\{H\}\} \cup \{c \setminus \{\neg H\} : c \in S \mid H \notin c\}, M \cup \{H\}) \vee \text{Has_cnfModel}_\theta(\{\{\neg H\}\} \cup \{c \setminus \{H\} : c \in S \mid \neg H \notin c\}, M \cup \{\neg H\});$$

THEOREM cnfModels_{20} . $\text{Is_cnfSat}_\theta(\{\{H\}\} \cup \{c \setminus \{\neg H\} : c \in S \mid H \notin c\}) \rightarrow \text{Is_cnfSat}_\theta(S)$.

The proof of the splitting principle runs as follows:

THEOREM cnfModels_{21} . $\text{Is_cnfSat}_\theta(S) \leftrightarrow \text{Is_cnfSat}_\theta(\{\{H\}\} \cup \{c \setminus \{\neg H\} : c \in S \mid H \notin c\}) \vee \text{Is_cnfSat}_\theta(\{\{\neg H\}\} \cup \{c \setminus \{H\} : c \in S \mid \neg H \notin c\})$. PROOF:

Suppose $\text{not}(s, h_0) \Rightarrow \text{AUTO}$

Let (s, h_0) be a counterexample. We consider the direct implication first:

Suppose $\Rightarrow \text{Is_cnfSat}_\theta(s) \ \& \ \neg(\text{Is_cnfSat}_\theta(\{\{h_0\}\} \cup \{c \setminus \{\neg h_0\} : c \in s \mid h_0 \notin c\}) \vee \text{Is_cnfSat}_\theta(\{\{\neg h_0\}\} \cup \{c \setminus \{h_0\} : c \in s \mid \neg h_0 \notin c\}))$

Use $\text{def}(\text{Is_cnfSat}_\theta) \Rightarrow \text{Stat1} : \langle \exists m \mid \text{Has_cnfModel}_\theta(s, m) \rangle \ \&$

$\text{Stat2} : \neg \langle \exists m \mid \text{Has_cnfModel}_\theta(\{\{h_0\}\} \cup \{c \setminus \{\neg h_0\} : c \in s \mid h_0 \notin c\}, m) \rangle \ \&$

$\text{Stat3} : \neg \langle \exists m \mid \text{Has_cnfModel}_\theta(\{\{\neg h_0\}\} \cup \{c \setminus \{h_0\} : c \in s \mid \neg h_0 \notin c\}, m) \rangle$

$\langle m_1 \rangle \hookrightarrow \text{Stat1}(\text{Stat1}, \star) \Rightarrow \text{Stat4} : \text{Has_cnfModel}_\theta(s, m_1)$

Using Theorem cnfModels_{18} we can construct a model for one of the two sets in the hypothesis

$\langle s, m_1, h_0 \rangle \hookrightarrow \text{TcnfModels}_{18}(\text{Stat4}, \star) \Rightarrow$

$\text{Has_cnfModel}_\theta(\{\{h_0\}\} \cup \{c \setminus \{\neg h_0\} : c \in s \mid h_0 \notin c\}, m_1 \cup \{h_0\}) \vee$

$\text{Has_cnfModel}_\theta(\{\{\neg h_0\}\} \cup \{c \setminus \{h_0\} : c \in s \mid \neg h_0 \notin c\}, m_1 \cup \{\neg h_0\})$

Both cases get discarded, as they contradict *Stat2* or *Stat3*, respectively

Suppose \Rightarrow *Stat5* : $\text{Has_cnfModel}_\Theta(\{\{h_0\}\} \cup \{c \setminus \{\neg h_0\} : c \in s \mid h_0 \notin c\}, m_1 \cup \{h_0\})$
 $\langle m_1 \cup \{h_0\} \rangle \hookrightarrow \text{Stat2}(\text{Stat5}, \star) \Rightarrow$ false
Discharge \Rightarrow *Stat6* : $\text{Has_cnfModel}_\Theta(\{\{\neg h_0\}\} \cup \{c \setminus \{h_0\} : c \in s \mid \neg h_0 \notin c\}, m_1 \cup \{\neg h_0\})$
 $\langle m_1 \cup \{\neg h_0\} \rangle \hookrightarrow \text{Stat3}(\text{Stat6}, \star) \Rightarrow$ false; Discharge \Rightarrow AUTO

We now consider the reverse implication, and suppose that the first statement holds. By Theorem cnfModels_{20} we arrive at a contradiction

Suppose \Rightarrow $\text{Is_cnfSat}_\Theta(\{\{h_0\}\} \cup \{c \setminus \{\neg h_0\} : c \in S \mid h_0 \notin c\})$
 $\langle h_0, s \rangle \hookrightarrow T\text{cnfModels}_{20}(\star) \Rightarrow$ false; Discharge \Rightarrow AUTO

It remains to be shown that the second statement holds. Given that $\neg\neg h_0 = h_0$, we can again apply cnfModels_{20} . We have discarded all possible cases, hence the proof is complete.

$\langle h_0 \rangle \hookrightarrow T\text{cnfModels}_0(\star) \Rightarrow$ $\neg\neg h_0 = h_0$
EQUAL \Rightarrow $\text{Is_cnfSat}_\Theta(\{\{\neg\neg h_0\}\} \cup \{c \setminus \{\neg\neg h_0\} : c \in s \mid \neg\neg h_0 \notin c\})$
 $\langle \neg\neg h_0, s \rangle \hookrightarrow T\text{cnfModels}_{20}(\star) \Rightarrow$ false; Discharge \Rightarrow QED

Some readers may be perplexed about an apparent terminology misuse in what precedes: in spite of our indication that the notions of model and satisfiability refer to formulae in conjunctive normal form, the formal definitions of the predicates $\text{Has_cnfModel}_\Theta(S, _)$ and $\text{Is_cnfSat}_\Theta(S)$ do not enforce any typing restriction reflecting the idea that S should be a set of sets of literals. The very requirement that the correspondence \smile be defined globally (i.e., that $\smile X$ yield a value for every set X) may look over-demanding, if one thinks that literals should be drawn from a specific set (say the set of non-null integer numbers) instead of from the class of all sets. The theory displayed in Figure 3 accommodates things for those who think that a better way of modeling negation would be by means of a self-inverse function whose domain is the specific set whose elements represent literals and truth constants. As a matter of fact, it provides a mechanism for extending any such “toggling map” into a global Galois correspondence. Once this patch is available, one will probably convene that working without the encumbrance of a typing discipline not only is safe but also offers some advantages to the proof designer.³

³ To clarify the theory interface shown in Figure 3, we must say that a pairing function $x, y \mapsto \langle x, y \rangle$ and projections $p \mapsto p^{[1]}$, $p \mapsto p^{[2]}$ have been defined globally so that $\langle \forall x, y \mid \langle x, y \rangle^{[1]} = x \ \& \ \langle x, y \rangle^{[2]} = y \rangle$. By the notation $\text{Svm}(F)$ we indicate that F is a *map*, i.e. a set satisfying the property $F = \{\langle p^{[1]}, p^{[2]} \rangle : p \in F\}$, and is also *single-valued* in the sense that $\{\langle p, q \rangle : p \in F, q \in F \mid p \neq q \ \& \ p^{[1]} = q^{[1]}\} = \emptyset$. The notation $F \upharpoonright X$, where F normally is a map—not a global function!—designates the *application* of F to X , defined as follows: $F \upharpoonright X =_{\text{Def}} \mathbf{arb}(\{p^{[2]} : p \in F \mid p^{[1]} = X\})$. Finally, the *inverse* F^\smile and the *domain* of a map F are defined to be the respective sets $\{\langle p^{[2]}, p^{[1]} \rangle : p \in F\}$ and $\{p^{[1]} : p \in F\}$.

<pre> THEORY globalizeTog (T) Svm(T) & T[←] = T & {p ∈ T p^[1] = p^[2]} = ∅ ⇒ (tog_∅) ⟨∀x ∈ domain(T) T x ≠ x & T (T x) = x⟩ ⟨∀x ∈ domain(T) tog_∅(x) = T x⟩ ⟨∀x tog_∅(x) ≠ x & tog_∅(tog_∅(x)) = x⟩ END globalizeTog </pre>

Fig. 3. A tool for globalizing a toggling map T to the entire universe of sets

4 The davisPutnam theory

After having developed the ability to construct a global Galois correspondence, a set of signed symbols, and an encoding of falsehood (to be passed on to our next theory via its formal parameters $\sim X$, lits and fail), we can now proceed to writing an “archetype” version of the Davis-Putnam algorithm for propositional formulae in conjunctive normal forms. We will define a recursive function dp_\emptyset that, given a set of clauses S , returns a model of S if S is satisfiable, otherwise returns a fictitious model containing the element fail . Let us note that the null set \emptyset will represent falsehood when viewed as a clause, whereas it will represent truth when viewed as a set of clauses.

Within the `davisPutnam` theory (see Appendix C), we will apply the `cnfModels` theory submitting as input parameter to it the same function $\sim X$, and renaming its output predicates as `Has_dpModel∅` and `Is_dpSat∅`, respectively.

A selection function $\text{sl}(S)$ will be supplied to `davisPutnam` as a fourth parameter: this is supposed to provide a literal appearing in S , unless S is a trivial formula (i.e., a blatantly satisfiable or unsatisfiable one). The simplest such function is

$$\text{sl}(S) \stackrel{\text{Def}}{=} \mathbf{arb}(\mathbf{arb}(S)).$$

A more efficient choice would be to select first literals within unit clauses, then pure literals, and finally arbitrary literals from the formula. Ouyang [8] analyzes the choice of branching rules, but such a discourse is beyond the scope of our paper.

The formulae which make sense as input for the satisfiability decision algorithm contain finitely many literals; accordingly, we define `clauSets∅` to be the set of all formulae with a finite number of literals, all belonging to `lits`:

$$\text{clauSets}_\emptyset \stackrel{\text{Def}}{=} \{s \subseteq \mathcal{P}\text{lits} \mid \text{Finite}(\bigcup s)\}.$$

Let us introduce the notation `settled∅` to represent, for any $s \in \text{clauSets}_\emptyset$, the set of all unit literals contained in s which are also pure literals:

$$\text{settled}(S)_\emptyset \stackrel{\text{Def}}{=} \{u : u \in S, h \in u \mid u = \{h\} \ \& \ \{h, \sim h\} \cap \bigcup(S \setminus \{u\}) = \emptyset\}.$$

Using the definition of $\text{Has_dpModel}_\theta$, we can deduce that $\bigcup \text{settled}_\theta(S)$ is included in any model of S ; therefore, in order to simplify S , it suffices to select literals from $S \setminus \text{settled}_\theta(S)$. Hence, we define the “picking” function:

$$\text{pk}_\theta =_{\text{def}} \{ [x, \text{sl}(x \setminus \text{settled}_\theta(x))] : x \in \text{clauSets}_\theta \}.$$

In sight of defining the Davis-Putnam procedure recursively, we must single out a well-founded relation over the family clauSets of all sets of clauses. (Incidentally, this approach will automatically ensure *termination* of the algorithm). A set of clauses is regarded as being “smaller” than another if its settled part is strictly included in the settled part of the other. The relation just defined is indeed a *well-founded* relation:

$$\begin{aligned} \text{THEOREM. } G \subseteq \text{clauSets}_\theta \ \& \ G \neq \emptyset \rightarrow \\ \left\langle \exists m \in G, \forall v \in G \mid \neg \left(\bigcup (v \setminus \text{settled}_\theta(v)) \subseteq \bigcup (m \setminus \text{settled}_\theta(m)) \right) \ \& \right. \\ \left. \bigcup (v \setminus \text{settled}_\theta(v)) \neq \bigcup (m \setminus \text{settled}_\theta(m)) \right\rangle. \end{aligned}$$

To be able to invoke the `wellfounded_recursive_fcn` theory already available in Ref (cf. [7]), we define an operator f3_dp_θ and a predicate P3_dp_θ , both ternary, which will play the role of formal input parameters. The third argument of these is meant to designate a “picking” function like the one just defined. As regards the operator f3_dp_θ , its first and second argument generally designate a set s of clauses and a doubleton collection of models, one modeling $s \cup \{\{p|s\}\}$ and one modeling $s \cup \{\{\neg(p|s)\}\}$; the pseudo-model $\{\text{fail}\}$ being used in place of either when a genuine model does not exist. When s is trivially satisfiable (e.g. null), as it coincides with its obvious part, then $\text{f3_dp}_\theta(s, -, -)$ supplies its model $\bigcup s$; when s is manifestly false, as flagged by the null clause present in it, $\text{f3_dp}_\theta(s, -, -)$ returns $\{\text{fail}\}$; when s is neither trivially satisfiable nor manifestly false, $\text{f3_dp}_\theta(s, m, p)$ draws a model of s (if any) from m , giving priority to the model of $s \cup \{\{p|s\}\}$ if this exist but picking the model of $s \cup \{\{\neg(p|s)\}\}$ when the former does not exist (of course if neither has a model, $m = \{\{\text{fail}\}\}$ will hold).

$$\begin{aligned} \text{f3_dp}_\theta(S, \mathcal{M}, P) =_{\text{def}} \text{if } \emptyset \in S \text{ then } \{\text{fail}\} \text{ else} \\ \text{if } \text{settled}_\theta(S) \supseteq S \text{ then } \bigcup S \text{ else} \\ \text{arb}(\{w \in \mathcal{M} \mid (\text{fail} \notin \bigcup \mathcal{M} \rightarrow P|S \in w) \ \& \ (\text{fail} \in w \rightarrow \langle \forall v \in \mathcal{M} \mid \text{fail} \in v \rangle)\}) \\ \text{fi} \\ \text{fi} \end{aligned}$$

As regards the predicate P3_dp_θ , its first argument again designates a set s of clauses, and its second designates the set s' resulting from the simplification of s when either the literal h or its complement $\neg h$ is assumed to be true, where h is selected from s by the picking function. It would be pointless to carry out the simplification relative to $\neg h$ when h is a pure literal, namely one whose complement does not occur in s : in this case the above-discussed operator f3_dp_θ will receive a singleton, instead of a doubleton, set of models. There are situations

in which the picking function is not guaranteed to return anything significant, but they cause no problem because in such cases the satisfiability check for s is obvious and can be performed directly by the operator f3_dp_θ without reduction of s to a simpler s' .

$$\begin{aligned} \text{P3_dp}_\theta(S, S', P) &\leftrightarrow_{\text{Def}} (\bigvee(P \upharpoonright S) \in \bigcup S \ \& \\ &S' = \{ \{ \bigvee(P \upharpoonright S) \} \} \cup \{ c \setminus \{ P \upharpoonright S \} : c \in S \mid \bigvee(P \upharpoonright S) \notin c \}) \vee \\ &S' = \{ \{ P \upharpoonright S \} \} \cup \{ c \setminus \{ \bigvee(P \upharpoonright S) \} : c \in S \mid P \upharpoonright S \notin c \} \end{aligned}$$

Let us note that in the case when $P \upharpoonright S$ is a pure literal, we have $\{ \{ P \upharpoonright S \} \} \cup \{ c \setminus \{ \bigvee(P \upharpoonright S) \} : c \in S \mid P \upharpoonright S \notin c \} = \{ \{ P \upharpoonright S \} \} \cup \{ c \in S \mid P \upharpoonright S \notin c \}$.
By applying the THEORY `well_recursive_fcn`, we obtain:

THEOREM. $\langle \forall x, p \mid x \in \text{clauSets}_\theta \rightarrow \text{dp0}_\theta(x, p) = \text{f3_dp}_\theta(x, \{ \text{dp0}_\theta(y, p) : y \in \text{clauSets}_\theta \mid (\bigcup(y \setminus \text{settled}_\theta(y)) \subseteq \bigcup(x \setminus \text{settled}_\theta(x)) \ \& \bigcup(y \setminus \text{settled}_\theta(y)) \neq \bigcup(x \setminus \text{settled}_\theta(x)) \ \& \text{P3_dp}_\theta(x, y, p) \}, p) \rangle$.

The following theorem states that the set resulting from the splitting rule is a finite set of literals and that it is smaller with respect to the well-founded relation introduced above.

THEOREM. $X \in \text{clauSets}_\theta \ \& \ H \in \bigcup(X \setminus \text{settled}_\theta(X)) \ \& \ Y = \{ \{ H \} \} \cup \{ c \setminus \{ \bigvee H \} : c \in X \mid H \notin c \} \rightarrow Y \in \text{clauSets}_\theta \ \& \ \bigcup(Y \setminus \text{settled}_\theta(Y)) \neq \bigcup(X \setminus \text{settled}_\theta(X)) \ \& \ \bigcup(Y \setminus \text{settled}_\theta(Y)) \subseteq \bigcup(X \setminus \text{settled}_\theta(X))$.

We are now ready to state the key theorem of our correctness verification:

THEOREM. $S \in \text{clauSets}_\theta \rightarrow \text{if fail} \in \text{dp0}_\theta(S, \text{pk}_\theta) \ \text{then } \neg \text{Is_dpSat}_\theta(S) \ \text{else Has_dpModel}_\theta(S, \text{dp0}_\theta(S, \text{pk}_\theta)) \ \text{fi}$.

This assertion gets proved by mathematical induction on the number $n = \# \bigcup(s \setminus \text{settled}_\theta(s))$. If $n = \emptyset$, then $s \setminus \text{settled}_\theta(s) = \emptyset$ or $s \setminus \text{settled}_\theta(s) = \{ \emptyset \}$. If the former alternative holds, then the value returned by dp0_θ will be $\bigcup s$, and actually we know from `cnfModels` (recalling the definition of `settledθ`) that this is a model of s . If the latter alternative holds, we have also $\emptyset \in s$ and thus dp0_θ will return the answer `fail` which is again correct by the definition of a model.

If $\emptyset \in n$, then pk_θ will pick a literal h from one of the clauses in $s \setminus \text{settled}_\theta(s)$. If $\bigvee(\text{pk}_\theta \upharpoonright s) \notin \bigcup s$ then the only set that satisfies the predicate P3_dp_θ is $s_1 = \{ \{ \text{pk}_\theta \upharpoonright s \} \} \cup \{ c \in s \mid (\text{pk}_\theta \upharpoonright s) \notin c \}$. As it has one fewer literal than s , we can apply the induction hypothesis and deduce that the resulting set m_1 is a model for s_1 .

Otherwise, both $s_1 = \{ \{ \bigvee(\text{pk}_\theta \upharpoonright s) \} \} \cup \{ c \setminus \{ \text{pk}_\theta \upharpoonright s \} : c \in s \mid \bigvee(\text{pk}_\theta \upharpoonright s) \notin c \}$ and $s_2 = \{ \{ \text{pk}_\theta \upharpoonright s \} \} \cup \{ c \setminus \{ \bigvee(\text{pk}_\theta \upharpoonright s) \} : c \in s \mid (\text{pk}_\theta \upharpoonright s) \notin c \}$ satisfy P3_dp_θ . As they have two fewer literals than s , by the induction hypothesis we deduce that

the models m_1 and m_2 returned by $dp0_\theta(s_1)$ and $dp0_\theta(s_2)$ are indeed models of s_1 and s_2 , respectively.

For all the cases above, using the theorems on the pure literal rule, unit literal rule and the splitting rule in `cnfModels`, we can prove the statement for s , and show that the induction holds, which completes our proof.

To refine our results, we define the function dp_θ that applies $dp0_\theta$ to the input formula deprived of all tautological clauses, and then enriches the model (if any) thus obtained, so that it becomes also a model of the initial formula.

DEF. $modelTaut_\theta(S, M) =_{\text{def}} \text{if } fail \in M \text{ then } \{fail\} \text{ else}$

$M \cup \{arb(\{h, \neg h\}) : c \in S, h \in c \setminus M \mid \neg h \in c \setminus M\};$

DEF. $dp_\theta(S) =_{\text{def}} modelTaut_\theta(S, dp0_\theta(S \setminus \{c : c \in S, h \in c \mid \neg h \in c\}, pk_\theta));$

THEOREM. $S \in clauSets_\theta \rightarrow (Is_dpSat_\theta(S) \leftrightarrow fail \notin dp_\theta(S));$

THEOREM. $S \in clauSets_\theta \rightarrow (Is_dpSat_\theta(S) \leftrightarrow Has_dpModel_\theta(S, dp_\theta(S))).$

5 Proof of the Compactness Theorem

It is remarkable that a single proof-development environment can provide support for the specification of a terminating algorithm (the Davis-Putnam procedure in our case-study), for the proof of its correctness, and also for the proof of such a general fact as the compactness theorem. For propositional logic, this fact can be stated by first introducing the notion of FINITE SATISFIABILITY, which refers to a set Ψ of sets S of finite clauses: in order that Ψ be finitely satisfiable, for every finite subset F of Ψ there must exist a model M which simultaneously satisfies all S in F . The compactness theorem states that any Ψ which is finitely satisfiable in this sense admits a model M which simultaneously satisfies all S in Ψ . A short, well-known proof of this fact can be derived from the Zorn lemma.

In rough outline, the proof goes as follows: let us consider the set Σ of all finitely satisfiable supersets Φ of Ψ such that

$$\{x : s \in \Phi, c \in s, y \in c, x \in \{y, \neg y\}\} = \{x : s \in \Psi, c \in s, y \in c, x \in \{y, \neg y\}\}.$$

It can be shown that every subset of Σ which is totally ordered by \subseteq has an upper bound (relative to \subseteq) in Σ ; therefore, by the Zorn lemma, Σ has at least one \subseteq -maximal element Φ_0 . It can be shown that for every finitely satisfiable set Φ and for each $X \in \{x : s \in \Phi, c \in s, y \in c, x \in \{y, \neg y\}\}$, either $\Phi \cup \{\{X\}\}$ or $\Phi \cup \{\{\neg X\}\}$ is finitely satisfiable: which means, when Φ is maximal (e.g. when $\Phi = \Phi_0$), that either $\{\{X\}\}$ or $\{\{\neg X\}\}$ (and only one of the two, else the finite satisfiability would be contradicted) belongs to it. Hence we can define a model M_0 by collecting all those X for which $\{\{X\}\} \in \Phi_0$. Consider now an $S \in \Phi_0$ and a clause $c \in S$. Assuming by contradiction that c does not intersect M_0 , we would have $\{\{\neg x\}\} \in \Phi_0$ for every $x \in c$, but then $\{S\} \cup \{\{\{\neg x\}\} : x \in c\}$ would not be satisfiable, contradicting the finite satisfiability of Φ_0 .

Carrying this argument out in Ref requires two proofs (one concerning the extensibility of finitely satisfiable sets, and one culminating in the compactness

theorem). We found that the natural placement for such proofs is within the `cnf-Models` theory discussed two sections ago. Approximately 220 lines of proofware were needed to formalize these proofs; their verification takes about 2 seconds.

Conclusions and Further Work

In this paper we have described how to prove the correctness of a well-known algorithm in the `Referee` theorem checker. The three theories introduced constitute the actual proof, showing that `Ref` can be used to tackle algorithm verification problems. Many algorithms can be specified, very naturally and in compact, high-level terms, by means of an executable language grounded on set theory (cf. [10]). Hence it would be desirable to enhance `Ref` with programming-specific notation, so that proving that a procedure behaves as desired could be done, in full, under the surveillance of the automated verifier.

This paper has also addressed, for the first time, the issue of how to “arithmetize” the syntax and the symbolic manipulations of a formal language; i.e., how to encode by means of set-theoretic constructions the formation rules, designation rules, deduction and/or rewriting rules of a formal system. The first step having now been done, we are confident that many situations will arise which can be tackled similarly. Indeed, thanks to the set-theoretic counterpart provided by the `THEORY signedSymbols` for the basic symbols of a language, \in -recursion can mimic structural recursion over the terms of its signature.

To give a quick example, let us denote by \mathcal{V}_ω the collection of all the hereditarily finite sets (namely, those sets whose rank is a finite ordinal). We can represent the algorithm which reduces an arbitrary formula of propositional logic to *negative-normal form* as follows:

$$\begin{aligned} \text{sgn}(Op, P) &=_{\text{Def}} \text{if } Op = 0 \text{ then } P \text{ elseif } P \in \text{lits} \text{ then } \neg P \\ &\quad \text{else } \{1 \setminus (\{Op\} \cap 1)\} \cup \{ \text{sgn}(1, y) : y \in P \setminus \mathcal{V}_\omega \} \text{ fi}; \\ \text{nnf}(P) &=_{\text{Def}} \text{if } P \in \text{lits} \text{ then } P \\ &\quad \text{else } \text{sgn}(\text{arb}(P \cap \mathcal{V}_\omega), \{ \text{nnf}(y) : y \in P \setminus \mathcal{V}_\omega \}) \text{ fi.} \end{aligned}$$

(Clue: $P \cap \mathcal{V}_\omega$ and $P \setminus \mathcal{V}_\omega$ represent opcode and argument of any set P , when P is viewed as encoding a syntactic object).

And here is how to evaluate a sentence of a propositional mono-modal logic with respect to a Kripke frame consisting of a set W of worlds and an accessibility relation R between worlds, and to a model M assigning a set of worlds to each propositional letter:

$$\begin{aligned} \text{m_vl}(Op, U, W, R) &=_{\text{Def}} \text{if } Op = 0 \text{ then } U \text{ elseif } Op = 1 \text{ then } W \setminus U \\ &\quad \text{else } \{x \in W \mid U \supseteq R \uparrow \{x\}\} \text{ fi}; \\ \text{m_eval}(E, W, R, M) &=_{\text{Def}} \text{if } E \in \text{domain}(M) \text{ then } M \upharpoonright E \text{ else} \\ &\quad \text{m_vl}(\text{arb}(E \cap \mathcal{V}_\omega), \bigcup \{ \text{m_eval}(y, W, R, M) : y \in E \setminus \mathcal{V}_\omega \}, W, R, M) \text{ fi.} \end{aligned}$$

Acknowledgements

The authors would like to thank Jacob T. Schwartz (New York University) for his assistance with the development of the Referee system.

We had pleasant and fruitful discussions with Giovanna D’Agostino (Univ. of Udine) and with Marianna Nicolosi Asmundo (Univ. of Catania), who contributed to the “arithmetization” (set-theoretic rendering in our case) of semantic notions referring to propositional (even non-classical) logics and to the proof of compactness.

Research partially funded by the INTAS project *Algebraic and deduction methods in non-classical logic and their applications to Computer Science*, and by the Italian MUR/PRIN project *Large-scale development of certified mathematical proofs*.

References

1. P. Letouzey and L. Thery, Formalizing Stalmarck’s algorithm in Coq, TPHOLs, 2000.
2. T. Nishihara, Y. Minamide Depth First Search, The Archive of Formal Proofs, 2004.
3. M. Davis, H. Putnam A Computing Procedure for Quantification Theory, Journal of the ACM, 7(3):201–215, 1960
4. M. Davis, G. Logemann, D. Loveland. A machine program for theorem-proving, Communications of the ACM, 5(7):394–397, 1962.
5. D. Cantone, E.G. Omodeo, J.T. Schwartz, P. Ursino. Notes from the logbook of a proof-checker’s project. In N. Dershowitz ed., *International symposium on verification (Theory and Practice)* celebrating Zohar Manna’s 1000000₂-th birthday. Springer-Verlag, LNCS 2772, pp. 182–207, 2003.
6. E.G. Omodeo, D. Cantone, A. Policriti, J.T. Schwartz. A Computerized Referee. In O. Stock and M. Schaerf (Eds.): *Reasoning, Action and Interaction in AI Theories and Systems*, Essays Dedicated to Luigia Carlucci Aiello, LNAI 4155, pp. 114–136, 2006.
7. E.G. Omodeo, J.T. Schwartz. A ‘Theory’ mechanism for a proof-verifier based on first-order set theory. In A. Kakas and F. Sadri (eds.), *Computational Logic: Logic Programming and beyond*, Essays in honour of Robert Kowalski, part II, Springer-Verlag, LNAI 2408, pp. 214–230, 2002.
8. M. Ouyang. How good are branching rules in DPLL?, Discrete Applied Mathematics, 89(1-3):281–286, 1998.
9. D. Prawitz, H. Prawitz, N. Voghera. A Mechanical Proof Procedure and its Realization in an Electronic Computer, Journal of the ACM, 7(2)102–128, 1960, reprinted (with a commentary) in Automation of Reasoning 1, Classical Papers on Computational Logic, pp. 202–28, J. Siekmann and G. Wrightson (eds), Springer-Verlag, 1983.
10. J. T. Schwartz, R. K. B. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1986.
11. http://www.settheory.com/Set12/Ref_user_manual.html

Appendix

A The signedSymbols THEORY

THEORY signedSymbols(atms)

\Rightarrow $\langle \text{aff}_\theta, \text{neg}_\theta, \text{lits}_\theta, \text{false}_\theta \rangle$
 $\langle \forall x, y \mid x \neq y \rightarrow \text{aff}_\theta(x) \neq \text{aff}_\theta(y) \rangle$
 $\langle \forall x \mid \text{neg}_\theta(\text{neg}_\theta(x)) = x \ \& \ \text{neg}_\theta(x) \neq x \rangle$
 $\langle \forall x, y \mid \text{aff}_\theta(x) \neq \text{neg}_\theta(\text{aff}_\theta(y)) \rangle$
 $\{ \text{aff}_\theta(x) : x \in \text{atms} \} \subseteq \text{lits}_\theta$
 $\{ \text{neg}_\theta(x) : x \in \text{lits}_\theta \} = \text{lits}_\theta$
 $\text{false}_\theta \notin \text{lits}_\theta$

END signedSymbols

B The cnfModels THEORY

THEORY cnfModels(\neg x)

$\langle \forall x \mid \neg\neg x = x \ \& \ \neg x \neq x \rangle$
 \Rightarrow $\langle \text{Has_cnfModel}_\theta, \text{Is_cnfSat}_\theta \rangle$
 $\langle \forall s \mid \text{Has_cnfModel}_\theta(\emptyset, \emptyset) \ \& \ \text{Is_cnfSat}_\theta(\emptyset) \ \& \ (\emptyset \in s \rightarrow \neg \text{Is_cnfSat}_\theta(s)) \rangle$
 $\langle \forall s \mid \{ h \in \bigcup s \mid \neg h \in \bigcup s \} = \emptyset \ \& \ \emptyset \notin s \rightarrow \text{Has_cnfModel}_\theta(s, \bigcup s) \ \& \ \text{Is_cnfSat}_\theta(s) \rangle$
 $\langle \forall s \mid \{ u : u \in s, h \in u \mid u = \{h\} \ \& \ \{h, \neg h\} \cap \bigcup (s \setminus \{u\}) = \emptyset \} \supseteq s \rightarrow$
 $\text{Has_cnfModel}_\theta(s, \bigcup s) \ \& \ \text{Is_cnfSat}_\theta(s) \rangle$
 $\langle \forall t, s, m \mid t \subseteq s \ \& \ \text{Has_cnfModel}_\theta(s, m) \rightarrow \text{Has_cnfModel}_\theta(t, m) \rangle$
 $\langle \forall t, s \mid t \subseteq s \ \& \ \text{Is_cnfSat}_\theta(s) \rightarrow \text{Is_cnfSat}_\theta(t) \rangle$
 $\langle \forall s, m \mid \text{Has_cnfModel}_\theta(s, m) \rightarrow \text{Has_cnfModel}_\theta(s, m \cap \bigcup s) \rangle$
 $\langle \forall s, m, t \mid \text{Has_cnfModel}_\theta(s, m) \rightarrow$
 $\text{Has_cnfModel}_\theta(\emptyset, m \cup \{ \text{arb}(\{k, \neg k\}) : c \in t, k \in c \setminus m \mid \neg k \in c \setminus m \}) \rangle$
 $\langle \forall s, m, t \mid \text{Has_cnfModel}_\theta(s, m) \rightarrow \text{Has_cnfModel}_\theta(s \cup \{ c : c \in t, h \in c \mid \neg h \in c \},$
 $m \cup \{ \text{arb}(\{h, \neg h\}) : c \in t, h \in c \setminus m \mid \neg h \in c \setminus m \}) \rangle$
 $\langle \forall s, m, h \mid \text{Has_cnfModel}_\theta(s, m) \ \& \ \neg \text{Has_cnfModel}_\theta(\{ \{h\} \} \cup \{ c \setminus \{ \neg h \} : c \in s \mid h \notin c \},$
 $m \cup \{ h \}) \rightarrow \neg h \in m \rangle$
 $\langle \forall s, m, h \mid \text{Has_cnfModel}_\theta(s, m) \rightarrow \text{Has_cnfModel}_\theta(\{ \{h\} \} \cup \{ c \setminus \{ \neg h \} : c \in s \mid h \notin c \},$
 $m \cup \{ h \}) \vee \text{Has_cnfModel}_\theta(\{ \{ \neg h \} \} \cup \{ c \setminus \{ h \} : c \in s \mid \neg h \notin c \}, m \cup \{ \neg h \}) \rangle$
 $\langle \forall h, s, m \mid \text{Has_cnfModel}_\theta(\{ \{h\} \} \cup \{ c \setminus \{ \neg h \} : c \in s \mid h \notin c \}, m) \rightarrow \text{Has_cnfModel}_\theta(s, m) \rangle$
 $\langle \forall h, s \mid \text{Is_cnfSat}_\theta(\{ \{h\} \} \cup \{ c \setminus \{ \neg h \} : c \in s \mid h \notin c \}) \rightarrow \text{Is_cnfSat}_\theta(s) \rangle$
 $\langle \forall s, h \mid \text{Is_cnfSat}_\theta(s) \leftrightarrow \text{Is_cnfSat}_\theta(\{ \{h\} \} \cup \{ c \setminus \{ \neg h \} : c \in s \mid h \notin c \}) \vee$
 $\text{Is_cnfSat}_\theta(\{ \{ \neg h \} \} \cup \{ c \setminus \{ h \} : c \in s \mid \neg h \notin c \}) \rangle$
 $\langle \forall h, s \mid \neg h \notin \bigcup s \rightarrow (\text{Is_cnfSat}_\theta(s) \leftrightarrow \text{Is_cnfSat}_\theta(\{ \{h\} \} \cup \{ c \in s \mid h \notin c \})) \rangle$
 $\langle \forall h, s, m \mid \{ h \} \in s \ \& \ \text{Has_cnfModel}_\theta(s, m) \rightarrow h \in m \ \& \ \neg h \notin m \rangle$
 $\langle \forall \Psi \mid \text{Is_cnfFinSat}_\theta(\Psi) \leftrightarrow \langle \forall f \subseteq \Psi \mid \text{Finite}(f) \rightarrow \langle \exists m, \forall s \in f \mid \text{Has_cnfModel}_\theta(s, m) \rangle \rangle \rangle$
 $\langle \forall \Psi \mid \text{Is_cnfFinSat}_\theta(\Psi) \ \& \ \langle \forall s \in \Psi, c \in s \mid \text{Finite}(c) \rangle \rightarrow$
 $\langle \exists m, \forall s \in \Psi \mid \text{Has_cnfModel}_\theta(s, m) \rangle \rangle$

END cnfModels

C The davisPutnam THEORY

THEORY davisPutnam ($\neg x$, lits, fail, sl(x))

$\langle \forall x \mid \neg\neg x = x \ \& \ \neg x \neq x \rangle$

$\langle \neg x : x \in \text{lits} \rangle = \text{lits}$

fail \notin lits

$\langle \forall s \mid s \neq \emptyset \ \& \ \emptyset \notin s \rightarrow \text{sl}(s) \in \text{Us} \rangle$

$\Rightarrow (\text{clauSets}_\theta, \text{Has_dpModel}_\theta, \text{Is_dpSat}_\theta, \text{dpModel}_\theta, \text{settled}_\theta, \text{pk}_\theta, \text{f3_dp}_\theta, \text{P3_dp}_\theta, \text{dp0}_\theta, \text{modelTaut}_\theta, \text{dp}_\theta)$

$\langle \forall h, s \mid \neg h \notin \text{Us} \rightarrow \{c \in s \mid h \notin c\} = \{c \setminus \{\neg h\} : c \in s \mid h \notin c\} \rangle$

$\langle \forall s \mid s \in \text{clauSets}_\theta \rightarrow (\text{Finite}(s) \ \& \ \text{Finite}(\text{Us}) \ \& \ \text{Us} \subseteq \text{lits} \ \& \ \text{fail} \notin \text{Us} \ \& \ \neg \text{fail} \notin \text{Us}) \rangle$

$\langle \forall s, c \mid (s \in \text{clauSets}_\theta \ \& \ c \in s) \rightarrow \text{Finite}(c) \rangle$

$\langle \forall s, t \mid (s \in \text{clauSets}_\theta \ \& \ (t \subseteq s \vee \text{Ut} \subseteq \text{Us})) \rightarrow t \in \text{clauSets}_\theta \rangle$

$\langle \forall s, m \mid \text{Has_dpModel}_\theta(s, m) \leftrightarrow \{h \in m \mid \neg h \in m\} = \emptyset \ \& \ \{c \in s \mid m \cap c = \emptyset\} = \emptyset \rangle$

$\langle \forall s \mid \text{Is_dpSat}_\theta(s) \leftrightarrow \langle \exists m \mid \text{Has_dpModel}_\theta(s, m) \rangle \rangle$

$\langle \forall g \mid (g \subseteq \text{clauSets}_\theta \ \& \ g \neq \emptyset) \rightarrow (\exists m \in g \mid \forall v \in g \mid \neg(\text{U}(v \setminus \text{settled}_\theta(v)) \subseteq \text{U}(m \setminus \text{settled}_\theta(m))) \ \& \ \text{U}(v \setminus \text{settled}_\theta(v)) \neq \text{U}(m \setminus \text{settled}_\theta(m))) \rangle$

$\langle \forall x, p \mid x \in \text{clauSets}_\theta \rightarrow \text{dp0}_\theta(x, p) = \text{f3_dp}_\theta \left(x, \{\text{dp0}_\theta(y, p) : y \in \text{clauSets}_\theta \mid$

$(\text{U}(y \setminus \text{settled}_\theta(y)) \subseteq \text{U}(x \setminus \text{settled}_\theta(x)) \ \& \ \text{U}(y \setminus \text{settled}_\theta(y)) \neq \text{U}(x \setminus \text{settled}_\theta(x))) \ \&$

$\text{P3_dp}_\theta(x, y, p)\} , p \right)$

$\langle \forall s \mid s \in \text{clauSets}_\theta \rightarrow \#\text{U}(s \setminus \text{settled}_\theta(s)) \in \text{Za} \rangle$

$\langle \forall x, h, y \mid x \in \text{clauSets}_\theta \ \& \ h \in \text{U}(x \setminus \text{settled}_\theta(x)) \ \& \ y = \{\{h\}\} \cup \{c \setminus \{\neg h\} : c \in x \mid h \notin c\} \rightarrow$

$y \in \text{clauSets}_\theta \ \& \ \text{U}(y \setminus \text{settled}_\theta(y)) \neq \text{U}(x \setminus \text{settled}_\theta(x)) \ \&$

$\text{U}(y \setminus \text{settled}_\theta(y)) \subseteq \text{U}(x \setminus \text{settled}_\theta(x)) \rangle$

$\langle \forall s \mid s \in \text{clauSets}_\theta \ \& \ \emptyset \notin s \ \& \ s \not\subseteq \text{settled}_\theta(s) \rightarrow \text{pk}_\theta \upharpoonright s \in \text{U}(s \setminus \text{settled}_\theta(s)) \ \& \ \text{pk}_\theta \upharpoonright s \neq \text{fail} \ \&$

$\neg(\text{pk}_\theta \upharpoonright s) \neq \text{fail} \ \& \ (\neg(\text{pk}_\theta \upharpoonright s) \in \text{Us} \rightarrow \neg(\text{pk}_\theta \upharpoonright s) \in \text{U}(s \setminus \text{settled}_\theta(s))) \rangle$

$\langle \forall s \mid s \in \text{clauSets}_\theta \ \& \ \emptyset \in s \vee s \subseteq \text{settled}_\theta(s) \vee \#\text{U}(s \setminus \text{settled}_\theta(s)) = \emptyset \rightarrow$

if fail $\in \text{dp0}_\theta(s, \text{pk}_\theta)$ **then** $\neg \text{Is_dpSat}_\theta(s)$ **else** $\text{Has_dpModel}_\theta(s, \text{dp0}_\theta(s, \text{pk}_\theta))$ **fi** \rangle

$\langle \forall s \mid s \in \text{clauSets}_\theta \rightarrow$

if fail $\in \text{dp0}_\theta(s, \text{pk}_\theta)$ **then** $\neg \text{Is_dpSat}_\theta(s)$ **else** $\text{Has_dpModel}_\theta(s, \text{dp0}_\theta(s, \text{pk}_\theta))$ **fi** \rangle

$\langle \forall s \mid s \in \text{clauSets}_\theta \rightarrow (\text{Is_dpSat}_\theta(s) \leftrightarrow \text{fail} \notin \text{dp}_\theta(s)) \rangle$

$\langle \forall s \mid s \in \text{clauSets}_\theta \rightarrow (\text{Is_dpSat}_\theta(s) \leftrightarrow \text{Has_dpModel}_\theta(s, \text{dp}_\theta(s))) \rangle$

END davisPutnam

A Dialogue Games Framework for the Operational Semantics of Logic Languages

Stefania Costantini Arianna Tocchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost,tocchio}@di.univaq.it

Abstract. We propose an approach to the operational semantics of agent-oriented logic programming languages based on the theory of dialogue games, that has its origin in the philosophy of argumentation. We define a formal dialogue game framework that focuses on rules of dialogue that the players of the game, i.e., the components of the language interpreter, can be assumed to adopt in interacting with each other. The operation of a component is thus modeled by means of the responses to utterances coming either from other components or from the outside. The approach leads to a natural, uniform and modular way of modeling all the components of the interpreter, including the communication component and the communication protocol. The interpreter behavior can be abstracted from the operational rules so as to define and prove useful properties.

1 Introduction

Logic Programming languages have been since the very beginning characterized by a rigorous definition of their declarative and procedural semantics [22] [1], formally linked to the operational semantics [12], [5]. As agents and multi-agent systems have become an emerging paradigm for programming distributed applications, several agent-oriented languages, formalisms and frameworks based on logic programming and computational logic have been proposed, among which [2], [18], [9], [29], [21], [10]. All of them have been developed with attention to the formal semantics, and to the compliance of the implementation w.r.t. the semantics.

Nevertheless, D'Inverno and Luck in [15] observe that *implementations have typically involved simplifying assumptions that have resulted in the loss of a strong theoretical foundation for them, while logics have had small relation to practical problems. Though this fragmentation into theoretical and practical aspects has been noted, and several efforts made in attempting to address this fragmentation in related areas of agent-oriented systems (for example, [14], [17], [23], [36]), there remains much to be done in bringing together the two strands of work.* An effort in this direction has been made via the formalization of operational semantics in an agent language. This has resulted for instance in the definition of AgentSpeak(L), which can be viewed as an abstraction of the implemented Procedural Reasoning System (PRS) [16] and the Distributed Multi-Agent Reasoning System (dMARS) [13], and which allows agent programs to be defined and interpreted [29].

The effort of providing an abstract formalization of agent behavior is aimed at the application of formal methods in a rigorous definition and analysis of agents functionalities. This may allow one to demonstrate interesting properties related to system correctness. Bracciali et al. in [8] synthesize the need of being able to prove agent behavior correctness as follows: *the ever-growing use of agents and multi-agent systems in practical applications poses the problem of formally verifying their properties; the idea being that by verifying properties of the overall system we can make informed judgments about the suitability of agents and multi-agent systems in solving problems posed within application domains.*

We can find a similar consideration in [6], where the authors explain the motivations that have persuaded them to describe the AgentSpeak(L) behavior with Model-Checking techniques: *...tools should be usable by a general computing audience, there should also be strong theoretical foundations for such tools, so that formal methods can be used in the design and implementation processes. In particular, the verification of multi-agent systems showing that a system is correct with respect to its stated requirements is an increasingly important issue, especially as agent.*

Formal properties of agent systems should refer in general to multi-agent contexts, where the correctness of agent interactions assumes a relevant role. In [4] and [3], some interesting properties are demonstrated via AUML. Wooldridge and Lomuscio in [37] present the VSK logic, a family of multi-modal logics for reasoning about the information properties of computational agents situated in some environment. Viroli and Omicini in [35] study the impact of thinking about agents and multi-agent systems in terms of their observable behavior. Bracciali et al. in [8] define the semantics of a multi-agent system via a definition of stability on the set of all actions performed by all agents in the system and specify properties of individual success of agents, overall success of a multi-agent system, robustness and world-dependence of a multi-agent system, as well as a number of properties of agents within systems. Finally, we mention the work of Bordini and Moreira reported in [7]: they exploit an operational semantics in order to investigate properties of AgentSpeak(L) agents.

McBurney and Parsons, in [25] [26] [28] [27], study argumentation-based dialogues between agents. They propose dialogue games as a formal framework for dialogues between autonomous agents, The authors discuss how to prove some properties of dialogues under a given protocol, in particular termination, dialogue outcomes, and complexity. Dialogue games are formal interactions between two or more participants, in which participants “move” by uttering statements according to pre-defined rules.

In this paper, we propose dialogue games as a general operational semantics approach for agent-oriented logic programming languages. We have applied this approach to the DALI language [10] [11]. In particular, in [34] the behavior of the DALI interpreter has been fully defined as a dialogue games whose players are the various modules composing the interpreter itself. I.e., the players are the component of the interpreter, considered as black boxes, that play an innocent game one towards the other.

A first advantage of the approach is that of being able to describe in a uniform way all aspects of an agent-oriented language, including communication. A second advantage is modularity: as a language interpreter is seen as composed of modules which are the players of a game, these modules can be composed, added, removed, replaced in

flexible ways provided that they respect the “rules” of the game. Another advantage is that one can take profit of game theory for proving properties of the interpreter, among which various properties of correctness. In [34], correctness of the DALI interpreter w.r.t. the declarative and procedural definition of DALI extended resolution is formally proved.

This approach is elaboration-tolerant w.r.t. the communication protocol, that can be specified via a separate set of transition rules. To demonstrate this point, we define compliance w.r.t. a protocol and demonstrate compliance for the simple case of the four-acts protocol introduced by Sadri et al. [31].

This research has been initiated in the context of the Ph.D. work of one of the authors, reported in [34]. Since then, the formal setting has been improved, and the effort reported in this paper has been that of generalizing the approach to logic languages in general. This has required to generalize and extend previous definitions.

The rest of the paper is organized as follows. In Section 2 we introduce dialogue games. In Section 3 we propose formal dialogue games as a tool for defining the operational semantics of agent-oriented logic programming languages, and introduce to this aim a specific dialogue game framework. In Section 4 we illustrate some examples of laws and rules of the proposed dialogue games, taken from the DALI language description but general enough to be quite directly applicable to other languages/formalisms. In Section 5 we show how we can abstract aspects of the operational behavior from given laws and rules, so as to define and prove useful properties. Finally, we conclude in Section 6.

2 Dialogue Games as a formal framework for dialogues between autonomous agents

Formal dialogue games have been studied in philosophy since at least the time of Aristotle. For a review of their history and applications see [25]. Recently, they have been applied in various contexts in computer science and artificial intelligence, particularly as the basis for interaction between autonomous software agents. Dialogue game protocols have been proposed for agent team formation, persuasion, negotiation over scarce resources, consumer purchase interactions and joint deliberation over a course of action in some situation ([20],[24],[32],[33]).

In particular, formal dialogue games are interactions between two or more players, where each player “moves” by making utterances, according to a defined set of rules.

In this section, we present a model of a generic formal dialogue game as reported in [25]. We assume that the topics of discussion between the players can be represented in some logical language. A dialogue game specification then consists of the following elements:

- **Commencement Rules:** Rules which define the circumstances under which a dialogue commences.
- **Locutions:** Rules which indicate what utterances are permitted.
- **Combination Rules:** Rules which define the dialogical contexts under which particular locutions are permitted or not, or obligatory or not.

- **Commitments:** Rules which define the circumstances under which participants express commitment to a proposition.
- **Termination Rules:** Rules that define the circumstances under which a dialogue ends.

For locutions, we will basically adopt (and illustrate in the following sections) the form proposed in [27]. The definition of a dialogue game is completed by the specification of *transition rules* that define how the state of the system (composed of the set of players) changes as a result of a certain locution having been uttered (or, also, as a result of a certain locution having *not* being uttered). Transition rules provide a formal linkage between locutions and their possible usages and effect, and make automated dialogues possible.

3 Operational Semantics as a Dialogue Game

We propose formal dialogue games as a tool for defining the operational semantics of agent-oriented logic programming languages. The motivation of this proposal relies in the nature of this languages, that reflects into the structure of the interpreter. Despite the differences, all agent-oriented languages have (at least) the following basic features:

- A logical “core” that for instance in both KGP and DALI is resolution-based.
- Reactivity, i.e., the capability of managing external stimula.
- Proactivity, i.e., the capability of managing internal “initiatives”.
- The capability of performing actions.
- The capability of recording what has happened and has been done in the past.
- The capability of managing communication with other agents. This can be seen as a composition of sub-capabilities: managing both out-coming and incoming messages according to a given protocol, and possibly applying ontologies to understand message contents.
- A basic cycle that interleaves the application of formerly specified capabilities. E.g., in DALI the basic cycle is integrated with the logical core into an extended resolution, while in KGP the basic cycle has a meta-level definition and thus can be varied.

All these components can be seen as “players” that play together and make “moves”, so as to coordinate themselves to generate the overall interpreter behavior. This means, our players participate in an “innocent” game where their objective is not to win, but rather is that of respecting the rules of the game itself. Thus, we expect each player to faithfully follow the laws and rules and produces a set of admissible moves. These moves will influence the other players and will determine the global game.

Advantages of the approach are the following:

- Each component can be seen as a “black-box”. I.e., the corresponding locutions and transition rules can be defined independently from the rest. Then, capabilities can be easily added/removed, at the expense of modifying only the basic cycle description, which is however by its very nature quite modular.

- There is no different formalism for the communication capability. Then, one can give the full description of the language interpreter in one and the same formalism. This has the positive consequence of elaboration-tolerance with respect of changes in the communication component. Moreover, properties that are not just related to communications, but rather involve communication in combination to other capabilities can be more easily proved.

We have experimented the proposed approach for defining the *full* operational semantics of the interpreter of the DALI language [34]. DALI [10] [11] is a Horn-clause agent oriented language, with a declarative and procedural semantics that have been obtained as an extension of the standard one. The operational semantics that we have defined has allowed us to prove some relevant properties. In particular, we have proved correctness of the interpreter w.r.t. DALI extended resolution, and compliance w.r.t. some communication protocols.

Below we propose how to define the operational semantics as a dialogue game.

First of all, we have to define what we mean by the *state* of an agent.

Definition 1 (State of an agent). Let Ag_x be the name of the agent. We define the internal state IS_{Ag_x} as the tuple $\langle E, A, G \rangle$ composed by the current sets of events perceived, actions to be performed and goals to be achieved.

Then, we propose the definition of a law for the particular kind of dialogue game that we are defining.

Definition 2 (Law). We define a law L_x as a framework composed by the following elements:

- **Name:** the name of law.
- **Locution:** The utterance, that expresses the purpose of the application of the law.
- **Preconditions:** The preconditions for the application the law.
- **Meaning:** The informal meaning of the law.
- **Response:** The effects of the application of the law.

Consider that, being the game innocent, the other player (and, ultimately, the overall interpreter) are expected to behave as specified by a law whenever it is applied. In particular, a law is applied with respect to the current state of the interpreter, which is defined as follows.

Definition 3. A state of the interpreter is a pair $\langle Ag_x, S_{Ag_x} \rangle$ where Ag_x is the name of the agent and the operational state S_{Ag_x} is a triple $\langle P_{Ag_x}, IS_{Ag_x}, Mode_{Ag_x} \rangle$. The first element is the logic program defining the agent, the second one is the agent current internal state, the third one is a particular attribute, that we call modality, which describes what the interpreter is doing.

The application of laws is regulated by transition rules that specify how the state of the interpreter changes when the laws are applied.

Definition 4 (Transition rule). A transition rule has the following form:

$\langle Ag_x, \langle P, IS, Mode \rangle \rangle \xrightarrow{L_i, \dots, L_j} \langle Ag_x, \langle NewP, NewIS, NewMode \rangle \rangle$
 where L_i, \dots, L_j are the rules which are applied in the given state thus obtaining a new state where some elements have possibly changed. Namely, $NewP$, $NewIS$ and $NewMode$ indicate, respectively, P , IS and $Mode$ updated after applying the laws.

A transition rule can also describe how an agent can influence another one. In this case, we will have:

Definition 5 (Inter-Agent Transition rule). *A transition rule involving two agents has the following form:*

$$\langle Ag_x, \langle P_{Ag_x}, IS_{Ag_x}, Mode_{Ag_x} \rangle \rangle \xrightarrow{L_i, \dots, L_j} \langle Ag_y, \langle P_{Ag_y}, IS_{Ag_y}, Mode_{Ag_y} \rangle \rangle$$

where $x \neq y$

About the general features of the game, we may notice that:

- **Commencement Rules** here define the activation of an agent, and imply as preconditions the acquisition of a syntactically correct logic program and of the possible initialization parameters. The response consists in the creation of the initial state and in the activation of the basic interpreter cycle.
- **Combination Rules** define in which order the transition rules should be applied if more than one is applicable (i.e., the preconditions of the corresponding laws are verified). In our case, the only rule is that the component corresponding to the basic interpreter cycle must regain the control after a predefined quantum of time. The basic cycle will then utter locutions that activate the other components according to its policy.
- **Commitments** are taken for granted, i.e., the players (components of the interpreter) always accept to give the needed responses and thus make the expected effects of transitions actual.
- **Termination Rules** should define under which circumstances an agent stops its activity. They may be missing if the agent is supposed to stay alive forever (or, in practice, as long as possible).

4 Examples of Laws and Transition Rules

In this Section we show some laws and rules that are taken from the operational semantics of DALI, but that are general enough to be easily adapted to other languages and formalisms. It may be interesting to notice that the DALI interpreter is described by 90 laws and 141 transition rules (fully reported in [34]).

4.1 Message Reception Player

We now present one of the players that belong to the DALI interpreter. Its function is that of receiving the incoming messages, and thus we may say that it is fairly general rather than specific of DALI. The only specific feature is that a DALI logic program includes a set of meta-rules that define the distinguished predicate *told* in order to specify constraints on the incoming messages. A message consists in general of: a sender, a primitive, a content, the language in which it is expressed and the adopted communication protocol. In DALI, if the message elements satisfy the constraints specified in the *told* rules, then the message is accepted; otherwise, it is discarded. This is why this player is called the TOLD player.

Below are the laws and rules for the TOLD player. Laws and rules are reported with the numbers occurring in [34]. The first law connects the agent to the input message space. Each agent checks, from time to time, if in incoming message space (that, in the case of DALI, is the Linda tuple space) there a message for it. In this case, it takes the message and starts to inspect it. The first law describes this process and extracts from the whole message the parameters which are relevant for TOLD rules: the sender agent and the content of the message.

L12: The **L12 receive_message(.)** law:

Locution: *receive_message*($Ag_x, Ag_y, Protocol, Ontology, Language, Primitive$)

Preconditions: this law can be applied when the agent Ag_x finds in input message space a message with its name.

Meaning: the agent Ag_x receives a message from Ag_y (environment, other agents,...). For the sake of simplicity, we consider the environment as an agent.

Response: The player considers the information about the language and the ontology and extracts the name of sender agent and the primitive contained in the message.

Law L_{13} verifies protocol compatibility. If the sender agent protocol is different from that of the receiver agent, then the message is discarded “a priori”.

L13: The **L13 receive_message(.)** law:

Locution: *receive_message*($Ag_x, Ag_y, Protocol, Ontology, Language, Primitive$)

Preconditions: The Protocol is compatible with the one of the receiver agent.

Meaning: This law discards all messages that the agent could not understand correctly considered the different protocols adopted. If the protocols coincide, then the message goes on.

Response: The message enters into the TOLD level.

L_{14} verifies whether the constraints for the received message are all true and, only in this case, allows the message to move forward.

L14: The **L14 TOLD_check_true(.)** law:

Locution: *TOLD_check_true*($Ag_y, Protocol, Primitive$)

Preconditions: The constraints of TOLD rules applicable to the name of the sender agent Ag_y and to the primitive and content must be true.

Meaning: The communication primitive is submitted to the check represented by TOLD rules.

Response: The primitive can be processed by the next step.

The TOLD player plays the game not only with itself and with other internal players but also with the environment. With itself, because it has an “its own opinion” (derived by means of the *told* rules) about the sender, primitive and content of the message (is the sender reliable? is the content compatible with its role and its knowledge base?) With the others, because it interplays moves with the META and interpreter players, where

the META player will apply ontologies for “understanding” the message contents. Finally, with the environment because the message contains information influencing its reception or elimination. For example, a non correct syntax of the message can determine its elimination apart from the “opinion” of the component.

The transition rule below specifies that an agent which is in modality *manage_perceptions* and finds an incoming message, will go into the *received_message* modality. It applies law L_{12} , that incorporates the precondition that an agent can receive a message only if a physical communication act has taken place in the server. We can see the neither the logic program nor the internal state of the agent change. This because the L_{12} law extracts the parameters from the message but does not involve internal processes of the agent.

$$R45 : \langle Ag_1, \langle P, IS, manage_incoming_messages_{Ag_1} \rangle \rangle \xrightarrow{L_{12}} \langle Ag_1, \langle P, IS, received_message(Message)_{Ag_1} \rangle \rangle$$

The R46 transition rule verifies the message protocol compatibility and only in the positive case allows the communicative act to go on. This initial filter is relevant because the protocol incompatibility can generate serious damages in the receiver agent. In fact, also if the primitive names are the same, the arguments can be different and this may result in a misleading interpretation.

$$R46 : \langle Ag_1, \langle P, IS, received_message(Message)_{Ag_1} \rangle \rangle \xrightarrow{L_{13}} \langle Ag_1, \langle P, IS, protocol_compatible(Protocol, Message)_{Ag_1} \rangle \rangle$$

Rule R47 forces the agent to eliminate a message with an incompatible protocol. The specification $not(L_{13})$ means that this transition takes place only in case law L_{13} cannot be applied.

$$R47 : \langle Ag_1, \langle P, IS, received_message(Message)_{Ag_1} \rangle \rangle \xrightarrow{not(L_{13})} \langle Ag_1, \langle P, IS, end_manage_incoming_messages_{Ag_1} \rangle \rangle$$

The R48 transition rule specifies how, once extracted the parameters Sender and Content and overcome the protocol control, the player invokes the corresponding TOLD rules. L_{14} acts in the situation where all constraints are verified and the message can be accepted by the agent. This step does not involve the logic program or the events queues, thus only the modality changes.

$$R48 : \langle Ag_1, \langle P, IS, protocol_compatible(Message)_{Ag_1} \rangle \rangle \xrightarrow{L_{14}} \langle Ag_1, \langle P, IS, TOLD(Sender, Content)_{Ag_1} \rangle \rangle$$

If instead at least one constraint is not verified, the precondition of L_{14} law becomes false and the expected response cannot be applied. This means that the message contains some information which is considered by the receiver agent to be either not interesting or harmful.

The move of the TOLD player that influences the behavior of META player is represented by the modality $TOLD(Sender, Content)$. In the DALI actual architecture, this move corresponds to the message overcoming of the TOLD filter level. Only in this case the message reaches the META level where the agent will try to “understand” the contents by possibly applying an ontology.

4.2 Communication Protocols

We have explicitly defined by means of suitable transition rules two communication protocols: the first one is the FIPA/DALI protocol, which is described in [34] by means of the R_{120} - R_{135} transition rules. The second one is the Four-acts protocol [31] and is dealt with by the R_{136} - R_{141} transition rules. We have described this protocol because its simplicity allows us to show all the related laws and rules. We notice however that the proposed operational semantics framework is in principle able to accommodate any other protocol.

In the Four-act protocol we have two agents, say Ag_1 and Ag_2 , each of which has a set of rules that determine its role in the dialogue. The dialogue subject is the request of a resource R by Ag_1 to Ag_2 . The agent Ag_1 will have the following protocol rules:

Ag₁ protocol rules

$ask(give(Resource), Ag_1) \Leftarrow init(Ag_1)$ (p₁)

$ok(give(Resource)Ag_1) \Leftarrow accept(give(Resource), Ag_2)$ (p₂)

$ask(give(Resource), Ag_1) \Leftarrow refuse(give(Resource), Ag_2)$ (p₃)

Ag_2 is the agent that receives the request and must decide if the resource can be given. Its choice depends on condition C , according to the following protocol rules:

Ag₂ protocol rules

$accept(give(Resource), Ag_2) \Leftarrow ask(give(Resource), Ag_1) \wedge C$ (p₄)

$refuse(give(Resource), Ag_2) \Leftarrow ask(give(Resource), Ag_1) \wedge \neg C$ (p₅)

The requesting resource process goes on indefinitely if the agent does not obtain the resource, so the above definition of the protocol has only a theoretical sense because, in the real world, the agents would presumably somehow put an end to the interaction.

Below are the laws and transition rules that operationally formalize this protocol. Notice that we have considered the original formulation of the Four-acts protocol, without trying either to extend it or to fix its problems, like e.g. termination. In fact, the Four-acts protocol is usually referred to in the literature as a simple example of “a” protocol, and not for its practical usability.

The first law makes the interpreter able to adopt this protocol (among the available ones), while the subsequent laws introduce the corresponding communication acts. The laws formalize both the asking and the responding roles, as an agent may alternatively take any of them.

L84: The **L84 adopt_four_acts_protocol(.)** law:

Location: $adopt_four_act_protocol(Protocol, Primitive, Content)$

Preconditions: No precondition.

Meaning: This law allows an agent to adopt Four-act protocol.

Response: The Four-acts protocol will be used to process the *Content* of the *Primitive*.

The R136 transition rule forces the interpreter to understand the communication primitive of incoming messages by using the Four-acts protocol:

$$R136 : \langle Ag_1, \langle P, IS, choose_protocol(Primitive, Content)_{Ag_1} \rangle \rangle \xrightarrow{L84} \langle Ag_1, \langle P, IS, process(Four - acts, Primitive(Content))_{Ag_1} \rangle \rangle$$

Law *L85* introduces the possibility for an agent to start the dialogue with the Sender agent that has issued a corresponding request. *L88* checks whether the agent is able to give the resource or not, while *L86* and *L87* are respectively responsible for acceptance and refusal. Finally, *L89* terminates the dialogue in case the requester agent has obtained the resource.

L85: The **L85 process(Four-acts, init(.))** law:

Locution: $process(Four - acts, init(Sender))$

Preconditions: No precondition.

Meaning: This law allows an agent to process a dialogue propose.

Response: The processing phase of this primitive continues.

L86: The **L86 process(Four-acts, ask(.))** law:

Locution: $process(Four - acts, ask(Resource, Sender))$

Preconditions: The dialogue is started through the *init* primitive.

Meaning: The agent *Sender* asks for the *Resource*.

Response: The processing phase of this primitive continues, thus allowing the agent to either accept or refuse this request.

L87: The **L87 process(Four-acts, accept(.))** law:

Locution: $process(Four - acts, accept(Resource, Sender))$

Preconditions: The agent has sent to the Sender the request to have a certain *Resource*.

Meaning: The agent *Sender* has accepted to give the *Resource*.

Response: The processing phase of this primitive continues, thus allowing the requester agent to obtain the *Resource*.

L88: The **L88 process(Four-acts, refuse(.))** law:

Locution: $process(Four - acts, refuse(Resource, Sender))$

Preconditions: The agent has sent to the *Sender* the request to obtain a certain *Resource*.

Meaning: The agent *Sender* has refused to give the *Resource*.

Response: The processing phase of this primitive continues, but the requester agent cannot obtain the *Resource*.

L89: The **L89 verify_request(.)** law:

Locution: $verify_request(Resource, Sender)$

Preconditions: The Sender agent has required the *Resource*.

Meaning: The agent verifies that it can give the *Resource* to the *Sender* agent and responds positively.

Response: The agent gives the resource.

L90: The **L90 process(Four-acts, ok(.))** law:

Locution: $process(Four - acts, ok(Resource, Sender))$

Preconditions: The agent has obtained the resource.

Meaning: This law allows an agent to conclude the dialogue having obtained the desired resource.

Response: The dialogue terminates.

The agent who wants to ask another agent for obtaining a resource starts the dialogue through the L_{85} law. In fact, *init* delineates the first communication step. The agent, after the initialization process, asks for the *Resource* by sending the message containing the action $give(Resource)$. We will indicate this request generically with *Action*:

$$R137 : \langle Ag_1, \langle P, IS, process(Four - acts, init(Sender)_{Ag_1}) \rangle \rangle \xrightarrow{L_{85}, L_{23}} \langle Ag_1, \langle P, NIS, send(Four - acts, ask(Action, Sender))_{Ag_1} \rangle \rangle$$

The agent receiving a resource request, accepts the proposal if the law L_{89} permits this:

$$R138 : \langle Ag_1, \langle P, IS, process(Four - acts, ask(Action, Sender)_{Ag_1}) \rangle \rangle \xrightarrow{L_{86}, L_{89}, L_{23}} \langle Ag_1, \langle P, NIS, send(Four - acts, accept(Action, Sender))_{Ag_1} \rangle \rangle$$

If the internal state does not authorize the resource transfer, the agent sends a *refuse* message.

$$R139 : \langle Ag_1, \langle P, IS, process(Four - acts, ask(Action, Sender)_{Ag_1}) \rangle \rangle \xrightarrow{L_{86}, not(L_{89}), L_{23}} \langle Ag_1, \langle P, NIS, send(Four - acts, refuse(Action, Sender))_{Ag_1} \rangle \rangle$$

A positive response to the assignment problem concludes the process and the interpreter goes into the *ok* modality.

$$R140 : \langle Ag_1, \langle P, IS, process(Four - acts, accept(Action, Sender)_{Ag_1}) \rangle \rangle \xrightarrow{L_{90}, L_{23}} \langle Ag_1, \langle P, NIS, send(Four - acts, ok(Action, Sender))_{Ag_1} \rangle \rangle$$

An agent that receives a refuse, persists stubbornly in its intent by returning to the *ask* modality:

$$R141 : \langle Ag_1, \langle P, IS, process(Four - acts, refuse(Action, Sender)_{Ag_1}) \rangle \rangle \xrightarrow{L_{86}, L_{23}} \langle Ag_1, \langle P, NIS, send(Four - acts, ask(Action, Sender))_{Ag_1} \rangle \rangle$$

5 Abstracting Agent Properties

We now propose some abstractions over the operational semantics behavior, that can be useful in order to define and prove properties. As a basic step, we define a function describing the agent behavior as the composition of the application of a given set of transition rules, starting from a specific agent operational state S_{Ag_x} . This abstraction process consists in formalizing the *Operational Semantics Call*:

Definition 6 (Operational Semantic Call). *Given a list of transition rules (R_1, \dots, R_n) and an initial operational state $S_{0_{Ag_x}}$ of an agent, we define the Operational Semantics Call $\Psi_{[k, \dots, n]}(Ag_x, S_{0_{Ag_x}}, (R_1, \dots, R_n))$ as follows:*

$$\Psi_{[R_1, \dots, R_n]}(Ag_x, S_{0_{Ag_x}}, (R_1, \dots, R_n)) = \langle Ag_x, S_{f_{Ag_x}} \rangle$$

such that

$\langle Ag_x, S_{0_{Ag_x}} \rangle \xrightarrow{R_1} \langle Ag_x, S_{1_{Ag_x}} \rangle \xrightarrow{R_2} \dots \xrightarrow{R_n} \langle Ag_x, S_{f_{Ag_x}} \rangle$ where Ag_x is the agent name and $S_{f_{Ag_x}}$ is the final operational state determined by the subsequent application of the given transition rules.

We then introduce a semantic function *Behavior*, that allows us to explore agent coherence by relating the perceptions that an agent receives in input to the actions that the agent performs, which constitute its observable output behavior. We consider the agent as a “black box” whose evolution is completely determined by the *Operational Semantics Call* Ψ computed on a given a list of transition rules and a given set of perceptions. Notice that the given perceptions are partly recorded in the “events” component of the initial state, and partly related to the laws which are applied by the given transition rules: as we have seen, there are laws that are applicable if there is an incoming perception, for instance an incoming message. Perceptions can take different forms and can be described in different ways depending upon the specific agent-oriented language/formalism and the application context at hand. Thus, we will not commit to any specific description. We only notice that in general: (i) A particular kind of perception consists in the reception of a message. (ii) A perception will be annotated with a time-stamp that indicates when the agent has become aware of it; then, perceptions are assumed to be totally ordered w.r.t. the time-stamp and can be compared on this basis.

The function *Behavior* operates an abstraction process that considers only the perceptions in input and the actions in output. As a preliminary step, we define the *Action Semantic function* that returns all operational states $S_{f_{Ag_x}}$ whose modality indicates that an action has been performed. A particular kind of action consists in sending a message.

Definition 7 (Action Semantic function). *Given the Operational Semantics Call Ψ computed on a list (R_1, \dots, R_n) of transition rules, we define the Action Semantic function Ξ_{Ag_x} as a function that, given $\Psi_{[R_1, \dots, R_n]}$, returns all $S_{f_{Ag_x}}$ ’s related to performing an action:*

$$\Xi_{Ag_x}(p, \Psi_{[R_1, \dots, R_n]}) = \{ S_{f_{Ag_x}} \mid \text{modality}(S_{f_{Ag_x}}) = \text{made}(\text{Action}) \vee \text{modality}(S_{f_{Ag_x}}) = \text{sent}(\text{Message/Action}) \}$$

where the function “modality” extracts the modality from $S_{f_{Ag_x}}$.

We now consider the set of actions that an agent performs *in response to a certain perception* p present in the initial state. It is obtained as the set of actions contained in

the states specified by the above *Action Semantic Function* when the agent receives the perception p as input.

Definition 8 (Semantic Behavior with respect to the perception p). Let $\Xi_{Ag_x}(p, \Psi_L)$ be the *Action Semantic Function* applied to a list L of transition rules and let the agent initial state contain only the perception p . We define the *Semantic Behavior*

$Beh_{Ag_x}^p(p, L)$ with respect to the perception p as the set of actions belonging to the $S_{f_{Ag_x}}$'s selected by Ξ :

$$Beh_{Ag_x}^p(p, L) = \{A \mid A \text{ is an action where } \exists S_{f_{Ag_x}} \in \Xi_{Ag_x}(p, \Psi_L) \text{ such that } A \in S_{f_{Ag_x}}\}$$

Below we extend the definition to consider a set P of perceptions.

Definition 9 (Semantic Behavior). Let $Beh_{Ag_x}^p(X, L)$ be the *Semantic Behavior* of the agent with respect to the single perception, let L be a list of transition rules, let $P = \{p_1, \dots, p_n\}$ be the set of perceptions contained in the agent initial state. We define the *Semantic Behavior* $Beh_{Ag_x}(P, L)$ with respect to P as

$$Beh_{Ag_x}(P, L) = \bigcup_{k=1, \dots, n} Beh_{Ag_x}^p(p_k, L)$$

The definition of the agent behavior through the operational semantics allows us to consider properties of our agents. An important property is conformance of an agent actual behavior with respect to the expected behavior (however defined, we do not enter into details here). We can say that an agent is *coherent* if it performs in response to a set of perceptions the set of actions that the designer expects from it.

Definition 10 (Coherent agent). Let $Beh_{Ag_x}(P, L)$ be the behavior (set of actions) of the agent Ag_x with respect to a set of perceptions P and a list L of transition rules. The agent Ag_x is coherent with respect to the expected behavior β_{Ag_x} if $\beta_{Ag_x} \subseteq Beh_{Ag_x}(P, L)$.

Definition 11 (Strongly Coherent agent). A coherent agent Ag_x is strongly coherent with respect to the expected behavior β_{Ag_x} if $\beta_{Ag_x} = Beh_{Ag_x}(P, L)$.

Let us restrict the set of perception that we consider to the incoming messages, and the expected behavior β_{Ag_x} to the out-coming message. Then, these two elements together may be assumed to describe the communication protocol Prt_{Ag_x} that the agent is expected to follow by an external observer. Then, strong coherence coincides with compliance w.r.t. the protocol.

Definition 12 (Protocol Compliance). Consider a protocol $Prt_{Ag_x} = \langle P, \beta_{Ag_x} \rangle$ where let P be a the set of incoming messages and β_{Ag_x} the corresponding expected set of out-coming messages. Ag_x is compliant w.r.t. Prt_{Ag_x} if it is strongly coherent w.r.t. P and β_{Ag_x} .

As a simple example of protocol compliance, it is easy to see that, referring to the operational semantics laws and rules defined in the previous section as OP :

Theorem 1. Agent Ag_x whose semantic behavior Beh_{Ag_x} is defined according to the transition rules in OP is compliant w.r.t. the Four-act protocol.

Intuitively, this is the case as each rule defining the protocol has a direct correspondence with a transition rule and a corresponding applied law in OP .

6 Conclusions

We have proposed a new approach to the operational semantics of logic languages. We have shown the effectiveness of the approach, and we have demonstrated how it allows useful properties to be defined and proved.

We have experimented the approach in the formalization of the DALI language. In [34] the full DALI interpreter is described, and such a complete working operational definition can hardly be found for other languages. The definition has been the basis of the working implementation of DALI.

We mean to consider in the future other languages/formalisms in order to emphasize, via the operational description, similarities, differences and general properties. Of course, we mean to consider formal properties of single agents or of multi-agent systems other less simple than those considered here.

References

1. K. R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming* 19/20, 1994.
2. K. A. Arisha, F. Ozcan, R. Ross, V. S. Subrahmanian, T. Eiter and S. Kraus. IMPACT: a platform for collaborating agents. *IEEE Intell. Systems* 14(2), 1999.
3. M. Baldoni, C. Baroglio, A. Martelli and V. Patti. Reasoning about logic-based agent interaction protocols. *Proc. of the Italian Conference on Computational Logic, CILC'04*, 2004.
4. M. Baldoni, C. Baroglio, A. Martelli, V. Patti and C. Schifanella. Verifying protocol conformance for logic-based communicating agents. *Post Proc. of the Fifth International Workshop on Computational Logic in Multi-Agent Systems, CLIMAV*, Springer-Verlag, 2005.
5. E. Boerger and D Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming* 24, Elsevier, 1995.
6. R. H. Bordini, M. Fisher, C. Pardavila and M. Wooldridge. Model checking AgentSpeak. *Second Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, ACM Press, 2003.
7. R. H. Bordini and A. F. Moreira. Proving BDI Properties of Agent-Oriented Programming Languages. *Ann. Math. Artif. Intell.* 42(1-3), 2004.
8. A. Bracciali, P. Mancarella, K. Stathis and F. Toni. On modelling declaratively multi-agent systems. *Proc. of Declarative Agent Languages and Technologies (DALT 2004)*, LNAI 3476, Springer-Verlag, 2004.
9. A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni. The KGP model of agency: Computational model and prototype implementation. *Global Computing: IST/FET International Workshop, Revised Selected Papers*, LNAI 3267, Springer-Verlag, 2005.
10. S. Costantini and A. Tocchio. A Logic Programming Language for Multi-agent Systems. *Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002*, LNAI 2424, Springer-Verlag, 2002.
11. S. Costantini, A. Tocchio. About declarative semantics of logic-based agent languages. *Declarative Agent Languages and Technologies (Post-Proc.)*, LNAI 3229, Springer-Verlag, Berlin, 2006.
12. S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog, *Journal of Logic Programming*, 1988.
13. M. d'Inverno and D. Kinny and M. Luck and M. Wooldridge. Formal Specification of dMARS. *ATAL '97: Proc. of the 4th Int. Works. on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, ISBN 3-540-64162-9, Springer Verlag, 1997.

14. M. d’Inverno and M. Luck. Formalising the Contract Net as a Goal-Directed System. Agents Breaking Away: Proc. of the Seventh Europ. Works. on Modelling Autonomous Agents in a Multi-Agent World, Lecture Notes in Artificial Intelligence, 1038, Springer-Verlag, 1996.
15. M. d’Inverno and M. Luck. Engineering AgentSpeak(L): A Formal Computational Model. Journal of Logic and Computation 8(3), 1998.
16. M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. Readings in Planning, Morgan-Kaufmann, 1990.
17. R. Goodwin. A formal specification of agent properties. Journal of Logic and Computation 5(6), 1995.
18. K. V. Hindriks, F. de Boer, W. van der Hoek, and J. C. Meyer. 1999. Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems 2(4), 1999.
19. R. A. Kowalski. How to be Artificially Intelligent - the Logical Way, Draft, revised February 2004, Available on line, URL <http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html>.
20. K. Larson and T. Sandholm. Bargaining with limited computation: Deliberation equilibrium. Artificial Intelligence 132(2), 2000.
21. J. A. Leite, J. J. Alferes and L. M. Pereira. *MINERVA*: A dynamic logic programming agent architecture. Proc. ATAL01, LNAI 2333, Springer Verlag, 2002.
22. J.W. Lloyd. Foundations of Logic Programming, Second Edition. Springer-Verlag, Berlin, 1987.
23. M. Luck, N. Griffiths and M. d’Inverno. From agent theory to agent construction: A case study. Intelligent Agents III: Proc. of the Third Int. Works. on Agent Theories, Architectures and Languages, LNAI 1038, Springer-Verlag, 1997.
24. D. C. Parkes. Optimal auction design for agents with hard valuation problems. Agent-Mediated Electronic Commerce Works. at the Int. Joint Conf. on Artificial Intelligence, Stockholm, 1999.
25. P. McBurney and S. Parsons. Dialogue Games in Multi-Agent Systems. Informal Logic 22(3), Special Issue on Applications of Argumentation in Computer Science, 2002.
26. P. McBurney and S. Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. Journal of Logic, Language and Information, 11 (3), Special Issue on Logic and Games, 2002.
27. P. McBurney, R. M. van Eijk, S. Parsons and L. Amgoud. A Dialogue Game protocol for agent purchase negotiation, Autonomous Agents and Multi-Agent Systems, 7 (3), 2003.
28. S. Parsons, M. Wooldridge and L. Amgoud. Properties and complexity of some formal inter-agent dialogues. Journal of Logic and Computation, 13 (3).
29. A. S. Rao AgentSpeak(L): BDI Agents speak out in a logical computable language. Agents Breaking Away: Proc. of the Seventh Europ. Works. on Modelling Autonomous Agents in a Multi-Agent World, Lecture Notes in Artificial Intelligence, 1038, Springer-Verlag, 1996.
30. A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. KR - 1992, DBLP <http://dblp.uni-trier.de>.
31. F. Sadri, F. Toni and P. Torroni. Dialogues for negotiation: agent varieties and dialogue sequences. Intelligent Agents VIII: 8th Inter. Works., ATAL 2001, Seattle, WA, USA, Revised Papers, LNAI 2333 Springer-Verlag, 2002.
32. T. Sandholm Unenforced e-commerce Transactions IEEE Internet Computing 1(6), 1997.
33. T. Sandholm, S. Suri, A. Gilpin and D. Levine. CABOB: A fast optimal algorithm for combinatorial auctions. Proc. of IJCAI-01, Seattle, WA, 2001.
34. A. Tocchio. Multi-agent systems in computational logic, Ph.D. Thesis, Dipartimento di Informatica, Università degli Studi di L’Aquila, 2005.
35. M. Viroli and A. Omicini. Multi-Agent Systems as Composition of Observable Systems WOA 2001 – Dagli oggetti agli agenti: tendenze evolutive dei sistemi software - <http://lia.deis.unibo.it/books/woa2001/pdf/22.pdf>.

36. M. Wooldridge and N. R. Jennings. Formalizing the cooperative problem solving process. Readings in agents, ISBN 1-55860-495-2, Morgan Kaufmann Publishers Inc., 1998.
37. M. Wooldridge and A. Lomuscio. A Logic of Visibility, Perception, and Knowledge: Completeness and Correspondence Results. Logic Journal of the IGPL 9(2), 2001.

A Heuristic Approach for P2P Negotiation

Stefania Costantini, Arianna Tocchio, and Panagiota Tsintza

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
{stefcost,tocchio,panagiota.tsintza}@di.univaq.it

Abstract. In this paper we present a formal and executable approach to automated multi-issue negotiation between two competitive agents. In particular, this approach is based on reasoning in terms of projections in convex regions of admissibility values and is an extension of previous work by Marco Cadoli in the area of proposal-based negotiation. Our goal is to develop a heuristic strategy to flexibly compute the offers and the counter-offers so as to fulfill the agent objectives and minimize the number of the agents' interactions. The proposed algorithm improves a fundamental parameter of the negotiation process: the transaction's complexity.

1 Introduction

Nowadays, negotiation and particularly automated negotiation, becomes more and more important as a consequence of the rapid development of web-based transactions and in particular of e-commerce. Negotiation is an important subject of study in the branch of Distributed Artificial Intelligence (DAI) and MAS (Multi-Agent Systems) ([1–4]).

The negotiation process can be defined as a particular form of interaction between two or more agents. As mentioned by Walton e Krabbe in ([5]), negotiation is a particular type of interaction in which a group of agents, with a desire to cooperate but with conflictive interests, work together in aim to reach a common goal. Governatori et al. emphasize that the negotiation can be seen as the process that involves at least two parts whose common goal is to achieve an agreement that is acceptable from all parts in the process ([6]).

Jennings et al. ([7]) provide a more formal definition of the process according to which: “The negotiation can be defined as a distributed research in a space of potential agreements”. In this sense, each participant involves a portion of the geometric space that delimitates her individual areas of interest (also called *negotiation spaces*). Therefore, each automated agent intends to reach all agreements in her area. Negotiation spaces can be represented by a set of constraints and then finding an agreement can be modeled as a *constraint satisfaction problem* (CSP). In particular, in multi-agent systems the negotiation process can be represented as a *distributed constraint satisfaction problem* (DCSP), since the constraints are distributed among different agents ([8]).

A large number of approaches to negotiation (e.g., *game theoretic, auction-based, heuristic-based, argumentation-based*([9])) has been recently developed.

We pay particular attention to *proposal-based negotiation*. In this case, the process involves a number of agents which usually have a limited common knowledge about the other's constraints, i.e., about the negotiation spaces. In proposal-based negotiation the information exchange between the parties is in the form of offers (internal points of the negotiation spaces) rather than constraints, preferences or argumentation. Each agent is able to compute the points of the individual areas in order to reach an agreement. During the process each negotiation space can evolve, for example expand or shrink, as a response to internal or external events. The negotiation terminates when the participants find a point, in the space of negotiation, that is mutually acceptable. Obviously, that point has to be included in the common area of all negotiation spaces (i.e., in the intersection of the areas of interests)([3]).

The agents, involved in the negotiation process need to interact and are usually *self-interested*, since each one has different constraints to satisfy and different benefits, in terms of utility functions, to maximize (or minimize). The utility functions can be represented as new constraints on the agents knowledge. The speed of the process, or time complexity, largely depends on the particular negotiation strategy adopted by each agent.

The research work reported here is an extension of previous work by Marco Cadoli, introduced in ([10]). This paper presents an heuristic strategy for proposal-based negotiation, whose goal is to minimize the number of the interactions between the automated agents involved in the process and thus to speed-up the search of an agreement. In this approach, the negotiation spaces are considered as convex, i.e., all points between two acceptable variable assignments are acceptable as well. The admissible offers are internal points of the negotiation areas, and those will be the only exchangeable information among the involved agents. Moreover, the agents are able to reason in means of projections. As discussed below, reasoning by means of projection can help the agents to compute subsequent offers as each one can exclude certain points of the individual negotiation areas.

The rest of this paper is structured as follows. Section 2 is an overview of related work. In section 3 we present the theoretical background and the basic approach to negotiation that we adopt, introduced by Marco Cadoli. In Section 4 we discuss our motivations for extending this basic approach. Section 5 is devoted to the presentation of the features of the extended negotiation model. In section 6 we present the implementation of the proposed strategy and in section 7 we analyze a case study and the related experimental outcomes. In Section 8 we conclude and outline future work.

2 An overview

Numerous strategies have been propose in order to improve the efficiency, completeness and robustness of the negotiation process(e.g., [10, 3, 11–20]). In [11] a number of agent strategies designed for the 2002 TAC (*Trading Agent Competition*) are reported. These techniques include machine learning, adapted, planning

and hybrid agents as well as heuristic-based strategies. The aim of [12], instead, is to determine how an agent (with firm deadlines) can select an optimal strategy based on an incomplete information about his opponent. STRATUM, reported in [13], is a methodology for guiding strategies for negotiating agents in non-game-theoretic domains.

Gomes in [14] analyzes how a group of agents can deal in the presence of *externalities*. This has been done by adding to the worth of a bilateral coalition the amount of the negative externalities, that is created for the excluded player. The equilibrium values are increased or decreased in the presence of negative or positive externalities. In [15] the line of research has produced a number of results on the hardness of dealing with positive and negative externalities and the maximization of the social welfare. In [18] Matos et al. present an empirical evaluation of a number of negotiation strategies in different types of automated agents and environments. Moreover, this research presents a service-oriented negotiation model and a number of tactics (as time-dependent, resource-dependent and behavior-dependent) as well as strategies capable to compute the offers and counter-offers used by the agents during the process.

Rahwan et. al in [19] have developed a protocol-independent theory of strategy. In this approach, various factors that influence the generation of strategies are defined, used by the automated agents. In [20], instead, a negotiation model and two types of negotiation strategies, called *concession* and *problem solving* strategies, are presented. This work considers three subclasses of strategies, called *starting high and conceding slowly*, *starting reasonable and conceding moderately* and *starting low and conceding rapidly*. The first one models an optimistic opening attitude and successive small concessions, the second one a realistic opening attitude and successive moderate concessions and, finally, the third one models a pessimistic opening attitude and successive large concessions.

The negotiation process is considered as a constraint-based model in [10],[17], and [3]. In [17] and [3] negotiation is considered as a distributed constraint satisfaction problem. Moreover, the exchange of information between the involved agents is limited to the offers. All other information items such as the negotiation space or the utility function are private and are considered as local knowledge of each agent. Each agent reduces the negotiation space by asserting new constraints in the agent's local knowledge. The mentioned research is focused on the introduction of an architecture that supports *one-to-one* and *one-to-many* negotiation.

3 Theoretical Background

In this paper, we discuss and extend the approach of reasoning by means of projections reported in ([10]). In this approach, negotiation is considered as a distributed constraint satisfaction problem and the negotiation spaces as convex regions: i.e., all points included in the individual regions are equally acceptable. The aspects that influence the definition of negotiation protocols are: 1) the objects of the negotiation, 2) the models (or tactics) that an agent adopts for

computing the various decisions, 3) the agents cooperation level and 4) the costs of the communication and computation, in terms of time, space, etc.

The interactions among agents involve the following factors:

- **Variables:** it is assumed that, before the beginning of the negotiation, the agents agree on the number and on the nature of the variables considered during the process. Variables represent issues that the agents wish to take into account. Variables are numeric (or boolean as special cases of integers). In the examples that we consider, variables are all real.
- **Constraints:** the negotiation areas of each agent can be represented as a set of constraints. A proposal can be accepted only if it is included on the individual negotiation area (that means that the set of constraints is satisfied). Note that the individual negotiation area (i.e. all the admissible offers) is private knowledge of each agent.
- **Information exchange:** it is assumed that the negotiation adopts a *proposal-based* mechanism. This means that the only information exchange among agents consists in offers and counter-offers. Moreover, the agent's interaction is vertex-based and therefore the agents are bound to select as offers only the areas vertices. I.e., this approach is applicable only in case of polyhedral negotiation areas. All other information items, such as constraints or preferences, will be considered as private. The agent's response to each offer can be either acceptance, that concludes the process, or rejection, that induces a counter-offer (if existing).
- **Protocols and agent's cooperation level:** the protocols can be considered to be the rules that each part involved in the negotiation process has to respect. In this context, the agents are bound to communicate only the offers included in their individual areas and that they are really willing to accept. In this sense, we consider the agents to be partially cooperative.
- **Strategy:** this approach adopts the strategy of concluding the process in the minimum number of interactions. This means that the approach tries to find an offer included in the common part of the negotiation areas by using the minimum number of interactions.
- **Objective function:** each agent is self-interested, which means that she/he ignores the other agent's preferences.

In [10] it has been proved that reasoning in terms of projections can lead to a protocol that always converges and in some cases allows for large savings in terms of number of proposed vertices (in the worst case, the number of agent's interactions is exponential in the number of variables and in the number of constraints). More specifically, an agent that has her offer rejected is capable of constructing a projection: i.e., new constraints obtained by connecting the proposals made so far. Projections help her compute the subsequent offers. In fact, after the projection construction the agent understands that all points included in her projection area can't be accepted from the opposer. In this way, she excludes all vertices which belong to this particular area thus avoiding unnecessary proposals, and proceeds with the remaining ones. Note that the number of vertices of a convex region is finite: then, the negotiation process ends

either when a point of agreement in the intersection of the negotiation areas is found or when it is proved that there is no such point.

3.1 Reasoning by means of projections

Below we introduce the main concepts related to negotiation by means of reasoning on projections. For the sake of clarity, we explain the approach by means of an example. Let us assume that the feasibility regions are finite, that means that they can be described by means of linear constraints. The example proposed here concerns a bilateral peer-to-peer negotiation process involving two issues. We also note that the negotiation areas are considered as polyhedral feasibility regions. This assumption enables a vertex-based interaction. For the moment in fact, let us assume that the only possible offers that an agent can make are the vertices of her feasibility region. Let us consider two agents, say *Seller* and *Buyer*, involved in the negotiation process and represented by the respective negotiation areas reported in figure 1. In this example, the negotiation

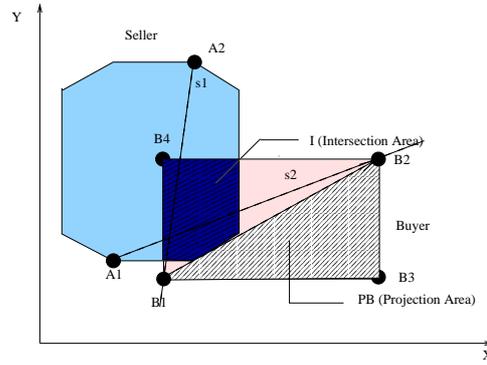


Fig. 1. An example

area (indicated as R_A) of agent *Seller* can be described by the set of constraints $C_A = \{X \geq 4, X \leq 20, Y \geq 13, Y \leq 40, X \geq -3Y+49, X \leq Y+4, X \geq (3/2)Y-50, X \leq -(3/4)Y+47\}$. The negotiation area (indicated as R_B) of the agent *Buyer* can be described by the set $C_B = \{X \geq 15, X \leq 40, Y \geq 10, Y \leq 25\}$.

By resolving each set of constraints, we can conclude that each area has four vertices that will be used as proposals during the agents interaction. Let $V_A = \{A_1 = (10, 13), A_2 = (17, 40), A_3 = (4, 15), A_4 = (17, 13), A_5 = (20, 16), A_6 = (20, 36), A_7 = (4, 36), A_8 = (10, 40)\}$ be the sets of possible proposals (set of vertices of the negotiation space) of the agents *Seller* and $V_B = \{B_1 = (15, 10), B_2 = (40, 25), B_3 = (40, 10), B_4 = (15, 25)\}$ be the possible proposals of the agent *Buyer*. The intersection area $I = R_A \cap R_B$ is clearly not empty, and therefore there is a potential agreement between the two agents.

We assume that the negotiation process starts with a proposal from the agent *Buyer* and that the sequence of proposals is as follows: B_1, A_1, B_2, A_2 . Each interaction has the side-effect of updating the knowledge base of the agents by storing all proposals (both made and received) and possible new constraints. The subsequent steps of the negotiation are motivated as follows:

- Since agent *Buyer* has received as counter-offer point A_1 not included into the buyer’s negotiation area, she rejects the proposal.
- After two interactions, the agents have exchanged four proposals, namely B_1, A_1, B_2, A_2 . None of them has been accepted. At this point, the agent *Buyer* computes a projection area by binding the couples of points (B_1, A_2) , (B_2, A_1) and (B_1, B_2) and by adding to her knowledge base the new linear constraints that represent the new lines s_1 and s_2 . The agent is able to conclude that the intersection of her negotiation area with the projection area (delimited by the lines s_1, s_2 and (B_1, B_2)) is empty and therefore all points of projection area can’t be accepted from the counter part of the process. The agent exploits the updated knowledge to select the next offer to make, excluding all initial points that are included in the projection area. In this way, the agent *Buyer* understands that her vertex B_3 can’t be accepted by the opposer, excludes this point from the set of proposals and proceeds by offering her point B_4 .
- Finally, this last offer is contained in the negotiation area of the agent *Seller* and therefore this proposal will be accepted. In this case, we say that the negotiation process terminated successfully.

In the case where the last offer is not contained in the negotiation area of the opposer, the agent *Buyer* would conclude that there is no further point to propose and would terminate the negotiation process, having proved that the intersection of the two negotiation areas is empty and therefore there is no possible agreement.

4 Limits of the Original Approach

We introduce an extension to the basic approach that we have previously described. The motivations of our extension are summarized in the following three points:

- Limiting the possible proposals to the vertices of the respective regions in ([10]) is efficient but, in some cases, inducts problematic trades, specially in cases in which the intersection area is not empty but not includes vertices. For example, in figure 2 there are two agents, *seller* and *buyer*, whose individual negotiation areas are expressed through convex regions. It is clear that there might be a potential agreement amongst the agents, since the intersection area includes various points. However in this case, after six interactions (the sequence of proposals is $B_1, A_1, B_2, A_2, B_3, A_3$) the seller agent perceives that there is no other vertex to propose (as she has previously excluded vertex

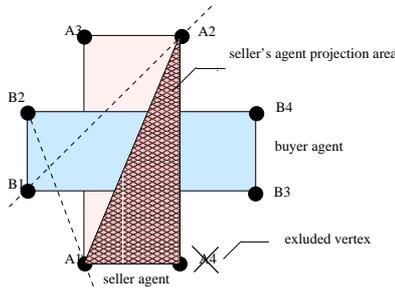


Fig. 2. A first problematic interaction

A_4) and so she concludes the negotiation process with a proof that there is not a point of agreement.

- Also when the intersection area contains one or more vertices determining a possible agreement, the exploration of the points internal to the area represents a more realistic and sometime advantageous method for concluding the negotiation process.
- The “flat” nature of proposals where all vertices are equally considered may lead in real applications to further problematic situations. In particular, it is natural for us to consider that one or more issues considered by an agent have greater priority than others. Let us assume that the agents try to reach an optimal point with respect to an objective function. This objective function can be chosen according to the particular context. In fact, the algorithm is independent of it. In figure 3 we consider two agents, *Business* and *Client*. We assume that Business desires to maximize the issue Y and that the sequence of interactions starts with a proposal (from Client) of the point B_1 and continues with A_1 , B_2 and A_2 . In this case, the approach of reasoning by means of projections does not allow us to obtain savings in terms of number of proposed vertices.
- As mentioned in [10], a problem of the approach of reasoning by means of projections is that, since the agents have to remember all former proposals (made, received and subsequent), it is hard to find algorithms and data structures which allow agents to store the entire sequence of proposals in polynomial space.

We discuss below how to solve (at least to a certain extent) the problems described in the previous three points by extending the approach so that the proposals are not only vertices but also internal points of the convex regions. Those points will not be randomly selected. Their selection, instead, will be based on the recent proposals made by the agent, which will be increased (or decreased) by a δ margin. This margin can be chosen, for example, as proposed in [21]. Besides, we will no longer memorize all proposals but, rather, the projection areas thus updating the agent’s knowledge base by adding new constraints.

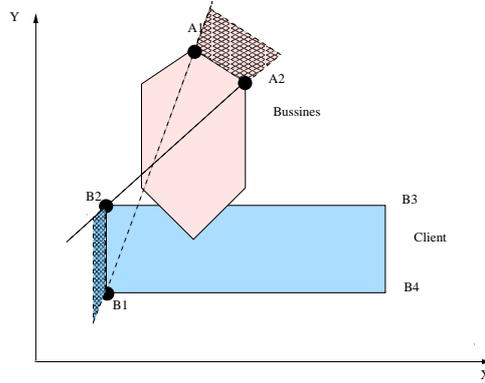


Fig. 3. A second problematic interaction

5 Extending the Approach

5.1 Model description

As mentioned in the previous section, we are going to introduce a formal and executable approach to automated multi-issue negotiation between two partially-cooperative agents. In this paper we present the approach for a 2-issue scenario because the multi-variables approach implies the exploitation of the projections in the space, a scenario on which we are working but that at this moment has not been evaluated. The adopted strategy overcomes a number of problems that can be identified in the original formulation. To this aim, we define the set of agents involved in the process as $AS = \{A_1, A_2, \dots, A_n\}$. Agent i maintains m issues, represented in terms of variables $x_i^1, x_i^2, \dots, x_i^{m_i} \in \mathbb{R}^+$.

For the sake of simplicity and without loss of generality, we explain the extension by considering only two agents A and B . Then, the set of agents is $AS = \{A_1, A_2\}$. We assume that the agents have already agreed on the number and on the nature of the variables to use during the process. Let us assume for instance that the agents have agreed upon using two variables denoting real numbers. In this way, the two negotiation areas can be represented in the cartesian plane. They will be denoted as R_A and R_B . In general, as discussed above the two bidding agents can reach an agreement only in the case where $R_A \cap R_B \neq \emptyset$.

In the proposed strategy the negotiation is considered to be interactive, that is the bidding agents need several rounds of interactions (of offers and counter-offers) before they conclude in an agreement. The agents are considered to be partially cooperative, in the sense that they cannot propose the same offer twice and cannot make a proposal that they are not willing to accept. That means that all offers have to be included in the individual negotiation areas. The extension to ([10]) is based on the change of the nature of the proposals which will be allowed to be internal points of the feasibility areas rather than just vertices.

The strategy takes further advantage from reasoning by means of projection. As before, during the process the bidding agents will never do proposals that are

included in the individual projection areas since those particular points will never be accepted from the antagonist. Furthermore, projection areas will be created dynamically during negotiation. In this way, the agents don't need to use all the internal points of the individual feasibility region: this will lead to a great saving in terms of offers. Note however that the number of offers remains exponential in the worst case. Nevertheless, after a large number of experiments we are confident about the fact that the proposed strategy speeds up the negotiation process in the average case.

5.2 The trade-off strategy in multi-variable regions

The tactic proposed in this paper thus tries to minimize the interaction complexity by excluding several points of the feasibility regions of agents. Assume that the two negotiation areas R_A and R_B are those represented in figure 4. The negotiation process initiates with a proposal, for example by agent A . The first proposal is assumed to be a random point of the individual convex region of the agent. The second proposal instead is identified as follows: each agent computes

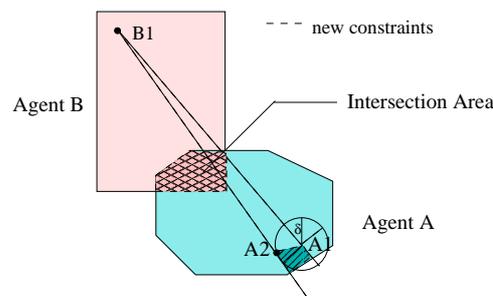


Fig. 4. The trade-off strategy in multi-variable regions

the circumference whose center corresponds to the point of the first proposal and whose radius is $R = \delta$, where the choice of the margin δ will depend on each specific application context and, chosen at the beginning of the negotiation phase, does not change at runtime. The proposal will be selected as a random point of the semi-circumference closer to the opposer's proposal. Clearly, we require it to be included on the individual feasibility region. In this way, the agent tries to approach the opposer's offer by proposing a point that is more likely to be accepted, and by adapting the individual profile to the one that can be assumed for the opposer. To do this, the agent has to add new constraints to her knowledge base. If no such point is found, then the next agent's proposal will be a new random point of the feasibility region. After that, each agent uses the reasoning on projection in the same way as in [10]. The points included in the projection area will be stored in the agents memory, by adding the new constraint that represent the projection area to her knowledge base.

All subsequent proposals will be selected in the same way as the second one and the center of the circumference will be the last proposal of the agent, adding one further constraint: the new proposals must not be included in the areas of the projection made so far. The projection areas may be described in terms of a set of linear dis-equations. In this way, the agent will find the new variables to offer working out a new, or extended, DCSP problem.

An advantage is that the agent does not have to store all the potential proposals. Rather, the only information the agent needs in order to construct the new projections (the constraints), are four trade-offs, that is the two most recently proposals from each agent. If a satisfactory contract has not been found yet, the agent continues with the next proposal and so on. Below is the precise definition of the algorithm:

Algorithm:

1. Find the first proposal $O_1 = (x_1, y_1)$ as a random point of the feasibility region and make the proposal.
2. If the counter-offer is not included in the region (no agreement is found), then do:
 - (a) Propose a new offer $O_2 = (x_2, y_2)$ by using the trade-off strategy.
 - (b) Generate the projections by using the reasoning by means of projection.
 - (c) Add the new constraints which represent the projections area to the agent's knowledge base.
 else accept offer
3. do
 - (a) Given the last offer of $O_i = (x_i, y_i)$ proposed by the other agent, and while no deadlock is observed, generate the projections by using the reasoning by means of projection.
 - (b) Add the new constraints which represent the projections area to the constraints found so far.
 - (c) propose a new offer $O_{i+1} = (x_{i+1}, y_{i+1})$ by using the trade-off strategy. while (the counter-offer is not included in the region (no agreement is found))

6 Implementation

6.1 The DALI language

The proposed approach has been implemented in the DALI language. DALI ([22–26]) is an Active Logic Programming language designed in the line of [27] for executable specification of logical agents. DALI is a prolog-like logic programming language with a prolog-like declarative and procedural semantics [28]. In order to introduce reactive and proactive capabilities, the basic logic language has been syntactically, semantically and procedurally enhanced by introducing several kinds of *events*, managed by suitable *reactive rules*. All the events and actions are timestamped, so as to record when they occurred. These features are summarized very briefly below.

An *external event* is a particular stimulus perceived by the agent from the environment. We define the set of external events perceived by the agent from

time t_1 to time t_n as a set $E = \{e_1 : t_1, \dots, e_n : t_n\}$ where $E \subseteq S$. and the e_i 's are atoms. indicated with postfix E in order to be distinguished from both plain atoms and other DALI events. External events allow an agent to react through a particular kind of rules, reactive rules, aimed at interacting with the external environment. When an event comes into the agent from its external world, the agent can perceive it and decide to react. The reaction is defined by a reactive rule which has in its head that external event. The special token $:>$, used instead of $:-$, indicates that reactive rules performs forward reasoning. A reactive rule has the form:

$ExtEvent_E :> Body$ where $Body$ has the usual (logic programming) syntax and intended meaning except that it may contain the DALI event and action atoms which are introduced below.

The *internal event* concept allow DALI agents to be proactive independently of the environment by reacting to its own conclusion (which can be considered as a form of introspection). More precisely: An internal event is syntactically indicated by postfix I and implies the definition of two rules. The first one contains the conditions (knowledge, past events, procedures, etc.) that must be true so that the reaction (in the second rule) may happen:

$IntEvent : -Conditions$

$IntEvent_I :> Body$

Internal events are automatically attempted with a default frequency customizable by means of user directives in the initialization file that can tune also other parameters such as how many times an agent must react to the internal event (forever, once, twice,...) and when (forever, when triggering conditions occur, ...); how long the event must be attempted (until some time, until some terminating conditions, forever).

Actions are the agent's way of affecting the environment, possibly in reaction to either an external or internal event. An action in DALI can be also a message sent by an agent to another one. An action atom is syntactically indicated by postfix A . Clearly, when an atom corresponding to an action occurs in the inference process, the action is supposed to be actually performed by suitable "actuators" that connect the agent with its environment. In DALI, actions may have or not preconditions: in the former case, the actions are defined by actions rules, in the latter case they are just action atoms. An action rule is just a plain rule, but in order to emphasize that it is related to an action, we have introduced the new token $:<$. External and internal events that have happened (i.e., that have been reacted to) and actions that have been performed are recorded as past events, that represent the agent's memory, and the basis of its "experience".

6.2 Implementation Outline

In this section, we present a snapshot of the code developed for the implementation of the above-discussed negotiation algorithm, paying particular attention to some reactive and proactive capabilities of the agents implemented in DALI. First of all, we note that all constraints (and therefore the representation of the negotiation area) as well as the margin δ to be used during the interaction

are stored in the agent profile and are loaded at runtime. This makes the implementation elaboration-tolerant w.r.t. changing the negotiation parameters. The proposals of the counter-part is received by the agent by means of a DALI reactive rule: $offerE(X, Y, A) \text{ :- } once(reconsider(X, Y, A))$.

Whenever an agent receives an offer she controls if the offer is included in the negotiation area and, if so, she responds. This is implemented via the following rule:

$reconsider(X, Y, A) \text{ :- } \text{--}area(X, Y), !, clause(agent(A), \text{--}),$
 $messageA(clientenew, send_message(accept_pr(X, Y, A), A)).$

In the opposite case the agent chooses, by using the rule *call_random_semicycle* ($X1, Y1, X, Y$), a (random) point of the closer semi-circumference to the opposer's proposal that is included in the negotiation area and not included in the projection area constructed so far. After that, the agent sends a message containing the counter-offer by means of an action of the kind *messageA*.

Finally, she updates the knowledge base by adding the new constraints that represent the projection areas and by updating the last four proposals (those made and received).

$reconsider(X, Y, A) \text{ :- } \text{--}out_of_area(X, Y), (Az > X, Bz > Y, out(\text{--}, \text{--}, \text{--}, X, Y)),$
 $call_random_semicycle(X1, Y1, X, Y),$
 $messageA(clientnew, send_message(new_of_fert(X1, Y1, A), A)),$
 $clause(agent(A), \text{--}), extract_of_f1(of_fert1(\text{--}, \text{--}), \text{--}, \text{--}),$
 $extract_of_f2(of_fert2(\text{--}, \text{--}), Ac, Bc), update_of_fert1(of_fert1(\text{--}, \text{--}), Ac, Bc),$
 $update_of_fert2(of_fert2(\text{--}, \text{--}), X, Y), update_proposal1(proposal1(\text{--}, \text{--}), Az, Bz),$
 $update_proposta2(proposta2(\text{--}, \text{--}), X1, Y1), clause(offers(L), \text{--}), append([Y], L, L1),$
 $append([X], L1, L2), assert(offers(L2)), retractall(offers(L)),$
 $clause(proposals(Lp), \text{--}), append([Y1], Lp, L3), append([X1], L3, L4),$
 $assert(proposals(L4)), retractall(proposals(Lp)).$

As an example of the pro-active capabilities of the agent, we show how the agent checks whether a point is included in the projection areas. This check employs an internal event, represented by a number of DALI rules. As mentioned above, the conclusion of the first couple of rules is automatically attempted from time to time. If it is true (i.e., it has been proved), possibly returning some values for input variables, then the body of the second rule (the reactive one) is executed, after assigning the values to the variables.

$update_history(\text{--}, \text{--}, X, Y) \text{ :- } \text{--}offerP(X, Y, \text{--}).$
 $update_historyI(Lo, Lp, X, Y) \text{ :- } \text{--}$
 $[\text{--}, \text{--}, Xo2, Yo2, Xo1, Yo1|L1] = Lo, [\text{--}, \text{--}, Xp2, Yp2, Xp1, Yp1|L2] = Lp,$
 $new_condition(X, Y, Xp2, Yp2, Xo2, Yo2, Xp1, Yp1, Xo1, Yo1),$
 $update_constraints([Xo2, Yo2, Xo1, Yo1|L1], [Xp2, Yp2, Xp1, Yp1|L2], X, Y).$
 $new_condition(X, Y, X1, Y1, X2, Y2, X3, Y3, X4, Y4) \text{ :- } \text{--}X4 > X1, X2 >= X4,$
 $Y2 >= Y4, X1 = X3, !, X1 >= X, coef_f2(X1, Y1, X2, Y2, X3, Y3, X4, Y4, M2),$
 $Cost2isY4 - (M2 * X4), (M2 * X) + Cost2 >= Y, coef_f3(X1, Y1, X2, Y2, X3,$
 $Y3, X4, Y4, M3), Cost3isY2 - (M3 * X2), Y >= (M3 * X) + Cost3.$

Here, $update_historyI(Lo, Lp, X, Y)$ is an internal event that is triggered each time the agent receives a new offer. The procedure $new_condition(X, Y, Xp2, Yp2, Xo2, Yo2, Xp1, Yp1, Xo1, Yo1)$ builds the new projection (by constructing the new constraints) and the $update_constraints([Xo2, Yo2, Xo1, Yo1|L1], [Xp2, Yp2,$

, $Xp1, Yp1|L2, X, Y$), updates the knowledge base of the agent by adding the new constraints.

7 Experiments

We have experimented the performance of the application by putting at work two intelligent agents in the wearing market context. An agent plays the role of the seller while the second one of the buyer. The exchanged products are described simply by two attributes: the price and the delivery days. This choice allows us to restrain to the cartesian plane.

In [29] we present many experiments, that have allowed us to elicit the impact of the main parameters on the algorithm performance. Here for the sake of brevity we present a single experiment.

Let us define some relevant experimentation parameters for the seller. Let X and Y be variables indicating respectively the delivery days and the price, with $3 < X < 33$ and $370 < Y < 740$ euros. For determining the area in which the offers can be identified, we suppose that a discount of the 50% can be applied to each product. The resulting area for the seller is 30 (days) for 370 (euros). The δ parameter has been set to 12. For the buyer, we have set an area of 25 days for 130 euros that we move in the plane in order to vary the intersection area with the one of the seller. The δ parameter for the buyer has been set to 10. The negotiation process is started by the buyer agent which proposes to the seller the next point in her area. The table 1 shows the results of the testing phase. The *Intersection Area* describes the size of the area where the seller and buyer

Intersection Area	Average	Max n. iterations	Min n. iterations
8 d. x 30 euro	9,59	15	2
8 d. x 20 euro	10,55	15	2
8 d. x 10 euro	12,67	17	2
1 d. x 1 euro	50,81	152	2

Table 1. The results of the testing phase

agents can find an agreement. The *Average* synthesizes how many interactions in average are necessary in order to conclude the negotiation process, while the *max* and *min* values represent respectively the highest and lowest number of interactions reached in the respective tests. The tests have been executed on a Pentium IV with 225 MB RAM.

We have to notice that the number of interactions is no more proportional in the number of vertices, as in the case of [10]. In our case, the number of interactions heavily depends upon the dimension of the intersection area and upon the distance between the first two offers. However, tests have showed that in the case in which the negotiation area has a relevant number of vertices, the

approach proposed in this paper results to be more efficient with respect to the original one, at least in average case. Note that in the best case, i.e., when the first (random) offer falls in the intersection area, the agreement is obtained after one interaction. Even though the algorithm complexity, in some extreme cases, can be considered high we claim that the granularity of the search space can justify this fact.

8 Conclusion and related work

The extension proposed in this paper to the original approach by Marco Cadoli is based upon adopting a heuristic algorithm which considers not only the vertices as possible offers but also internal points of the feasibility regions.

By comparing the proposed algorithm with the one reported in [10] we conclude that our work overcomes a number of problems, even though in our case the number of interactions it is no more proportional to the number of vertices. However, experiments show that our algorithm has a reasonable complexity in the average case, and in some cases it can be even more efficient than the original one.

The additional complexity, according to the tests, is a reasonable price to pay for the extra features and for the possibility of other extensions. In fact, the approach can be further extended, e.g. by adding new protocols, objective and utility functions. Furthermore, the approach can be improved by studying a peer-to-peer negotiation where the parts involved in the process use more than two issues, involving in such way a multidimensional space. We have been studying the possibility of considering as negotiation spaces not only convex areas but also non-convex ones. This result can be obtained by converting a non-convex region in a convex one and by excluding all points that are not part of the original negotiation area [30]. The proposed approach can be adopted either as a stand-alone strategy of negotiation or as a constraint-based technique, in the context of many general architectures like for instance the one in [17].

There is basic difference between our approach and other related research such as the one in [17]: these works introduce architectures that supports *one-to-one* and *one-to-many* negotiation, without however considering complexity issues. Instead, the nature of the offers and the selection of variables assignment in our approach are mainly aimed at obtaining a reasonable complexity in terms of number of interactions steps.

Among the various other approaches that can be found in the literature, several consider negotiation via *argumentation* and take therefore ac different point of view w.r.t. our *proposal-based* approach. In other works, such as [8] the negotiation is defined as a single-issue process. The speed of negotiation is tackled in [31], but in this research variables are boolean while in our approach are real numbers.

Acknowledgments

This work started as a cooperation with Marco Cadoli. Unfortunately, Marco left us in November 2006 after a long illness. This paper is dedicated to him.

References

1. Guttman, R.H., Moukas, A.G., Maes, P.: Agent-mediated electronic commerce: a survey. *Knowl. Eng. Rev.*, Cambridge University Press, New York, NY, USA **13**(2) (1998) 147–159
2. Guttman, R.H., Maes, P.: Cooperative vs. competitive multi-agent negotiations in retail electronic commerce. In: *CIA '98: Proceedings of the 2th International Workshop on Cooperative Information Agents II, Learning, Mobility and Electronic Commerce for Information Discovery on the Internet*, London, UK, Springer-Verlag (1998) 135–147
3. Kowalczyk, R., Bui, V.: On constraint-based reasoning in e-negotiation agents. In: *Agent-Mediated Electronic Commerce III, Current Issues in Agent-Based Electronic Commerce Systems* (includes revised papers from AMEC 2000 Workshop), London, UK, Springer-Verlag (2001) 31–46
4. Bozzano, M., Delzanno, G., Martelli, M., Mascardi, V., Zini, F.: Logic programming and multi-agent system: A synergic combination for applications and semantics (1999)
5. Walton, D.N., Krabbe, E.C.W.: *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. Albany : State University of New York Press (1995)
6. Governatori, G., ter Hofstede, A.H., Oaks, P.: Defeasible logic for automated negotiation. In Swatman, P., Swatman, P., eds.: *Proceedings of COLLECTeR*, Deakin University (2000) Published on CD.
7. Jennings, N.R., Parsons, S., Sierra, C., Faratin, P.: Automated negotiation. In *Proceedings of 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Systems (PAAM 2000)*
8. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering* **10**(5) (1998) 673–685
9. Karunatilake, N.C., Jennings, N.R., Rahwan, I., Norman, T.J.: Argument-based negotiation in a social context. In: *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, New York, NY, USA, ACM Press (2005) 1331–1332
10. Cadoli, M.: Proposal-based negotiation in convex regions. In: *CIA*. (2003) 93–108
11. Greenwald, A., The, E.: The 2002 trading agent competition: An overview of agent strategies. Amy Greenwald (ed.), *AI Magazine*. (2002)
12. Fatima, S.S., Wooldridge, M., Jennings, N.R.: Optimal negotiation strategies for agents with incomplete information. Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), John-Jules Meyer and Milind Tambe, *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pp. 53-68 (2001)
13. Rahwan, I., Sonenburg, L., Jennings, N.R., McBurney, P.: Stratum: A methodology for designing heuristic agent negotiation strategies. *International Journal of Applied Artificial Intelligence* 21 (2007)

14. Gomes, A.R.: Valuations and dynamics of negotiations. Rodney L. White Center for Financial Research Working Paper No. 21-99 (July 2004)
15. Conitzer, V., Sandholm, T.: Expressive negotiation in settings with externalities. In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05), pp. 255-260 (2005) Pittsburgh, Pennsylvania, USA.
16. Kazmer, D., Zhu, L., Hatch, D.: Process window derivation with an application to optical media manufacturing. *Journal of Manufacturing Science and Engineering* **123**, Issue 2 (May 2001) 301–311
17. Rahwan, I., Kowalczyk, R., Pham, H.H.: Intelligent agents for automated one-to-many e-commerce negotiation. In: ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2002) 197–204
18. Matos, N., Sierra, C., Jennings, N.R.: Determining successful negotiation strategies: An evolutionary approach. In Demazeau, Y., ed.: Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS-98), Paris, France, IEEE Press (1998) 182–189
19. Rahwan, I., McBurney, P., Sonenberg, L.: Towards a theory of negotiation strategy. I. Rahwan, P. McBurney, and L. Sonenberg. Towards a theory of negotiation strategy (a preliminary report). In S. Parsons and P. Gmytrasiewicz, editors, Proceedings of the 5th Workshop on Game Theoretic and Decision Theoretic Agents (GTDT-2003), pages 73–80, 2003. (2003)
20. Lopes, F., Mamede, N.J., Novais, A.Q., Coelho, H.: Negotiation strategies for autonomous computational agents. In: ECAI. (2004) 38–42
21. Somefun, K., Gerding, E., Bohte, S., La, H.: Automated negotiation and bundling of information goods. In: Proceedings of the 5th Workshop on AgentMediated Electronic Commerce (AMEC V), Melbourne, Australia, July. (2003)
22. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
23. Costantini, S., Tocchio, A.: The dali logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
24. Tocchio, A.: Multi-agent systems in computational logic. Ph.D. Thesis, Dipartimento di Informatica, Università degli Studi di L'Aquila (2005)
25. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: Declarative Agent Languages and Technologies. LNAI 3229. Springer-Verlag, Berlin (2006) Post-Proc. of DALT 2005.
26. Costantini, S., Tocchio, A., Verticchio, A.: A game-theoretic operational semantics for the dali communication architecture. In: Proc. of WOA04, Turin, Italy, December 2004, ISBN 88-371-1533-4. (2004)
27. Kowalski, A.: How to be artificially intelligent - the logical way. Draft, revised February 2004, Available on line (2006) <http://www.lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html>.
28. Lloyd, J.W.: Foundations of Logic Programming (Second, Extended Edition). Springer-Verlag, Berlin (1987)
29. Costantini, S., Tocchio, A., Tsintza, P.: Experimental evaluation of a heuristic approach for p2p negotiation. Submitted Paper (2007)
30. Siegel, A.: A historical review of the isoperimetric theorem in 2-d, and its place in elementary plane geometry ,<http://www.cs.nyu.edu/faculty/siegel/SCIAM.pdf>.

31. Wooldridge, M., Parson, S.: Languages for negotiation. In Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI 2000) pages 393-397, 2000.

Using Unfounded Sets for Computing Answer Sets of Programs with Recursive Aggregates

Mario Alviano, Wolfgang Faber, and Nicola Leone

Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
{alviano, faber, leone}@mat.unical.it

Abstract. Answer set programming with aggregates (ASP^A) allows for modelling many problems in a more concise way than with standard answer set programming (ASP). Previous works have focused on semantical and theoretical issues, and only few works have addressed computational issues, such as presenting algorithms and methods for implementing ASP^A .

In this paper, we fix a rich ASP language supporting recursive aggregates, and study means for implementing a reasoning engine for it. In particular, we leverage work on unfounded sets for ASP^A , and show how they can be used for characterizing and computing answer sets. Furthermore, we introduce an operator for effectively computing the greatest unfounded set (GUS), and study how it can be evaluated in a modular way, providing a means both for pruning the search space and for answer set checking. We have implemented these ideas and provide a brief sketch of the prototype architecture.

1 Introduction

The introduction of aggregate atoms [1–10] is one of the major syntactic extensions to Answer Set Programming of the recent years. While both semantic and computational properties of standard (aggregate-free) logic programs have been deeply investigated, relatively few works have focused on logic programs with aggregates; some of their semantic properties and their computational features are still far from being fully clarified. In particular, there is a lack of studies on algorithms and optimization methods for implementing recursive aggregates in ASP efficiently.

In this paper, we try to overcome this deficiency and make a step towards a more efficient implementation of recursive aggregates in ASP. To this end, we first focus on the properties of unfounded sets for programs with aggregates¹. Unfounded sets are at the basis of the implementation of virtually all currently available ASP solvers. Indeed, *native* ASP solvers like, e.g., DLV and Smodels, use unfounded sets for pruning the search space (through the well-founded operators); and *SAT-based* ASP solvers like, e.g., AS-SAT and Cmodels, use the related concept of loop formulas [13, 14] for checking. Thus, an in-depth study of the properties of unfounded sets for programs with aggregates is a valuable contribution for the implementation efficient ASP^A systems.

We provide a new notion of unfounded sets for ASP^A , explain how unfounded sets can be profitably employed both for pruning the search space and for checking answer

¹ We use the notion of unfounded sets in the sense of [11] rather than in the sense of [12]

sets in ASP^A computations. We then design an operator for computing the greatest unfounded set (GUS), and, in order to support a more efficient implementation, we provide a method for the modular computation of GUS, and sketch the architecture of our implementation of ASP^A .

We adopt the ASP^A semantics defined in [8], which seems to be receiving a consensus. Recent works, such as [15, 16] give further support for the plausibility of this semantics by relating it to established constructs for aggregate-free programs. In particular, [15] presented a semantics for very general programs, and showed that it coincides with [8] on ASP^A programs. We consider the rich ASP^A fragment allowing for disjunction, nonmonotonic negation, and both monotonic and anti-monotonic recursive aggregates; we denote this language by $DLP_{a,m}^A$.

Roughly, the main contributions of the paper are the following.

- We define a new and intuitive notion of unfounded set for $DLP_{a,m}^A$, relate it to previous notions showing also that it agrees with the standard notions of unfounded sets on aggregate-free programs, and characterize its properties.
- We show that unfounded sets can be profitably employed for pruning the search space in $DLP_{a,m}^A$ computations, by formally proving the properties of greatest unfounded sets (GUS) w.r.t. pruning.
- We demonstrate the formal properties allowing us to exploit greatest unfounded sets for answer-set checking in $DLP_{a,m}^A$ programs.
- We specify an operator $\mathcal{R}_{\mathcal{P},I}$ for computing the greatest unfounded sets.
- We design a modular evaluation technique for computing $\mathcal{R}_{\mathcal{P},I}$ component wise to allow for a more efficient implementation.
- We implement the above results in DLV, obtaining a system supporting the $DLP_{a,m}^A$ language, which is available for experimenting with recursive aggregates.²

To the best of our knowledge, our work provides the first implementation of recursive aggregates in disjunctive ASP. Previous implementations of aggregates in ASP either forbid recursive aggregates [5] or disallow disjunction [17, 18, 4, 10].³

2 Logic Programs with Aggregates

In this section, we recall syntax, semantics, and some basic properties of logic programs with aggregates.

2.1 Syntax

We assume that the reader is familiar with standard LP; we refer to the respective constructs as *standard atoms*, *standard literals*, *standard rules*, and *standard programs*. Two literals are said to be complementary if they are of the form p and $\text{not } p$ for some atom p . Given a literal L , $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$. For further background, see [20, 21].

² Note that before our extension DLV supported only nonrecursive aggregates.

³ Note that Cmodels [19] disallows aggregates in disjunctive rules.

Set Terms. A DLP^A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard atoms.⁴ A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where \bar{t} is a list of constants and $Conj$ is a ground (variable free) conjunction of standard atoms.

Aggregate Functions. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

Example 1. In the examples, we adopt the syntax of DLV to denote aggregates. Aggregate functions currently supported by the DLV system are: $\#count$ (number of terms), $\#sum$ (sum of non-negative integers), $\#times$ (product of positive integers), $\#min$ (minimum term), $\#max$ (maximum term)⁵.

Aggregate Literals. An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard. Also, an *aggregate atom* may have the form $T_1 \prec_1 f(S) \prec_2 T_2$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{<, \leq\}$, and T_1 and T_2 are terms.

Example 2. The following aggregate atoms are in DLV notation, where the latter contains a ground set and could be a ground instance of the former:

$$\#max\{Z : r(Z), a(Z, V)\} > Y \quad \#max\{\langle 2 : r(2), a(2, k) \rangle, \langle 2 : r(2), a(2, c) \rangle\} > 1$$

An *atom* is either a standard atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, L is an *aggregate literal*.

DLP^A Programs. A DLP^A *rule* r is a construct

$$a_1 \vee \dots \vee a_n :- b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m.$$

where a_1, \dots, a_n are standard atoms, b_1, \dots, b_m are atoms, and $n \geq 1, m \geq k \geq 0$. The disjunction $a_1 \vee \dots \vee a_n$ is referred to as the *head* of r while the conjunction $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$ is the *body* of r . We denote the set of head atoms by $H(r)$, and the set $\{b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m\}$ of the body literals by $B(r)$. $B^+(r)$ and $B^-(r)$ denote, respectively, the sets of positive and negative literals in $B(r)$. Note that this syntax does not explicitly allow integrity constraints (rules without head atoms). They can, however, be simulated in the usual way by using a new symbol and negation.

A DLP^A *program* is a set of DLP^A rules. In the sequel, we will often drop DLP^A, when it is clear from the context. A *global* variable of a rule r appears in a standard atom of r (possibly also in other atoms); all other variables are *local* variables.

⁴ Intuitively, a symbolic set $\{X : a(X, Y), p(Y)\}$ stands for the set of X -values making $a(X, Y), p(Y)$ true, i.e., $\{X \mid \exists Y \text{ s.t. } a(X, Y), p(Y) \text{ is true}\}$.

⁵ The first two aggregates roughly correspond, respectively, to the cardinality and weight constraint literals of Smodels. $\#min$ and $\#max$ are undefined for empty set.

Safety. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r ; (ii) each local variable of r appearing in a symbolic set $\{Vars : Conj\}$ appears in an atom of $Conj$; (iii) each guard of an aggregate atom of r is a constant or a global variable. A program \mathcal{P} is safe if all $r \in \mathcal{P}$ are safe. In the following we assume that DLP^A programs are safe.

2.2 Answer Set Semantics

Universe and Base. Given a DLP^A program \mathcal{P} , let $U_{\mathcal{P}}$ denote the set of constants appearing in \mathcal{P} , and $B_{\mathcal{P}}$ be the set of standard atoms constructible from the (standard) predicates of \mathcal{P} with constants in $U_{\mathcal{P}}$.

Instantiation. A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r* ; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for S* . Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of S* is the following ground set of pairs $inst(S)$:

$\{\langle \gamma(Vars) : \gamma(Conj) \rangle \mid \gamma \text{ is a local substitution for } S\}$.⁶

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r ; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program \mathcal{P} is the set of all possible instances of the rules of \mathcal{P} .

Interpretations. An *interpretation* for a DLP^A program \mathcal{P} is a consistent set of standard ground literals, that is $I \subseteq (B_{\mathcal{P}} \cup \neg B_{\mathcal{P}})$ such that $I \cap \neg I = \emptyset$. A standard ground literal L is true (resp. false) w.r.t I if $L \in I$ (resp. $L \in \neg I$). If a standard ground literal is neither true nor false w.r.t I then it is undefined w.r.t I . We denote by I^+ (resp. I^-) the set of all atoms occurring in standard positive (resp. negative) literals in I . We denote by \bar{I} the set of undefined atoms w.r.t. I (i.e. $B_{\mathcal{P}} \setminus I^+ \cup I^-$). An interpretation I is *total* if \bar{I} is empty (i.e., $I^+ \cup \neg I^- = B_{\mathcal{P}}$), otherwise I is *partial*. A *totalization* of a (partial) interpretation I is a total interpretation J containing I , (i.e., J is a total interpretation and $I \subseteq J$).

An interpretation also provides a meaning for aggregate literals. Their truth value is first defined for total interpretations, and then generalized to partial ones.

Let I be a total interpretation. A standard ground conjunction is true (resp. false) w.r.t I if all (resp. some) of its literals are true (resp. false). The meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. I . More precisely, let $I(S)$ denote the multiset $[t_1 \mid \langle t_1, \dots, t_n : Conj \rangle \in S \wedge Conj \text{ is true w.r.t. } I]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f , $I(f(S)) = \perp$ (where \perp is a fixed symbol not occurring in \mathcal{P}).

⁶ Given a substitution σ and a DLP^A object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each occurrence of variable X in Obj by $\sigma(X)$.

A ground aggregate atom A of the form $f(S) \prec k$ is *true w.r.t. I* if: (i) $I(f(S)) \neq \perp$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal $\text{not } A = \text{not } f(S) \prec k$ is *true w.r.t. I* if (i) $I(f(S)) \neq \perp$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, $\text{not } A$ is false.

If I is a *partial* interpretation, an aggregate literal A is true (resp. false) w.r.t. I if it is true (resp. false) w.r.t. *each* totalization J of I ; otherwise it is undefined.

Example 3. Consider the atom $A = \#\text{sum}\{\langle 1:p(2,1) \rangle, \langle 2:p(2,2) \rangle\} > 1$. Let S be the ground set in A . For the interpretation $I = \{p(2,2)\}$, each extending total interpretation contains either $p(2,1)$ or not $p(2,1)$. Therefore, either $I(S) = [2]$ or $I(S) = [1, 2]$ and the application of $\#\text{sum}$ yields either 2 or 3, hence A is true w.r.t. I , since they are both greater than 1.

Our definitions of interpretation and truth values preserve “knowledge monotonicity”. If an interpretation J extends I (i.e., $I \subseteq J$), then each literal which is true w.r.t. I is true w.r.t. J , and each literal which is false w.r.t. I is false w.r.t. J as well.

Minimal Models. Given an interpretation I and a ground rule r , the head of r is *true w.r.t. I* if some literal in $H(r)$ is true w.r.t. I ; the body of r is *true w.r.t. I* if all literals in $B(r)$ are true w.r.t. I ; rule r is *satisfied w.r.t. I* if its head is true w.r.t. I whenever its body is true w.r.t. I . A total interpretation M is a *model* of a DLP^A program \mathcal{P} if all $r \in \text{Ground}(\mathcal{P})$ are satisfied w.r.t. M . A model M for \mathcal{P} is (subset) *minimal* if no model N for \mathcal{P} exists such that $N^+ \subset M^+$. Note that, under these definitions, the word *interpretation* refers to a possibly partial interpretation, while a *model* is always a total interpretation.

Answer Sets. We now recall the generalization of the Gelfond-Lifschitz transformation and answer sets for DLP^A programs from [8]: Given a ground DLP^A program \mathcal{P} and a total interpretation I , let \mathcal{P}^I denote the transformed program obtained from \mathcal{P} by deleting all rules in which a body literal is false w.r.t. I . I is an answer set of a program \mathcal{P} if it is a minimal model of $\text{Ground}(\mathcal{P})^I$.

Example 4. Consider interpretation $I_1 = \{p(a)\}$, $I_2 = \{\text{not } p(a)\}$ and two programs $P_1 = \{p(a) :- \#\text{count}\{X : p(X)\} > 0.\}$ and $P_2 = \{p(a) :- \#\text{count}\{X : p(X)\} < 1.\}$.

$\text{Ground}(P_1) = \{p(a) :- \#\text{count}\{a : p(a)\} > 0.\}$ and $\text{Ground}(P_1)^{I_1} = \text{Ground}(P_1)$, $\text{Ground}(P_1)^{I_2} = \emptyset$. Furthermore, $\text{Ground}(P_2) = \{p(a) :- \#\text{count}\{a : p(a)\} < 1.\}$, and $\text{Ground}(P_2)^{I_1} = \emptyset$, $\text{Ground}(P_2)^{I_2} = \text{Ground}(P_2)$ hold.

I_2 is the only answer set of P_1 (since I_1 is not a minimal model of $\text{Ground}(P_1)^{I_1}$), while P_2 admits no answer set (I_1 is not a minimal model of $\text{Ground}(P_2)^{I_1}$, and I_2 is not a model of $\text{Ground}(P_2) = \text{Ground}(P_2)^{I_2}$).

Note that any answer set A of \mathcal{P} is also a model of \mathcal{P} because $\text{Ground}(\mathcal{P})^A \subseteq \text{Ground}(\mathcal{P})$, and rules in $\text{Ground}(\mathcal{P}) \setminus \text{Ground}(\mathcal{P})^A$ are satisfied w.r.t. A .

Monotonicity. Given two interpretations I and J we say that $I \leq J$ if $I^+ \subseteq J^+$ and $J^- \subseteq I^-$. A ground literal ℓ is *monotone*, if for all interpretations I, J , such that $I \leq J$, we have that: (i) ℓ true w.r.t. I implies ℓ true w.r.t. J , and (ii) ℓ false w.r.t. J implies ℓ

false w.r.t. I . A ground literal ℓ is *antimonotone*, if the opposite happens, that is, for all interpretations I, J , such that $I \leq J$, we have that: (i) ℓ true w.r.t. J implies ℓ true w.r.t. I , and (ii) ℓ false w.r.t. I implies ℓ false w.r.t. J . A ground literal ℓ is *nonmonotone*, if it is neither monotone nor antimonotone.

Note that positive standard literals are monotone, whereas negative standard literals are antimonotone. Aggregate literals may be monotone, antimonotone or nonmonotone, regardless whether they are positive or negative. Nonmonotone literals include the sum over (possibly negative) integers and the average.

Example 5. All ground instances of $\#count\{Z : r(Z)\} > 1$ and not $\#count\{Z : r(Z)\} < 1$ are monotone, while for $\#count\{Z : r(Z)\} < 1$, and not $\#count\{Z : r(Z)\} > 1$ they are antimonotone.

We denote by $DLP_{m,a}^A$ the fragment of DLP^A in which monotone and antimonotone literals may occur. In the following, by *program* we will usually refer to a $DLP_{m,a}^A$ program. Given a rule r of a $DLP_{m,a}^A$ program, we denote with $Mon(B(r))$ and $Ant(B(r))$, respectively, the set of *monotone* and *antimonotone* literals in $B(r)$. Note that, as described in [22], many programs with nonmonotone literals can be polynomially rewritten into $DLP_{m,a}^A$ programs. Some important examples include programs containing aggregate atoms of the form $T_1 \prec_1 f(S) \prec_2 T_2$ and $f(S) = T$ (which per se are nonmonotone independent of $f(S)$), which can be rewritten to conjunctions $T_1 \prec_1 f(S), f(S) \prec_2 T_2$ and $f(S) \geq T, f(S) \leq T$, respectively.

3 Unfounded Sets

We now give a definition of unfounded set for DLP^A programs with monotone and antimonotone aggregates, extending the one of [16].

In the following we denote by $S_1 \dot{\cup} \neg.S_2$ the set $(S_1 \setminus S_2) \cup \neg.S_2$, where S_1 and S_2 are sets of standard ground literals.

Definition 1 (Unfounded Set). A set X of ground atoms is an unfounded set for a $DLP_{a,m}^A$ program \mathcal{P} w.r.t. an interpretation I if, for each rule $r \in \mathcal{P}$ such that $H(r) \cap X \neq \emptyset$, at least one of the following conditions holds:

1. $Ant(B(r))$ is false w.r.t. I .
2. $Mon(B(r))$ is false w.r.t. $I \dot{\cup} \neg.X$.
3. $H(r)$ is true w.r.t. $I \dot{\cup} \neg.X$.

While condition 1 declares that rule satisfaction does not depend on atoms in X , conditions 2 and 3 ensure that the rule is satisfied also if the atoms in X are switched to false. Note that condition 3 is equivalent to $(H(r) \setminus X) \cap I \neq \emptyset$, and \emptyset is always an unfounded set, independent of interpretation and program.

Example 6. Consider the following program P :

$$a(1) \vee a(2). \quad a(1) :- \#count\{\{1:a(2)\}\} > 1. \quad a(2) :- \#count\{\{1:a(1)\}\} > 1.$$

and $I = \{a(1), a(2)\}$. Then $\{a(1)\}$ and $\{a(2)\}$ are unfounded sets for P w.r.t. I . $\{a(1), a(2)\}$ is not an unfounded set for P w.r.t. I , as for the first rule none of the three conditions holds.

Theorem 1. *A set X of ground atoms is an unfounded set for a $\text{DLP}_{a,m}^A$ program \mathcal{P} w.r.t. an interpretation I according to Def. 1 iff it is an unfounded set for \mathcal{P} w.r.t. I according to Def. 1 of [23].*

Proof. According to Def. 1 of [23], a set X of ground atoms is an unfounded set for a program \mathcal{P} w.r.t. an interpretation I if, for each rule r in $\text{Ground}(\mathcal{P})$ having some atoms from X in the head, at least one of the following conditions holds: a) some literal of $B(r)$ is false w.r.t. I , b) some literal of $B(r)$ is false w.r.t. $I \dot{\cup} \neg.X$, or c) some atom of $H(r) \setminus X$ is true w.r.t. I .

First of all, let us observe that conditions 1 and 2 of Def. 1 trivially imply conditions a) and b), respectively, and that, as noted earlier, condition 3 of Def. 1 is equivalent to condition c).

Now, observe that if a monotone body literal is false w.r.t. I , it is also false w.r.t. $I \dot{\cup} \neg.X$. In a similar way, if an antimonotone body literal of r is false w.r.t. $I \dot{\cup} \neg.X$, then it is false also w.r.t. I . Therefore, if condition a) holds for a monotone literal, condition 2 holds for this literal; if condition a) holds for an antimonotone literal, condition 1 holds. Likewise, if condition b) holds for a monotone literal, condition 2 holds; if condition b) holds for an antimonotone literal, condition 1 holds.

Thus, on $\text{DLP}_{m,a}^A$ our definition of unfounded set specializes Def. 1 of [23] by imposing stricter properties in conditions 1 and 2.

From this equivalence and results in [23] it follows that unfounded sets as defined in Def. 1 also coincide with other definitions of unfounded sets for various language fragments.

Corollary 1. *For a non-disjunctive, aggregate-free program \mathcal{P} and an interpretation I , any unfounded set w.r.t. Def. 1 is a standard unfounded set (as defined in [11]).*

For an aggregate-free program \mathcal{P} and interpretation I , any unfounded set w.r.t. Def. 1 is an unfounded set as defined in [24].

For a non-disjunctive $\text{LP}_{m,a}^A$ program \mathcal{P} and an interpretation I , any unfounded set w.r.t. Def. 1 is an unfounded set as defined in [16].

We next state an important monotonicity property of unfounded sets.

Proposition 1. *Let I be a partial interpretation for a $\text{DLP}_{m,a}^A$ program \mathcal{P} and X an unfounded set for \mathcal{P} w.r.t. I . Then, for each $J \supseteq I$, X is an unfounded set for \mathcal{P} w.r.t. J as well.*

Proof. If X is an unfounded set for \mathcal{P} w.r.t. I , then for each $a \in X$ and for each $r \in \mathcal{P}$ with $a \in H(r)$, (1) $\text{Ant}(B(r))$ is false w.r.t. I , or (2) $\text{Mon}(B(r))$ is false w.r.t. $I \dot{\cup} \neg.X$, or (3) $H(r)$ is true w.r.t. $I \dot{\cup} \neg.X$ holds. Now, note that since $I \subseteq J$ holds, then also $I \dot{\cup} \neg.X \subseteq J \dot{\cup} \neg.X$ holds. So, if (1) holds, it holds also for J , and if (2) or (3) hold, then they hold also for $J \dot{\cup} \neg.X$.

We next define the central notion in the remainder of this work, the *Greatest Unfounded Set* (GUS), as the union of all unfounded sets.

Definition 2. *Let I be an interpretation for a program \mathcal{P} . Then, let $GUS_{\mathcal{P}}(I)$ (the GUS for \mathcal{P} w.r.t. I) denote the union of all unfounded sets for \mathcal{P} w.r.t. I .*

From Proposition 1 it follows that the GUS of an interpretation I is always contained in the GUS of a superset of I .

Proposition 2. *Let I be an interpretation for a program \mathcal{P} . Then, $GUS_{\mathcal{P}}(I) \subseteq GUS_{\mathcal{P}}(J)$, for each $J \supseteq I$.*

Note that despite its name, the GUS is not always guaranteed to be an unfounded set. In the non-disjunctive case, the union of two unfounded sets is an unfounded set as well, also in presence of monotone and antimonotone aggregates [16], and so for these programs a GUS is necessarily an unfounded set. However, in the presence of disjunctive rules, this property does no longer hold, as shown in [24]. Therefore it also does not hold for $DLP_{m,a}^A$, and as a consequence a GUS need not be an unfounded set.

Observation 2 *If X_1 and X_2 are unfounded sets for a $DLP_{m,a}^A$ program \mathcal{P} w.r.t. I , then $X_1 \cup X_2$ is not necessarily an unfounded set.*

By virtue of Theorem 1, Proposition 1 of [23] carries over to unfounded sets of Definition 1.

Proposition 3. *If X_1 and X_2 are unfounded sets for a program \mathcal{P} w.r.t. I and both $X_1 \cap I = \emptyset$ and $X_2 \cap I = \emptyset$ hold, then $X_1 \cup X_2$ is an unfounded set for \mathcal{P} w.r.t. I .*

This allows for defining the class of unfounded-free interpretations for which the GUS is guaranteed to be an unfounded set.

Definition 3 (Unfounded-free Interpretation). *Let I be an interpretation for a program \mathcal{P} . I is unfounded-free if $I \cap X = \emptyset$ for each unfounded set X for \mathcal{P} w.r.t. I .*

As an easy consequence we obtain:

Proposition 4. *Let I be an unfounded-free interpretation for a program \mathcal{P} . Then, $GUS_{\mathcal{P}}(I)$ is an unfounded set.*

Next, we show an interesting property for total interpretations.

Proposition 5. *Let I be a total interpretation for a program \mathcal{P} . Then, I is unfounded-free iff no non-empty set $X \subseteq I^+$ is an unfounded set for \mathcal{P} w.r.t. I .*

Proof. (\implies) If a non-empty subset Y of I^+ is an unfounded set for \mathcal{P} w.r.t. I , then I is not unfounded-free.

(\impliedby) If I is not unfounded-free, then there exists a non-empty subset of I^+ which is an unfounded set for \mathcal{P} w.r.t. I . Let X be an unfounded set for \mathcal{P} w.r.t. I such that $Y = X \cap I \neq \emptyset$. Note that $I \dot{\cup} \neg.X = I \dot{\cup} \neg.Y$, then Y is also an unfounded set for \mathcal{P} w.r.t. I .

4 Answer Set Checking via Unfounded Sets

Unfounded sets can be used to characterize models and answer sets; these characterizations can be profitably used for answer set checking. Given Theorem 1, the following results are consequences of Theorem 4 and Corollary 6 of [23].

Proposition 6. *Let M be a total interpretation for a program \mathcal{P} . Then M is a model for \mathcal{P} iff M^- is an unfounded set for \mathcal{P} w.r.t. M .*

Proposition 7. *Let M be a model for \mathcal{P} . M is an answer-set for \mathcal{P} iff M is unfounded-free for \mathcal{P} .*

Furthermore, we can show that unfounded sets also characterize minimal models.

Proposition 8. *Let M be a model for a positive program \mathcal{P} . M is a minimal model for \mathcal{P} iff it is unfounded-free.*

Proof. (\Leftarrow) If M is not minimal then there exists another model M_1 such that $M_1^+ \subset M^+$, and so $X = M^+ \setminus M_1^+ \neq \emptyset$. Then, for each $r \in \mathcal{P}$ such that $H(r) \cap X \neq \emptyset$, (i) $H(r) \cap M_1^+ \neq \emptyset$, or (ii) $Ant(B(r))$ is false w.r.t. M_1 , or (iii) $Mon(B(r))$ is false w.r.t. M_1 . Note that $M_1 = (M \setminus X) \cup \neg.X = M \dot{\cup} \neg.X$, and then: from (i) follows that $H(r)$ is true w.r.t. $M \dot{\cup} \neg.X$, from (ii) follows that $Ant(B(r))$ is false w.r.t. M (because $M_1 \leq M$), from (iii) follows that $Mon(B(r))$ is false w.r.t. $M \dot{\cup} \neg.X$. So, X is an unfounded set for \mathcal{P} w.r.t. M , and then M is not unfounded-free.

(\Rightarrow) Assume, by contradiction, that M is not unfounded-free. Then, by Proposition 5, there exists a non-empty $X \subseteq M^+$ which is an unfounded set for \mathcal{P} w.r.t. M . Now, we show that the total interpretation $M_1 = M \dot{\cup} \neg.X$ is a model for \mathcal{P} (contradicting the minimality of M). Let r be a rule of \mathcal{P} such that $H(r)$ is true w.r.t. M , and $H(r)$ is false w.r.t. M_1 . Then, $H(r) \cap X \neq \emptyset$. But X is an unfounded set for \mathcal{P} w.r.t. M , then (1) $Ant(B(r))$ is false w.r.t. M (and then it is false w.r.t. M_1 , because $M_1 \leq M$), or (2) $Mon(B(r))$ is false w.r.t. $M \dot{\cup} \neg.X = M_1$, or (3) $H(r)$ is true w.r.t. $M \dot{\cup} \neg.X = M_1$. Note that (3) cannot hold by assumption. Then, r is satisfied w.r.t. M_1 by body, contradicting the minimality of M .

We next show that the condition of being unfounded-free is invariant for a program and its reduct.

Lemma 1. *Let M be a total interpretation for a program \mathcal{P} . M is unfounded-free for \mathcal{P} iff it is unfounded-free for \mathcal{P}^M .*

Proof. (\Rightarrow) If X is not an unfounded set for \mathcal{P} w.r.t. M , then for each $a \in X$ there exists $r \in \mathcal{P}$ such that r violates all condition of Definition 1. Then, from condition (1) and (2), $B(r)$ is true w.r.t. M . Therefore, the image r' of r is in \mathcal{P}^M . Clearly, r' violates all conditions of Definition 1 for \mathcal{P}^M w.r.t. M . Now, if M is unfounded-free for \mathcal{P} , then, by Proposition 5, every non-empty $X \subseteq M^+$ is not an unfounded set for \mathcal{P} w.r.t. M , and then it is not an unfounded set for \mathcal{P}^M w.r.t. M . So, M is unfounded-free for \mathcal{P}^M . (\Leftarrow) Let X be an unfounded set for \mathcal{P} w.r.t. M . Then, for each $a \in X$ and for each $r \in \mathcal{P}$ with $a \in H(r)$, (1) $Ant(B(r))$ is false w.r.t. M , or (2) $Mon(B(r))$ is false w.r.t.

$M \dot{\cup} \neg.X$, or (3) $H(r)$ is true w.r.t. $M \dot{\cup} \neg.X$. Case (1) or (2) imply that r has no image in \mathcal{P}^M or condition (2) holds for r' . Case (3) imply that condition (3) holds also for r' . So, X is an unfounded set also for \mathcal{P}^M w.r.t. M . Therefore, if M is not unfounded-free for \mathcal{P} , then it is not unfounded-free for \mathcal{P}^M . It follows that M unfounded-free for \mathcal{P}^M implies M unfounded-free for \mathcal{P} .

Furthermore, $GUS_{\mathcal{P}}(I)$ permits to check whether I is unfounded-free, and then whether it is an answer set.

Theorem 3. *Let I be a total interpretation for a program \mathcal{P} . I is unfounded-free if and only if $I^- = GUS_{\mathcal{P}}(I)$.*

Proof. (\Leftarrow) It is easy to see that each unfounded set X for \mathcal{P} w.r.t. I is a subset of I^- , and then $I \cap X = \emptyset$ holds.

(\Rightarrow) For each unfounded set X for \mathcal{P} w.r.t. I , $I \cap X = \emptyset$ holds. Since I is total, this is equivalent to $X \subseteq I^-$, and then $GUS_{\mathcal{P}}(I) \subseteq I^-$. By Proposition 6, it follows that $I^- \subseteq GUS_{\mathcal{P}}(I)$, and then $I^- = GUS_{\mathcal{P}}(I)$.

Corollary 2. *Given a total interpretation I for a program \mathcal{P} , I is an answer set if and only if $I^- = GUS_{\mathcal{P}}(I)$ and I^- is an unfounded set w.r.t. \mathcal{P} and I .*

These results allow for checking whether a model or an interpretation is an answer set just by using the notion of unfounded sets.

5 Pruning via Unfounded Sets

In this section we show some properties of $GUS_{\mathcal{P}}(I)$, which may be used during the computation of the answer sets, for pruning the search space and detecting useless branches of the computation.

Theorem 4. *Given an interpretation I for a program \mathcal{P} , if $I \cap GUS_{\mathcal{P}}(I) \neq \emptyset$, then no totalization of I is an answer set for \mathcal{P} .*

Proof. If $I \cap GUS_{\mathcal{P}}(I) \neq \emptyset$, then there exists an unfounded set X for \mathcal{P} w.r.t. I such that $I \cap X \neq \emptyset$. Let J be a totalization of I . Then, by Proposition 1, X is an unfounded set for \mathcal{P} w.r.t. J . Clearly, $J \cap X \neq \emptyset$, so J is not unfounded-free. By Proposition 7 J is not an answer-set for \mathcal{P} .

Thus, during the construction of answer sets one may want to compute the GUS with respect to the interpretation so far and test whether it contains some element of the interpretation. If so, one should abandon the construction and backtrack, as no answer set can be found in the current branch. Moreover, the GUS also serves as an inference operator for pruning the search space.

Theorem 5. *Given an interpretation I for a program \mathcal{P} , if J is an answer set containing I , then J contains $I \dot{\cup} \neg.GUS_{\mathcal{P}}(I)$ as well.*

Proof. Assume $J \not\supseteq I \dot{\cup} \neg.GUS_{\mathcal{P}}(I)$ then $J \cap GUS_{\mathcal{P}}(I) \neq \emptyset$. From Proposition 2 it follows that $J \cap GUS_{\mathcal{P}}(J) \neq \emptyset$, and then, by Theorem 4, J is not an answer set for \mathcal{P} .

In other words, $\neg.GUS_{\mathcal{P}}(I)$ is contained in all answer sets extending I , so when constructing answer set candidates we can safely add these literals to the candidate.

6 Computing Greatest Unfounded Sets

We now define an operator for computing the Greatest Unfounded Set of a $DLP_{m,a}^A$ program \mathcal{P} w.r.t. an interpretation I : the operator $\mathcal{R}_{\mathcal{P},I}$ that, given a set X of ground atoms, discards the elements in X that do not satisfy any of the unfoundedness conditions of Definition 1.

Definition 4. Let \mathcal{P} be a $DLP_{m,a}^A$ program and I an interpretation. Then we define the operator $\mathcal{R}_{\mathcal{P},I}$ as a mapping $2^{B_{\mathcal{P}}} \rightarrow 2^{B_{\mathcal{P}}}$ as follows:

$$\mathcal{R}_{\mathcal{P},I}(X) = \{a \in X \mid \forall r \in \text{ground}(\mathcal{P}) \text{ with } a \in H(r), \text{Ant}(B(r)) \text{ is false w.r.t. } I, \\ \text{or } \text{Mon}(B(r)) \text{ is false w.r.t. } I \dot{\cup} \neg.X, \\ \text{or } H(r) \text{ is true w.r.t. } I \dot{\cup} \neg.\{a\}\}$$

Given a set $X \subseteq B_{\mathcal{P}}$, the sequence $R_0 = X$, $R_n = \mathcal{R}_{\mathcal{P},I}(R_{n-1})$ decreases monotonically and converges finitely to a limit that we denote by $\mathcal{R}_{\mathcal{P},I}^{\omega}(X)$. We next show that $\mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$ is an unfounded set, and we can therefore use this operator to detect undefined atoms that can be safely switched to false, reducing the search space.

Proposition 9. Given a $DLP_{m,a}^A$ program \mathcal{P} and an interpretation I , $\mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$ is an unfounded set for \mathcal{P} w.r.t. I .

Proof. Let $X = \mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$. By definition of $\mathcal{R}_{\mathcal{P},I}$, we have $X \subseteq B_{\mathcal{P}} \setminus I$, and hence $X \cap I = \emptyset$. Now, for each $a \in X$ and for each $r \in \mathcal{P}$, $a \in H(r)$ implies that $\text{Ant}(B(r))$ is false w.r.t. I , or $\text{Mon}(B(r))$ is false w.r.t. $I \dot{\cup} \neg.X$, or $H(r)$ is true w.r.t. $I \dot{\cup} \neg.\{a\}$. If the last holds, since $X \cap I = \emptyset$, also $H(r)$ is true w.r.t. $I \dot{\cup} \neg.X$. Then, X is an unfounded set for \mathcal{P} w.r.t. I .

Importantly, $\mathcal{R}_{\mathcal{P},I}$ does not discard any unfounded set contained in the input set.

Proposition 10. Let \mathcal{P} be a $DLP_{m,a}^A$ program, I be an interpretation for \mathcal{P} , and $J \subseteq B_{\mathcal{P}}$. Every unfounded set for \mathcal{P} w.r.t. I which is contained in J is also contained in $\mathcal{R}_{\mathcal{P},I}^{\omega}(J)$.

Proof. Let $X \subseteq J$ be an unfounded set for \mathcal{P} w.r.t. I . For each $a \in X$ and for each rule $r \in \mathcal{P}$ such that $a \in H(r)$, $\text{Ant}(B(r))$ is false w.r.t. I , or $\text{Mon}(B(r))$ is false w.r.t. $I \dot{\cup} \neg.X$, or $H(r)$ is true w.r.t. $I \dot{\cup} \neg.X$ holds. If the last holds, since $\{a\} \subseteq X$, $H(r)$ is true w.r.t. $I \dot{\cup} \neg.\{a\}$ as well. Then, from the definition of $\mathcal{R}_{\mathcal{P},I}$, $\mathcal{R}_{\mathcal{P},I}(X) = X$ holds and, since X is monotonic and $X \subseteq J$, $\mathcal{R}_{\mathcal{P},I}^{\omega}(J)$ must contain X .

Using the above propositions, we can prove that $\mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$ computes the greatest unfounded set for \mathcal{P} w.r.t. I .

Theorem 6. Let \mathcal{P} be a $DLP_{m,a}^A$ program and I an unfounded-free interpretation for it. Then, $\mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I) = \text{GUS}_{\mathcal{P}}(I)$.

Proof. (\supseteq) Since I is unfounded-free, $I \cap X = \emptyset$ holds for each unfounded set for \mathcal{P} w.r.t. I , and then $X \subseteq B_{\mathcal{P}} \setminus I$. So, by Proposition 10, also $X \subseteq \mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$ holds, and then $\text{GUS}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ is contained in $\mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$.

(\subseteq) By Proposition 9, $\mathcal{R}_{\mathcal{P},I}^{\omega}(B_{\mathcal{P}} \setminus I)$ is an unfounded set for \mathcal{P} w.r.t. I , and then, by definition of GUS, it is contained in $\text{GUS}_{\mathcal{P}}(I)$.

It is easy to see that the fixpoint of the $\mathcal{R}_{\mathcal{P},I}$ operator is efficiently computable. Thus, from the above theorem one can employ $\mathcal{R}_{\mathcal{P},I}$ as powerful and efficient pruning operator for unfounded-free interpretations. Actually, on the large class of head-cycle free programs [25], the $\mathcal{R}_{\mathcal{P},I}$ allows us to always compute the greatest unfounded set, even if the interpretation is not unfounded-free, and can be therefore employed both for pruning and answer-set checking. In the next section, we show how this can be done in an efficient way providing an algorithm for the modular computation of GUS via $\mathcal{R}_{\mathcal{P},I}$.

7 Modular Evaluation of Greatest Unfounded Sets

In this section, we show how we can localize the computation of unfounded sets. To this end, we define the notion of *dependency graph*, the strongly connected components of which define the modules, on which the local computation will work.

With every ground program \mathcal{P} , we associate a directed graph $DG_{\mathcal{P}} = (\mathcal{N}, E)$, called the *dependency graph* of \mathcal{P} , in which (i) each atom of \mathcal{P} is a node in \mathcal{N} and (ii) there is an arc in E directed from a node a to a node b iff there is a rule r in \mathcal{P} such that $b \in H(r)$ and a is a standard atom in $Mon(B(r))$ or an atom appearing in the ground set of an aggregate literal in $Mon(B(r))$.

An important and well-known class of programs are *head-cycle-free (HCF)* programs: A program \mathcal{P} is HCF iff there is no rule r in \mathcal{P} such that two predicates occurring in the head of r are in the same cycle of $DG_{\mathcal{P}}$. In our implementation for $DLP_{m,a}^A$, described in Section 8, we consider only HCF programs. This class of programs has recently been shown to be the largest class of programs for which standard reasoning tasks are still in NP (cf. [26]). The main result of this section, Theorem 7, is therefore also stated for HCF programs.

We can partition the set of ground atoms occurring in \mathcal{P} in strongly connected components. Two atoms a and b are in the same component if there is both a path from a to b and a path from b to a in $DG_{\mathcal{P}}$. Also, we can define a partial order \preceq for components: $C_1 \preceq C_2$ iff there exist $a \in C_1, b \in C_2$ such that there is a path from a to b . Moreover, the subprogram $\mathcal{P}_C \subseteq \mathcal{P}$ associated with a component C consists of all rules \mathcal{P} which contain an atom of C in their heads. Before introducing the algorithm, we show some properties that hold for the $\mathcal{R}_{\mathcal{P},I}$ operator.

Lemma 2. *Let \mathcal{P} be a program and I be an interpretation. For each sets X and Y such that $X \subseteq Y$, $\mathcal{R}_{\mathcal{P},I}^\omega(X) \subseteq \mathcal{R}_{\mathcal{P},I}^\omega(Y)$ holds.*

Proof. By induction. The only condition of Def. 4 that depends on the starting set X is “ $Mon(B(r))$ is false w.r.t. $I \dot{\cup} \neg.X$ ”. If this holds for some atom in $\mathcal{R}_{\mathcal{P},I}(X)$ and some rule r in \mathcal{P} , then $Mon(B(r))$ is false also w.r.t. $I \dot{\cup} \neg.Y$, because $I \dot{\cup} \neg.Y \leq I \dot{\cup} \neg.X$. So, $\mathcal{R}_{\mathcal{P},I}(X) \subseteq \mathcal{R}_{\mathcal{P},I}(Y)$. Assuming $\mathcal{R}_{\mathcal{P},I}^{(i)}(X) \subseteq \mathcal{R}_{\mathcal{P},I}^{(i)}(Y)$, then $\mathcal{R}_{\mathcal{P},I}^{(i+1)}(X) = \mathcal{R}_{\mathcal{P},I}(\mathcal{R}_{\mathcal{P},I}^{(i)}(X)) \subseteq \mathcal{R}_{\mathcal{P},I}(\mathcal{R}_{\mathcal{P},I}^{(i)}(Y)) = \mathcal{R}_{\mathcal{P},I}^{(i+1)}(Y)$.

Lemma 3. *Let \mathcal{P} be a program and I an interpretation, C a component of \mathcal{P} and \mathcal{P}_C the subprogram associated to C . Then, for each $X \subseteq C$, $\mathcal{R}_{\mathcal{P}_C,I}(X) = \mathcal{R}_{\mathcal{P},I}(X)$.*

Proof. Clearly, each rule r of \mathcal{P} with $H(r) \cap X \neq \emptyset$ is also in \mathcal{P}_C .

Theorem 7. Let C_1, C_2, \dots, C_n be a total order for the components of an HCF program \mathcal{P} such that $C_i \preceq C_j$ implies $i \leq j$. Starting from $I_0 := I$ and then, for each $i = 1, \dots, n$, computing $X_i := \mathcal{R}_{\mathcal{P}_{C_i}, I_{i-1}}^\omega(C_i \setminus I)$, $I_i := I_{i-1} \cup \neg.X_i$, it holds that I_n is equal to $I \cup \neg.GUS_{\mathcal{P}}(I)$.

Proof. We prove that at each step of the computation $X_i = \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_i$ holds. Base (\subseteq). From Lemma 3 and Lemma 2 it follows that $X_1 = \mathcal{R}_{\mathcal{P}_{C_1}, I}^\omega(C_1 \setminus I) = \mathcal{R}_{\mathcal{P}, I}^\omega(C_1 \setminus I) \subseteq \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I)$. So, $X_1 \subseteq \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_1$, because $X_1 \subseteq C_1$. Base (\supseteq). For each $a \in \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_1$ and for each $r \in P$ with $a \in H(r)$, (1) $Ant(B(r))$ is false w.r.t. I , or (2) $Mon(B(r))$ is false w.r.t. $I \dot{\cup} \neg.\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I)$, or (3) $H(r)$ is true w.r.t. $I \dot{\cup} \neg.\{a\}$. Note that, since \mathcal{P} is HCF, a is the only atom in $H(r)$ belonging to C_1 , so from (3) it follows that $H(r)$ is true w.r.t. $I \dot{\cup} \neg.(\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_1)$. Also, note that $Mon(B(r))$ depends only on atoms in C_1 . Then, from (2) it follows that $Mon(B(r))$ is false w.r.t. $\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_1$. So, $\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_1$ is an unfounded set for \mathcal{P}_{C_1} w.r.t. I and, by Def. 4, it is a subset of X_1 .

Suppose that $X_i = \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_i$.

(\subseteq) For each $a \in X_{i+1}$ and for each $r \in \mathcal{P}_{C_{i+1}}$ with $a \in H(r)$, $Ant(B(r))$ is false w.r.t. I_i (and w.r.t. I because $I_i \leq I$), or $Mon(B(r))$ is false w.r.t. $I_i \dot{\cup} \neg.X_{i+1}$ ($= I \dot{\cup} \neg.(X_{i+1} \cup (I_i^- \setminus I^-))$), or $H(r)$ is true w.r.t. $I_i \dot{\cup} \neg.\{a\}$ (and therefore also w.r.t. $I \dot{\cup} \neg.(X_{i+1} \cup (I_i^- \setminus I^-))$ because $I_i^+ = I^+$ and a is the only atom belonging to some C_j , for $j = 1, \dots, i+1$). No other rule in $\mathcal{P} \setminus \mathcal{P}_{C_{i+1}}$ has a in head, and then $X_{i+1} \cup (I_i^- \setminus I^-)$ is an unfounded set for \mathcal{P} w.r.t. I . So, from Proposition 10, X_{i+1} is a subset of $\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I)$.

(\supseteq) For each $a \in \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_{i+1}$ and for each $r \in \mathcal{P}$ with $a \in H(r)$, (1) $Ant(B(r))$ is false w.r.t. I (and so w.r.t. I_i because $I_i \supseteq I$), or (2) $Mon(B(r))$ is false w.r.t. $I \dot{\cup} \neg.\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I)$, or (3) $H(r)$ is true w.r.t. I (and so w.r.t. I_i). From (2) it follows that $Mon(B(r))$ is false w.r.t. $Y = I \dot{\cup} \neg.(\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap (C_1 \cup C_2 \cup \dots \cup C_{i+1}))$ (because $Mon(B(r))$ depends only from atoms in C_1, \dots, C_{i+1}).

But $I \dot{\cup} \neg.(\mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap (C_1 \cup C_2 \cup \dots \cup C_i)) = I_i$ and $X_{i+1} \subseteq \mathcal{R}_{\mathcal{P}, I}^\omega(B_{\mathcal{P}} \setminus I) \cap C_{i+1}$. Then $Y^+ = (I_i \dot{\cup} \neg.X_{i+1})^+$ and $Y^- \supseteq (I_i \dot{\cup} \neg.X_{i+1})^-$, so $Y \leq I_i \dot{\cup} \neg.X_{i+1}$ and $Mon(B(r))$ is false also w.r.t. $I_i \dot{\cup} \neg.X_{i+1}$.

8 Prototype Architecture

We have implemented the approach described in Section 7, modifying the system DLV, which already processes nonrecursive aggregates. For a thorough description of the DLV architecture, we refer to [27]. The main structure of the system is reported in Figure 1.

The input, after having possibly been processed by some frontends, is handed to the DLV core, in particular to the grounding, which produces a ground version of the input, which is guaranteed to have the same answer sets as the input. Control is then handed over to the model generator, which performs a backtracking heuristic search for models, which serve as answer set candidates. During this search, various pruning techniques are employed, among them also unfounded set computations (cf. [28, 29]). Each of the found answer set candidates is then submitted to the model checker, which

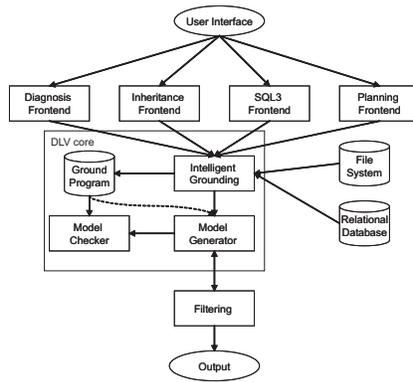


Fig. 1. DLV system architecture

verifies whether the model is an answer set (cf. [30]). When the input program is HCF, this check need not be done, as any produced candidate is known to be an answer set.

Therefore, for our prototype we had to modify the grounding and generator modules. In the grounding phase, in the case of non-recursive aggregates all instances of predicates inside an aggregate are known at the time the aggregate is instantiated. When supporting also recursive aggregates, this assumption no longer holds, and therefore a somewhat more complex grounding strategy has to be employed. Concerning the model generator, a large part of the existing machinery for aggregates could be re-used. In order to treat recursive aggregates correctly, unfounded set computation involving aggregates, which has not been present in DLV so far, is necessary. We have implemented unfounded set computations using an optimized implementation of the method described in Section 7, which further localize the computation by focusing only on the components that have been affected by the last propagation step. The system prototype is available at <http://www.dlvsystem.com/dlvRecAggr>, and supports a superset of hcf programs, requiring head-cycle freeness only on the components with recursive aggregates. Preliminary results of experiments on Companies Control examples indicate that the implementation offers good performance on medium-size instances.

References

1. Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In: ISLP'91, MIT Press (1991) 387–401
2. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In Codognet, P., ed.: ICLP-2001, (2001) 212–226
3. Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic. Logic Programming and Beyond. LNCS 2408 (2002) 413–451
4. Simons, P., Niemelä, I., Sooinen, T.: Extending and Implementing the Stable Model Semantics. *Artificial Intelligence* **138** (2002) 181–234
5. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI 2003, Acapulco, Mexico, (2003) 847–852
6. Pelov, N., Truszczyński, M.: Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In: NMR 2004. (2004) 327–334
7. Pelov, N., Denecker, M., Bruynooghe, M.: Partial stable models for logic programs with aggregates. In: LPNMR-7. LNCS 2923

8. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. LNCS 3229
9. Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. Theory and Practice of Logic Programming (2007) Accepted for publication, available in CoRR as cs.AI/0601051.
10. Son, T.C., Pontelli, E., Elkabani, I.: On Logic Programming with Aggregates. Tech. Report NMSU-CS-2005-006, New Mexico State University (2005)
11. Van Gelder, A., Ross, K., Schlipf, J.: The Well-Founded Semantics for General Logic Programs. JACM **38**(3) (1991) 620–650
12. Aczel, P.: Non-well-founded sets. Number 14. CSLI Lecture Notes (1988)
13. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)
14. Lee, J., Lifschitz, V.: Loop Formulas for Disjunctive Logic Programs. In: Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03), (2003) 451–465
15. Ferraris, P.: Answer Sets for Propositional Theories. <http://www.cs.utexas.edu/users/otto/papers/proptheories.ps> (2004)
16. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 406–411
17. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07). (2007) To appear.
18. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS 2923
19. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662
20. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
21. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
22. Faber, W.: Decomposition of Nonmonotone Aggregates in Logic Programming. WLP 2006 164–171
23. Faber, W.: Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates. In: LPNMR'05. LNCS 3662
24. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation **135**(2) (1997) 69–112
25. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87
26. Faber, W., Leone, N.: On the Complexity of Answer Set Programming with Aggregates. In: Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07, Tempe, Arizona, 2007, Proceedings. (2007) To appear.
27. 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
28. Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, TU Wien (2002)
29. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Disjunctive Logic Programming Systems. Fundamenta Informaticae **71**(2–3) (2006) 183–214

30. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. *Artificial Intelligence* **15**(1-2) (2003) 177-212

Extending Agent-oriented Requirements with Declarative Business Processes: a Computational Logic-based Approach*

Volha Bryl¹, Marco Montali², Paola Mello², Paolo Torroni², and Nicola Zannone¹

¹ DIT, University of Trento – Via Sommarive 14, 38050 Povo (TN), Italy
{bryl | zannone}@dit.unitn.it

² DEIS, University of Bologna – V.le Risorgimento 2, 40136 Bologna, Italy
{pmello | mmontali | ptorroni}@deis.unibo.it

Abstract. The analysis of business requirements and the specification of business processes are fundamental for the development of information system. The focus of this paper is on the combination of these two phases, that is, on linking the business goals and requirements to the business process model. To this end, we propose to extend the Tropos framework, which is used to model system and business requirements, with declarative business process-oriented constructs, inspired by DecSerFlow and ConDec languages. We also show how the proposed framework can be mapped into SCIFF, a computational logic-based framework, for properties and conformance verification.

1 Introduction

Modeling and analyzing requirements of information systems in terms of agents and their goals has been a topic of a considerable interest during the last decades [1]. Tropos [2] is one of the existing approaches to agent-oriented software engineering, which emphasizes the concepts of agent and goal from the early phases of the system development. Agent-oriented requirements analysis helps to understand the organizational setting in which a system will operate, to model stakeholders’ strategic interests and thus to represent the rationale beyond the introduction of the system and the design choices made.

The next step to be made after modeling and analyzing early system requirements is defining the corresponding business process. As it was pointed out in [3], linking the “strategic” business goals and requirements to the business process model is an utmost important issue. In this setting, many problems arise from organizational theory and strategic management perspectives due to limits on particular resources (e.g., cost, time, etc.). Business strategies have a fundamental impact on the structure of enterprises leading to efficiency in coordination

* This work has been partially funded by EU SENSORIA and SERENITY projects, by the MIUR-FIRB TOCAI project, by the MIUR PRIN 2005-011293 project, and by the PAT MOSTRO project.

and cooperation within economic activities. However, one of the drawbacks of Tropos, as well as many other agent-oriented modeling approaches, is in that the passage from a requirements model to a business process model is not clearly defined. For example, Tropos does not allow modeling temporal and data constraints between the tasks an agent is assigned to, which is essential when specifying the partial ordering between activities of a business process. Ability to represent start and completion times, triggering events, deadlines, etc. is strictly necessary when defining a business process model.

To support the development, optimization, and management of enterprise day-by-day activities, we propose a framework for facilitating the interaction between requirements analysis and the definition of business processes. In particular, we propose to extend Tropos with declarative business process-oriented constructs, inspired by two novel graphical languages, namely DecSerFlow [4] and ConDec [5]. In this way, the typical goal-oriented approach of Tropos agents is augmented with a high-level reactive, process-oriented dimension. We refer to the extended framework as to \mathcal{B} -Tropos. Furthermore, we show how both these complementary aspects could be mapped into a unique underlying formalism, called SCIFF [6], a computational logic-based framework for the specification and verification of interaction protocols in an open multi-agent setting. Thanks to this mapping it is possible to exploit the possibility of directly using the SCIFF specification to implement logic-based agents [7], as well as to perform different kinds of verification, such as properties verification [8] or conformance verification of a given execution trace w.r.t. the model it should follow [6]. To make the discussion more concrete, the proposed approach is applied to modeling and analyzing an intra-enterprise organizational model, focusing on the coordination of economic activities among different units of an enterprise collaborating to produce a specific product.

The structure of the paper is as follows. Section 2 briefly presents the Tropos methodology. Section 3 describes our process-oriented extensions of Tropos. The SCIFF framework is presented in Section 4, whereas Section 5 defines the mapping of \mathcal{B} -Tropos concepts to SCIFF specifications. The paper ends with the overview of related work and conclusive remarks in Sections 6 and 7, respectively.

2 The Tropos Methodology

Tropos [2] is an agent-oriented software engineering methodology tailored to describe and analyze socio-technical systems along the whole development process from requirements analysis up to implementation. One of its main advantages is the importance given to early requirements analysis. This allows one to capture *why* a piece of software is developed, behind the *what* or the *how*.

The methodology is founded on models that use the concepts of actor (i.e., agent and role), goal, task, resource, and social dependency. An *actor* is an active entity that has strategic goals and performs actions to achieve them. A *goal* represents a strategic interest of an actor. A *task* represents a particular course of actions that produces a desired effect. A *resource* represents a physical or an in-

formational entity without intentionality. A *dependency* between two actors indicates that one actor depends on another to achieve some goal, execute some task, or deliver some resource. The former actor is called *depender*, while the latter is called *dependee*. The object around which the dependency centres is called *dependum*. In the graphical representation, actors are represented as circles; goals, plans and resources are respectively represented as ovals, hexagons and rectangles; and dependencies have the form $depender \rightarrow dependum \rightarrow dependee$.

From a methodological perspective, Tropos is based on the idea of building a model of the system that is incrementally refined and extended. Specifically, goal analysis consists of refining goals and eliciting new social relationships among actors. Goal analysis is conducted from the perspective of single actors using three reasoning techniques: means-end analysis, AND/OR decomposition, and contribution analysis. *Means-end analysis* aims at identifying tasks to be executed in order to achieve a goal. Means-end relations are graphically represented as arrows without any label on them. *AND/OR decomposition* combines AND and OR refinements of a root goal or a root task into subparts. In essence, AND-decomposition is used to define the process for achieving a goal or a task, whereas OR-decomposition defines alternatives for achieving a goal or executing a task. *Contribution analysis* identifies the impact of the achievement of goals and tasks over the achievement of other goals and tasks. This impact can be positive or negative and is graphically represented as edges labeled with “+” and “-”, respectively.

Example 1. Figure 1 presents the requirements model of a product development process. In this scenario, different divisions of a company have to cooperate in order to produce a specific product. The Customer Care division is responsible for deploying products to customers, which refines it into subgoals *manufacture product*, for which it depends on the Manufacturing division, and *present product*, for which it depends on the Sales division. In turn, Manufacturing decomposes the appointed goal into subgoals *define solution for product*, for which it depends on the Research & Development (R&D) division, and *make product* that it achieves through task *execute production line*. To achieve goal *define solution for product*, R&D has to achieve goals *provide solution*, which it achieves through task *design solution*, *evaluate solution*, and *deploy solution*, which it achieves through task *define production plan*. The evaluation of the solution is performed in terms of costs and available resources. R&D executes task *assess costs*, which consists in calculating bill of quantities and evaluating bill of quantities, to achieve goal *evaluate costs*, and depends on the Warehouse for *evaluate available resources*. The Warehouse either can query the databases to find available resources or ask the Purchases division to buy resources from external Supplier. In the latter case, Purchases searches in company’s databases for possible Suppliers and selects the one who provides the best offer.

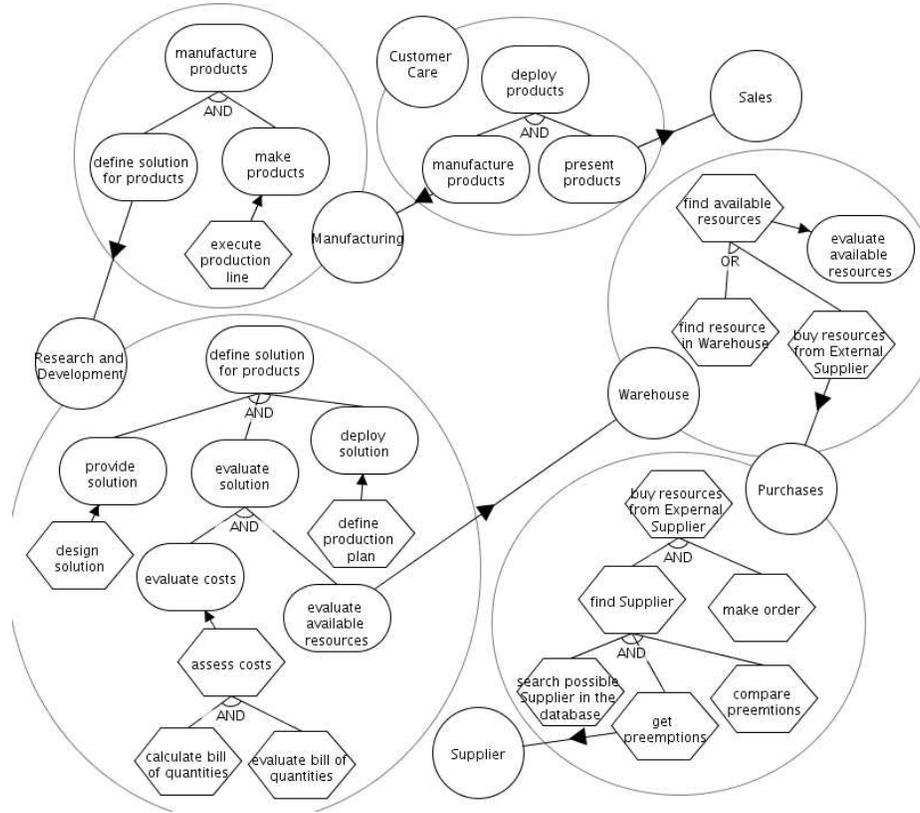


Fig. 1. Product Development Process in Tropos

3 Towards declarative process-oriented annotations

How business processes can be obtained from requirements analysis is an urgent issue for the development of a system. Unfortunately, Tropos is not able to cope with this issue mainly due to the lack of temporal constructs. In this section we discuss how Tropos can be extended in order to deal with high-level process-oriented aspects. The proposed extensions intend to support designers in defining durations, absolute time, and data-based decision constraints of goals and tasks as well as declaratively specifying relations between them. The latter extension is based on DecSerFlow [4] and ConDec [5], two novel graphical languages recently proposed by van der Aalst et al. to represent in a declarative and graphical way service flows and flexible business processes. We call Tropos extended with declarative business process-oriented constructs \mathcal{B} -Tropos.

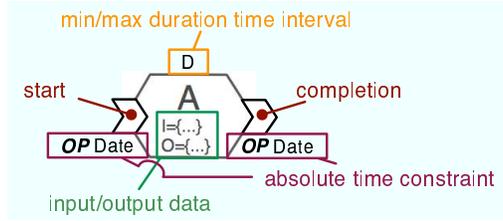


Fig. 2. Extended notation for tasks and goals

3.1 Some definitions

For the sake of clarity, we now give some informal definitions, which will be used to describe the Tropos extensions introduced in this section.

Definition 1 (Time interval). A time interval is a definite length of time marked off by two (non negative) instants (T_{min} and T_{max}), which could be considered both in an exclusive or inclusive manner. As usually, we use parentheses $(...)$ to indicate exclusion and square brackets $[...]$ to indicate inclusion.

Definition 2 (Relative time interval). A time interval is relative if initial instant and final instant are defined in function of another instant. Given a time interval TI marked off by T_{min} and T_{max} and a time instant T , two relative time intervals could be defined w.r.t. T

- TI^{+T} to denote the time interval marked off by $T + T_{min}$ and $T + T_{max}$;
- TI^{-T} to denote the time interval marked off by $T - T_{max}$ and $T - T_{min}$.

For example, $[10, 15)^{+T_1} \equiv [T_1 + 10, T_1 + 15)$ and $(0, 7]^{-T_2} \equiv [T_2 - 7, T_2)$.

Definition 3 (Absolute time constraint). An absolute time constraint is a unary constraint of the form T **OP** Date, where T is a time variable, Date is a date and **OP** $\in \{at, after, after_or_at, before, before_or_at\}$ (with their intuitive meaning).

Definition 4 (Data-based decision). A data-based decision formalizes a data-driven choice in terms of a CLP [12] constraint or Prolog predicate.

Definition 5 (Condition). A condition is a conjunction of data-based decisions and absolute time constraints.

3.2 Tasks/Goals extension

In order to support the modeling and analysis of process-oriented aspects of systems, we have annotated goals and tasks with temporal information such as *start* and *completion* time (the notation is shown in Fig. 2). Each task/goal can also be described in terms of its allowed *duration* (D in Fig. 2). This allows one to constrain, for instance, the completion time to the start one: *completion time* $\in D^{+source\ time}$. Additionally, absolute temporal constraints can be used to define start and completion times of goals and tasks. Tasks can also be specified in terms of their *input* and *output*. Finally, goals and tasks can be annotated with a *fulfillment* condition, which defines when they are successfully executed.

	relation	weak relation	negation
responded presence			
co-existence			
response			
precedence			
succession			

Table 1. Tropos extensions to capture process-oriented constraints (grouped negation connections share the same intended meaning, as described in [4]).

3.3 Process-oriented constraints

To refine a requirements model into a high-level and declarative process-oriented view, we have introduced different connections between goals and tasks, namely *relation*, *weak relation*, and *negation* (see Table 1). These connections allow designers to specify partial orderings between tasks under both temporal and data constraints. To make the framework more flexible, connections are not directly linked to tasks but to their start and completion time. A small circle is used to denote the connection source, which determines when the triggering condition is satisfied (co-existence and succession connections associate the circle to both end-points, since they are bi-directional).

Relation and negation connections are based on DecSerFlow [4] and ConDec [5] template formulas, extended with the possibility of bounding execution times (e.g., deadlines) and representing data-based and absolute time constraints. Conditions can be specified on both start and completion time and are delimited by curly braces (see $\{c\}$, $\{r\}$ and $\{cr_i\}$ in Table 1); the source condition is a triggering condition whereas the target condition represents a restriction on time and/or data.

The intended meaning of a responded presence relation is: if the source happens s.t. c is satisfied, then the target should happen and satisfy r . The co-existence relation applies the responded presence relation in both directions, by imposing that the two involved tasks, when satisfying cr_1 and cr_2 , should co-exist (namely either none or both are executed). Other relation connections extend the responded presence relation by specifying a temporal ordering between source and target events; optionally, a relative time interval (denoted with T_b in Table 1) could be attached to these connections, bounding when the target is expected to happen w.r.t. the time at which the source happened.³

In particular, the response relation constrains the target to happen *after* the source. If T_b is specified, the minimum and maximum time are respectively

³ If T_b is not specified, the default interval is $(0, \infty)$.

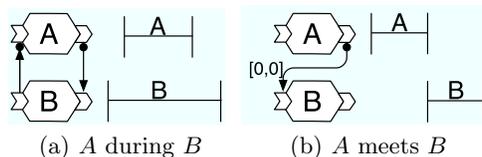


Fig. 3. Representation of two simple Allen's intervals in \mathcal{B} -Tropos

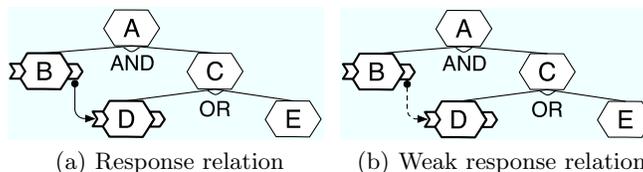


Fig. 4. Integrating process-oriented and goal-directed dimensions in \mathcal{B} -Tropos

treated as a delay and a deadline, i.e. the target should occur between the minimum and the maximum time after the source ($target\ time \in T_b^{+source\ time}$). The precedence relation is opposite to response relation, in the sense that it constrains the target to happen *before* the source. A succession relation is used to mutually specify that two tasks are the response and precedence of each other. By mixing different relation connections, we can express complex temporal dependencies and orderings, such as Allen's intervals [9] (see Fig. 3). For example, Allen's *meets* relation is formalized by imposing that A 's completion should be equal to B 's start (see Fig. 3(b)).

As in DecSerFlow and ConDec, we assume an open approach. Therefore, we have to explicitly specify not only what is expected, but also what is forbidden. These “negative” dependencies are represented by negation connections, the counter-part of relation connections. For example, the negation co-existence between two task states that when one task is executed, the other task shall never be executed, either before or after the source.

Summarizing, through relation and negation connections designers can add a horizontal declarative and high level process-oriented dimension to the vertical goal-directed decomposition of goals and tasks. It is worth noting that, in presence of OR decompositions, adding connections may affect the semantics of the requirements model. The decomposition of task A in Fig. 4(a) shows that A can be satisfied by satisfying D or E . On the contrary, the response relation between B 's completion and D 's start makes D mandatory (B has to be performed because of the AND-decomposition, hence D is expected to be performed after B). This kind of interaction is not always desirable. Therefore, we have introduced *weak relation* connections that relax relation connections. Their intended meaning is: whenever both the source and the target happen, then the target must satisfy the connection semantics and the corresponding restriction. The main difference between relations and weak relations is that in weak relations the execution is constrained a posteriori, after both source and target have happened.

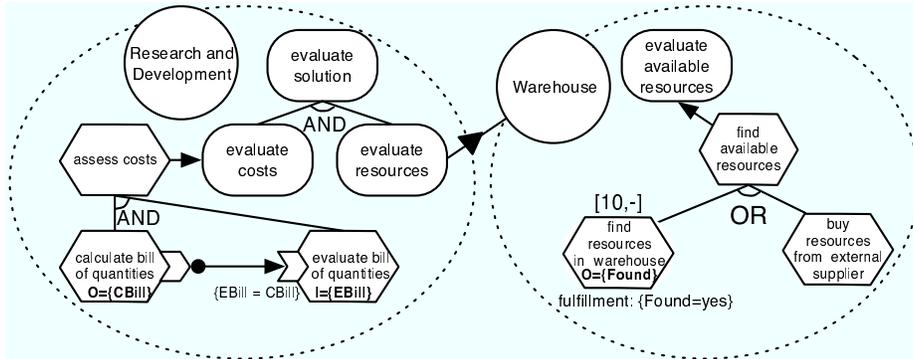


Fig. 5. Process-oriented extensions applied on a fragment of Fig. 1

Differently from Fig. 4(a), in Fig. 4(b) the response constraint between B and D should be satisfied only if D is executed.

Finally, \mathcal{B} -Tropos permits to constrain non-leaf tasks, leading to the possibility of expressing some process-oriented patterns [10]. For instance, a relation connection whose source is the completion of a task, which is AND-decomposed into two subtasks, triggers when both subtasks have been executed. Therefore, the connection resembles the concept of a synchronizing merge on the leaf tasks.

To show how process-oriented constraints could be added to a Tropos model, we extend a fragment of the diagram represented in Fig. 1; the result is shown in Fig. 5. The first extension concerns the decomposition of task *assess costs*: the bill of quantities can be evaluated only after having been calculated. Such a constraint could be modeled in \mathcal{B} -Tropos by (1) indicating that the calculation produces a bill of quantities, whereas the evaluation takes a bill as an input, and (2) attaching a response relation connection between the completion of task *calculate bill of quantities* and the start of task *evaluate bill of quantities*. The second extension has the purpose of better detailing task *find resources in Warehouse*, namely representing that (1) task duration is at least of 10 time units, (2) the task produces as an output a datum (called *Found*), which describes whether or not resources have been found in the Warehouse, and (3) the task is considered fulfilled only if resources have been actually found, i.e., *Found* is equal to *yes*.

4 SCIFF

SCIFF [6] is a formal framework based on abductive logic programming [11], developed in the context of the SOCS project⁴ for specifying and verifying interaction protocols in an open multi-agent setting. SCIFF introduces the concept of event as an atomic observable and relevant occurrence triggered at execution time. The designer has the possibility to decide what has to be considered as an

⁴ Societies of heterogeneous Computees, EU-IST-2001-32530 (home page <http://lia.deis.unibo.it/research/SOCS/>).

event; this generality allows him to decide how to model the target domain at the desired abstraction level, and to exploit *SCIFF* for representing any evolving process where activities are performed and information is exchanged.

We distinguish between the description of an *event*, and the fact that an event has happened. Happened events are represented as atoms $\mathbf{H}(Ev, T)$, where Ev is a *term* and T is an integer, representing the discrete time point at which the event happened. The set of all the events happened during a protocol execution constitutes its log (or execution trace). Furthermore, the *SCIFF* language supports the concept of *expectation* as first-class object, pushing the user to think of an evolving process in terms of reactive rules of the form “*if A happened, then B is expected to happen*”. Expectations about events come with form $\mathbf{E}(Ev, T)$ where Ev and T are variables, eventually grounded to a particular term/value.

The binding between happened events and expectations is given by means of *Social Integrity Constraints (ICs)*. They are forward rules, of the form $Body \rightarrow Head$, where $Body$ can contain literals and (conjunctions of happened and expected) events and $Head$ can contain (disjunctions of) conjunctions of expectations. CLP constraints and Prolog predicates can be used to impose relations or restrictions on any of the variables, for instance, on time (e.g., by expressing orderings or deadlines). Intuitively, \mathcal{IC} allows the designer to define how an interaction should evolve, given some previous situation represented in terms of happened events; the static knowledge of the target domain is instead formalized inside the *SCIFF* Knowledge Base. Here we find pieces of knowledge of the interaction model as well as the global society goal and/or objectives of single participants. Indeed, *SCIFF* considers interaction as goal-directed, i.e., envisages environments in which each actor, as well as the overall society, could have some objective only achievable through interaction; by adopting such a vision, the same interaction protocol could be seamlessly exploited for achieving different strategic goals. This knowledge is expressed in the form of clauses (i.e., a logic program); a clause’s body may contain expectations about the behavior of participants, defined literals, and constraints, while their heads are atoms. As advocated in [13], this vision reconciles in a unique framework forward reactive reasoning with backward, goal-oriented deliberative reasoning.

In *SCIFF* an interaction model is interpreted in terms of an Abductive Logic Program (ALP)[11]. In general, an ALP is a triple $\langle P, A, IC \rangle$, where P is a logic program, A is a set of predicates named *abducibles*, and IC is a set of Integrity Constraints. Roughly speaking, the role of P is to define predicates, the role of A is to fill-in the parts of P that are unknown, and the role of IC is to control the way elements of A are hypothesized, or “abduced”. Reasoning in abductive logic programming is usually goal-directed (being G a goal), and it accounts to finding a set of abduced hypotheses Δ built from predicates in A such that $P \cup \Delta \models G$ and $P \cup \Delta \models IC$. The idea underlying *SCIFF* is to adopt abduction to dynamically *generate* the expectations and to perform the *conformance checking* between expectations and happened events (to ensure that they are following the interaction model). Expectations are defined as abducibles: the framework makes hypotheses about how participants should behave. Conformance is verified by

trying to confirm the hypothesized expectations: a concrete running interaction is evaluated as conformant if it *fulfills* the specification. Operationally, expectations are generated and verified by the *SCIFF* proof procedure,⁵ a transition system which has been proved sound and complete w.r.t. the declarative semantics [6]. The proof procedure is embedded within *SOCS-SI*,⁶ a JAVA-based tool capable to accept different event-sources (or previously collected execution traces) and to check if the actual behavior is conformant w.r.t. a given *SCIFF* specification.

5 Mapping *B*-Tropos concepts to the *SCIFF* framework

In this section we present the mapping of *B*-Tropos concepts into *SCIFF* specifications, briefly describing how the obtained formalization is used to implement the skeleton of logic-based agents.

Table 2 summarizes the formalization of the goal-oriented part of *B*-Tropos in *SCIFF*. Tasks and goals refer to a whatsoever actor *X*. Being goal-oriented, all the concepts of such a part are modeled inside the *SCIFF* knowledge base. When formalizing this part, two fundamental concepts emerge: the achievement of a goal and the execution of a task. Both concepts are modeled in *SCIFF* by considering the actor who is trying to achieve the goal or executing the task and the involved start and completion times; such times should satisfy both duration and absolute time constraints eventually associated to the goal/task. Furthermore, by taking into account a specific goal, different (possibly overlapping) cases may arise:

- AND/OR-decompositions and means-end relations can be trivially translated to Prolog.
- Positive contributions are implemented with a clause specifying that the target is achieved if the contribution’s source is achieved.
- Negative contributions are implemented as denials, by imposing that achieving both the involved goals leads to inconsistency.
- In goal and task dependencies, it is expected that the depender will communicate to the dependee that he/she requires the goal to be achieved inside a certain time interval. The communication of this kind of delegation is explicit (i.e. observable), so it can be directly mapped to a *SCIFF* expectation about depender’s behavior.
- In some cases the designer may prefer to keep the model at an abstract level, so goals can be neither refined nor associated to tasks. Abduction allows us to face such a lack of information by reasoning on goal’s achievement in a hypothetical way; in particular, we introduce a new abducible called **achieved** to hypothesize that the actor has actually reached the goal.

Task execution mainly differs from goal achievement in that task start and completion events are verified by a fulfillment condition. As for dependency

⁵ Available at <http://lia.deis.unibo.it/research/sciff/>.

⁶ Available at http://www.lia.deis.unibo.it/research/socs_si/socs_si.shtml.

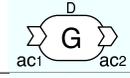
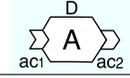
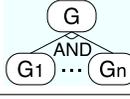
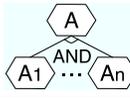
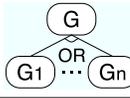
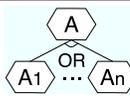
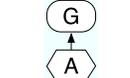
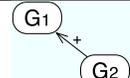
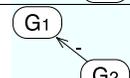
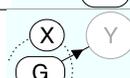
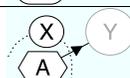
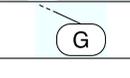
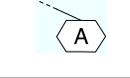
\mathcal{B} -Tropos Goal/ Task		$achieve(X, G, T_i, T_f) \leftarrow T_f \in D^{+T_i}, ac_1, ac_2, \dots$
		$execute(X, A, T_i, T_f) \leftarrow T_f \in D^{+T_i}, ac_1, ac_2, \dots$
AND decomposition		$achieve(X, G, T_i, T_f) \leftarrow$ $achieve(X, G_1, T_{i1}, T_{f1}), \dots, achieve(X, G_n, T_{in}, T_{fn}),$ $T_i = \min\{T_{i1}, \dots, T_{in}\}, T_f = \max\{T_{f1}, \dots, T_{fn}\}.$
		$execute(X, A, T_i, T_f) \leftarrow$ $execute(X, A_1, T_{i1}, T_{f1}), \dots, execute(X, A_n, T_{in}, T_{fn}),$ $T_i = \min\{T_{i1}, \dots, T_{in}\}, T_f = \max\{T_{f1}, \dots, T_{fn}\}.$
OR decomposition		$achieve(X, G, T_i, T_f) \leftarrow achieve(X, G_1, T_i, T_f).$ \dots $achieve(X, G, T_i, T_f) \leftarrow achieve(X, G_n, T_i, T_f).$
		$execute(X, A, T_i, T_f) \leftarrow execute(X, A_1, T_i, T_f).$ \dots $execute(X, A, T_i, T_f) \leftarrow execute(X, A_n, T_i, T_f).$
Means-end		$achieve(X, G, T_i, T_f) \leftarrow execute(X, A, T_i, T_f).$
Positive contribution		$achieve(X, G_1, T_i, T_f) \leftarrow achieve(X, G_2, T_i, T_f).$
Negative contribution		$achieve(X, G_1, T_i, T_f), achieve(X, G_2, T_i, T_f) \rightarrow \perp$
Goal Dependency		$achieve(X, G, T_i, T_f) \leftarrow \mathbf{E}(delegate(X, Y, G, T_f), T_i).$
Task Dependency		$execute(X, A, T_i, T_f) \leftarrow \mathbf{E}(delegate(X, Y, A, T_f), T_i).$
Leaf goal		$achieve(X, G, T_i, T_f) \leftarrow \mathbf{achieved}(X, A, T_i, T_f).$
Leaf task		$execute(X, A, T_i, T_f) \leftarrow \mathbf{E}(event(start, X, A), T_i),$ $\mathbf{E}(event(compl, X, A), T_f),$ $fulfillment_condition, T_f > T_i.$

Table 2. Mapping of the goal-oriented proactive part of \mathcal{B} -Tropos in SCIFF.

relations, these events are mapped to expectations and should appear in the execution trace.

The reactive part of \mathcal{B} -Tropos encompasses both the reaction to a request for achieving a goal and process-oriented constraints. As already pointed out, process-oriented constraints are inspired by DecSerFlow/ConDec template formulas, for which a preliminary mapping to SCIFF has been already established

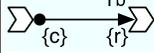
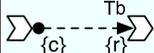
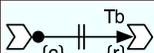
Response		$\mathbf{hap}(\mathit{event}(Ev, A, X), T_1) \wedge c$ $\rightarrow \mathbf{exp}(\mathit{event}(Ev, A, X), T_2) \wedge r \wedge T_2 \in T_b^{+T_1}$.
Weak Response		$\mathbf{hap}(\mathit{event}(Ev, A, X), T_1) \wedge c$ $\wedge \mathbf{hap}(\mathit{event}(Ev, A, X), T_2) \rightarrow r \wedge T_2 \in T_b^{+T_1}$.
Negation Response		$\mathbf{hap}(\mathit{event}(Ev, A, X), T_1) \wedge c$ $\wedge \mathbf{hap}(\mathit{event}(Ev, A, X), T_2) \wedge r \wedge T_2 \in T_b^{+T_1} \rightarrow \perp$.

Table 3. Mapping of \mathcal{B} -Tropos response connections in \mathcal{SCIFF} .

[14]. Connections belonging to the same family (i.e. relations, weak relations and negations) are translated to very similar \mathcal{IC} s: the only main difference is the way in which the involved times are constrained, to reflect the connection semantics. An example is given in Table 3, where response connections have been formalized; they specify in a straightforward way the informal description given in Section 3.

Predicates **hap** and **exp** respectively represent the happening and the expectation of a complex or simple event (remember indeed that also non-leaf tasks could be constrained). Since the start and completion of leaf tasks are considered as observable events, then for a leaf-task A ($Ev \in \{start, completion\}$):

$$\begin{aligned} \mathbf{hap}(\mathit{event}(Ev, A, X), T) &\leftarrow \mathbf{H}(\mathit{event}(Ev, A, X), T). \\ \mathbf{exp}(\mathit{event}(Ev, A, X), T) &\leftarrow \mathbf{E}(\mathit{event}(Ev, A, X), T). \end{aligned}$$

Complex events recursively follows the AND/OR decomposition philosophy:

- the start/completion of an OR-decomposed task happen (resp. is expected to happen) when one of its (sub)tasks start/completion happens (resp. is expected to happen);
- the start of an AND-decomposed task happens (resp. is expected to happen) when its first (sub)task is started (resp. expected to started);
- the completion of an AND-decomposed task happens (resp. is expected to happen) when its last (sub)task is completed (expected to be completed).

To model the reaction to a request for achieving a goal G , we simply assume that when a dependee Y receives from depender X a request for achieving goal G , then Y should react by assuming the commitment of actually achieving G :⁷

$$\mathbf{H}(\mathit{delegate}(X, Y, G, T_f), T_d) \rightarrow \mathit{achieve}(Y, G, T_i, T_f) \wedge T_i > T_d.$$

Table 5.1 shows the \mathcal{SCIFF} formalization corresponding to the \mathcal{B} -Tropos diagram of Figure 5. Here Research & Development and Warehouse are respectively represented as $r\&d$ and wh , and the equality symbol $=$ is used to denote unification.

The provided formalization could be used to directly implement the skeleton of logic-based agents, as for example the ones described in [7]. Such agents follow

⁷ Anyway, more complex dependency protocols should be seamlessly modeled in \mathcal{SCIFF} .

Table 5.1 Formalization of the \mathcal{B} -Tropos model fragment shown in Figure 5

$KB_{r\&d} : \text{achieve}(r\&d, \text{eval_solution}, T_i, T_f) \leftarrow \text{achieve}(r\&d, \text{eval_costs}, T_{i1}, T_{f1}),$ $\text{achieve}(r\&d, \text{eval_resources}, T_{i2}, T_{f2}),$ $\min(T_i, [T_{i1}, T_{i2}]), \max(T_f, [T_{f1}, T_{f2}]).$ $\text{achieve}(r\&d, \text{eval_costs}, T_i, T_f) \leftarrow \text{execute}(r\&d, \text{assess_costs}, T_i, T_f).$ $\text{execute}(r\&d, \text{assess_costs}, T_i, T_f) \leftarrow \text{execute}(r\&d, \text{calc_bill}, T_{i1}, T_{f1}),$ $\text{execute}(r\&d, \text{eval_bill}, T_{i2}, T_{f2}),$ $\min(T_i, [T_{i1}, T_{i2}]), \max(T_f, [T_{f1}, T_{f2}]).$ $\text{execute}(r\&d, \text{calc_bill}, T_i, T_f) \leftarrow \mathbf{E}(\text{event}(\text{start}, r\&d, \text{calc_bill}), T_i),$ $\mathbf{E}(\text{event}(\text{compl}, r\&d, \text{calc_bill}, [CBill]), T_f), T_f > T_i.$ $\text{execute}(r\&d, \text{eval_bill}, T_i, T_f) \leftarrow \mathbf{E}(\text{event}(\text{start}, r\&d, \text{eval_bill}, [EBill]), T_i),$ $\mathbf{E}(\text{event}(\text{compl}, r\&d, \text{eval_bill}), T_f), T_f > T_i.$ $\text{achieve}(r\&d, \text{eval_resources}, T_i, T_f) \leftarrow \mathbf{E}(\text{delegate}(r\&d, wh, \text{eval_resources}, T_f), T_i).$	$KB_{wh} : \text{achieve}(wh, \text{eval_resources}, T_i, T_f) \leftarrow \text{execute}(wh, \text{find_resources}, T_i, T_f).$ $\text{execute}(wh, \text{find_resources}, T_i, T_f) \leftarrow \text{execute}(wh, \text{find_in_wh}, T_i, T_f).$ $\text{execute}(wh, \text{find_resources}, T_i, T_f) \leftarrow \text{execute}(wh, \text{buy}, T_i, T_f).$ $\text{execute}(wh, \text{find_resources}, T_i, T_f) \leftarrow \mathbf{E}(\text{event}(\text{start}, wh, \text{find_in_wh}), T_i),$ $\mathbf{E}(\text{event}(\text{compl}, wh, \text{find_in_wh}, Found), T_f),$ $T_f \geq T_i + 10, Found = \text{yes}.$ $\text{execute}(wh, \text{buy}, T_i, T_f) \leftarrow \mathbf{E}(\text{event}(\text{start}, wh, \text{buy}), T_i),$ $\mathbf{E}(\text{event}(\text{compl}, wh, \text{buy}), T_f), T_f > T_i.$
<hr/>	
$IC_{s_{r\&d}} : \mathbf{H}(\text{event}(\text{compl}, r\&d, \text{calc_bill}, [CBill]), T_1) \rightarrow \mathbf{E}(\text{event}(\text{start}, r\&d, \text{eval_bill}, [EBill]), T_2)$ $\wedge T_2 > T_1, EBill = CBill.$	
<hr/> $IC_{s_{wh}} : \mathbf{H}(\text{delegate}(r\&d, wh, \text{eval_resources}, T_f), T_i) \rightarrow \text{achieve}(wh, \text{eval_resources}, T_i, T_f).$ <hr/>	

the Kowalsky-Sadri cycle for intelligent agents, by realizing the *think* phase with the SCIFF proof-procedure and the *observe* and *act* phases in JADE. The proof-procedure embedded into SCIFF-agents is equipped with the possibility to transform expectations about the agent itself into happened events, and with a selection rule for choosing a behavior when more different choices are available.

In particular, each actor represented in a \mathcal{B} -Tropos model could be mapped into a SCIFF-agent whose deliberative pro-active part (formalized in the agent's knowledge base) is driven by the goal/task decomposition of its root goal, and whose reactive behavior (formalized as a set of ICs) is determined by the delegation mechanism and the process-oriented constraints. The agent that wants to achieve the global goal (such as Customer Care in Figure 1) starts by decomposing it, whereas other actors wait until an incoming request from a depender is observed; in this case, the delegation reactive rule of the agent is triggered, and the agent tries to achieve its root goal. The root goal is decomposed until finally one or more expectations are generated. Such expectations could be either requests or start/completions of tasks, and thus are transformed to happened events, i.e. actions performed by the agent.

Table 5.1 shows how the formalized SCIFF specification is assigned to the two agents under study, i.e., the Warehouse and R&D unit. To have an intuition about how the two agents act and interact, let us consider the case in which the R&D unit should achieve its top goal (because it has received the corresponding delegation from the Manufacturing division). The unit will decompose the goal obtaining, at last, the following set of expectations about itself:⁸

$$\begin{aligned} & \mathbf{E}(\text{event}(\text{start}, r\&d, \text{calc_bill}), T_{scb}), \dots, \\ & \mathbf{E}(\text{event}(\text{compl}, r\&d, \text{calc_bill}, [\text{Bill}]), T_{ccb}), T_{ccb} > T_{scb}, \\ & \mathbf{E}(\text{event}(\text{start}, r\&d, \text{eval_bill}, [\text{Bill}]), T_{seb}), T_{seb} > T_{ccb}, \\ & \mathbf{E}(\text{event}(\text{compl}, r\&d, \text{eval_bill}), T_{ceb}), T_{ceb} > T_{seb}, \\ & \mathbf{E}(\text{del}(r\&d, wh, \text{eval_resources}, T_{cer}), T_{ser}). \end{aligned}$$

This set of expectations could be read as an execution plan, consisting of two concurrent parts: (1) a sequence about start/completion of leaf tasks, ordered by the response relation which constrains the bill calculation and evaluation; (2) the delegation of resources evaluation, which should be communicated to the Warehouse. In particular, when the expectation about the delegation is transformed to a happened event by the R&D agent, the Warehouse agent is committed to achieve the delegated goal inside the time interval (T_{ser}, T_{cer}) .

Besides the implementation of logic-based agents, SCIFF can also be used to perform different kinds of verification, namely performance verification and conformance verification. Performance verification is devoted to prove that stakeholders can achieve their strategic goals in a given time. Such a verification can also be used to evaluate different design alternatives in terms of system performances. Conformance verification [6] is related to the auditing measures that can be adopted for monitoring the activities performed by actors within the system. The idea underlying conformance verification is to analyze system logs and compare them with the design of the system. This allows system administrators to understand whether or not stakeholders have achieved their goals and, if it is not the case, predict future actions. For the lack of space, we do not discuss here the details of these kinds of verification.

6 Related Works

Several formal frameworks have been developed to support the Tropos methodology. For instance, Giorgini et al. [15] proposed a formal framework based on logic programming for the analysis of security and privacy requirements. However, the framework does not take into account temporal aspects of the system. In [16] a planning approach has been proposed to analyze and evaluate design alternatives. Though this framework explores the space of alternatives and determines a (sub-)optimal plan, that is, a sequence of actions, to achieve the goals of stakeholders, it does not permit to define temporal constraints among

⁸ By imposing, through a special integrity constraint, that two different expectations about the same event should be fulfilled by one happened event.

tasks. Fuxman et al. [17] proposed Formal Tropos that extends Tropos with annotations that characterize the temporal evolution of the system, describing for instance how the network of relationships evolves over time. Formal Tropos provides a temporal logic-based specification language for representing Tropos concepts together with temporal constructs, which are verified using a model-checking technique such as the one implemented in NuSMV. This framework has been used to verify the consistency of the requirements model [17] as well as business processes against business requirements and strategic goal model [3]. However, Formal Tropos does not support conformance verification.

The use of computational logic for the flexible specification and rigorous verification of agent interaction is adopted by many proposals. While other works [18] use temporal logics to model the temporal dimension of interaction, *SCIFF* exploits a constraint solver and adopts an explicit representation of time, making it possible to specify and reason upon expressive temporal constraints (such as deadlines). With [19, 20], *SCIFF* shares the vision of a multi-agent system as an open society of heterogeneous and autonomous interacting entities. While in [20] Event Calculus is applied to commitment-based protocol specification, *SCIFF* semantics is given in terms of an Abductive Logic Program. Many abductive proof-procedures exist in literature (e.g., [13, 21]), but none of them deals with hypotheses confirmation used in *SCIFF* to verify if the abduced expectations have indeed a corresponding happened event.

7 Conclusions

In this work we have proposed *B-Tropos*, an extension of Tropos with declarative process-oriented constraints, to the aim of making the first step towards the definition of a business process from an early requirements model. More specifically, we have introduced the possibility to mutually constrain task/goal execution times, by using connections inspired by DecSerFlow and ConDec languages. Augmenting a Tropos model with such constraints has the effect that both the proactive agents behavior and the reactive, process-oriented one could be captured within the same diagram.

We have also shown how both goal-oriented and process-oriented dimensions of *B-Tropos* can be mapped into the *SCIFF* framework. Such a mapping makes it possible to directly implement logic-based agents following what is prescribed by the model, as well as to perform different kinds of verification, namely to check if the model satisfies a given property and to monitor if the execution trace of a real system is actually compliant with the model.

The work presented here it is a first step towards the integration of a business process in the requirements model. The next step will be the generation of executable business process specifications (such as BPEL) from *B-Tropos* models. Moreover, we intend to better exploit the underlying *SCIFF* constraint solver by introducing more complex scheduling and resource constraints in order to capture more details of business requirements and agent interactions.

References

1. Henderson-Sellers, B., Giorgini, P., eds.: *Agent-Oriented Methodologies*. Idea Group Publishing (2005)
2. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS* **8**(3) (2004) 203–236
3. Kazhamiakin, R., Pistore, M., Roveri, M.: A Framework for Integrating Business Processes and Business Requirements. In: *EDOC'04*, IEEE Press (2004) 9–20
4. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: *WS-FM'06*. Volume of LNCS 4184., Springer (2006)
5. van der Aalst, W.M.P., Pesic, M.: A declarative approach for flexible business processes management. In: *BPM'06*. LNCS 4103, Springer (2006) 169–180
6. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. *TOCL* (2007) Accepted for publication.
7. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P.: A verifiable logic-based agent architecture. In: *ISMIS'06*. LNCS 4203, Springer (2006) 188–197
8. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Security Protocols Verification in Abductive Logic Programming: A Case Study. In: *ESAW'05*. LNCS 3963, Springer (2005) 106–124
9. Allen, J., Ferguson, G.: Actions and events in interval temporal logic. *Journal of Logic and Computation* **4**(5) (1995)
10. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. *Distributed and Parallel Databases* **14**(1) (2003) 5–51
11. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. *Journal of Logic and Computation* **2**(6) (1993) 719–770
12. Jaffar, J., Maher, M.: Constraint logic programming: a survey. *Journal of Logic Programming* **19-20** (1994) 503–582
13. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. *Journal of Logic Programming* **33**(2) (1997) 151–165
14. Chesani, F., Mello, P., Montali, M., Storari, S.: Towards a decserflow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy (2007)
15. Giorgini, P., Massacci, F., Zannone, N.: Security and Trust Requirements Engineering. In: *FOSAD 2004/20005*. LNCS 3655, Springer (2005) 237–272
16. Bryl, V., Massacci, F., Mylopoulos, J., Zannone, N.: Designing Security Requirements Models through Planning. In: *CAiSE'06*. LNCS 4001, Springer (2006) 33–47
17. Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and Analyzing Early Requirements in Tropos. *REJ* **9**(2) (2004) 132–150
18. Venkatraman, M., Singh, M.P.: Verifying compliance with commitment protocols. *JAAMAS* **2**(3) (1999) 217–236
19. Artikis, A., Pitt, J., Sergot, M.J.: Animated specifications of computational societies. In: *AAMAS'02*. (2002) 1053–1061
20. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: applying event calculus planning using commitments. In: *AAMAS'02*, ACM (2002) 527–534
21. Kakas, A., Mancarella, P.: On the relation between truth maintenance and abduction. In: *PRICAI'90*. (1990) 438–443

A tableau for Right Propositional Neighborhood Logic over trees

Davide Bresolin¹, Angelo Montanari², and Pietro Sala²

¹ Department of Computer Science, University of Verona,
Verona, Italy bresolin@sci.univr.it

² Department of Mathematics and Computer Science, University of Udine,
Udine, Italy angelo.montanari@dimi.uniud.it
pietro.sala@dimi.uniud.it

Abstract. Propositional interval temporal logics come into play in many areas of artificial intelligence and computer science. Unfortunately, most of them turned out to be (highly) undecidable. In the last years, various decidable fragments have been identified and systematically studied. In this paper, we address the decision problem for the future fragment of Propositional Neighborhood Logic (Right Propositional Neighborhood Logic) interpreted over trees and we positively solve it by providing a sound, complete, and terminating tableau-based decision procedure.

1 Introduction

Interval temporal logics play an important role in many areas of computer science. Interval-based formal systems for representing and reasoning about time have indeed been developed in artificial intelligence (reasoning about action and change, qualitative reasoning, planning, and natural language processing), theoretical computer science (specification and automatic verification of programs), and databases (temporal and spatio-temporal databases). A comprehensive and up-to-date survey on the main propositional and first-order interval temporal logics proposed in the literature can be found in [10]. In this paper, we restrict our attention to the class of propositional interval temporal logics, which includes Halpern and Shoham's Modal Logic of Time Intervals (HS) [12], Venema's CDT logic, interpreted over linear and partial orders [8,11,19], Moszkowski's Propositional Interval Temporal Logic (PITL) [15], Propositional Neighborhood Logic (PNL), investigated by Goranko et al. [2,3,4,5,6,9], and the temporal logics of subinterval relations [1,17,18].

The most expressive propositional interval temporal logics, e.g., HS, CDT, and PITL, turned out to be (highly) undecidable, but various syntactic and/or semantic decidable fragments of them have been identified and systematically studied. As pointed out in [13], different paths to decidability have been explored. One can get decidability by making a suitable choice of the interval modalities. This is the case with the $\langle B \rangle \langle \bar{B} \rangle$ (*begins/begun by*) and $\langle E \rangle \langle \bar{E} \rangle$ (*ends/ended by*) fragments of HS. Goranko et al. proved the decidability of $\langle B \rangle \langle \bar{B} \rangle$ (the case

of $\langle E \rangle \langle \bar{E} \rangle$ is similar) by embedding it into the propositional temporal logic of linear time LTL[F,P] with temporal modalities F (sometime in the future) and P (sometime in the past) [10]. Formulae of $\langle B \rangle \langle \bar{B} \rangle$ are translated into formulae of LTL[F,P] by a mapping that replaces $\langle B \rangle$ by P and $\langle \bar{B} \rangle$ by F . LTL[F,P] has the finite model property and is decidable. Decidability can also be achieved by constraining the classes of temporal structures over which the interval logic is interpreted. This is the case with the so-called Split Logics (SLs) [14]. SLs are propositional interval logics equipped with operators borrowed from HS and CDT, but interpreted over specific structures, called split structures, where every interval can be ‘chopped’ in at most one way. The decidability of various SLs has been proved by embedding them into first-order fragments of monadic second-order theories of time granularity, which are proper decidable extensions of the well-known monadic second-order theory of one successor S1S. Finally, decidability can be achieved by constraining the relation between the truth value of a formula over an interval and its truth value over subintervals of that interval. As an example, one can constrain a propositional letter to be true over an interval if and only if it is true at its starting point (*locality*) or can constrain it to be true over an interval if and only if it is true over all its subintervals (*homogeneity*). A decidable fragment of PITL extended with quantification over propositional letters (QPITL) has been obtained by imposing the *locality* constraint [15]. By exploiting such a constraint, decidability of QPITL can be proved by embedding it into quantified LTL.

In this paper, we focus our attention on the class of propositional temporal logics of temporal neighborhood (PNLs for short). They feature modalities for right and left interval neighborhoods, namely, the *after* operator $\langle A \rangle$ and its transpose $\langle \bar{A} \rangle$, such that $\langle A \rangle \varphi$ (resp. $\langle \bar{A} \rangle \varphi$) holds over $[d_0, d_1]$ if φ holds over $[d_1, d_2]$ (resp. $[d_2, d_0]$) for some $d_2 > d_1$ (resp. $d_2 < d_0$). The decidability of various logics in this class has been proved by developing suitable tableau-based decision procedures. The decidability of the future fragment of PNL (RPNL for short) over the natural numbers has been proved by Bresolin et al. in [4,6]. They basically show that an RPNL formula is satisfiable if and only if there exist a finite model, or an ultimately periodic (infinite) one, with a finite representation of *bounded size*. By exploiting this result, they devise a tableau-based decision procedure of optimal complexity (NEXPTIME). This result has been later extended to full PNL interpreted over the integers in [5], where an optimal NEXPTIME decision procedure is provided. Even though the solution for full PNL exploits the same idea at the basis of the one for RPNL, its proof turns out to be much more difficult. A third decidability result is given in [2], where the satisfiability problem for PNL has been shown to be decidable in NEXPTIME with respect to several classes of linear orders by reducing it to the satisfiability problem for the decidable 2-variable fragment of first-order logic extended with a linear order [16]. Here, we address the decision problem for RPNL interpreted over trees and we positively solve it by providing a sound, complete, and terminating tableau-based decision procedure.

The paper is organized as follows. In Section 2 we introduce syntax and semantics of RPNL interpreted over trees. In Section 3 we illustrate the problems that one must face when linear structures are replaced with trees. Then, in Section 4 we develop a tableau-based decision procedure for RPNL interpreted over trees, we determine its complexity, and we prove its soundness and completeness. Conclusions provide an assessment of the work and outline future research directions.

2 Right Propositional Neighborhood Logic over trees

According to a commonly accepted perspective [7], the temporal structure underlying branching-time temporal logics is a tree, where each time point may have many successor time points. We assume that the timeline defined by every path in the tree is either finite or isomorphic to $\langle \mathbb{N}, < \rangle$ and we allow a node in the tree to have infinitely many (possibly, uncountably many) successors. It will turn out that, as far as our logic is concerned, such trees are indistinguishable from trees with finite branching.

Given a directed graph $\mathbb{G} = \langle G, S \rangle$, a *finite S-sequence* over \mathbb{G} is a sequence of nodes $g_1 g_2 \dots g_n$, with $n \geq 2$ and $g_i \in G$ for $i = 1, \dots, n$, such that $S(g_i, g_{i+1})$ for $i = 1, \dots, n - 1$. *Infinite S-sequences* can be defined analogously. We define a *path* ρ in \mathbb{G} as a finite or infinite *S-sequence*. In the following, we shall take advantage of a relation $S^+ \subseteq G \times G$ such that $S^+(g_i, g_j)$ if and only if g_i and g_j are respectively the first and the last element of a finite *S-sequence*. In such a case, we say that g_j is *S-reachable* from g_i .

Infinite trees are defined as follows.

Definition 1. *A tree is a directed graph $\mathbb{T} = \langle T, S \rangle$, where T is the set of nodes, called time points, and $S \subseteq T \times T$ is the set of edges. The set T contains a distinguished element t_0 , called the root of the tree, and the relation S is such that:*

- for every $t \in T$ if $t \neq t_0$ then $S^+(t_0, t)$, that is, every time point is *S-reachable* from the root;
- for every $t \in T$ if $t \neq t_0$ then there exists at most one $t' \in T$ such that $S(t', t)$ (together with the previous one, this condition guarantees that every time point different from the root has exactly one *S-predecessor*);
- there exists no t' such that $S(t', t_0)$, that is, t_0 has no *S-predecessors*.

Given a tree $\mathbb{T} = \langle T, S \rangle$, we can define a partial order $<$ over T such that, for every $t, t' \in T$, $t < t'$ if and only if $S^+(t, t')$. It can be easily shown that, for every infinite path ρ in \mathbb{T} , $\langle \rho, < \rangle$ is isomorphic to $\langle \mathbb{N}, < \rangle$.

Given a tree $\mathbb{T} = \langle T, S \rangle$, and the corresponding partial ordering $\langle T, < \rangle$, an *interval* on \mathbb{T} is an ordered pair $[t_i, t_j]$ such that $t_i, t_j \in T$ and $t_i < t_j$. The set of all intervals will be denoted by $\mathbb{I}(\mathbb{T})^-$. We use the superscript $-$ to indicate that point-intervals $[b, b]$ are excluded. The pair $\langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle$ is called an *interval*

structure. For every pair of intervals $[t_i, t_j], [t'_i, t'_j] \in \mathbb{I}(\mathbb{T})^-$, we say that $[t'_i, t'_j]$ is a *right neighbor* of $[t_i, t_j]$ if and only if $t_j = t'_i$.

The language of *Right Propositional Neighborhood Logic* (RPNL⁻ for short) consists of a set AP of propositional letters, the classical connectives \neg and \vee , and the modal operator $\langle A \rangle$.

The *formulae* of RPNL⁻, denoted by φ, ψ, \dots , are recursively defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle A \rangle\varphi.$$

The remaining classical connectives, as well as the logical constants \top (true) and \perp (false), can be defined as usual. Moreover we define a derived modal operator $[A]$ such that $[A]\varphi \equiv \neg\langle A \rangle\neg\varphi$. We denote by $|\varphi|$ the length of φ , that is, the number of symbols in φ (in the following, we shall use $|\cdot|$ to denote the cardinality of a set as well). Whenever there are no ambiguities, we call an RPNL⁻ formula just a formula. A formula of the forms $\langle A \rangle\psi$ or $[A]\psi$ is called a *temporal formula* (from now on, we identify $\neg\langle A \rangle\psi$ with $[A]\neg\psi$ and $\neg[A]\psi$ with $\langle A \rangle\neg\psi$), in particular, we call a formula of the form $\langle A \rangle\psi$ a *temporal request*.

A *model* for an RPNL⁻ formula is a pair $\mathbf{M} = \langle \langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle, \mathcal{V} \rangle$, where $\langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle$ is an interval structure and $\mathcal{V} : \mathbb{I}(\mathbb{D}) \rightarrow 2^{AP}$ is a *valuation function* assigning to every interval the set of propositional letters true on it. Given a model $\mathbf{M} = \langle \langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle, \mathcal{V} \rangle$ and an interval $[d_i, d_j] \in \mathbb{I}(\mathbb{T})^-$, the semantics of RPNL⁻ is defined recursively by the *satisfiability relation* \models as follows:

- for every propositional letter $p \in AP$, $\mathbf{M}, [t_i, t_j] \models p$ iff $p \in \mathcal{V}([t_i, t_j])$;
- $\mathbf{M}, [t_i, t_j] \models \neg\psi$ iff $\mathbf{M}, [t_i, t_j] \not\models \psi$;
- $\mathbf{M}, [t_i, t_j] \models \psi_1 \vee \psi_2$ iff $\mathbf{M}, [t_i, t_j] \models \psi_1$, or $\mathbf{M}, [t_i, t_j] \models \psi_2$;
- $\mathbf{M}, [t_i, t_j] \models \langle A \rangle\psi$ iff $\exists t_k \in T, t_k > t_j$ such that $\mathbf{M}, [t_j, t_k] \models \psi$.

We place ourselves in the most general setting and we do not impose any constraint on the valuation function. In particular, given interval $[d_i, d_j]$, it may happen that $p \in \mathcal{V}([d_i, d_j])$ and $p \notin \mathcal{V}([d'_i, d'_j])$ for all intervals $[d'_i, d'_j]$ (properly) included in $[d_i, d_j]$.

3 RPNL⁻ over linear vs. branching structures

As a preliminary step, we point out the differences between interpreting RPNL⁻ over linear structures and over branching ones. In [3], Bresolin and Montanari propose a branching-time extension of RPNL⁻, called BTNL[R]⁻, that interleaves operators that quantify over possible timelines with RPNL⁻ operators that quantify over intervals belonging to a given timeline, and they develop a doubly-exponential tableau-based decision procedure for it. In this paper, we keep the logic RPNL⁻ unchanged, but we interpret it over trees instead of linear orders. It is not difficult to show that whenever an RPNL⁻ formula is satisfiable over a linear structure it is also satisfiable over its branching variant. However, the opposite does not hold in general as shown by the following examples.

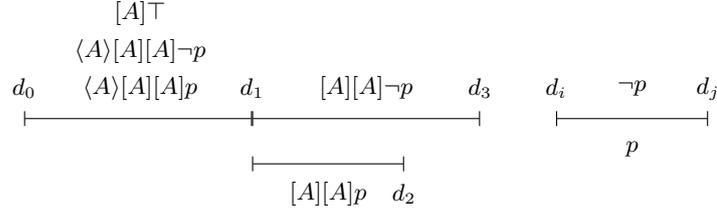


Fig. 1. Unsatisfiability of the formula φ_1 over a linear ordering.

As a first example consider the formula $\varphi_1 \equiv \langle A \rangle [A][A]p \wedge \langle A \rangle [A][A]\neg p \wedge [A]\langle A \rangle \top$ which states that there exists an interval in the future of the current one such that p holds over every interval in its future (the double $[A]$ allows us to refer to all intervals strictly to the right of the current one), there exists an interval in the future of the current one such that $\neg p$ holds over every interval in its future and the model is infinite (last conjunct). The formula φ_1 is clearly unsatisfiable on a linear order, as shown in Figure 1, because it requests the existence of an interval (in fact, infinite ones) over which both p and $\neg p$ hold. On the contrary, one can easily satisfy φ_1 on a branching structure imposing that one condition holds on a given branch of the model and the other condition holds on another one, as shown in Figure 2.

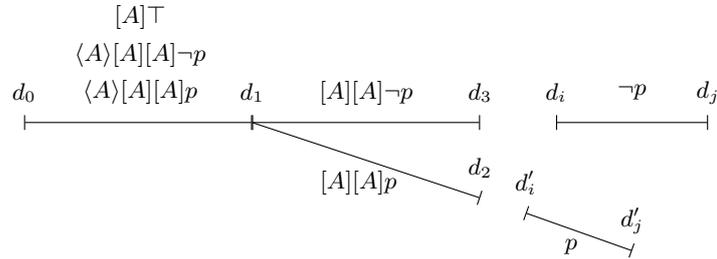


Fig. 2. A branching structure satisfying φ_1 .

As another example, consider the formula $\varphi_2 = \langle A \rangle (\langle A \rangle \top \wedge [A]\langle A \rangle \top) \wedge \langle A \rangle [A] \perp$. If we interpret it over an interval of a branching structure, it imposes that such an interval belongs to both an infinite path and a finite one (see Figure 3). On the contrary, such a formula is clearly unsatisfiable in a linear structure.

The differences between interpreting an RPNL^- formula over a branching structure instead of a linear one can be explained as follows. As we shall prove in the next section, in any model for an RPNL^- formula, atoms sharing their

right endpoints have the same temporal requests ($\langle A \rangle \psi$ and $[A] \psi$ formulae) and thus one can associate with any time point its set of temporal requests. This is true for both linear and branching structures. However, linear and branching structures differ from each other in two fundamental characteristics.

On the one hand, in linear structures there exists a single path over which all existential requests associated with a time point must be fulfilled, and thus the order in which they are fulfilled often plays a crucial role. In [4] Bresolin and Montanari prove a small model theorem that bounds the number of occurrences of time points with the same set of existential requests in a finite model, as well as in a finite psuedo-model representing an infinite model, for a RPNL^- formula over linear discrete orders. In branching structures, every existential request can be immediately satisfied in a distinct branch, that is, we can introduce as many branches as the existential requests associated with a given time point are and satisfy distinct requests over distinct branches.

On the other hand, in the case of linear structures, for any pair of time points, either the past of the first one includes that of the second one or vice versa, while in the case of branching structures the pasts of a pair of points can be only partially overlapped. Formally, the past of a time point can be described as a set of sets of temporal requests associated with different points in its past. In the linear case, we may need to consider an exponential number of such sets (exponential in the number of temporal requests), while in the branching case we may need to take into account a doubly exponential number of sets (we have an exponential number of distinct sets of temporal requests, and for each of them we may need to consider an exponential number of sets).

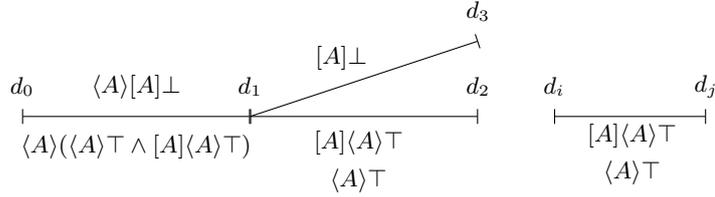


Fig. 3. A branching structure satisfying φ_2 .

4 A tableau for RPNL^- over branching structures

In this section, we define a tableau-based decision procedure for RPNL^- , we analyze its computational complexity, and we prove its soundness and completeness. We first introduce some preliminary notions. Let φ be an RPNL^- formula to be checked for satisfiability and let AP be the set of its propositional letters. For the sake of brevity, we use $(A)\psi$ as a shorthand for both $\langle A \rangle \psi$ and $[A] \psi$.

Definition 2. The closure $\text{CL}(\varphi)$ of φ is the set of all subformulae of φ and of their negations (we identify $\neg\neg\psi$ with ψ).

Definition 3. The set of temporal requests of φ is the set $\text{TF}(\varphi)$ of all temporal formulae in $\text{CL}(\varphi)$, that is, $\text{TF}(\varphi) = \{(A)\psi \in \text{CL}(\varphi)\}$.

By induction on the structure of φ , we can easily prove the following proposition.

Proposition 1. For every formula φ , $|\text{CL}(\varphi)|$ is less than or equal to $2 \cdot |\varphi|$, while $|\text{TF}(\varphi)|$ is less than or equal to $2 \cdot (|\varphi| - 1)$.

The notion of φ -atom is defined in the standard way.

Definition 4. A φ -atom is a set $A \subseteq \text{CL}(\varphi)$ such that:

- for every $\psi \in \text{CL}(\varphi)$, $\psi \in A$ iff $\neg\psi \notin A$;
- for every $\psi_1 \vee \psi_2 \in \text{CL}(\varphi)$, $\psi_1 \vee \psi_2 \in A$ iff $\psi_1 \in A$ or $\psi_2 \in A$.

We denote the set of all φ -atoms by A_φ . We have that $|A_\varphi| \leq 2^{|\varphi|}$. For any atom A we define the set of temporal requests of A as the set $\text{REQ}(A) = A \cap \text{TF}(\varphi)$. It's easy to prove that given a model $\mathbf{M} = \langle \langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle, \mathcal{V} \rangle$ for a RPNL^- formula φ one can construct a unique function $\mathcal{V}_A : \mathbb{I}(\mathbb{T})^- \rightarrow A_\varphi$ that associate to every interval $[d_i, d_j] \in \mathbb{I}(\mathbb{T})^-$ an atom A such that for every $\psi \in \text{CL}(\varphi)$, $\psi \in A$ if and only if $\mathbf{M}, [d_i, d_j] \models \psi$. Now we prove the following theorem that constrains atoms sharing the same endpoints.

Theorem 1. Let $\mathbf{M} = \langle \langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle, \mathcal{V} \rangle$ a model for a RPNL^- formula φ and $d_e, d_i, d_j \in \mathbb{T}$ with $d_i < d_e$ and $d_j < d_e$, if $\mathcal{V}_A([d_i, d_e]) = A$ and the atom associated to $\mathcal{V}_A([d_j, d_e]) = A'$ then $\text{REQ}(A) = \text{REQ}(A')$.

Proof. Suppose by contraddiction that $\text{REQ}(A) \neq \text{REQ}(A')$ then we can assume without loss of generality that exists $\langle A \rangle \psi \in \text{REQ}(A)$ and $\langle A \rangle \psi \notin \text{REQ}(A')$, by the definition of atom we have that $[A]\neg\psi \in \text{REQ}(A')$, since \mathbf{M} is a model there exists $d_h > d_e$ such that $\mathbf{M}, [d_e, d_h] \models \psi$ (for $\langle A \rangle \psi \in \text{REQ}(A)$) and for every $d_k > d_e$ $\mathbf{M}, [d_e, d_k] \models \neg\psi$ (for $[A]\neg\psi \in \text{REQ}(A')$) then $\mathbf{M}, [d_e, d_k] \models \neg\psi$ this leads to a contraddiction.

Atoms are connected by the following binary relation.

Definition 5. Let R_φ be a binary relation over A_φ such that, for every pair of atoms $A, A' \in A_\varphi$, $A R_\varphi A'$ if and only if, for every $[A]\psi \in \text{CL}(\varphi)$, if $[A]\psi \in A$, then $\psi \in A'$.

A tableau for RPNL^- is a suitable decorated tree \mathcal{T} . The decoration of each node n in \mathcal{T} , denoted by $\nu(n)$, is a triple $\langle [d_i, d_j], A, \mathbb{D} \rangle$, where $\mathbb{D} = \langle D, < \rangle$ is a prefix of \mathbb{N} , d_i and d_j , with $d_i < d_j$, belong to D , and A is an atom. The root r of \mathcal{T} is labeled by the empty decoration $\langle \emptyset, \emptyset \rangle$. Given a node n , we denote by $A(n)$ the atom component of $\nu(n)$.

Expansion rules. The construction of a tableau is based on the following expansion rules. Let n be a node in \mathcal{T} , decorated with $\langle [d_i, d_j], A_n, \mathbb{D}_n \rangle$. The following expansion rules can be possibly applied to n :

1. *Step rule*: if $d_j \geq d$ for all $d \in D_n$ and there exists at least one $\langle A \rangle$ -formula in A_n , then choose a set of atoms $\{A'_1, \dots, A'_k\}$ such that, $A_n R_\varphi A'_i$ for every A'_i and for every $\langle A \rangle \psi \in A_n$ there exists A'_i such that $\psi \in A'_i$. For every A'_i add an immediate successor n' to n decorated with $\langle [d_j, d_{j+1}], A'_i, \mathbb{D}' \rangle$, where \mathbb{D}' is obtained from \mathbb{D}_n by adding a new point d_{j+1} after all points in \mathbb{D}_n .
2. *Fill-in rule*: if there exists a point $d_k > d_j$ such that there are no ancestors n' of n with decoration $\langle [d_j, d_k], A', \mathbb{D}' \rangle$, for some A' and \mathbb{D}' , then take any atom A'' such that $A_n R_\varphi A''$ and $\text{REQ}(A'') = \text{REQ}(\bar{A})$, for all ancestors \bar{n} of n with $\nu(\bar{n}) = \langle [\bar{d}, d_k], \bar{A}, \bar{\mathbb{D}} \rangle$ (since d_k belongs to \mathbb{D}_n , there exists at least one of them), for some \bar{d}, \bar{A} , and $\bar{\mathbb{D}}$, and add an immediate successor n'' to n , with $\nu(n'') = \langle [d_k, d_j], A'', \mathbb{D}_n \rangle$.

Both rules add new successors to n . However, while the step rule decorates such nodes with a new interval ending at a new point d_{j+1} , the fill-in rule decorates them with a new interval whose endpoints were already in \mathbb{D}_n .

From the definition of the *Fill-in rule*, it follows that all atoms associated with intervals ending at the same point d agree on their temporal requests. Hereinafter, we denote by $\text{REQ}(d)$ their common set of requests.

Blocking condition. To guarantee the termination of the method, we need a suitable *blocking condition* to avoid the infinite application of the expansion rules in the case of infinite models. We say that a node n , decorated with $\langle [d_i, d_j], A_n, \mathbb{D}_n \rangle$ is *blocked* if there exists an ancestor n' of n , decorated with $\langle [d_k, d_l], A_{n'}, \mathbb{D}_{n'} \rangle$, with $d_l < d_j$, such that:

- $\text{REQ}(A_n) = \text{REQ}(A_{n'})$;
- for all $d_h < d_j$ there exists $d_m < d_l$ such that $\text{REQ}(d_h) = \text{REQ}(d_m)$.

Roughly speaking we block a node n if it has an ancestor n' with the same set of temporal requests and every set of temporal requests that occurs in a path from n' to n occurs in an ancestor of n' .

Expansion strategy. Given a decorated tree \mathcal{T} and a node n , we say that an expansion rule is *applicable to n* if n is non-blocked and the application of the rule generates at least one new node. The *branch expansion strategy* for a leaf node n , decorated with $\langle [d_i, d_j], A_n, \mathbb{D}_n \rangle$, is the following one:

1. if the fill-in rule is applicable, apply the fill-in rule to n ;
2. if the fill-in rule is not applicable and there exists a point $d_k < d_j$, such that there are no ancestors of n with decoration $\langle [d_k, d_j], A', \mathbb{D}' \rangle$, for some $A' \in A_\varphi$, *close* the node n ;
3. if the fill-in rule is not applicable, n is not closed, then apply the step rule to n .

Tableau. Let φ be the formula to be checked for satisfiability, an *initial tableau* for φ is a decorated tree with only one node $\langle [d_0, d_1], A, \{d_0 < d_1\} \rangle$, where A is an atom containing φ .

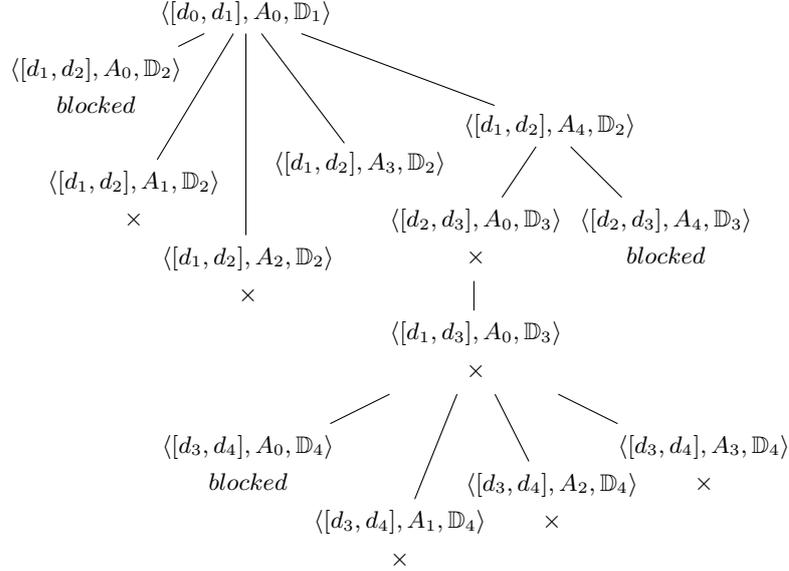


Fig. 4. A tableau for the formula $\varphi_2 \equiv \langle A \rangle (\langle A \rangle \top \wedge [A] \langle A \rangle \top) \wedge \langle A \rangle [A] \perp$

A *tableau* for φ is any decorated tree \mathcal{T} obtained by expanding the initial tableau for φ through successive applications of the branch-expansion strategy to currently existing branches, until the branch-expansion strategy cannot be applied anymore.

Pruning the tableau. Given a tableau \mathcal{T} for a formula φ , we apply the following pruning procedure until no nodes can be removed:

1. remove any closed node from the tableau;
2. remove any node n without successor such that the fill-in rule has been applied to it in the tableau construction;
3. remove any node n such that the step rule has been applied to it in the tableau construction and there exists $\langle A \rangle \psi \in A_n$ such that there are no successors of n labeled with ψ ;
4. remove any node that is not reachable from the root.

It will turn out that a formula φ is satisfiable if and only if there exists a final tableau for it that is not empty.

4.1 An example

To understand how the tableau-based procedure works consider the formula $\varphi_2 \equiv \langle A \rangle (\langle A \rangle \top \wedge [A] \langle A \rangle \top) \wedge \langle A \rangle [A] \perp$ introduced in Section 3. A non-empty

final tableau for φ_2 is represented in Figure 4. We denote with ψ the formula $\langle A \rangle \top \wedge [A] \langle A \rangle \top$. The atoms used in the tableau construction are $A_0 = \{\langle A \rangle \top, \langle A \rangle [A] \perp, \langle A \rangle \psi, \neg \psi, \varphi_2\}$, $A_1 = \{[A] \perp, \langle A \rangle [A] \perp, \langle A \rangle \psi, \neg \psi, \varphi_2\}$, $A_2 = \{[A] \perp, [A] \langle A \rangle \top, \langle A \rangle \psi, \neg \psi, \neg \varphi_2\}$, $A_3 = \{[A] \perp, [A] \langle A \rangle \top, [A] \neg \psi, \neg \psi, \neg \varphi_2\}$, and $A_4 = \{\langle A \rangle \top, [A] \langle A \rangle \top, \langle A \rangle \psi, \psi, \neg \varphi_2\}$. \mathbb{D}_i stands for the linear order $\{d_0 < \dots < d_i\}$.

The root of the tableau is the node $\langle [d_0, d_1], A_0, \mathbb{D}_1 \rangle$. We apply the step rule on it choosing the set $\{A_0, A_1, A_2, A_3, A_4\}$. The node $\langle [d_1, d_2], A_0, \mathbb{D}_2 \rangle$ satisfies $\langle A \rangle \top$ and it is blocked since A_0 is repeated. $\langle A \rangle [A] \perp$ is satisfied by the node $\langle [d_1, d_2], A_3, \mathbb{D}_2 \rangle$. Since A_3 does not contain $\langle A \rangle$ -formulae this node is open and it is not expanded further. We satisfy $\langle A \rangle \psi$ with the node $\langle [d_1, d_2], A_4, \mathbb{D}_2 \rangle$ that, as we will show later, is open. Every node containing A_1 or A_2 is closed because these atoms contain both the formula $[A] \perp$, that imposes on the current node to have no successors, and an $\langle A \rangle$ -formula that imposes on the node to have at least one successor. Consider now the node $\langle [d_1, d_2], A_4, \mathbb{D}_2 \rangle$: to satisfy $\langle A \rangle \top$ and $\langle A \rangle \psi$, we apply the step rule choosing the set $\{A_0, A_4\}$. Then we apply the fill-in rule to the successor labeled $\langle [d_2, d_3], A_0, \mathbb{D}_3 \rangle$ and we add a successor labelled $\langle [d_1, d_3], A_0, \mathbb{D}_3 \rangle$ to it. Now we have to apply the step rule on $\langle [d_1, d_3], A_0, \mathbb{D}_3 \rangle$: we choose to use the same successors we used before for $\langle [d_0, d_1], A_0, \mathbb{D}_1 \rangle$. With this choice for the successors, we can't satisfy the formula $\langle A \rangle [A] \perp$ since the node $\langle [d_3, d_4], A_3, \mathbb{D}_4 \rangle$ is closed because the fill-in rule is not applicable on $[d_2, d_4]$ for this branch (an atom cannot contain both $\langle A \rangle \top$ and $[A] \perp$). For this reason $\langle [d_1, d_3], A_0, \mathbb{D}_3 \rangle$ will be removed by the pruning procedure, as well as its predecessor $\langle [d_2, d_3], A_0, \mathbb{D}_3 \rangle$, and as well as the node $\langle [d_3, d_4], A_0, \mathbb{D}_4 \rangle$ (which will be removed since it is no more reachable from the root). However, since the node $\langle [d_2, d_3], A_4, \mathbb{D}_3 \rangle$ is blocked and satisfies all the $\langle A \rangle$ -formulae of $\langle [d_1, d_2], A_4, \mathbb{D}_2 \rangle$, the latter node will not be removed. This shows that the final tableau depicted in Figure 4 is not empty, from which we can correctly conclude that φ_2 is satisfiable.

4.2 Computational complexity

As a preliminary step, we show that the proposed tableau method terminates; then we analyze its computational complexity.

In order to prove termination of the tableau method, we give a bound on the length of any branch B of any tableau for φ :

1. by the blocking condition, after at most $O(2^{2 \cdot TF(\varphi)}) = O(2^{2 \cdot n})$, with $n = |\varphi|$, applications of the step rule, the expansion strategy cannot be applied anymore to a branch ($REQ(d)$ can take $2^{TF(\varphi)}$ different values and there can be at most $2^{TF(\varphi)}$ different sets of requests associated with time points $d' < d$);
2. given a branch B , between two successive applications of the step rule, the fill-in rule can be applied at most k times, where k is the current number of elements in the linear ordering D_n labelling the last node of B (k is exactly the number of applications of the step rule up to that point).

Hence, the length of a branch is (at most) exponential in $|\varphi|$.

Theorem 2 (Termination). *The tableau method for RPNL^- terminates.*

Proof. Given a formula φ , let \mathcal{T} be a tableau for φ . Since, by construction, every node of \mathcal{T} has a finite outgoing degree and every branch of it is of finite length, by König's Lemma, \mathcal{T} is finite.

Since the outgoing degree of every node is bounded by the number of requests, that is, $|\text{REQ}(\varphi)| \leq n$, the size of a final tableau is $O(2^{2^n})$. However, the proposed decision procedure does not need to explicitly generate the whole tableau. It can indeed keep track of a branch at a time and expand it in a non-deterministic way, until it is either blocked or closed (in this case the procedure returns fail). Given that the length of any branch is at most $O(2^{2^n})$, the proposed decision procedure is in EXPSPACE.

4.3 Soundness and completeness

The soundness and completeness of the proposed method can be proved as follows. Soundness is proved by showing how it is possible to construct a model satisfying φ from a non-empty final tableau \mathcal{T} for φ . The proof must encompass both the case of blocked nodes and that of non-blocked ones. Proving completeness consists in showing that for any satisfiable formula φ there exists a non-empty final tableau for it.

Theorem 3 (Soundness). *Given a formula φ and a final tableau \mathcal{T} for φ , if \mathcal{T} is non-empty, then φ is satisfiable.*

Proof. Let \mathcal{T} be a non-empty final tableau for φ . We show that, by visiting \mathcal{T} , we can build up a model satisfying φ . We start from the root n_0 of the tableau. Since \mathcal{T} is not-empty, we choose one of the successors of n_0 . By the definition of initial tableau, such a successor n is decorated with $\langle [d_0, d_1], A_n, \{d_0 < d_1\} \rangle$. Thus, we consider the two-nodes tree $\mathbb{T}_0 = \{d_0, d_1\}$ and we define the initial model $\mathbf{M}_0 = \langle \langle \mathbb{T}_0, \mathbb{I}(\mathbb{T}_0)^- \rangle, \mathcal{V}_0 \rangle$ where $\mathcal{V}[d_0, d_1] = \{p \in A_n\}$. Now, suppose that we have built a partial model \mathbf{M} and that we have reached a node n in the tableau decorated with $\langle [d_i, d_j], A_n, \mathbb{D}_n \rangle$. We proceed by induction on the expansion rule that has been applied to n .

- *No expansion rules have been applied to n , but n is not blocked.* In this case there are no $\langle A \rangle$ -formulae in A_n and we do not need to expand the model.
- *The step rule has been applied to n .* For every formula $\langle A \rangle \psi \in A_n$ we take a successor n_ψ of n such that $\psi \in A(n_\psi)$, we add an immediate successor d_ψ to d_j in the tree and expand the model by putting $\mathcal{V}[d_j, d_\psi] = \{p \in A(n_\psi)\}$.
- *The fill-in rule has been applied to n .* The successor of n is labelled with an interval $[d_k, d_j]$ and with an atom A' . We expand the model by putting $\mathcal{V}[d_k, d_j] = \{p \in A'\}$.
- *n is blocked.* In this case there exists an ancestor n' of n such that $\text{REQ}(A_n) = \text{REQ}(A_{n'})$ and for all $d_h < d_j$ there exists $d_m < d_l$ such that $\text{REQ}(d_h) = \text{REQ}(d_m)$. Since no new atoms are introduced between n' and n we can proceed with the construction from n repeating the construction that starts from n' .

At the end of such a (possibly infinite) construction, we have built a model satisfying φ .

Theorem 4 (Completeness). *Given a satisfiable formula φ , there exists a non-empty final tableau \mathcal{T} for φ .*

Proof. Let φ be a satisfiable formula and let $\mathbf{M} = \langle \langle \mathbb{T}, \mathbb{I}(\mathbb{T})^- \rangle, \mathcal{V} \rangle$ be a model for it. We prove that there exists a non-empty final tableau \mathcal{T} which corresponds to \mathbf{M} . Since \mathbf{M} is a model for φ , there exists an interval $[d_0, d_1]$ such that $\mathbf{M}, [d_0, d_1] \Vdash \varphi$. Let $A_0 = \{\psi \in \text{CL}(\varphi) : \mathbf{M}, [d_0, d_1] \Vdash \psi\}$. By the definition of initial tableau there exists a successor n_0 of the root labelled with $\langle [d_0, d_1], A_0, \{d_0 < d_1\} \rangle$. We start the construction of the non-empty tableau for φ from n_0 , and we proceed by induction on the expansion rules. Suppose that we have built a partial tableau \mathcal{T} and that we are considering a node n decorated with $\langle [d_i, d_j], A_n, \mathbb{D}_n \rangle$. Three cases arise.

- *The fill-in rule is applicable to n .* Let d_k be the point such that there are no nodes labelled with $[d_k, d_j]$. We add a successor n' to n labelled with $\langle [d_k, d_j], A', \mathbb{D}' \rangle$, where $A' = \{\psi \in \text{CL}(\varphi) : \mathbf{M}, [d_k, d_j] \Vdash \psi\}$, and we proceed in the construction from n' .
- *The step rule is applicable to n .* Hence, for every $\langle A \rangle \psi \in A_n$ there exists an interval $[d_j, d_\psi]$ such that $\mathbf{M}, [d_j, d_\psi] \Vdash \psi$. Let $A_\psi = \{\theta \in \text{CL}(\varphi) : \mathbf{M}, [d_j, d_\psi] \Vdash \theta\}$. For every $\langle A \rangle \psi \in A_n$ we add a successor n_ψ to n decorated with $\langle [d_j, d_\psi], A_\psi, \mathbb{D}_n \cup \{d_\psi\} \rangle$ and we proceed inductively on such n_ψ .
- *No rules are applicable to n .* Since \mathbf{M} is a model for φ , n cannot be a closed node. Hence, either there are no $\langle A \rangle$ -formulae in A_n or n is blocked.

It can be proved that the tableau \mathcal{T} obtained at the end of the above procedure is a non empty tableau for φ .

5 Conclusions

In this paper, we focused our attention on interval logics of temporal neighborhood. We addressed the satisfiability problem for the future fragment of strict Neighborhood Logic (RPNL⁻), interpreted over trees, and we showed how to solve it by providing a sound and complete tableau-based decision procedure of EXPSPACE complexity.

We do not know if our procedure is of optimal complexity or not. A first development of this work will be the study of the complexity of the satisfiability problem for the logic RPNL⁻ interpreted over trees. A more valuable, yet more difficult, development would be the extension of the proposed tableau method to the case of full PNL interpreted over trees and over general partial orders with the linear interval property (where intervals are always linear).

References

1. D. Bresolin, V. Goranko, A. Montanari, and P. Sala. Tableau systems for logics of subinterval structures over dense orderings. In *Proc. of TABLEAUX 2007*, volume 4548 of *LNAI*, pages 73–89. Springer, 2007.
2. D. Bresolin, V. Goranko, A. Montanari, and G. Sciavicco. On Decidability and Expressiveness of Propositional Interval Neighborhood Logics. In *Proc. of the International Symposium on Logical Foundations of Computer Science (LFCS)*, volume 4514 of *LNCS*, pages 84–99. Springer, 2007.
3. D. Bresolin and A. Montanari. A tableau-based decision procedure for a branching-time interval temporal logic. In H. Schlingloff, editor, *Proc. of the 4th Int. Workshop on Methods for Modalities*, pages 38–53, 2005.
4. D. Bresolin and A. Montanari. A tableau-based decision procedure for right propositional neighborhood logic. In *Proc. of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 63–77. Springer, 2005.
5. D. Bresolin, A. Montanari, and P. Sala. An optimal tableau-based decision algorithm for Propositional Neighborhood Logic. In *Proc. of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 4393 of *LNCS*, pages 549–560. Springer, 2007.
6. D. Bresolin, A. Montanari, and G. Sciavicco. An optimal decision procedure for Right Propositional Neighborhood Logic. *Journal of Automated Reasoning*, 38(1-3):173–199, 2007.
7. E. Emerson and J. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.
8. V. Goranko, A. Montanari, and G. Sciavicco. A general tableau method for propositional interval temporal logic. In *Proc. of TABLEAUX 2003*, volume 2796 of *LNAI*, pages 102–116. Springer, 2003.
9. V. Goranko, A. Montanari, and G. Sciavicco. Propositional interval neighborhood temporal logics. *Journal of Universal Computer Science*, 9(9):1137–1167, 2003.
10. V. Goranko, A. Montanari, and G. Sciavicco. A road map of interval temporal logics and duration calculi. *Journal of Applied Non-Classical Logics*, 14(1–2):9–54, 2004.
11. V. Goranko, A. Montanari, G. Sciavicco, and P. Sala. A general tableau method for propositional interval temporal logics: Theory and implementation. *Journal of Applied Logic*, 4(3):305–330, 2006.
12. Joseph Y. Halpern and Yoav Shoham. A propositional modal logic of time intervals. *Journal of the ACM*, 38(4):935–962, October 1991.
13. A. Montanari. Propositional interval temporal logics: some promising paths. In *Proc. of the 12th International Symposium on Temporal Representation and Reasoning (TIME)*, pages 201–203. IEEE Computer Society Press, 2005.
14. A. Montanari, G. Sciavicco, and N. Vitacolonna. Decidability of interval temporal logics over split-frames via granularity. In *Proc. of the 8th European Conference on Logics in AI*, volume 2424 of *LNAI*, pages 259–270. Springer, 2002.
15. B. Moszkowski. *Reasoning about digital circuits*. Tech. rep. stan-cs-83-970, Dept. of Computer Science, Stanford University, Stanford, CA, 1983.
16. M. Otto. Two variable first-order logic over ordered domains. *Journal of Symbolic Logic*, 66(2):685–702, 2001.
17. I. Shapirovsky. On PSPACE-decidability in Transitive Modal Logic. In R. Schmidt, I. Pratt-Hartmann, M. Reynolds, and H. Wansing, editors, *Advances in Modal Logic*, volume 5, pages 269–287. King’s College Publications, London, 2005.

18. I. Shapirovsky and V. Shehtman. Chronological future modality in Minkowski spacetime. In P. Balbiani, N. Y. Suzuki, F. Wolter, and M. Zakharyashev, editors, *Advances in Modal Logic*, volume 4, pages 437–459. King's College Publications, London, 2003.
19. Y. Venema. A modal logic for chopping intervals. *Journal of Logic and Computation*, 1(4):453–476, 1991.

KLM Logics of Nonmonotonic Reasoning: Calculi and Implementations

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

¹ Dipartimento di Informatica - Università del Piemonte Orientale “A. Avogadro” -
via Bellini 25/G - 15100 Alessandria, Italy

`laura@mf.n.unipmn.it`

² Dipartimento di Informatica - Università degli Studi di Torino, corso Svizzera 185 -
10149 Turin - Italy

`{gliozzi,pozzato}@di.unito.it`

³ LSIS - UMR CNRS 6168 Université Paul Cézanne (Aix-Marseille 3), Avenue
Escadrille Normandie-Niemen 13397 Marseille Cedex 20 - France

`nicola.olivetti@univ-cezanne.fr`

Abstract. We present proof methods for the logics of nonmonotonic reasoning defined by Kraus, Lehmann and Magidor (KLM). We introduce tableaux calculi (called \mathcal{TS}^T) for all KLM logics. We provide decision procedures for KLM logics based on these calculi, and we study their complexity. Finally, we describe KLMLean 2.0, a theorem prover implementing the calculi \mathcal{TS}^T inspired by the “lean” methodology. KLMLean 2.0 is implemented in SICStus Prolog and it also comprises a graphical interface written in Java¹.

1 Introduction

In the early 90' [13, 14] Kraus, Lehmann and Magidor (from now on KLM) proposed a formalization of non-monotonic reasoning that was early recognized as a landmark. Their work stemmed from two sources: the theory of nonmonotonic consequence relations initiated by Gabbay [8] and the preferential semantics proposed by Shoham [16] as a generalization of Circumscription. Their works lead to a classification of nonmonotonic consequence relations, determining a hierarchy of stronger and stronger systems.

According to the KLM framework, defeasible knowledge is represented by a (finite) set of nonmonotonic conditionals or assertions of the form

$$A \sim B$$

whose reading is *normally (or typically) the A's are B's*. The operator “ \sim ” is nonmonotonic, in the sense that $A \sim B$ does not imply $A \wedge C \sim B$. For instance, a

¹ This research has been partially supported by “Progetto Lagrange - Fondazione CRT” and by the projects “MIUR PRIN05: Specification and verification of agent interaction protocols” and “GALILEO 2006: Interazione e coordinazione nei sistemi multi-agenti”.

knowledge base K may contain $\text{football_lover} \sim \text{bet}$, $\text{football_player} \sim \text{football_lover}$, $\text{football_player} \sim \neg \text{bet}$, whose meaning is that people loving football typically bet on the result of a match, football players typically love football but they typically do not bet (especially on matches they are going to play...). If \sim were interpreted as classical (or intuitionistic) implication, one would get $\text{football_player} \sim \perp$, i.e. typically there are not football players, thereby obtaining a trivial knowledge base.

In KLM framework, one can derive new conditional assertions from the knowledge base by means of a set of inference rules. The set of adopted inference rules defines some fundamental types of inference systems, namely, from the strongest to the weakest: Rational (**R**), Preferential (**P**), Loop-Cumulative (**CL**), and Cumulative (**C**) logic. In all these systems one can infer new assertions without incurring the trivializing conclusions of classical logic: in the above example, in none of the systems, one can infer $\text{football_player} \sim \text{bet}$. In cumulative logics (both **C** and **CL**) one can infer $\text{football_lover} \wedge \text{football_player} \sim \neg \text{bet}$, giving preference to more specific information; in Preferential logic **P** one can also infer that $\text{football_lover} \sim \neg \text{football_player}$; in the rational case **R**, if one further knows that $\neg(\text{football_lover} \sim \text{rich})$, that is to say it is not the case that football lovers are typically rich persons, one can also infer that $\text{football_lover} \wedge \neg \text{rich} \sim \text{bet}$.

From a semantic point of view, to the each logic (**R**, **P**, **CL**, **C**) corresponds one kind of models, namely, possible-world structures equipped with a preference relation among worlds or states. More precisely, for **P** we have models with a preference relation (an irreflexive and transitive relation) on worlds. For the stronger **R** the preference relation is further assumed to be *modular*. For the weaker logic **CL**, the preference relation is defined on *states*, where a state can be identified, intuitively, with a set of worlds. In the weakest case of **C**, the preference relation is on states, as for **CL**, but it is no longer assumed to be transitive. In all cases, the meaning of a conditional assertion $A \sim B$ is that B holds in the *most preferred* worlds/states where A holds.

A recent result by Halpern and Friedman [6] has shown that preferential and rational logic are quite natural and general systems: surprisingly enough, the axiom system of preferential (likewise of rational logic) is complete with respect to a wide spectrum of semantics, from ranked models, to parametrized probabilistic structures, ϵ -semantics and possibilistic structures. The reason is that all these structures are examples of *plausibility structures* and the truth in them is captured by the axioms of preferential (or rational) logic. These results, and their extensions to the first order setting [7] are the source of a renewed interest in KLM framework.

Even if KLM was born as an inferential approach to nonmonotonic reasoning, curiously enough, there has not been much investigation on deductive mechanisms for these logics. In short, the state of the art is as follows:

- Lehmann and Magidor [14] have proved that validity in **P** is **coNP**-complete. Their decision procedure for **P** is more a theoretical tool than a practical algorithm, as it requires to guess sets of indexes and propositional evaluations. They have also provided another procedure for **P** that exploits its reduction

- to **R**. However, the reduction of **P** to **R** breaks down if boolean combinations of conditionals are allowed, indeed it is exactly when such combinations are allowed that the difference between **P** and **R** arises.
- A tableau proof procedure for **C** has been given in [1]. Their tableau procedure is fairly complicated; it uses labels and it contains a cut-rule. Moreover, it is not clear how it can be adapted to **CL** and **P**.
 - In [11] it is defined a labelled tableau calculus for the conditional logic **CE** whose flat fragment (i.e. without nested conditionals) corresponds to **P**. That calculus needs a fairly complicated loop-checking mechanism to ensure termination. It is not clear if it matches complexity bounds and if it can be adapted in a simple way to **CL**.
 - Finally, decidability of **P** and **R** has also been obtained by interpreting them into standard modal logics, as it is done by Boutilier [4]. However, his mapping is not very direct and natural, as we discuss below.

In this work we present tableau procedures for all KLM logics called \mathcal{TS}^T , where **S** stands for $\{\mathbf{R}, \mathbf{P}, \mathbf{CL}, \mathbf{C}\}$. Our approach is based on a novel interpretation of **P** into modal logics. As a difference with previous approaches (e.g. Crocco and Lamarre [5] and Boutilier [4]), that take S4 as the modal counterpart of **P**, we consider here Gödel-Löb modal logic of provability G (see for instance [12]). Our tableau method provides a sort of run-time translation of **P** into modal logic G. The idea is simply to interpret the preference relation as an accessibility relation: a conditional $A \sim B$ holds in a model if B is true in all minimal A -worlds w (i.e. worlds in which A holds and that are minimal). An A -world w is a minimal A -world if all smaller worlds are not A -worlds. The relation with modal logic G is motivated by the fact that we assume, following KLM, the so-called *smoothness condition*, which is related to the well-known *limit assumption*. This condition ensures that minimal A -worlds exist whenever there are A -worlds, by preventing infinitely descending chains of worlds. This condition therefore corresponds to the finite-chain condition on the accessibility relation (as in modal logic G). Therefore, our interpretation of conditionals is different from the one proposed by Boutilier, who rejects the smoothness condition and then gives a less natural (and more complicated) interpretation of **P** into modal logic S4. We are able to extend our approach to the cases of **CL** and **C** by using a second modality which takes care of states. As a difference with **P** and **CL**, the calculus for **C** requires a sort of (analytic) cut rule to account for the smoothness condition. We also consider the case of the strongest logic **R**; as for the other weaker systems, our approach is based on an interpretation of **R** into an extension of modal logic G, including modularity of the preference relation (previous approaches [5, 4] take S4.3 as the modal counterpart of **R**). As a difference with the tableau calculi introduced for **P**, **CL**, and **C**, here we present a *labelled* tableau system, which seems to be the most natural approach in order to capture the modularity of the preference relation.

The tableau calculi mentioned above can be used to define a systematic procedure which allows the satisfiability problem for **R**, **P**, and **CL** to be decided in nondeterministic polynomial time; moreover, we obtain a complexity bound

for these logics, namely that they are **coNP**-complete. For **R** and **P**, this is in accordance with the known complexity results, whereas for **CL** this bound is new, to the best of our knowledge. The calculus \mathcal{TC}^T gives nonetheless a decision procedure for this logic too.

Finally, we present KLMLean 2.0, a theorem prover implementing \mathcal{TS}^T calculi in SICStus Prolog. KLMLean 2.0 is inspired by the “lean” methodology [3], and it also comprises a graphical interface written in Java. For the rational logic **R**, KLMLean 2.0 offers two different versions: 1. a simple version, where Prolog *constants* are used to represent \mathcal{TR}^T 's labels; 2. a more efficient one, where labels are represented by Prolog *variables*, inspired by the free-variable tableau presented in [2]. To the best of our knowledge, KLMLean 2.0 is the first theorem prover for KLM logics.

The plan of the paper is as follows: in section 2, we recall KLM logics, and we show how their semantics can be represented by standard Kripke models. In section 3, we present the tableaux calculi \mathcal{TS}^T and we elaborate them in order to obtain a terminating procedure. Also, we summarize complexity results obtained by analyzing \mathcal{TS}^T . In section 4 we present KLMLean 2.0, a theorem prover written in SICStus Prolog implementing the \mathcal{TS}^T calculi.

2 KLM Logics

We briefly recall the axiomatizations and semantics of the KLM systems. For the sake of exposition, we present the systems in the order from the strongest to the weakest: **R**, **P**, **CL**, and **C**. For a complete picture of KLM systems, see [13, 14]. The language of KLM logics consists just of conditional assertions $A \sim B$. We consider a richer language allowing boolean combinations of assertions and propositional formulas. Our language \mathcal{L} is defined from a set of propositional variables ATM , the boolean connectives and the conditional operator \sim . We use A, B, C, \dots to denote propositional formulas (that do not contain conditional formulas), whereas F, G, \dots are used to denote all formulas (including conditionals); Γ, Δ, \dots represent sets of formulas, whereas X, Y, \dots denote sets of sets of formulas. The formulas of \mathcal{L} are defined as follows: if A is a propositional formula, $A \in \mathcal{L}$; if A and B are propositional formulas, $A \sim B \in \mathcal{L}$; if F is a boolean combination of formulas of \mathcal{L} , $F \in \mathcal{L}$.

The axiomatization of **R** consists of all axioms and rules of propositional calculus together with the following axioms and rules. We use \vdash_{PC} to denote provability in the propositional calculus, whereas \vdash is used to denote provability in **R**:

- REF. $A \sim A$ (reflexivity)
- LLE. If $\vdash_{PC} A \leftrightarrow B$, then $\vdash (A \sim C) \rightarrow (B \sim C)$ (left logical equivalence)
- RW. If $\vdash_{PC} A \rightarrow B$, then $\vdash (C \sim A) \rightarrow (C \sim B)$ (right weakening)
- CM. $((A \sim B) \wedge (A \sim C)) \rightarrow (A \wedge B \sim C)$ (cautious monotonicity)
- AND. $((A \sim B) \wedge (A \sim C)) \rightarrow (A \sim B \wedge C)$
- OR. $((A \sim C) \wedge (B \sim C)) \rightarrow (A \vee B \sim C)$

– RM. $((A \sim B) \wedge \neg(A \sim \neg C)) \rightarrow ((A \wedge C) \sim B)$ (rational monotonicity)

REF states that A is always a default conclusion of A . LLE states that the syntactic form of the antecedent of a conditional formula is irrelevant. RW describes a similar property of the consequent. This allows to combine default and logical reasoning [6]. CM states that if B and C are two default conclusions of A , then adding one of the two conclusions to A will not cause the retraction of the other conclusion. AND states that it is possible to combine two default conclusions. OR states that it is allowed to reason by cases: if C is the default conclusion of two premises A and B , then it is also the default conclusion of their disjunction. RM is the rule of *rational monotonicity*, which characterizes the logic **R**: if $A \sim B$ and $\neg(A \sim \neg C)$ hold, then one can infer $A \wedge C \sim B$. This rule allows a conditional to be inferred from a set of conditionals in absence of other information. More precisely, “it says that an agent should not have to retract any previous defeasible conclusion when learning about a new fact the negation of which was not previously derivable” [14].

The axiomatization of **P** can be obtained from the axiomatization of **R** by removing the axiom RM.

The axiomatization of **CL** can be obtained from the axiomatization of **P** by removing the axiom OR and by adding the following infinite set of LOOP axioms:

LOOP. $(A_0 \sim A_1) \wedge (A_1 \sim A_2) \dots (A_{n-1} \sim A_n) \wedge (A_n \sim A_0) \rightarrow (A_0 \sim A_n)$

and the following axiom CUT:

CUT. $((A \sim B) \wedge (A \wedge B \sim C)) \rightarrow (A \sim C)$

Notice that these axioms are derivable in **P** (and therefore in **R**). The weakest logical system considered by KLM is Cumulative Logic **C**, whose axiom system can be obtained from the one for **CL** by removing the set of (LOOP) axioms.

The semantics of KLM logics is defined by considering possible world (or possible states) structures with a *preference relation* $w < w'$ among worlds (or states), whose meaning is that w is preferred to w' . $A \sim B$ holds in a model \mathcal{M} if B holds in all *minimal worlds (states)* where A holds. This definition makes sense provided minimal worlds for A exist whenever there are A -worlds (A -states): this is ensured by the *smoothness condition* defined below.

Definition 1 (Rational and Preferential models). *A rational model is a triple*

$$\mathcal{M} = \langle \mathcal{W}, <, V \rangle$$

where \mathcal{W} is a non-empty set of items called worlds, $<$ is an irreflexive, transitive and modular² relation on \mathcal{W} , and V is a function $V : \mathcal{W} \mapsto 2^{ATM}$, which assigns to every world w the set of atoms holding in that world. The truth conditions for a formula F are as follows:

² A relation $<$ is *modular* if the following condition holds: for each u, v, w , if $u < v$, then either $w < v$ or $u < w$.

- if F is a boolean combination of formulas, $\mathcal{M}, w \models F$ is defined as for propositional logic, namely:
 - if F is an atom $P \in ATM$, then $\mathcal{M}, w \models P$ iff $P \in V(w)$;
 - if F is a negation $\neg G$, then $\mathcal{M}, w \models \neg G$ iff $\mathcal{M}, w \not\models G$;
 - if F is a conjunction $G_1 \wedge G_2$, then $\mathcal{M}, w \models G_1 \wedge G_2$ iff $\mathcal{M}, w \models G_1$ and $\mathcal{M}, w \models G_2$;
 - if F is a disjunction $G_1 \vee G_2$, then $\mathcal{M}, w \models G_1 \vee G_2$ iff $\mathcal{M}, w \models G_1$ or $\mathcal{M}, w \models G_2$;
 - if F is an implication $G_1 \rightarrow G_2$, then $\mathcal{M}, w \models G_1 \rightarrow G_2$ iff $\mathcal{M}, w \not\models G_1$ or $\mathcal{M}, w \models G_2$.

Let A be a propositional formula; we define $Min_{<}(A) = \{w \in \mathcal{W} \mid \mathcal{M}, w \models A \text{ and } \forall w', w' < w \text{ implies } \mathcal{M}, w' \not\models A\}$. We define:

- $\mathcal{M}, w \models A \sim B$ if for all w' , if $w' \in Min_{<}(A)$ then $\mathcal{M}, w' \models B$.

We also define the smoothness condition on the preference relation: if $\mathcal{M}, w \models A$, then $w \in Min_{<}(A)$ or $\exists w' \in Min_{<}(A)$ s.t. $w' < w$. Validity and satisfiability of a formula are defined as usual.

A preferential model is defined as the rational/preferential model, with the only difference that the preference relation $<$ is no longer assumed to be modular.

Observe that the above definition of rational model extends the one given by KLM to boolean combinations of formulas. Notice also that the truth conditions for conditional formulas are given with respect to single possible worlds for uniformity sake. Since the truth value of a conditional only depends on global properties of \mathcal{M} , we have that: $\mathcal{M}, w \models A \sim B$ iff $\mathcal{M} \models A \sim B$.

Models for (loop-)cumulative logics also comprise states:

Definition 2 (Loop-Cumulative and Cumulative models). A (loop-)cumulative model is a tuple

$$\mathcal{M} = \langle S, \mathcal{W}, l, <, V \rangle$$

where S is a set of states and $l : S \mapsto 2^{\mathcal{W}}$ is a function that labels every state with a nonempty set of worlds; $<$ is defined on S , it satisfies the smoothness condition and it is irreflexive and transitive in **CL**, whereas it is only irreflexive in **C**.

For $s \in S$ and A propositional, we let $\mathcal{M}, s \models A$ if $\forall w \in l(s)$, then $\mathcal{M}, w \models A$, where $\mathcal{M}, w \models A$ is defined as for propositional logic. Let $Min_{<}(A)$ be the set of minimal states s such that $\mathcal{M}, s \models A$. We define $\mathcal{M}, s \models A \sim B$ if $\forall s' \in Min_{<}(A)$, $\mathcal{M}, s' \models B$. The relation \models can be extended to boolean combinations of conditionals in the standard way. We assume that $<$ satisfies the smoothness condition.

The language of the tableau calculi that we will introduce in the next section extends \mathcal{L} by formulas of the form $\Box A$, where A is propositional, whose intuitive meaning is that $\Box A$ holds in a world/state w if A holds in all the worlds/states preferred to w (i.e. in all w' such that $w' < w$). We extend the notion of KLM model to provide an evaluation of boxed formulas as follows:

Definition 3 (Truth condition of modality \Box). We define the truth condition of a boxed formula as follows:

- (**R** and **P**): $\mathcal{M}, w \models \Box A$ if, for every $w' \in \mathcal{W}$, if $w' < w$ then $\mathcal{M}, w' \models A$
- (**CL** and **C**): $\mathcal{M}, s \models \Box A$ if, for every $s' \in S$, if $s' < s$ then $\mathcal{M}, s' \models A$

From definition of $Min_{<}(A)$ in Definitions 1 and 2 above, and Definition 3, it follows that for any formula A , $w \in Min_{<}(A)$ iff $\mathcal{M}, w \models A \wedge \Box \neg A$ (resp. $s \in Min_{<}(A)$ iff $\mathcal{M}, s \models A \wedge \Box \neg A$).

3 Tableaux Calculi \mathcal{TS}^T

In this section we present analytic tableaux calculi \mathcal{TS}^T for KLM logics, where **S** stands for $\{\mathbf{R}, \mathbf{P}, \mathbf{CL}, \mathbf{C}\}$. The basic idea is simply to interpret the preference relation as an accessibility relation. The calculi for **R** and **P** implement a sort of run-time translation into (extensions of) Gödel-Löb modal logic of provability G. This is motivated by the fact that we assume the smoothness condition, which ensures that minimal A -worlds exist whenever there are A -worlds, by preventing infinitely descending chains of worlds. This condition therefore corresponds to the finite-chain condition on the accessibility relation (as in modal logic G). This approach is extended to the cases of **CL** and **C** by using a second modality L which takes care of states. For a broader discussion on those calculi, see [9, 15, 10].

The rules of the calculi manipulate sets of formulas Γ . We write Γ, F as a shorthand for $\Gamma \cup \{F\}$. Moreover, given Γ we define the following sets:

- $\Gamma^\Box = \{\Box \neg A \mid \Box \neg A \in \Gamma\}$
- $\Gamma^{\Box^\perp} = \{\neg A \mid \Box \neg A \in \Gamma\}$
- $\Gamma^{\sim^\pm} = \{A \sim B \mid A \sim B \in \Gamma\} \cup \{\neg(A \sim B) \mid \neg(A \sim B) \in \Gamma\}$
- $\Gamma^{L^\perp} = \{A \mid LA \in \Gamma\}$.

As mentioned, the calculus for rational logic **R** makes use of *labelled* formulas (see Figure 1), where the labels are drawn from a denumerable set \mathcal{A} ; there are two kinds of formulas: 1. *world formulas*, denoted by $x : F$, where $x \in \mathcal{A}$ and $F \in \mathcal{L}$; 2. *relation formulas*, denoted by $x < y$, where $x, y \in \mathcal{A}$, representing the preference relation. Rules of the calculus \mathcal{TR}^T also manipulate the following set of formulas: $\Gamma_{x \rightarrow y}^M = \{y : \neg A, y : \Box \neg A \mid x : \Box \neg A \in \Gamma\}$. The following definition states the truth conditions for labelled formulas:

Definition 4 (Truth conditions of formulas of \mathcal{TR}). Given a model $\mathcal{M} = \langle \mathcal{W}, <, V \rangle$ and a label alphabet \mathcal{A} , we consider a mapping $I : \mathcal{A} \mapsto \mathcal{W}$. Given a formula α of the calculus \mathcal{TR} , we define $\mathcal{M} \models_I \alpha$ as follows: $\mathcal{M} \models_I x : F$ iff $\mathcal{M}, I(x) \models F$; $\mathcal{M} \models_I x < y$ iff $I(x) < I(y)$.

We say that a set Γ of formulas of \mathcal{TR} is satisfiable if, for all formulas $\alpha \in \Gamma$, we have that $\mathcal{M} \models_I \alpha$, for some model \mathcal{M} and some mapping I .

TR^T	$\text{(AX)} \Gamma, x : P, x : \neg P \text{ with } P \in \text{ATM}$ $\text{(AX)} \Gamma, x < y, y < x$ $(\sim^+) \frac{\Gamma, u : A \vdash B, x : \neg A}{\Gamma, u : A \vdash B, x : \neg \Box \neg A} \quad \Gamma, u : A \vdash B$ $(\sim^-) \frac{\Gamma, u : \neg(A \vdash B)}{\Gamma, x : A, x : \Box \neg A, x : \neg B} \quad x \text{ new label}$ $(\Box^-) \frac{\Gamma, x : \neg \Box \neg A}{\Gamma, y < x, y : A, y : \Box \neg A, \Gamma_{x \rightarrow y}^M} \quad y \text{ new label}$ $(<) \frac{\Gamma, x < y}{\Gamma, x < y, x < z, \Gamma_{z \rightarrow x}^M} \quad \Gamma, x < y, z < y, \Gamma_{y \rightarrow z}^M \quad \begin{array}{l} z \text{ occurs in } \Gamma \text{ and} \\ \{x < z, z < y\} \cap \Gamma = \emptyset \end{array}$
TP^T	$\text{(AX)} \Gamma, P, \neg P \text{ with } P \in \text{ATM}$ $(\sim^+) \frac{\Gamma, A \vdash B}{\Gamma, A \vdash B, \neg A} \quad \Gamma, A \vdash B, \neg \Box \neg A \quad \Gamma, A \vdash B, B$ $(\sim^-) \frac{\Gamma, \neg(A \vdash B)}{\Gamma^{\sim^\pm}, A, \Box \neg A, \neg B}$ $(\Box^-) \frac{\Gamma, \neg \Box \neg A}{\Gamma^\Box, \Gamma^{\sim^\pm}, \Gamma^{\Box^\pm}, A, \Box \neg A}$
TCL^T	$(\sim^+) \frac{\Gamma, A \vdash B}{\Gamma, A \vdash B, \neg LA} \quad \Gamma, A \vdash B, \neg \Box \neg LA \quad \Gamma, A \vdash B, LB$ $(\sim^-) \frac{\Gamma, \neg(A \vdash B)}{\Gamma^{\sim^\pm}, LA, \Box \neg LA, \neg LB}$ $(\Box^-) \frac{\Gamma, \neg \Box \neg LA}{\Gamma^\Box, \Gamma^{\sim^\pm}, \Gamma^{\Box^\pm}, LA, \Box \neg LA}$ $(L^-) \frac{\Gamma, \neg LA}{\Gamma^{L^\pm}, \neg LA}$ $(L^-) \frac{\Gamma}{\Gamma^{L^\pm}} \text{ if } \Gamma \text{ does not contain negated } L \text{ - formulas}$
TC^T	$(\sim^+) \frac{\Gamma, A \vdash B}{\Gamma, A \vdash B, \neg LA} \quad \Gamma^{\sim^\pm}, \Gamma^{\Box^\pm}, A \vdash B, LA, \Box \neg LA \quad \Gamma, A \vdash B, LA, \Box \neg LA, LB$ $(\sim^-) \frac{\Gamma, \neg(A \vdash B)}{\Gamma^{\sim^\pm}, LA, \Box \neg LA, \neg LB}$ $(L^-) \frac{\Gamma, \neg LA}{\Gamma^{L^\pm}, \neg LA}$ $(L^-) \frac{\Gamma}{\Gamma^{L^\pm}} \text{ if } \Gamma \text{ does not contain negated } L \text{ - formulas}$

Fig. 1. Tableau systems \mathcal{TS}^T . To save space, we omit the standard rules for boolean connectives. For \mathcal{TCL}^T and \mathcal{TC}^T the axiom (AX) is as in \mathcal{TP}^T .

While the calculus for **R** makes use of labelled formulas, the calculi \mathcal{TS}^T for all other KLM logics, presented in Figure 1, do not make use of labels. A tableau is a tree whose nodes are sets of formulas Γ . A branch is a sequence of sets of formulas $\Gamma_1, \Gamma_2, \dots, \Gamma_n, \dots$, where each node Γ_i is obtained by its immediate predecessor Γ_{i-1} by applying a rule of \mathcal{TS}^T , having Γ_{i-1} as the premise and Γ_i as one of its conclusions. A branch is closed if one of its nodes is an instance of (AX), otherwise it is open. We say that a tableau is closed if all its branches are closed.

The rules (\sim^-) and (\Box^-) are called *dynamic*, since they introduce a new world in their conclusions. The other rules are called *static*. In order to prove that $\text{football_lover} \sim \neg \text{football_player}$ can be derived in preferential logic **P** from the knowledge base $\{\text{football_lover} \vdash \text{bet}, \text{football_player} \sim \text{football_lover}, \text{football_player}$

$\frac{(\sim^-) \frac{L \vdash B, P \vdash L, P \vdash \neg B, \neg(L \vdash \neg P)}{L \vdash B, P \vdash L, P \vdash \neg B, L, \Box \neg L, \neg \neg P}}{L \vdash B, P \vdash L, P \vdash \neg B, L, \Box \neg L, P}$		
$\frac{(\sim^+) \dots, L, \neg L}{\times}$	$\frac{(\Box^-) \dots, \neg \Box \neg L, \Box \neg L}{\dots, L, \Box \neg L, \neg L}$	$\frac{(\sim^+) \frac{L \vdash B, P \vdash L, P \vdash \neg B, L, \Box \neg L, P, B}{L \vdash B, P \vdash L, P \vdash \neg B, L, \Box \neg L, P, B, \neg \Box \neg P} \dots, \neg B, B}{L \vdash B, P \vdash L, P \vdash \neg B, \neg L, P, \Box \neg P}$
$\frac{(\sim^+) \dots, \neg P, P}{\times}$	$\frac{(\Box^-) \dots, \Box \neg P, \neg \Box \neg P}{\dots, \neg P, P, \Box \neg P}$	$\frac{(\sim^+) \dots, \neg L, L}{\times}$

Fig. 2. A derivation in \mathcal{TP}^T for $\{\text{football_lover} \sim \text{bet}, \text{football_player} \sim \text{football_lover}, \text{football_player} \sim \neg \text{bet}, \neg(\text{football_lover} \sim \neg \text{football_player})\}$. To increase readability, we use P to denote football_player , L to denote football_lover and B for bet .

$\sim \neg \text{bet}\}$, one needs to search a closed tableau in \mathcal{TP}^T for the set of formulas $\{\text{football_lover} \sim \text{bet}, \text{football_player} \sim \text{football_lover}, \text{football_player} \sim \neg \text{bet}, \neg(\text{football_lover} \sim \neg \text{football_player})\}$. In Figure 2 a closed tableau in \mathcal{TP}^T for the above unsatisfiable set of formulas is presented.

The calculi \mathcal{TS}^T are sound and complete wrt the semantics:

Theorem 1 (Soundness and completeness of \mathcal{TS}^T , [9, 15, 10]). *Given a set of formulas Γ of \mathcal{L} , it is unsatisfiable if and only if there is a closed tableau in \mathcal{TS}^T having Γ as a root.*

3.1 Termination and Complexity Results

The calculi \mathcal{TS}^T do not ensure termination. However, we are able to turn them into terminating calculi. In general, non-termination in tableau calculi can be caused by two different reasons: 1. dynamic rules can generate infinitely-many worlds, creating infinite branches; 2. some rules copy their principal formula in the conclusion, thus can be reapplied over the same formula without any control.

Concerning point 1, we are able to show that the generation of infinite branches due to the interplay between rules (\sim^+) and (\Box^-) cannot occur. Intuitively, the application of (\Box^-) to a formula $\neg \Box \neg A$ (introduced by (\sim^+)) adds the formula $\Box \neg A$ to the conclusion, so that (\sim^+) can no longer consistently introduce $\neg \Box \neg A$. This is due to the properties of \Box in G. Furthermore, the (\sim^-) rule can be applied only once to a given negated conditional on a branch, thus infinitely-many worlds cannot be generated on a branch.

Concerning point 2, we have to control the application of the (\sim^+) rule, which can otherwise be applied without any control since it copies its principal formula $A \sim B$ in all its conclusions, then the conclusions have a higher complexity than the premise. We can show that it is useless to apply (\sim^+) *more than once in the same world/state*, therefore the calculi \mathcal{TS}^T are modified to keep track of positive conditionals already considered in a world/state by moving them in an additional set Σ in the conclusions of (\sim^+) , and restrict the application of this rule to unused conditionals only. The dynamic rules (\sim^-) and (\Box^-) , whose

conclusions represent a *different* world/state wrt the corresponding premise, reintroduce formulas from Σ in order to allow further applications of (\sim^+) in the other worlds/states. This machinery is standard. Concerning the labelled calculus \mathcal{TR}^T , the same mechanism is applied by equipping each positive conditional with the list L of worlds-labels in which (\sim^+) has already been applied, and restricting its application by using worlds not belonging to L . In [15, 9, 10] it is shown that no other machinery is needed to ensure termination, except for \mathcal{TC}^T , which needs a further standard loop-checking machinery.

Theorem 2 (Termination of \mathcal{TS}^T , [9, 15, 10]). *For all KLM logic S , \mathcal{TS}^T ensures a terminating proof search.*

We conclude this section by summarizing complexity results for \mathcal{TS}^T . In [9, 15, 10], we have shown that for the logics \mathbf{R} , \mathbf{P} , and \mathbf{CL} , the tableaux calculi above can be used to define a **coNP** decision procedure to check the validity of a set of formulas of \mathcal{L} . The same cannot be done for the weakest logic \mathbf{C} .

We can prove the following Theorem:

Theorem 3 (Complexity of \mathbf{R} , \mathbf{P} , and \mathbf{CL}). *The problem of deciding validity for KLM logics \mathbf{R} , \mathbf{P} , and \mathbf{CL} is **coNP**-complete.*

Concerning \mathbf{R} and \mathbf{P} , this result matches the known complexity results for these logics. Concerning the logic \mathbf{CL} , this complexity bound is new, to the best of our knowledge.

4 KLMLean 2.0: a Theorem Prover for KLM Logics

In this section we describe an implementation of \mathcal{TS}^T calculi in SICStus Prolog. The program, called KLMLean 2.0, is inspired by the “lean” methodology [3] (even if it does not fit its style in a rigorous manner): the Prolog program consists in a set of clauses, each one representing a tableau rule or axiom; the proof search is provided for free by the mere depth-first search mechanism of Prolog, without any additional ad hoc mechanism. KLMLean 2.0 can be downloaded at [http://www.di.unito.it/~pozzato/klmlean 2.0](http://www.di.unito.it/~pozzato/klmlean%202.0).

Let us first describe the implementation of non-labelled calculi for \mathbf{P} , \mathbf{CL} , and \mathbf{C} . We represent each node of a proof tree (i.e. set of formulas) by a Prolog list. The tableaux calculi are implemented by the predicate

prove(Gamma,Sigma,Tree).

which succeeds if and only if the set of formulas Γ , represented by the list **Gamma**, is unsatisfiable. **Sigma** is the list representing the set Σ of *used conditionals*, and it is used in order to control the application of the (\sim^+) rule, as described in the previous section. When **prove** succeeds, **Tree** contains a representation of a

closed tableau³. For instance, to prove that $A \sim B \wedge C, \neg(A \sim C)$ is unsatisfiable in **P**, one queries KLMLean 2.0 with the following goal: `prove([a => (b and c), neg (a => c)], [], Tree)`. The string “=>” is used to represent the conditional operator \sim , “and” is used to denote \wedge , and so on. Each clause of `prove` implements one axiom or rule of the tableaux calculi; for example, the clauses implementing (**AX**) and (\sim^-) are as follows:

```
prove(Gamma,_,tree(...)):-member(F,Gamma),member(neg F,Gamma),!.
prove(Gamma,Sigma,tree(...)):-select(neg (A => B),Gamma,NewGamma),
  conditionals(NewGamma,Cond),append(Cond,Sigma,DefGamma),
  prove([neg B|[box neg A|[A|DefGamma]]],[],...).
```

The clause for (**AX**) is applied when a formula F and its negation $\neg F$ belong to Γ . Notice that F is a formula of the language \mathcal{L} , even complex; KLMLean 2.0 extends (**AX**) to a generic formula F in order to increase its performances, without losing the soundness of the calculi. The clause for (\sim^-) is applied when a formula $\neg(A \sim B)$ belongs to Γ . The predicate `select` removes $\neg(A \sim B)$ from `Gamma`, then the auxiliary predicate `conditionals` is invoked to compute the set $\Gamma \sim^\pm$ on the resulting list `NewGamma`; finally, the predicate `prove` is recursively invoked on the only conclusion of the rule. Notice that, since (\sim^-) is a dynamic rule, the conditionals belonging to Σ move to Γ in the conclusion (execution of `append`), in order to allow further applications of (\sim^+) . To search for a derivation of a set of formulas Γ , KLMLean 2.0 proceeds as follows: first of all, if Γ is an instance of (**AX**), the goal will succeed immediately by using the clauses for the axioms. If it is not, then the first applicable rule will be chosen, e.g. if `Gamma` contains a formula `neg(neg F)`, then the clause for (\neg) rule will be used, invoking `prove` on its unique conclusion. KLMLean 2.0 proceeds in a similar way for the other rules. The ordering of the clauses is such that the boolean rules are applied before the other ones. In the case of cumulative logic **C**, KLMLean 2.0 implements a loop-checking machinery by equipping the `prove` predicate with an additional argument, called `Analyzed`, representing the list of sets of formulas already considered in the current branch. Clauses implementing \mathcal{TC}^T are invoked only if the current set of formulas has not yet been considered, i.e. if it does not belong to `Analyzed`.

The theorem prover for rational logic **R** implements *labelled* tableau calculi \mathcal{TR}^T . It makes use of Prolog constants to represent labels: world formulas $x : A$ are represented by a Prolog list `[x,a]`, and relation formulas $x < y$ are represented by a list `[x,<,y]`. As for the other systems, each clause of the predicate `prove` implements a tableau rule or axiom. As an example, here is the clause implementing the rule $(<)$, capturing the modularity of the preference relation:

³ More in detail, the argument `Tree` is instantiated by a functor `tree(Rule,F,G,SubTree1,[SubTree2])`, tracing that the rule `Rule` has been applied to the principal formula `F` `Rule` `G` in the current set of formulas. `SubTree1` and `SubTree2` are functors `tree` representing the closed tableau for each conclusion of the rule.

```

prove(Gamma,Labels,Cond,tree(...)):-
  member([X,<,Y],Gamma),member(Z,Labels),X\=Z, Y\=Z,
  \+member([X,<,Z],Gamma),\+member([Z,<,Y],Gamma),!,
  gammaM(Gamma,Y,Z,ResLeft),gammaM(Gamma,Z,X,ResRight),
  append(ResLeft,Gamma,LeftConcl),append(ResRight,Gamma,RightConcl),
  prove([[Z,<,Y]|LeftConcl],Labels,Cond,...),!,
  prove([[X,<,Z]|RightConcl],Labels,Cond,...).

```

The predicate `gammaM(Gamma,X,Y,...)` computes the set $\Gamma_{x \rightarrow y}^M$ defined in the previous section. In this system, `Cond` is used in order to control the application of the (\vdash^+) rule: it is a list whose elements have the form `[x,a => b]`, representing that (\vdash^+) has been applied to $A \vdash B$ in the current branch by using the label x . In order to increase its performances, KLMLean for **R** adopts a heuristic approach (not very “lean”) to implement the crucial (\vdash^+) rule: the predicate `prove` chooses the “best” positive conditional to which apply the rule, and the “best” label to use. Roughly speaking, an application of (\vdash^+) is considered to be better than another one if it leads to an immediate closure of more conclusions. Even if (\vdash^+) is invertible, choosing the right label in the application of (\vdash^+) is highly critical for the performances of the theorem prover. To postpone this choice as much as possible, for the logic **R** we have defined a more efficient version of the prover, inspired by the free-variable tableaux introduced in [2]. It makes use of *Prolog variables* to represent all the labels that can be used in a single application of the (\vdash^+) rule. This version represents labels by integers starting from 1; by using integers we can easily express constraints on the range of the variable-labels. To this regard, the library `clpfd` is used to manage free-variables domains. In order to prove $\Gamma, u : A \vdash B$, KLMLean 2.0 will call `prove` on the following conclusions: $\Gamma, u : A \vdash B, Y : \neg A$; $\Gamma, u : A \vdash B, Y : \neg \Box \neg A$; $\Gamma, u : A \vdash B, Y : B$, where Y is a Prolog variable. Y will then be instantiated by Prolog’s pattern matching to close a branch with an axiom. Predicate `Y in 1..Max` is used to define the domain of Y , where `Max` is the maximal integer occurring in the branch (i.e. the last label introduced). The list `Cond` here contains elements of the form `[a => b,Used]`, where `Used` is the list of free variables already introduced to apply (\vdash^+) in the current branch. In order to ensure termination, the clause implementing (\vdash^+) is applied *only if* `|Used| < Max`; the predicate `all_different([Y|Used])` is then invoked to ensure that all variables used to apply (\vdash^+) on the same conditional will assume different values. On the unsatisfiable set $\neg(A \vee D \vdash F \vee \neg C \vee \neg B \vee A), (A \wedge B) \vee (C \wedge D) \vdash E \wedge F, \neg(P \vdash E \wedge F), \neg((A \wedge B) \vee (C \wedge D) \vdash G)$, the free-variables version succeeds in less than 2 ms, whereas the “standard” version requires 1.9 s.

The performances of KLMLean 2.0 are promising. We have tested the implementation for **R** over 300 sets of formulas: it terminates its computation in 236 cases in less than 2.5 s (204 in less than 100 ms). The results for **R** and for the other KLM logics are reported in Table 1:

KLMLean 2.0 has also a graphical user interface (GUI) implemented in Java. The GUI interacts with the SICStus Prolog implementation by means of the package `se.sics.jasper`. Thanks to the GUI, one does not need to know how to call

KLM logic	1 ms	10 ms	100 ms	1 s	2.5 s
R (with free-variables)	176	178	204	223	236
P	166	164	185	206	211
CL	119	118	136	150	159
C	76	77	92	110	123

Table 1. Some statistics for KLMLean 2.0.

the predicate `prove`, or if the program implements a labelled or an unlabelled deductive system; one just introduces the formulas of a knowledge base K (or the set of formulas to prove to be unsatisfiable) and a formula F , in order to prove if one can infer F from K , in a text box and searches a derivation by clicking a button. Moreover, one can choose the intended system of KLM logic, namely **R**, **P**, **CL** or **C**. When the analyzed set $K \cup \{\neg F\}$ of formulas is unsatisfiable, KLMLean offers these options:

- display a proof tree of the set in a special window;
- build a \LaTeX file containing the same proof tree: compiling this file with \LaTeX , one can obtain the closed tree in a pdf file, or ps, or dvi, and then can print it.

The theorem prover for rational logic **R** offers another functionality. If the initial set of formulas $K \cup \{\neg F\}$ is satisfiable, i.e. the predicate `prove` answers `no`, then KLMLean 2.0 also displays a model satisfying $K \cup \{\neg F\}$.

Some pictures of KLMLean 2.0 are presented in Figure 3.

5 Conclusions and Future Work

In this paper, we have presented tableau calculi \mathcal{TS}^T for all the logics of the KLM framework for nonmonotonic reasoning. We have shown a tableau calculus for rational logic **R**, preferential logic **P**, loop-cumulative logic **CL**, and cumulative logic **C**. The calculi presented give a decision procedure for the respective logics. Moreover, for **R**, **P** and **CL** we have shown that we can obtain **coNP** decision procedures by refining the rules of the respective calculi. In case of **C**, we obtain a decision procedure by adding a suitable loop-checking mechanism. In this case, our procedure gives an hyper exponential upper bound. Further investigation is needed to get a more efficient procedure. On the other hand, we are not aware of any tighter complexity bound for this logic.

All the calculi presented in this paper have been implemented by a theorem prover called KLMLean 2.0, which is a SICStus Prolog implementation of \mathcal{TS}^T inspired to the “lean” methodology, whose basic idea is to write short programs and exploit the power of Prolog’s engine as much as possible. To the best of our knowledge, KLMLean 2.0 is the first theorem prover for KLM logics.

We plan to extend our calculi to first order case. The starting point will be the analysis of first order preferential and rational logics by Friedman, Halpern

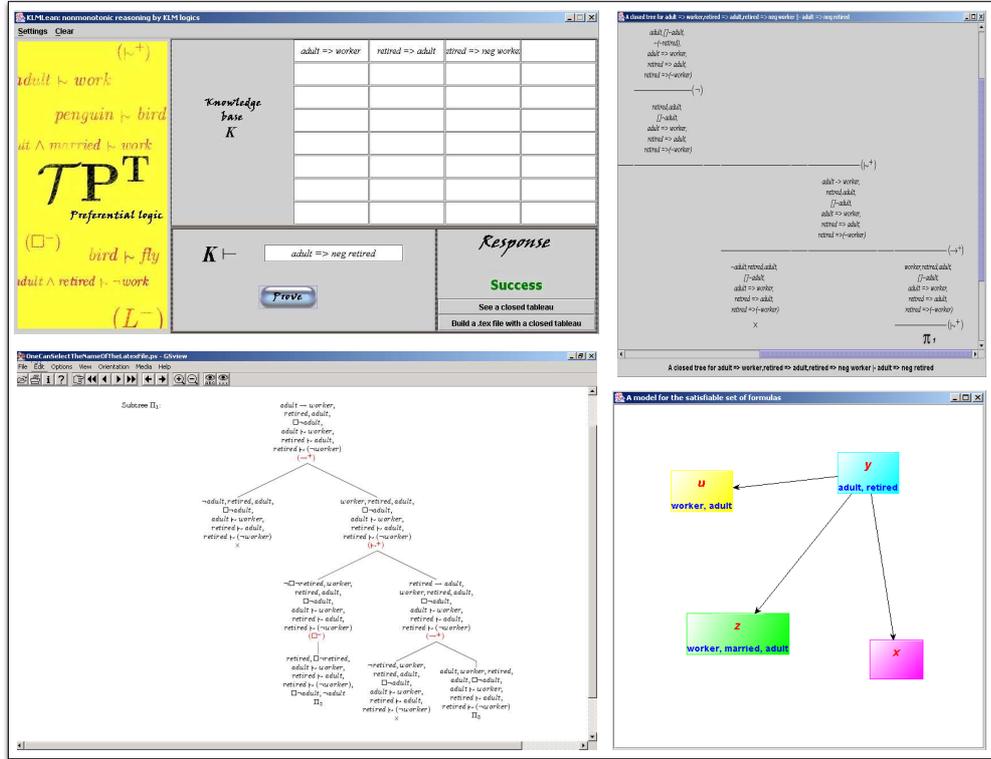


Fig. 3. Some pictures of KLMLean 2.0

and Koller in [7]. Moreover, we intend to increase the performances of KLMLean 2.0 by experimenting standard refinements and heuristics.

References

1. A. Artosi, G. Governatori, and A. Rotolo. Labelled tableaux for non-monotonic reasoning: Cumulative consequence relations. *J. of Logic and Computation*, 12(6):1027–1060, 2002.
2. B. Beckert and R. Goré. Free variable tableaux for propositional modal logics. In *Proc. of TABLEUX 1997 (Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 1227 of *LNAI*, Springer-Verlag, pages 91–106, 1997.
3. B. Beckert and J. Posegga. leantap: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
4. C. Boutilier. Conditional logics of normality: a modal approach. *Art. Int.*, 68(1):87–154, 1994.
5. G. Crocco and P. Lamarre. On the connection between non-monotonic inference systems and conditional logics. In *Proc. of KR 92*, pages 565–571, 1992.
6. N. Friedman and J. Y. Halpern. Plausibility measures and default reasoning. *Journal of the ACM*, 48(4):648–685, 2001.

7. N. Friedman, J. Y. Halpern, and D. Koller. First-order conditional logic for default reasoning revisited. *ACM TOCL*, 1(2):175–207, 2000.
8. D. Gabbay. Theoretical foundations for non-monotonic reasoning in expert systems. *Logics and models of concurrent systems*, Springer, pages 439–457, 1985.
9. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Analytic Tableaux for KLM Preferential and Cumulative Logics. In *Proc. of LPAR 2005, LNAI 3835*, pages 666–681. Springer, 2005.
10. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Analytic Tableaux Calculi for KLM Rational Logic R. In *Proc. of JELIA 2006*, volume 4160 of *LNAI*, pages 190–202. Springer, 2006.
11. L. Giordano, V. Gliozzi, N. Olivetti, and C. Schwind. Tableau calculi for preference-based conditional logics. In *Proc. of TABLEAUX 2003, LNAI 2796*, pages 81–101. Springer, 2003.
12. G.E. Hughes and M.J. Cresswell. *A Companion to Modal Logic*. Methuen, 1984.
13. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
14. D. Lehmann and M. Magidor. What does a conditional knowledge base entail? *Artificial Intelligence*, 55(1):1–60, 1992.
15. G. L. Pozzato. *Proof Methods for Conditional and Preferential Logics*. PhD thesis, 2007.
16. Y. Shoham. A semantical approach to nonmonotonic logics. In *Proc. of Logics in Computer Science*, pages 275–279, 1987.

Answer Set Programming with Resources^{*}

Stefania Costantini¹ and Andrea Formisano²

¹ Università di L'Aquila, Dipartimento di Informatica. stefcost@di.univaq.it

² Università di Perugia, Dip. di Matematica e Informatica. formis@dipmat.unipg.it

Abstract. In this paper we propose an extension of Answer Set Programming (ASP) to support declarative reasoning on consumption and production of resources. We call the proposed extension RASP, standing for Resourced ASP. Resources are modeled by specific atoms, to which we associate *quantities* that represent the available amount. The firing of a RASP-rule can both consume and produce resources. We define the semantics for RASP programs by extending the usual answer set semantics and we propose an implementation based on standard ASP-solvers.

Key words: ASP, quantitative reasoning, language extensions.

Introduction

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 [10, 11]. 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 [3, 17, 1] among many). The ASP formulations in these and other fields might take profit from the possibility of performing (at least to some extent) forms of *quantitative* reasoning like those that are possible in Linear Logics [8] and Description Logics [2]. In this direction, in this paper we propose an extension of ASP to support declarative reasoning on consumption and production of resources. We call the proposed extension RASP, standing for Resourced ASP. Resources are modeled by specific atoms to which we associate *quantities* that represent the available amount. Thus, an atom of the form $q:n$ indicates a quantity n of resource q . The firing of a RASP-rule can both consume and produce resources. If several tasks require the same resource, different allocation choices are possible.

We provide syntax of RASP programs (Sect. 1) by allowing amount atoms to occur as positive literals in both the head and the body of rules. Amount atoms in the body of rules indicate which resources are *consumed*

^{*} This research is partially supported by GNCS.

by that rule, and in which quantity. Correspondingly, amount atoms in the head (as we will see, in the general case we will admit several amount atoms in the head of a rule) indicate which resources are *produced* by that rule, and in which quantity.

Let us focus on a simple example that should clarify the motivations of the approach. Consider a situation where, if one has three eggs, three hundred grams of flour and three hundred grams of sugar, then s(he) can cook a cake. Similarly, if s(he) has two eggs, three hundred grams of sugar and two hundred milliliters of milk, then s(he) can prepare an ice-cream. Depending on the number n of available eggs, it might be the case that the ingredients are sufficient to prepare both, one, or none of the desserts. This could be a RASP program encoding such a situation:

γ_1 :	cake :- egg:3, flour:3, sugar:3.	γ_3 :	egg: n .
γ_2 :	ice_cream :- egg:3, sugar:2, milk:2.	γ_4 :	flour:8.
		γ_5 :	sugar:6.

where the amounts occurring in the body of a rule correspond to resources which are consumed whenever the rule is used. Facts describe initially available resources. Clearly, one could be interested in finding all possible alternative allocations of resources, possibly subject to further specific constraints or consumption policies (cf., Sect. 4).

A semantics for RASP programs is provided in Sect. 2, by combining usual answer set semantics with an interpretation of resource amounts, where different allocation choices correspond to different answer sets. We discuss extensions to the basic paradigm (Sect. 3) where, in the context of the same semantics, one can define both constraints on quantities and negative quantities. Also, we allow the same rule to be (optionally) fired more than once, if the available quantities of consumed resources are sufficient for constructing more than one instance of the produced resources. In Sect. 4 we discuss how to introduce a filter on the answer sets of a RASP program, so as to specify different *policies* for production and consumption of resources. Finally, an implementation of RASP based on standard ASP-solvers is proposed in Sect. 5.

1 Syntax of RASP-programs

Let $\langle \Pi, \mathcal{C}, \mathcal{V} \rangle$ be a structure 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.¹

¹ Intuitively, we are partitioning the symbols of the underlying language in *Program* symbols and *Resource* symbols.

The elements of \mathcal{C}_R are said *amount-symbols*, while the elements of Π_R are said *resource-symbols*. A *program-term* is a variable or a constant symbol. An *amount-term* is either a variable or an amount-symbol.

Let $\mathcal{A}(X, Y)$ denote the collection of all atoms of the form $p(t_1, \dots, t_n)$, with $p \in X$ and $\{t_1, \dots, t_n\} \subseteq Y$. Then, a *program-atom* is an element of $\mathcal{A}(\Pi_P, \mathcal{C} \cup \mathcal{V})$. An *amount-atom* is a writing of the form $q:a$ where $q \in \mathcal{A}(\Pi_R, \mathcal{C} \cup \mathcal{V})$ and a is an amount-term. As usual, a *program-literal* L is A or $\text{not } A$ where A is a program-atom. If $L = A$ (resp., $L = \text{not } A$) then \bar{L} denotes $\text{not } A$ (resp., A). A *resource-literal* (r-literal) is either a program-literal or an amount-atom.

A *RASP-rule* (rule, for short) γ is a writing of the form $H \leftarrow B_1, \dots, B_k$ where B_1, \dots, B_k are r-literals and either H is a program-atom or a (non-empty) list of amount-atoms. If no amount-atom occurs in γ , then γ is said to be a *program-rule*, otherwise it is a *resource-rule* (r-rule, for short). The *head* and the *body* of a rule are defined as usual. A rule with empty body (i.e. $k = 0$) is a *fact*. The case in which a rule γ has an empty head is admitted only if γ is a program-rule (i.e. γ is an ASP *constraint*). An *r-program* is a finite set of RASP-rules.

As customary, a term (atom, literal, rule, ...) is ground if no variable occurs in it. The grounding of an r-program P is the set of all ground instances of rules of P , obtained through ground substitutions over the Herbrand universe of P . In order to ensure finiteness of the grounding process we assume that each variable occurring as amount-term in a rule, also occurs in a program-atom of the body. In any r-program only a finite number of resource-symbols of \mathcal{C}_R occurs. Hence, as far as amount-atoms are concerned, a finite number of ground instances can be generated by the grounding process. Let $\mathcal{B}_H(X, Y)$ denote the collection of all ground atoms built from predicate symbols in X and terms in Y .

2 Semantics of RASP-programs

In order to define semantics of r-programs, we have to fix an interpretation for amount-symbols. This is done by choosing a collection Q of *quantities*, a natural choice being $Q = \mathbb{Z}$, and the operations to combine and compare quantities. To this aim, we consider given a mapping $\kappa : \mathcal{C}_R \rightarrow \mathbb{Z}$ that associate integers to amount-symbols. (Alternative options for Q are possible. For instance, one could choose Q to be the set \mathbb{Q} of rationals.) Positive (resp. negative) integers will be used to model produced (resp. consumed) amounts of resources.

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. Given a collection Z of integers, $\Sigma(Z)$ denotes their sum.

We introduce now the notion of r-interpretation for r-programs, by considering the ground case in the first place. As we will see, for any fixed resource symbol, an interpretation of a (ground) r-program P must determine an allocation of amounts for all occurrences of such symbol in each rule of P . We restrain to those allocations having a non-negative global balance. The collection \mathbb{S}_P of all potential allocations is the following set of mappings:

$$\mathbb{S}_P = \left\{ F \in (\mathcal{FM}(Q))^P \mid 0 \leq \Sigma\left(\bigcup_{\gamma \in P} F(\gamma)\right) \right\} \quad (1)$$

Plainly, to interpret a r-program, further restrictions will be imposed to identify suitable elements of \mathbb{S}_P . Hence, an r-interpretation of a ground r-program P is defined by providing a mapping $\mu : \Pi_R \rightarrow \mathbb{S}_P$. The function μ determines, for each resource-symbol $q \in \Pi_R$, a mapping $\mu(q) \in \mathbb{S}_P$. In turns, such a mapping assigns to each rule $\gamma \in P$ a (possibly empty) multi-set $\mu(q)(\gamma)$ of quantities for each amount-symbols a such that $q:a$ occurs in γ . We have the following definition.

Definition 1. *An r-interpretation for a (ground) r-program P is a pair $\mathcal{I} = \langle I, \mu \rangle$, with $I \subseteq \mathcal{BH}(\Pi_P, \mathcal{C})$ and $\mu : \Pi_R \rightarrow \mathbb{S}_P$.*

Notice that an r-interpretation (and hence a model) for an r-program can be seen after two different though related points of view. In fact, two aspect have to be taken into account: the truth of program literals and the allocation of resources. Hence, the firing of a r-rule, involving consumption/production of resources. Then, the rule can be fired if and only if the truth values of the program-literals satisfy the rule. We reflect the fact that the satisfaction of a r-rule γ depends on the truth of its program-literals by introducing a fragment of ASP program $\hat{\gamma}$. Let the r-rule γ , having L_1, \dots, L_k as program-literals and R_1, \dots, R_h as amount-atoms. The ASP-program $\hat{\gamma}$ is so defined:

$$\hat{\gamma} = \begin{cases} \{ \leftarrow \overline{L_1}, \dots, \leftarrow \overline{L_k} \} & \text{if the head of } \gamma \text{ consists of amount-atoms} \\ \{ \leftarrow \overline{L_1}, \dots, \leftarrow \overline{L_k}, \\ \quad H \leftarrow L_1, \dots, L_k \} & \text{if the head } H \text{ of } \gamma \text{ is a program-atom} \\ & \text{and } h > 0 \\ \{ \gamma \} & \text{otherwise.} \end{cases}$$

Notice that if γ is a program-rule then $\hat{\gamma} = \{\gamma\}$. The following definition states that in order to be a model, an r-interpretation that allocates non-void amounts to the resource-symbols of γ (i.e., γ is *fired*, see below), has to model the ASP-rules in $\hat{\gamma}$.

Definition 2. Let $\mathcal{I} = \langle I, \mu \rangle$ 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 $\gamma \in P$

$$\left(\forall q \in \Pi_R (\mu(q)(\gamma) = \emptyset) \right) \vee \left(\forall q \in \Pi_R (\mu(q)(\gamma) = \mathcal{R}(q, \gamma)) \right) \quad (2)$$

where $\mathcal{R}(q, \gamma)$ is the multiset so defined:²

$$\begin{aligned} \mathcal{R}(q, \gamma) = \{ \{ v \in \mathbb{Z} \mid v = -\kappa(a) \text{ for } q:a \text{ in the body of } \gamma \\ \text{or } v = \kappa(a) \text{ for } q:a \text{ in the head of } \gamma \} \} \quad (3) \end{aligned}$$

- I is a stable model for the ASP-program \hat{P} , so defined

$$\hat{P} = \bigcup \left\{ \hat{\gamma} \mid \begin{array}{l} \gamma \text{ is a program-rule in } P, \text{ or} \\ \gamma \text{ is a resource-rule in } P \text{ and } \exists q \in \Pi_R (\mu(q)(\gamma) \neq \emptyset) \end{array} \right\}$$

An r-rule γ in P is *fired* if $\exists q \in \Pi_R (\mu(q)(\gamma) \neq \emptyset)$ (i.e., through its satisfaction, some resources are consumed and/or produced). Given an answer set $\langle I, \mu \rangle$ for an r-program P , the *resource balance* for P , w.r.t. $\langle I, \mu \rangle$, is the mapping $\varphi : \Pi_R \rightarrow \mathbb{Z}$ defined as: $\varphi(q) = \Sigma\{\Sigma(\mu(q)(\gamma)) \mid \gamma \in P\}$, which summarizes consumptions and productions of all resources.

An r-interpretation \mathcal{I} is a answer set of an r-program P if it is a answer set for the grounding of P .

Example 1. Let us consider the simple r-program P made of the rules $\gamma_1, \dots, \gamma_5$ mentioned in the Introduction, where we identify \mathcal{C}_R with \mathbb{Z} and we have $\Pi_R = \{\text{egg, flour, sugar, milk}\}$, $\Pi_P = \{\text{cake, ice_cream}\}$. Putting $n = 4$, an r-interpretation for P is $\langle I, \mu \rangle$ with $I = \{\text{cake}\}$ and μ such that

$$\begin{aligned} \mu(\text{egg})(\gamma_1) &= \{ \{ -3 \} \}, & \mu(\text{egg})(\gamma_3) &= \{ \{ 4 \} \}, & \mu(\text{egg})(\gamma_i) &= \emptyset \text{ otherwise;} \\ \mu(\text{flour})(\gamma_1) &= \{ \{ -4 \} \}, & \mu(\text{flour})(\gamma_4) &= \{ \{ 8 \} \}, & \mu(\text{flour})(\gamma_i) &= \emptyset \text{ otherwise;} \\ \mu(\text{sugar})(\gamma_1) &= \{ \{ -3 \} \}, & \mu(\text{sugar})(\gamma_5) &= \{ \{ 6 \} \}, & \mu(\text{sugar})(\gamma_i) &= \emptyset \text{ otherwise;} \\ \mu(\text{milk})(\gamma_i) &= \emptyset \text{ for all } i. \end{aligned}$$

The program \hat{P} is made of the fact `cake` (i.e., $\hat{\gamma}_1$). Hence, I is a stable model for \hat{P} . This r-interpretation fires the rule γ_1 . The resource balance φ is such that $\varphi(\text{egg}) = 1$, $\varphi(\text{flour}) = 4$, $\varphi(\text{sugar}) = 3$, and $\varphi(\text{milk}) = 0$.

² As it will be clear in the sequel, the use of multisets allows us to handle multiple copies of the same amount-atom. Each of these copies must be taken into account, since it corresponds to a different amount of resource.

The next example involves interactions between r-rules and ASP rules: through the use of variables, the values admitted for resource amounts can be controlled by means of program predicate; also, compound program atoms and terms can be used to define different ground instance of r-rules (cf., the first rule of the program below).

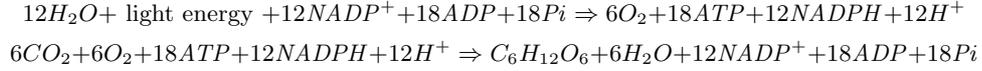
Example 2. A simple example of a “toy heating system”. It is a dual-fuel system and comprises two fuel sources: electricity as a primary system and alternate fuel such as gas or fuel oil as a secondary source. The control switches the electric heat off and the backup fuel on during peak load conditions, typically on the coldest days of the winter. We can model this situation by the following RASP program (assuming $a_1, a_2, a_3 \in \mathbb{Z}$):

```
warm :- fuel(Type):Q, load(Load), requiredFuel(Load,Type), needed(Type,Q).
:- not warm.                                1{temperature(5), temperature(20)}1.
money:a3.                                    is_winter.
load(low) :- temperature(Celsius), Celsius>10, is_winter.
load(high) :- temperature(Celsius), Celsius<=10, is_winter.
load(none) :- not is_winter.
requiredFuel(high,oil).      requiredFuel(low,electricity).
needed(oil, a2).             needed(electricity, a1).
fuel(oil):a1 :- money:a3.    fuel(electricity):a2 :- money:a3.
```

There are two expected solutions, discriminated through a cardinality constraint (in the second line, [13]), which states that the temperature can be either 5 or 20 degrees. In the former case we have a high peak load (`load(high)` is true) and oil is used. The other solution, when the temperature is 20 degrees (`load(low)` is true), involve using electricity.

As discussed in Sect. 5, a solver for RASP can be obtained by exploiting existing ASP-solvers through compilation into ASP. Once κ has been chosen, it can be encoded in r-programs through program-predicates and usual ASP-rules. Alternatively, one can directly employ numerical terms and rely on the built-in features of the specific ASP-solver.

Example 3. We take advantage from having (multi)sets of amount-atoms as heads of rules, to model a chemical reaction. We consider a simplified description of the photosynthesis in green plants. Photosynthesis uses the energy of light to convert carbon dioxide (CO_2) and water (H_2O) into glucose ($C_6H_{12}O_6$) and oxygen (O_2). The process can be roughly described as two interleaved phases, called respectively *light-dependent reaction* (which converts solar energy into chemical energy stored in specific molecules, *NADPH* and *ATP*) and *carbon fixation* (which captures carbon dioxide and makes the precursors of glucose). Simple general equations that schematize the two phases are (adapted from [12]):



The following rules encode such reactions:

```
oxygen:A, atp:C, nadph:B, proton:B :- water:B, adp:C, nadp:B, pi:C,
light, B=2*A, C=3*A, A=6*Moles
reagents1(Moles,A,B,C,D).

glucose:D, water:A, adp:C, nadp:B, pi:C :- nadph:B, proton:B, atp:C, oxygen:A,
carbDioxide:A, A=6*D, B=2*A, C=3*A,
reagents2(A,B,C,D).
```

where we generically used the two predicates `reagents1,2` (defined elsewhere in the program) to indicate that values for `A,B,C,D,Moles` can be obtained through other program-rules.

3 Extending the basic framework

Multiple firing of resource-rules. Let us enrich the RASP language by admitting r-rules γ of the form

$$[N_1, N_2]: H \leftarrow B_1, \dots, B_k \quad (4)$$

(where H and the B_i are as before). The intended meaning is that, whenever γ can be fired, this can be done any number n of times provided that $N_1 \leq n \leq N_2$ holds. Consequently, after grounding, N_1 and N_2 have to be instantiated to positive integers (namely, $N_1, N_2 \in \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$). To this purpose, both variables N_1 and N_2 have to occur in program atoms of the body. Notice that for $N_1 = N_2 = 1$ we obtain r-rules as introduced in Sect. 1. To deal with this extension it suffices to modify the definition of r-interpretation as follows: An r-interpretation for a program P is a triple $\mathcal{I} = \langle I, \mu, \xi \rangle$, where I and μ are as before, while ξ is a mapping $\xi: P \rightarrow \mathbb{N}^+$. In view of this notion of r-interpretation, the definition of answer set is obtained from Def. 2 by replacing condition (2) with:³

$$\forall q \in \Pi_R \mu(q)(\gamma) = \emptyset \quad \vee \quad \forall q \in \Pi_R \mu(q)(\gamma) = \xi(\gamma) \cdot \mathcal{R}(q, \gamma) \wedge N_1 \leq \xi(\gamma) \leq N_2 \quad .$$

Negative amount-atoms. A *negative* amount-atom is of the form $-q:a$, where $q:a$ is an amount-atom. The easiest manner of interpreting negative amount-atoms is through the inverse function on \mathbb{Z} . Formally, it suffices to modify the definition of $\mathcal{R}(q, \gamma)$ (cf., (3) in Def. 2) as follows:

$$\mathcal{R}(q, \gamma) = \{ \{ v \in \mathbb{Z} \mid v = -\kappa(a) \text{ for } q:a \text{ in the body or } -q:a \text{ in the head of } \gamma \\ \text{or } v = \kappa(a) \text{ for } q:a \text{ in the head or } -q:a \text{ in the body of } \gamma \} \}$$

³ Given a multiset S and $n \in \mathbb{N}$, we are denoting by $n \cdot S$ the multiset $\{ \{ n \cdot s \mid s \in S \} \}$.

This use of the inverse function corresponds to imposing a duality between consumption and production of resources. Hence, a negative amount-atom $-q:a$ in the body of a fired rule models the consumption of $-\kappa(a)$ units of resource q . This actually means that an amount $\kappa(a)$ of resource q is produced. A dual argument applies to negative amount-atoms appearing in the head of r-rules.

Constraints on global resource balances. In the framework introduced so far, one can easily express constraints on the consumption/production of amounts of resources. For instance, assume that an amount-atom $q:X$ (with $X \in \mathcal{V}$) occurs in a r-rule γ and that the firing of γ should be avoided for particular amounts of q . This can be ensured by using program-literals in the body of γ that inhibit the firing of γ whenever X assumes such values (as consequence of grounding). Constraints of this kind are, in a sense, *local* to the specific rule being considered.

It is easy to extend the basic framework to allow the assertion of constraints on the global resource balance. We describe now two simple forms of such *global* constraints. The (qualified) fact *leaveAtLeast* : $q:a$ where $q:a$ is a ground amount-atom, can be used to weed-out all models in which an amount of q smaller than $\kappa(a)$ is left unused.⁴ A similar constraint is imposed through the fact *leaveAtMost* : $q:a$. In this case, in each answer set no more than $\kappa(a)$ units of q can be left unused.

For a resource q , in order to fulfill the constraints *leaveAtLeast* : $q:a_q$ and *leaveAtMost* : $q:b_q$, we filter the potential r-interpretations for an r-program P by refining the definition of the mapping μ (cf., (1), page 4). In particular, we restrain its codomain \mathbb{S}_P as follows:

$$\mu : \Pi_R \rightarrow \{F \in (\mathcal{FM}(\mathbb{Z}))^P \mid \kappa(a_q) \leq \Sigma(\bigcup_{\gamma \in P} F(\gamma)) \leq \kappa(b_q)\}$$

Similarly, other constraint imposing different requirements on the global resource balance can be dealt with.

4 Budget Policies

As we have seen, a answer set for an r-program is constituted of a set of (true) program-atoms I and a mapping μ establishing the amounts for each consumption/production of resources. There might be the case that different answer sets agree on I but involve different mappings. I.e.,

⁴ Notice that, for a given resource symbol q , it makes little sense to impose more than one constraint of this kind. It suffices, in fact, to consider the most restrictive one.

fixed the set I , more that one mapping might satisfy the conditions expressed in Def. 2. In such a situation, it could be valuable to apply some criteria to filter the collection of all possible answer sets. In particular, different policies could be adopted in selecting a specific strategy for the consumption/production of resources (i.e., in firing r-rules). We identify three basic possibilities, among many (for the sake of simplicity, let us focus on grounded programs):

Thrifty. An r-rule γ is fired only if this is forced. (For instance, because the truth of a program-atom occurring as head of γ is required by effect of other parts of the program.)

Prodigal. Whenever an r-rule γ can be fired, it must be fired.

Optional. This is the most general policy and it admits the solutions obtained by the previous policies. In this case, different resource allocations can enable the firing of different rules, possibly in antithetic manners. There is no preference among the different mappings.

Example 4. Recall the example mentioned in the Introduction (r-rules $\gamma_1 \dots \gamma_5$), where we put $n = 7$ and add a further rule:

$$\gamma_6: \text{milk:3.}$$

Four potential solutions are possible: make none, one, or both the desserts. The Thrifty policy selects the answer set in which no dessert is made, since no firing of rules is forced. The Prodigal policy selects the answer set in which both desserts are made (this is possible because enough ingredients are available for both recipes). Using the Optional policy all of the four possibilities are admissible.

The first two policies can be easily characterized by considering a partial order on the collection of mappings μ . Such an order can be directly induced by a partial order \sqsubseteq on $\mathcal{FM}(\mathbb{Z})$ simply defined as: $\forall m \in \mathcal{FM}(\mathbb{Z}) (\emptyset \sqsubseteq m)$. Consequently, a partial order \sqsubseteq' on \mathbb{S}_P can be so defined: $\forall F_1, F_2 \in \mathbb{S}_P (F_1 \sqsubseteq' F_2 \leftrightarrow \forall \gamma \in P (F_1(\gamma) \sqsubseteq F_2(\gamma)))$. Finally, a partial order \sqsubseteq'' on the collection of mappings turns out to be so definable: $\forall \mu_1, \mu_2 \in (\mathbb{S}_P)^{\Pi_R} (\mu_1 \sqsubseteq'' \mu_2 \leftrightarrow \forall q \in \Pi_R (\mu_1(q) \sqsubseteq' \mu_2(q)))$.

Given an r-program P , consider the collection of those answer sets $\langle I, \mu \rangle$ of P which agree on I . The adoption of the Prodigal policy corresponds to focusing on answer sets that are maximal w.r.t. the order \sqsubseteq'' . Conversely, the Thrifty policy selects \sqsubseteq'' -minimal answer sets. Notice that, for a given (ground) r-program, by Def. 2, fixing \sqsubseteq as above implies that $\mu_1 \sqsubseteq'' \mu_2$ if μ_2 fires all the rules in P fired by μ_1 . A refined choice for \sqsubseteq , viable for instance in presence of multiple firings, could be: $\forall m_1, m_2 \in \mathcal{FM}(\mathbb{Z}) (m_1 \sqsubseteq m_2 \leftrightarrow \Sigma(m_1) \preceq \Sigma(m_2))$.

5 Towards a running implementation of ground RASP

In this section we explore a way of characterizing the semantics of r-programs by describing a translation of ground r-programs into plain ASP, so as to establish a correspondence between the answer sets of an r-program and the answer sets of a suitable ASP-program. (Needless to say that such a translation should be carried out by a compiler and then it would be transparent to the user.) In what follows we deal with ground r-programs only. We remark that the grounding process has to take care of multiple amount-atoms that could appear in a rule as effect of ground substitution. Multiple copies of the same amount-atom must not be deleted, since each occurrence of an amount-atom corresponds to a different amount of resource. Keeping track of multiple copies of amount-atoms reflects, in the translation into ASP code, the use of multisets in defining the semantics of r-programs.

In implementing RASP one has to render the group \mathbb{Z} (or \mathbb{Q}), together with all needed operations and functions (namely, sum, inversion, comparison, etc.), as well as the mapping κ . All these ingredients could be realized in at least two ways (not necessarily antithetic). On the one hand, since, as mentioned, a finite portion of \mathbb{Z} enters into play in a ground program, we can encode it and the corresponding restrictions of all the needed operations as fragments of ASP code (e.g., by explicit tabulation). Alternatively, one could profit from some form of built-in or external source of computation. Examples of features supporting external evaluation in ASP-solvers are the user-defined functions and the API of `lparse` [13], external evaluation in `d1v` [5, 4], or even the integration with other programming paradigms and tools [14].

ASP encoding of r-rules. In encoding resource handling in ASP, in order to use solvers such as `smodels` or `d1v`, we have to cope with a limitation of these tools relative to dealing with negative numbers. Some solvers (e.g. `d1v` [9]) explicitly restrain the integers occurring in a program to be non-negative. Others, namely those based on the front-end `lparse` [15], do not explicitly impose such a restriction. Nevertheless, some unexpected behaviors may be experimented when negative quantities enter into play in the search/construction of answer sets.⁵ To overcome this limitation, we are forced to separate the treatment of negative (i.e., consumed) and

⁵ Consider, for instance, two variants of the following program obtained by substituting i with 3 and -3 , respectively. `Lparse` (vers. 1.0.17) produces unexpected answers: `nump(i)` belongs to the answer set only for the case of $i = 3$. (Similar

positive (i.e., produced) quantities. This slightly complicates the translation of an r-program into an ASP program, that would be simpler if the ASP-solver at hand treated negative integers directly.

Let γ be the a ground r-rule $p \leftarrow L_1, \dots, L_n, q_1:a_1, \dots, q_k:a_k$ (with $k, n \geq 0$), then its rendering in ASP code is made of these ASP rules:⁶

- (1) $p :- L_1, \dots, L_n, \text{fired}(n_\gamma).$
- (2) $:- \bar{L}_i, \text{fired}(n_\gamma).$ for $j = 1, \dots, n$
- (3) $\text{fired}(n_\gamma) :- \text{use_n}(n_\gamma, 1, q_1, a_1), \dots, \text{use_n}(n_\gamma, k, q_k, a_k).$
- (4) $\text{cons}(n_\gamma, j, q_j, a_j).$ for $j = 1, \dots, k$
- (5) $:- 1\{\text{use_n}(n_\gamma, 1, q_1, a_1), \dots, \text{use_n}(n_\gamma, k, q_k, a_k)\}k - 1.$

where n_γ is a new constant uniquely associated to γ . The cardinality constraint used in rule (5) (see, [13] for a detailed description), imposes that either none or all of the literals $\text{use_n}(n_\gamma, 1, q_1, a_1), \dots, \text{use_n}(n_\gamma, k, q_k, a_k)$ are true. Notice the use of a specific argument of the predicate **cons** (namely, the second one) in order to distinguish among different occurrences of identical amount atoms by indexing the amount atoms of the rule; we assume defined a domain predicate $\text{idx}(0..m)$, where m is the maximum number of amount-atoms in a rule. For a rule of the form $q_0:a_0 \leftarrow L_1, \dots, L_n, q_1:a_1, \dots, q_k:a_k$, the encoding changes as follows:

- (2') $:- \bar{L}_i, \text{fired}(n_\gamma).$ for $j = 1, \dots, n$
- (3') $\text{fired}(n_\gamma) :- \text{use_p}(n_\gamma, 0, q_0, a_0), \text{use_n}(n_\gamma, 1, q_1, a_1), \dots, \text{use_n}(n_\gamma, k, q_k, a_k).$
- (4') $\text{prod}(n_\gamma, 0, q_0, a_0).$ for $j = 1, \dots, k$
- (5') $:- 1\{\text{use_p}(n_\gamma, 0, q_0, N), \text{use_n}(n_\gamma, 1, q_1, a_1), \dots, \text{use_n}(n_\gamma, k, q_k, a_k)\}k.$

(Where (3')-(5') should be suitably generalized in case of rules having more than one amount-atom in the head.) Notice the separation between positive/produced and negative/consumed amounts by means of the predicate symbols **use_p/prod** and **use_n/cons**. Negative amount-atoms $-q:a$ are treated by exchanging the use of **cons** and **prod**. Ordinary program-rules are left unchanged.

Ensuring correct usage of resources. The following code imposes correct usage of resources in firing a rule:

- (10) $\text{use_p}(G, I, R, Q) :- \text{fired}(G), \text{prod}(G, I, R, Q).$
- (11) $\text{use_n}(G, I, R, Q) :- \text{fired}(G), \text{cons}(G, I, R, Q).$
- (12) $:- \text{use_p}(G, I, R, N), N \neq Q, \text{val}(N), \text{prod}(G, I, R, Q).$
- (13) $:- \text{use_n}(G, I, R, N), N \neq Q, \text{val}(N), \text{cons}(G, I, R, Q).$

(where **val** encodes a domain predicate for quantities, needed by **lparse**

behaviors have been reported in [6, Sec. 3.1].)

```
val(-5..8).      nump(X) :- num(X), val(X).
g :- num( i ).   num( i ) :- g.          g.
```

⁶ Here we are encoding assuming **lparse** as front-end of the solver.

for domain restriction during grounding). The balance for each resource is evaluated by the following fragment of code. Observe the rather counterintuitive programming trick exploited to evaluate sums (e.g., lines (16)-(18)) due to limitations of `lparse`'s language.

```
(14) 0{fired(G)}1 :- rule(G).          rule(nγ).  for each resource-rule γ
(15) res(qj).                          for each resource symbol qj
(16) #weight use_n(X,Y,W,Z)=Z.        #weight use_p(X,Y,W,Z)=Z.
(17) consumed(R,N) :- N[use_n(G,I,R,Q):val(Q):idx(I):rule(G)]N, res(R),val(N).
(18) 1{consumed(T,N) : val(N)}1 :- res(T).
(19) produced(R,N) :- N[use_p(G,I,R,Q):val(Q):idx(I):rule(G)]N, res(R),val(N).
(20) 1{produced(T,N) : val(N)}1 :- res(T).
(21) :- consumed(R,N), produced(R,P), res(R), val(P;N), P<N.
```

Rules (16)-(20) evaluates the global consumed/produced amounts. Such an involved use of weight literals to compute sums could be avoided in systems providing aggregate functions (e.g., `dlv`). Rule (21) imposes a positive global balance for each resource (cf., (1), page 4). To impose other forms of global constraints on the balance of a specific resource, one has to introduce modified versions of rule (21).

Dealing with multiple firing. As regards multiple firing of r-rules, if γ is of the form shown in (4), it suffices to add this fragment of code to the resulting ASP-program:

```
(22) firings(nγ,N1..N2).          for each rule [N1,N2] : γ
(23) 1{counter(G,N):firings(G,N)}1 :- fired(G), rule(G).
(24) :- not fired(G), counter(G,N), firings(G,N), rule(G).
```

where `counter` encodes the mapping ξ which associates a number n of firings to each fired rule, subject to the restriction $N_1 \leq n \leq N_2$ (imposed by the predicate `firings`).

To take into account repeated firings in the resource's balances, rules (3),(5),(10)-(13) (as well as (3'),(5')) have to be slightly modified as follows:

```
(3) fired(nγ):- use_n(nγ,1,q1,C*a1), ..., use_n(nγ,k,qk,C*ak),
               counter(nγ,C), firings(nγ,C).
(5) :- 1 {use_n(nγ,1,q1,C*a1), ..., use_n(nγ,k,qk,C*ak)} k-1,
       counter(nγ,C), firings(nγ,C).
(10) use_p(G,I,R,C*Q) :- fired(G), counter(G,C), firings(G,C), prod(G,I,R,Q).
(11) use_n(G,I,R,C*Q) :- fired(G), counter(G,C), firings(G,C), cons(G,I,R,Q).
(12) :- use_p(G,I,R,N), N!=C*Q, val(N), counter(G,C),
       firings(G,C), prod(G,I,R,Q).
(13) :- use_n(G,I,R,N), N!=C*Q, val(N), counter(G,C),
       firings(G,C), cons(G,I,R,Q).
```

Rendering of budget policies. The translation described so far realizes the Optional policy. The Prodigious policy is obtained by adding constraints that weed-out the answer sets in which a rule could be fired but it is not:

```
(30) balance(R,P-N) :- res(R), consumed(R,N), produced(R,P), P>=N, val(P;N).
(31) #weight cons(X,Y,W,Z)=Z.          #weight prod(X,Y,W,Z)=Z.
(32) consumes(G,R,N) :- N [cons(G,Idx,R,Q) : val(Q) : idx(Idx)] N,
                        rule(G), res(R), val(N).
(33) 1{consumes(G,T,N):val(N)}1 :- rule(G), res(T).
(34) produces(G,R,N) :- N [prod(G,Idx,R,Q) : val(Q) : idx(Idx)] N,
                        rule(G), res(R), val(N).
(35) 1{produces(G,T,N):val(N)}1 :- rule(G), res(T).
(36) :- enabled(G), rule(G).
(37) enbld(G,R) :- balance(R,U), produces(G,R,P1), consumes(G,R,N1),
                    (U+P)>=N, val(U;N;P), rule(G), res(R).
```

and instances of (38) and (39) are added for each r-rule γ :

```
(38) enabled(n $\gamma$ ) :- counter(n $\gamma$ ,C), firings(n $\gamma$ ,More), firings(n $\gamma$ ,C),
                    enbld(n $\gamma$ ,q $_1$ ), ..., enbld(n $\gamma$ ,q $_k$ ), L $_1$ , ..., L $_n$ .
(39) enabled(n $\gamma$ ) :- not fired(n $\gamma$ ), enbld(n $\gamma$ ,q $_1$ ), ..., enbld(n $\gamma$ ,q $_k$ ),
                    L $_1$ , ..., L $_n$ .
```

Note that the predicate **balance** encodes the component φ of the answer set of r-programs (see page 5).

An analogous treatment can be designed for other policies, such as the Thrifty policy. It is also easy to single-out an answer set corresponding to maximal numbers of firings by using the **maximize** instruction offered by **lpars**. For example (using **num** as auxiliary domain predicate):

```
#weight counter(X,Y)=Y.          maximize[counter(G,C):num(C):rule(G)].
```

The above-outlined translation from RASP into ASP, has been checked on several examples. For the time being, the translation is done by hand and the implementation of an automated compiler is part of our future works. Let us conclude this section by reporting on the output given by **smodels** for a short example.

Example 5. Different cakes require different amounts of ingredients:

```
egg:15.    flour:15.    sugar:13.    chocolate:4.
[1,5]: cake(small):1 :- egg:2, flour:3, sugar:3.
[1,2]: cake(big):1 :- egg:7, flour:6, sugar:5, chocolate:2.
```

By translation into ASP, under the Prodigious policy, and by running the ASP solver, we obtain the models:

```
Answer 1: balance(cake(big),2) balance(cake(small),0)
balance(chocolate,0) balance(egg,1) balance(flour,3) balance(sugar,3)
fired(g1) counter(g1,2) ...
```

```

Answer 2: balance(cake(big),0) balance(cake(small),4)
balance(chocolate,4) balance(egg,7) balance(flour,3) balance(sugar,1)
fired(g2) counter(g2,4) ...
Answer 3: balance(cake(big),1) balance(cake(small),2)
balance(chocolate,2) balance(egg,4) balance(flour,3) balance(sugar,2)
fired(g1) fired(g2) counter(g1,1) counter(g2,2) ...

```

6 Concluding remarks

In this work we have introduced RASP, an extension of ASP with the possibility of defining resources with their amounts. Resources can be produced and consumed by rules' firings, that can also be multiple, taking into account various kinds of *global* constraints on resource consumption/production, as well as policies to customize and filter resource allocation. Semantics of resources' amounts relies upon the algebraic structure of \mathbb{Z} which models quantities, operations, and relations among them. (Other choices for the domain of quantities are possible, e.g., \mathbb{Q} in place of \mathbb{Z} .)

Different allocations of resources correspond to different answer sets. A compilation process into plain ASP has been outlined. This translation makes use of specific features offered by the front-end `lparse`, but the approach can be easily rephrased for other ASP solvers [16]. The extension RASP can be useful to model more easily and directly several kinds of production processes and planning problems, including configuration problems.

An envisaged extension of the framework consists in allowing amounts in amount-atoms to be described explicitly as intervals or sets of values. Similarly, explicit expressions or compound resource-terms could be considered. Concerning negation-as-failure applied to amount-atoms, there might be several approaches to assess their semantics. It is not clear which is the most intuitive meaning of saying that “an amount-atom $q:a$ is not true”. Should it mean that the resource p is not produced at all? Or it means that p can be produced in any amount but different from a ? Moreover, what should be considered as the *scope* of this constraint? A single rule or the whole program? Actually, some of these approaches can be simulated by using the features described in Sect. 3 (cf., the constraints on global resource balance). For these reasons, it seemed more natural to us not to permit `naf` amount-atoms.

References

- [1] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See also asparagus.cs.uni-potsdam.de.
- [2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
- [3] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
- [4] F. Calimeri and G. Ianni. External sources of computation for answer set solvers. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proc. of the 8th Intl. Conference on Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *LNCS*, pages 105–118. Springer, 2005.
- [5] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In L. P. Kaelbling and A. Saffiotti, editors, *Proc. of the 19th Intl. Joint Conference on Artificial Intelligence*, pages 90–96. Professional Book Center, 2005.
- [6] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74, 2006.
- [7] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th Intl. Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.
- [8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [9] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.
- [10] V. Lifschitz. Answer set planning. In *Proc. of the 16th Intl. Conference on Logic Programming*, pages 23–37, 1999.
- [11] V. W. Marek and M. Truszczyński. *Stable logic programming - an alternative logic programming paradigm*, pages 375–398. Springer, 1999.
- [12] D. Nelson and M. Cox. *Lehninger Principles of Biochemistry*. Freeman & Co., 2004.
- [13] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proc. of the 8th Workshop on Non-Monotonic Reasoning*, 2000.
- [14] M. Osorio and E. Corona. The A-Pol system. In M. D. Vos and A. Provetti, editors, *Answer Set Programming, Advances in Theory and Implementation, Proc. of the 2nd Intl. ASP'03*, volume 78 of *CEUR Workshop Proc.*, 2003.
- [15] P. Simons, I. Niemelä, and T. Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
- [16] Web references for some ASP solvers. ASSAT: assat.cs.ust.hk; Ccalc: www.cs.utexas.edu/users/tag/ccalc; 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.
- [17] WASP-WP5 Report: Model applications and proofs-of-concept, 2005. Working Group on Answer Set Programming (WASP): www.kr.tuwien.ac.at/projects/WASP/report.html.

A Goal-Directed Calculus for Standard Conditional Logics

Nicola Olivetti¹ and Gian Luca Pozzato²

¹ LSIS - UMR CNRS 6168 Université Paul Cézanne (Aix-Marseille 3), Avenue
Escadrille Normandie-Niemen 13397 Marseille Cedex 20 - France

`nicola.olivetti@univ-cezanne.fr`

² Dipartimento di Informatica - Università degli Studi di Torino, corso Svizzera 185 -
10149 Turin - Italy

`pozzato@di.unito.it`

Abstract. In this paper we focus on proof methods for conditional logics. We present *US'*, a goal-directed calculus for the basic normal conditional logic CK and its standard extensions ID, MP, and ID+MP. *US'* is derived from some labelled sequent calculi, called *SeqS'*, and it is based on the notion of uniform proofs. We also introduce *GOALDUCK*, a simple implementation of *US'* written in SICStus Prolog¹.

1 Introduction

Conditional logics have a long history. They have been studied first by Lewis [26, 28, 3, 36] in order to formalize a kind of hypothetical reasoning (if *A* were the case then *B*), that cannot be captured by classical logic with material implication. In particular they were introduced to capture *counterfactual sentences*, i.e. conditionals of the form “if *A* were the case then *B* would be the case”, where *A* is false. If we interpret the *if...then* in the above sentence as a classical implication, we obtain that all counterfactuals are trivially true. Nonetheless one may want to reason about counterfactuals sentences and hence be capable of distinguishing among true and false ones [4].

In the last years, there has been a considerable amount of work on applications of conditional logics to various areas of artificial intelligence and knowledge representation such as non-monotonic reasoning, hypothetical reasoning, belief revision, and even diagnosis.

The application of conditional logics to nonmonotonic reasoning was firstly investigated by Delgrande [8] who proposed a conditional logic for prototypical reasoning; the understanding of a conditional $A \Rightarrow B$ in his logic is “the *A*'s have typically the property *B*”. For instance, one could have:

$$\begin{aligned} \forall x(Penguin(x) \rightarrow Bird(x)), \forall x(Penguin(x) \rightarrow \neg Fly(x)), \\ \forall x(Bird(x) \Rightarrow Fly(x)) \end{aligned}$$

¹ This research has been partially supported by “*Progetto Lagrange - Fondazione CRT*” and by the projects “*MIUR PRIN05: Specification and verification of agent interaction protocols*” and “*GALILEO 2006: Interazione e coordinazione nei sistemi multi-agenti*”.

The last sentence states that birds typically fly. Observe that replacing \Rightarrow with the classical implication \rightarrow , the above knowledge base is consistent only if there are no penguins. The study of the relations between conditional logics and non-monotonic reasoning has gone much further since the seminal work by Kraus, Lehmann, and Magidor [24] (KLM framework), who proposed a formalization of the properties of a nonmonotonic consequence relation: their system comprises nonmonotonic assertions of the form $A \sim B$, interpreted as “ B is a plausible conclusion of A ”. It turns out that all forms of inference studied in KLM framework are particular cases of well-known conditional axioms [6]. In this respect the KLM language is just a fragment of conditional logics. We refer to [10] for a broader discussion on the application of conditional logics to nonmonotonic reasoning.

Conditional logics have also been used to formalize knowledge update and revision. For instance, Grahne presents a conditional logic (a variant of Lewis’ VCU) to formalize knowledge-update as defined by Katsuno and Mendelzon [23]. More recently, in [16] and [15], it has been shown a tight correspondence between AGM revision systems [1] and a specific conditional logic, called BCR. The connection between revision/update and conditional logics can be intuitively explained in terms of the so-called Ramsey Test (RT): the idea is that $A \Rightarrow B$ “holds” in a knowledge base K if and only if B “holds” in the knowledge base K revised/updated with A ; this can be expressed by (RT) $K \vdash A \Rightarrow B$ iff $K \circ A \vdash B$, where \circ denotes a revision/update operator.

Conditional logics have also been used to model hypothetical queries in deductive databases and logic programming; the conditional logic CK+ID is the basis of the logic programming language CondLP defined in [12]. In that language one can have hypothetical goals of the form (quoting the old Yale’s Shooting problem) $Load_gun \Rightarrow (Shoot \Rightarrow Dead)$ and the idea is that the hypothetical goal succeeds if $Dead$ succeeds in the state “revised” first by $Load_gun$ and then by $Shoot$ ².

In a related context, conditional logics have been used to model causal inference and reasoning about action execution in planning [35, 22]. In the causal interpretation the conditional $A \Rightarrow B$ is interpreted as “ A causes B ”; observe that identity (i.e. $A \Rightarrow A$) is not assumed to hold.

Moreover, conditional logics have found some applications in diagnosis, where they can be used to reason counterfactually about the expected functioning of system components in face of the observed faults [29].

In spite of their significance, very few proof systems have been proposed for conditional logics. One possible reason of the underdevelopment of proof-methods for conditional logics is the lack of a universally accepted semantics for them. This is in sharp contrast to modal and temporal logics which have a consolidated semantics based on a standard kind of Kripke structures.

Similarly to modal logics, the semantics of conditional logics can be defined in terms of possible world structures. In this respect, conditional logics can be

² The language CondLP comprises a nonmonotonic mechanism of revision to preserve the consistency of a program potentially violated by an hypotheticalal assumption.

seen as a generalization of modal logics (or a type of multi-modal logic) where the conditional operator is a sort of modality indexed by a formula of the same language. The two most popular semantics for conditional logics are the so-called *sphere semantics* [26] and the *selection function semantics* [28]. Both are possible-world semantics, but are based on different (though related) algebraic notions. Here we adopt the selection function semantics, which is more general and considerably simpler than the sphere semantics.

With the selection function semantics, truth values are assigned to formulas depending on a world; intuitively, the selection function f selects, for a world w and a formula A , the set of worlds $f(w, A)$ which are “most-similar to w ” or “closer to w ” given the information A . In *normal* conditional logics, the function f depends on the set of worlds satisfying A rather than on A itself, so that $f(w, A) = f(w, A')$ whenever A and A' are true in the same worlds (normality condition). A conditional sentence $A \Rightarrow B$ is true in w whenever B is true in every world selected by f for A and w . Since we adopt the selection function semantics, CK is the fundamental system [28]; it has the same role as the system K (from which it derives its name) in modal logic: CK-valid formulas are exactly those ones that are valid in every selection function model.

In this paper we first present a sequent calculus for CK and for some of its standard extensions, namely $\{\text{ID}, \text{MP}, \text{ID+MP}\}$. This calculus is called SeqS', and it makes use of labels following the line of [37] and [11]. Two types of formulas are involved in the rules of the calculi: world formulas of the form $x : A$ representing that A holds at world x and transition formulas of the form $x \xrightarrow{A} y$ representing that $y \in f(x, A)$. The rules manipulate both kinds of formulas.

We then show that SeqS' calculi can be the starting point to develop goal-directed proof procedures, according to the paradigm of Uniform Proofs by Miller and others [27, 13]. Calculi of these kind are suitable for logic programming applications. The basic idea of goal-directed proof search is as follows: given a sequent $\Gamma \vdash G$, one can interpret Γ as the program or database, and G as a goal whose proof is searched. The backward proof search of the sequent is driven by the goal G , in the sense that the goal is stepwise decomposed according to its logical structure. In a goal-directed proof method, the logical connectives can be interpreted operationally as simple and fixed search instructions. A proof of this sort is called a *uniform proof*. Given a sequent calculus for a logic L, in general, not every provable sequent admits a uniform proof: one must identify a significant fragment of L that allows uniform proofs. Usually, this fragment (in the propositional case) is alike to the Harrop-fragment of intuitionistic logic (see [27]). To specify this fragment one distinguish between the formulas which can occur in the database (D-formulas) and the formulas that can be asked as goals (G-formulas).

We present goal-directed proof procedures for a selected fragment of CK and its extensions with axioms ID and MP. We call these calculi \mathcal{US}' , where S' stands for $\{\text{CK}, \text{ID}, \text{MP}, \text{ID+MP}\}$.

Finally, we introduce GOALD \mathcal{U} CK, a very simple SICStus Prolog implementation of the calculi \mathcal{US}' . As far as we know, no other goal-directed theorem

prover for the above standard conditional logics has been previously described in the literature.

The plan of the paper is as follows. In section 2 we introduce the conditional systems we consider, then we present the sequent calculi SeqS' for conditional logics in section 3. In section 4 we present the goal-directed proof procedure *US'* derived from SeqS'. In section 5 we present the theorem prover *GOALDUCK*. In section 6 we discuss some related work and possible future research.

2 Conditional Logics

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

A propositional conditional language \mathcal{L} is defined from: a set of propositional variables *ATM*; the symbols of *false* \perp and *true* \top ; a set of connectives $\neg, \rightarrow, \vee, \wedge, \Rightarrow$. We define formulas of \mathcal{L} as follows:

- \perp, \top , and the propositional variables of *ATM* are *atomic formulas*;
- if A is a formula, then $\neg A$ is a *complex formula*;
- if A and B are formulas, $A \rightarrow B, A \vee B, A \wedge B$, and $A \Rightarrow B$ are *complex formulas*.

Similarly to modal logics, the semantics of conditional logics can be defined in terms of possible world structures. In this respect, conditional logics can be seen as a generalization of modal logics (or a type of multi-modal logic) where the conditional operator is a sort of modality indexed by a formula of the same language. The two most popular semantics for conditional logics are the so-called *sphere semantics* [26] and the *selection function semantics* [28]. Both are possible-world semantics, but are based on different (though related) algebraic notions. In this work, we adopt the more general solution of the selection function semantics. We consider a non-empty set of possible worlds \mathcal{W} . Intuitively, the selection function f selects, for a world w and a formula A , the set of worlds of \mathcal{W} which are *closer* to w given A . A conditional formula $A \Rightarrow B$ holds in a world w if the formula B holds in *all the worlds selected by f for w and A* .

Definition 1. A selection function model is a triple $\mathcal{M} = \langle \mathcal{W}, f, [\] \rangle$ where:

- \mathcal{W} is a non empty set of items called worlds;
- f is the so-called selection function and has the following type:
 $f: \mathcal{W} \times 2^{\mathcal{W}} \longrightarrow 2^{\mathcal{W}}$
- $[\]$ is the evaluation function, which assigns to an atom $P \in \text{ATM}$ the set of worlds where P is true, and is extended to the other formulas as follows:
 - $[\perp] = \emptyset$
 - $[\top] = \mathcal{W}$
 - $[\neg A] = \mathcal{W} - [A]$
 - $[A \rightarrow B] = (\mathcal{W} - [A]) \cup [B]$
 - $[A \vee B] = [A] \cup [B]$
 - $[A \wedge B] = [A] \cap [B]$

- $[A \Rightarrow B] = \{w \in \mathcal{W} \mid f(w, [A]) \subseteq [B]\}$

Notice that we have defined f taking $[A]$ rather than A ; this is equivalent to define f on formulas, i.e. $f(w, A)$ but imposing that if $[A] = [A']$ in the model, then $f(w, A) = f(w, A')$. This condition is called *normality*.

The semantics above characterizes the *basic normal conditional system*, called CK. An axiomatization of the CK system is given by:

- any axiomatization of classical propositional logic
- (Modus Ponens)
$$\frac{A \quad A \rightarrow B}{B}$$
- (RCEA)
$$\frac{A \leftrightarrow B}{(A \Rightarrow C) \leftrightarrow (B \Rightarrow C)}$$
- (RCK)
$$\frac{(A_1 \wedge \dots \wedge A_n) \rightarrow B}{(C \Rightarrow A_1 \wedge \dots \wedge C \Rightarrow A_n) \rightarrow (C \Rightarrow B)}$$

As in modal logic, extensions of the basic system CK are obtained by assuming further properties on the selection function. Here we consider the following standard extensions:

Name	Axiom	Model condition
ID	$A \Rightarrow A$	$f(w, [A]) \subseteq [A]$
MP	$(A \Rightarrow B) \rightarrow (A \rightarrow B)$	if $w \in [A]$ then $w \in f(w, [A])$

3 Sequent Calculi SeqS'

In this section we present some sequent calculi for conditional logics. These calculi are called SeqS', where S' stands for $\{\text{CK}, \text{ID}, \text{MP}, \text{ID} + \text{MP}\}$ ³, and they are inspired to labelled deductive systems introduced in [11] and in [37].

In Figure 1 we present SeqS'; the calculi make use of *labelled formulas*, where the labels are drawn from a denumerable set \mathcal{A} ; there are two kinds of formulas:

- *world formulas*, denoted by $x : A$, where $x \in \mathcal{A}$ and $A \in \mathcal{L}$;
- *transition formulas*, denoted by $x \xrightarrow{A} y$, where $x, y \in \mathcal{A}$ and $A \in \mathcal{L}$.

A world formula $x : A$ is used to represent that A holds in the possible world represented by the label x ; a transition formula $x \xrightarrow{A} y$ represents that $y \in f(x, [A])$. A *sequent* is a pair $\langle \Gamma, \Delta \rangle$, usually denoted with $\Gamma \vdash \Delta$, where Γ and Δ are multisets of labelled formulas. The intuitive meaning of $\Gamma \vdash \Delta$ is: every model that satisfies all labelled formulas of Γ in the respective worlds (specified by the labels) satisfies at least one of the labelled formulas of Δ (in those worlds). The rigorous definition of sequent validity is stated as follows:

³ In [33] sequent calculi SeqS' are also extended to the normal conditional logics allowing axioms CS and CEM. These calculi are called SeqS.

Definition 2 (Sequent validity). Given a model $\mathcal{M} = \langle \mathcal{W}, f, [\] \rangle$ for \mathcal{L} , and a label alphabet \mathcal{A} , we consider any mapping $I : \mathcal{A} \rightarrow \mathcal{W}$.

Let F be a labelled formula, we define $\mathcal{M} \models_I F$ as follows:

- $\mathcal{M} \models_I x : A$ iff $I(x) \in [A]$
- $\mathcal{M} \models_I x \xrightarrow{A} y$ iff $I(y) \in f(I(x), [A])$

We say that $\Gamma \vdash \Delta$ is valid in \mathcal{M} if for every mapping $I : \mathcal{A} \rightarrow \mathcal{W}$, if $\mathcal{M} \models_I F$ for every $F \in \Gamma$, then $\mathcal{M} \models_I G$ for some $G \in \Delta$. We say that $\Gamma \vdash \Delta$ is valid in a system (CK or one of its extensions) if it is valid in every \mathcal{M} satisfying the specific conditions for that system (if any).

$(\mathbf{AX}) \Gamma, x : P \vdash \Delta, x : P \quad (P \in \text{ATM})$	
$(\mathbf{A}\perp) \Gamma, x : \perp \vdash \Delta$	$(\mathbf{A}\top) \Gamma \vdash \Delta, x : \top$
$(\neg L) \frac{\Gamma \vdash \Delta, x : A}{\Gamma, x : \neg A \vdash \Delta}$	$(\neg R) \frac{\Gamma, x : A \vdash \Delta}{\Gamma \vdash \Delta, x : \neg A}$
$(\wedge L) \frac{\Gamma, x : A, x : B \vdash \Delta}{\Gamma, x : A \wedge B \vdash \Delta}$	$(\wedge R) \frac{\Gamma \vdash \Delta, x : A \quad \Gamma \vdash \Delta, x : B}{\Gamma \vdash \Delta, x : A \wedge B}$
$(\vee L) \frac{\Gamma, x : A \vdash \Delta \quad \Gamma, x : B \vdash \Delta}{\Gamma, x : A \vee B \vdash \Delta}$	$(\vee R) \frac{\Gamma \vdash \Delta, x : A, x : B}{\Gamma \vdash \Delta, x : A \vee B}$
$(\rightarrow L) \frac{\Gamma \vdash \Delta, x : A \quad \Gamma, x : B \vdash \Delta}{\Gamma, x : A \rightarrow B \vdash \Delta}$	$(\rightarrow R) \frac{\Gamma, x : A \vdash \Delta, x : B}{\Gamma \vdash \Delta, x : A \rightarrow B}$
$(\Rightarrow L) \frac{\Gamma, x : A \Rightarrow B \vdash \Delta, x \xrightarrow{A} y \quad \Gamma, x : A \Rightarrow B, y : B \vdash \Delta}{\Gamma, x : A \Rightarrow B \vdash \Delta}$	$(\Rightarrow R) \frac{\Gamma, x \xrightarrow{A} y \vdash \Delta, y : B}{\Gamma \vdash \Delta, x : A \Rightarrow B}$
$(EQ) \frac{u : A \vdash u : B \quad u : B \vdash u : A}{\Gamma, x \xrightarrow{A} y \vdash \Delta, x \xrightarrow{B} y}$	
$(MP) \frac{\Gamma \vdash \Delta, x \xrightarrow{A} x, x : A}{\Gamma \vdash \Delta, x \xrightarrow{A} x}$	$(ID) \frac{\Gamma, x \xrightarrow{A} y, y : A \vdash \Delta}{\Gamma, x \xrightarrow{A} y \vdash \Delta}$

Fig. 1. Sequent calculi SeqS'. Rules (ID) and (MP) are only used in corresponding extensions of the basic system SeqCK.

Systems combining two or more semantic conditions are characterized by all rules capturing those conditions.

As an example, in Figure 2 we present a derivation in SeqID of the valid sequent $\vdash x : (P \rightarrow Q) \Rightarrow (P \rightarrow Q)$.

$$\begin{array}{c}
\frac{x \xrightarrow{P \rightarrow Q} y, y : P \vdash y : Q, y : P \quad x \xrightarrow{P \rightarrow Q} y, y : Q, y : P \vdash y : Q}{x \xrightarrow{P \rightarrow Q} y, y : P \rightarrow Q, y : P \vdash y : Q} (\rightarrow L) \\
\frac{x \xrightarrow{P \rightarrow Q} y, y : P \rightarrow Q, y : P \vdash y : Q}{x \xrightarrow{P \rightarrow Q} y, y : P \vdash y : Q} (ID) \\
\frac{x \xrightarrow{P \rightarrow Q} y, y : P \vdash y : Q}{x \xrightarrow{P \rightarrow Q} y \vdash y : P \rightarrow Q} (\rightarrow R) \\
\frac{x \xrightarrow{P \rightarrow Q} y \vdash y : P \rightarrow Q}{\vdash x : (P \rightarrow Q) \Rightarrow (P \rightarrow Q)} (\Rightarrow R)
\end{array}$$

Fig. 2. A derivation in SeqID for $\vdash x : (P \rightarrow Q) \Rightarrow (P \rightarrow Q)$.

SeqS' calculi are sound and complete with respect to the semantics:

Theorem 1 (Soundness and completeness [33]). A sequent $\Gamma \vdash \Delta$ is valid if and only if $\Gamma \vdash \Delta$ is derivable in SeqS'.

4 Goal-directed Proof Procedure for Conditional Logics

In this section we investigate how SeqS' calculi can be used in order to develop goal-directed proof procedures for conditional logics, following the paradigm of Uniform Proof by Miller and others [27, 13].

The paradigm of uniform proof can be seen as a generalization of conventional logic programming. Given a sequent $\Gamma \vdash G$, one can interpret Γ as the *program* or the *knowledge base* or the *database*, whereas G can be seen as a *goal* whose proof is searched. Intuitively, the basic idea is that the backward proof of $\Gamma \vdash G$ is driven by the goal G ; roughly speaking, the goal G is stepwise decomposed according to its logical structure by the rules of the calculus, until its constituents are reached. The connectives in G can be interpreted operationally as search instructions. To prove an atomic goal Q , the proof search mechanism checks if Γ contains a “clause” whose head matches with Q and then tries to prove the “body” of the clause.

Given a sequent calculus, not every valid sequent admits a uniform proof of this kind; in order to describe a goal-directed proof search one must identify a fragment of the corresponding logic that allows uniform proofs.

Here we present a simple goal-directed calculus US' , where S' stands for $\{\text{CK, ID, MP, ID+MP}\}$. First of all, we specify the fragment of the conditional language \mathcal{L} we consider. We distinguish between the formulas which can occur in the program (or knowledge base or database), called D -formulas, and the formulas that can be asked as goals, called G -formulas.

Definition 3 (Language for uniform proofs). We consider the fragment of the conditional language \mathcal{L} , called $\mathcal{L}US'$, comprising:

The rule $(\mathcal{U} \Rightarrow)_{\mathbf{CK}}$ belongs to $\mathcal{U}\mathbf{CK}$ and $\mathcal{U}\mathbf{MP}$ only, whereas $(\mathcal{U} \Rightarrow)_{\mathbf{ID}}$ is used in $\mathcal{U}\mathbf{ID}$ and $\mathcal{U}\mathbf{ID}+\mathbf{MP}$. The rule $(\mathcal{U} \mathbf{trans})_{\mathbf{MP}}$ only belongs to the calculi $\mathcal{U}\mathbf{MP}$ and $\mathcal{U}\mathbf{ID}+\mathbf{MP}$.

A *proof* is a tree whose branches are sequences of nodes $\Gamma_i \vdash_{GD} \gamma_i$, where γ_i is a goal (i.e. either a formula $x : G$ or a transition formula $x \xrightarrow{A} y$). Each node $\Gamma_i \vdash_{GD} \gamma_i$ is obtained by its immediate predecessor $\Gamma_{i-1} \vdash_{GD} \gamma_{i-1}$ by applying a rule of $\mathcal{U}\mathbf{S}'$, i.e. $\Gamma_{i-1} \vdash_{GD} \gamma_{i-1} \Rightarrow \Gamma_i \vdash_{GD} \gamma_i$. A branch is closed if one of its nodes is an instance of $(\mathcal{U} \top)$ or of $(\mathcal{U} \mathbf{ax})$, otherwise it is open. A *derivation* is a proof whose branches are all closed. We give the following (standard) definition:

Definition 4. *If Γ is a database and γ is a goal, then $\Gamma \vdash_{GD} \gamma$ is derivable in $\mathcal{U}\mathbf{S}'$ if there is a derivation in $\mathcal{U}\mathbf{S}'$ starting with $\Gamma \vdash_{GD} \gamma$.*

Given a formula of type A , i.e. either an atomic formula or a boolean combination of conjunctions and disjunctions of atomic formulas, the operation $\text{Flat}(x : A)$ has the effect of *flatten* the conjunction/disjunction into sets of atomic formulas:

Definition 5 (Flat Operation).

- $\text{Flat}(x : Q) = \{\{x : Q\}\}$, with $Q \in \mathbf{ATM}$;
- $\text{Flat}(x : F \vee G) = \text{Flat}(x : F) \cup \text{Flat}(x : G)$
- $\text{Flat}(x : F \wedge G) = \{S_F \cup S_G \mid S_F \in \text{Flat}(x : F) \text{ and } S_G \in \text{Flat}(x : G)\}$

This operation is needed when *rules introducing a formula of type A in the database* are applied, since an A -formula might not be a D -formula. These rules, namely $(\mathcal{U} \mathbf{trans})$ and $(\mathcal{U} \Rightarrow)_{\mathbf{ID}}$, introduce a formula of type A in the left hand side of the sequent, i.e. a formula possibly being a combination of conjunctions and disjunctions of atoms. This formula needs to be decomposed in its atomic components. Indeed, in order to search a derivation for a transition formula $x \xrightarrow{A} y$, when $(\mathcal{U} \mathbf{trans})$ is applied by considering a transition $x \xrightarrow{A'} y$ in the program (or database), then the calculus leads to search a derivation for both $u : A' \vdash_{GD} u : A$ and $u : A \vdash_{GD} u : A'$; intuitively, this step corresponds to an application of the (EQ) rule in SeqS' . As an example, suppose that A has the form $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n$; in this case, in order to prove $u : A \vdash_{GD} u : A'$, we need to flat the database, thus proving the following sequent: $u : Q_1, u : Q_2, \dots, u : Q_n \vdash_{GD} u : A'$. In case A has the form $Q_1 \vee Q_2$, then we need to prove both $u : Q_1 \vdash_{GD} u : A'$ and $u : Q_2 \vdash_{GD} u : A'$. The same happens when $(\mathcal{U} \Rightarrow)_{\mathbf{ID}}$ is applied to $\Gamma \vdash_{GD} x : A \Rightarrow B$, and the computation steps to prove $\Gamma, x \xrightarrow{A} y, y : A \vdash_{GD} y : B$, and $y : A$ needs to be decomposed as defined here above.

As another example, suppose that the formula $x : A$ is introduced in the database, and that $A = P \vee (Q \wedge (R \vee S))$. The Flat operation returns the following databases: $\text{Flat}(x : A) = \{\{P\}, \{Q, R\}, \{Q, S\}\}$.

Moreover, we write $\Gamma, \text{Flat}(x : A) \vdash_{GD} \gamma$ to denote that the goal γ can be derived from *all* the databases obtained by applying Flat to $x : A$ (and adding formulas in Γ), that is to say:

Definition 6. Given a database Γ , a formula A and a goal G , let $\text{Flat}(x : A) = \{S_1, S_2, \dots, S_n\}$. We write $\Gamma, \text{Flat}(x : A) \vdash_{GD} \gamma$ if and only if $\forall i = 1, 2, \dots, n$ we have that $\Gamma, S_i \vdash_{GD} \gamma$.

The goal-directed calculi \mathcal{US}' are sound and complete wrt to the semantics, that is to say it can be shown that:

Theorem 2 (Soundness and Completeness of \mathcal{US}'). If Γ is a database, γ is a goal (i.e. either a formula $x : G$ or a transition formula $x \xrightarrow{A} y$), then $\Gamma \vdash_{GD} \gamma$ is derivable in \mathcal{US}' if and only if $\Gamma \vdash \gamma$ is derivable in the corresponding system SeqS' .

In order to save space, we omit the proof, which is similar to the one of Theorem 6.5 presented in [33]. Soundness and completeness wrt the semantics immediately follow from the fact that SeqS' calculi are sound and complete wrt selection function models [33].

As an example of usage of the goal-directed proof procedures \mathcal{US}' , consider the following knowledge base Γ , representing the functioning of a simple DVD recorder:

- (1) $x : \text{seton} \Rightarrow ((\text{pressRecButton} \Rightarrow \text{recording}) \rightarrow \text{readyToRec})$
- (2) $x : \text{seton} \Rightarrow \text{pressRecButton} \Rightarrow (\text{sourceSelected} \rightarrow \text{recording})$
- (3) $x : \text{seton} \Rightarrow \text{pressRecButton} \Rightarrow (\top \rightarrow \text{sourceSelected})$

We should interpret a conditional formula $A \Rightarrow B$ as “if the current state is updated with A then B holds” or, if A were an action, as “as an effect of A , B holds”, or “having performed A , B holds”. In this respect, clauses in Γ can be interpreted as: (1) “having set the device on, it is ready to record whenever it starts to record after pressing the REC button”; (2) “having set the device on and then having pressed the REC button, if the registration source has been selected, then the device will start to record”; (3) “having set the device on and then having pressed the REC button, it results that the registration source has been selected (the default source)”. For a broader discussion on plausible interpretations of conditionals, we remind the reader to [35, 22, 12]; here we just observe that they have been widely used to express update/action/causation.

We show that the goal “Having set the DVD recorder on, is it ready to record?”, formalized as $x : \text{seton} \Rightarrow \text{readyToRec}$ derives from Γ in \mathcal{UCK} . The derivation is presented in Figure 4.

5 The Theorem Prover GoalD \mathcal{UCK}

In this section we present GOALD \mathcal{UCK} , a very simple implementation of the calculi \mathcal{US}' . GOALD \mathcal{UCK} is a SICStus Prolog program consisting of only eight clauses (resp. nine clauses in systems allowing MP), each one of them implementing a rule of the calculus⁴, with the addition of some auxiliary predicates and of

⁴ For technical reasons, GOALD \mathcal{UCK} splits the rule (\mathcal{U} **prop**) in two clauses, one taking care of applying it to the specific case of a goal $x : Q$ by using a clause $x : G \rightarrow Q$.


```

prove([X,Q],Gamma,Trans,Labels):-
  member([Y,F=>(G->Q)],Gamma),atom(Q),
  prove([X,G],Gamma,Trans,Labels),extract(F,List),
  proveList(X,Y,List,Gamma,Trans,Labels).

prove([X,A,Y],_,Trans,_):-
  member([X,AP,Y],Trans),flat(A,FlattenedA),flat(AP,FlattenedAP),
  proveFlat([x,A],[],[],[x],x,FlattenedAP),
  proveFlat([x,AP],[],[],[x],x,FlattenedA).

```

The clause implementing $(\mathcal{U} \Rightarrow)_{\text{CK}}$ is very intuitive: if $A \Rightarrow G$ is the current goal, then the `generateLabel` predicate introduces a new label Y , then the predicate `prove` is called to prove the goal $y : G$ from a knowledge base enriched by the transition $x \xrightarrow{A} y$. Obviously, in the versions of `GOALDUCK` supporting `CK+ID{+MP}`, i.e. implementing the goal-directed calculi $\mathcal{UID}\{+MP\}$, this clause is replaced by the one implementing the rule $(\mathcal{U} \Rightarrow)_{\text{ID}}$.

The clause implementing $(\mathcal{U} \text{ prop})$ proceeds as follows: first, it searches for a clause $y : F \Rightarrow (G \rightarrow Q)$ in the database Γ , then it checks if Q is an atom, i.e. if $Q \in \text{ATM}$; second, it makes a recursive call to `prove` in order to find a derivation for the goal $x : G$; finally, it invokes two auxiliary predicates, `extract` and `proveList`, having the following functions:

- `extract` builds a list of the form $[A_1, A_2, \dots, A_n]$, where $F = A_1 \Rightarrow A_2 \Rightarrow \dots \Rightarrow A_n$;
- `proveList` invokes recursively the predicate `prove` in order to find a uniform proof for each goal $x_i \xrightarrow{A_{i+1}} x_{i+1}$ such that A_{i+1} belongs to the list generated by `extract`.

Given a goal $[X,A,Y]$, the clause implementing $(\mathcal{U} \text{ trans})$ is invoked. It first searches for a transition formula $[X,AP,Y]$ in the database (i.e. in the list `Trans`), then it calls the predicate `flat` on both formulas A and AP ; this predicate implements the `Flat` operation, building a *list of databases* obtained by flattening A (resp. AP) as defined in Definition 5. In order to prove `Flat`($u : A$) $\vdash_{GD} u : AP$ and `Flat`($u : AP$) $\vdash_{GD} u : A$, another auxiliary predicate, called `proveFlat`, is finally invoked by the clause implementing $(\mathcal{U} \text{ prop})$; `proveFlat` recursively invokes the predicate `prove` by using *all different databases* built by `flat`.

The performances of `GOALDUCK` are promising. We have tested its performances and compared them with `CondLean`'s. `CondLean` is a theorem prover implementing `SeqS`' calculi written in `SICStus Prolog` [30, 31]. We have implemented a `SICStus Prolog` program testing both `GOALDUCK` and `CondLean`, versions for `CK`, in order to compare their performances. This program randomly generates a database containing a specific set of formulas, obtained by combining a fixed set of propositional variables ATM . Moreover, the program builds a set of goals, whose cardinality is specified as a parameter, that can be either derivable or not from the generated database. Each goal is obtained from a set ATM° of variables which is a *subset* of the set ATM of variables in the database, in order to study situations in which several formulas in the database

are *useless* to prove a goal. *GOALDUCK* seems to offer better performances than *CondLean*. We have tested the two theorem provers over 100 goals, obtaining that *GOALDUCK* is able to prove 81 goals, whereas *CondLean* answers positively in 72 cases. Moreover, *GOALDUCK* concludes its work in 97 cases over 100 within the fixed time limit of 1 ms, even with a finite failure in 16 cases, whereas *CondLean* results in a time out in 28 cases.

We conclude this section by remarking that goal-directed proof methods usually do not ensure a terminating proof search. *GOALDUCK* does not ensure termination too. Indeed, given a database Γ and a goal $x : G$, it could happen that, after several steps, the goal-driven computation leads to search a derivation for the same goal $x : G$ from a database Γ' such that $\Gamma' \supseteq \Gamma$ (that is to say: Γ' either corresponds to Γ or it is obtained from Γ by adding new facts). This problem is also well known in standard logic programming. As an example, consider the database containing the fact $x : (Q \wedge \top) \rightarrow Q$: querying *GOALDUCK* with the goal $x : Q$, one can observe that the computation does not terminate: *GOALDUCK* tries to apply (**U prop**) by using the only clause of the program (database), then it searches a derivation of the two subgoals $x : \top$ (and the computation succeeds) and $x : Q$. In order to prove this goal, *GOALDUCK* repeats the above steps, then incurring in a loop.

6 Conclusions and Future Works

In this paper we have provided goal-directed calculi for normal conditional logics called \mathcal{US}' . These calculi are sound and complete wrt to the semantics, if the language considered is restricted to a specific fragment allowing uniform proofs. Moreover, we have presented *GOALDUCK*, a SICStus Prolog implementation of the calculi \mathcal{US}' . *GOALDUCK* is a simple program, consisting of only seven clauses. Each clause of *GOALDUCK* implements an axiom or rule of the calculi \mathcal{US}' . To the best of our knowledge, no other goal-directed calculi/theorem provers for conditional logics have been previously proposed in the literature. Further investigation could lead to the development of extensions of logic programming based on conditional logics. Related works on proof methods for conditional logics in the literature have concentrated on extensions of CK and do not present implementations of the deductive systems introduced.

De Swart [7] and Gent [14] give sequent/tableau calculi for the strong conditional logics VC and VCS. Their proof systems are based on the entrenchment connective \leq , from which the conditional operator can be defined.

Crocco and Fariñas [5] give sequent calculi for some conditional logics including CK, CEM, CO and others. Their calculi comprise two levels of sequents: principal sequents with \vdash_P correspond to the basic deduction relation, whereas auxiliary sequents with \vdash_a correspond to the conditional operator: thus the constituents of $\Gamma \vdash_P \Delta$ are sequents of the form $X \vdash_a Y$, where X, Y are sets of formulas.

Artosi, Governatori, and Rotolo [2] develop labelled tableau for the *first-degree* fragment (i.e. without nested conditionals) of the conditional logic CU

that corresponds to cumulative non-monotonic logics. In their work they use labels similarly to SeqS'. Formulas are labelled by path of worlds containing also variable worlds.

Lamarre [25] presents tableau systems for the conditional logics V, VN, VC, and VW. Lamarre's method is a consistency-checking procedure which tries to build a system of sphere falsifying the input formulas.

Groeneboer and Delgrande [9] have developed a tableau method for the conditional logic VN which is based on the translation of this logic into the modal logic S4.3.

In [19] and [20] a labelled tableau calculus for the logic CE and some of its extensions is presented. The flat fragment of CE corresponds to the non-monotonic preferential logic P and admits a semantics in terms of preferential structures (possible worlds together with a family of preference relations). The tableau calculus makes use of pseudo-formulas, that are modalities in a hybrid language indexed on worlds.

In [17, 18, 34] tableau calculi for nonmonotonic KLM logics are presented. These calculi, called \mathcal{TS}^T , are obtained by introducing suitable modalities to interpret conditional assertions. Moreover, these calculi have been implemented in SICStus Prolog: the program, called KLMLearn, is inspired by the "lean" methodology and follows the style of CondLean [32, 21].

In future research we aim to develop goal-directed calculi for other conditional logics, in order to extend GOALDUCK to support all of them. We also intend to investigate a broader language allowing uniform proofs.

Finally, we intend to evaluate the performances of GOALDUCK by executing some other tests. As a consequence, in our future research we plan to increase its performances by experimenting standard refinements and heuristics.

References

1. C. E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510–530, 1985.
2. A. Artosi, G. Governatori, and A. Rotolo. Labelled tableaux for non-monotonic reasoning: Cumulative consequence relations. *Journal of Logic and Computation*, 12(6):1027–1060, 2002.
3. B. F. Chellas. Basic conditional logics. *Journal of Philosophical Logic*, 4:133–153, 1975.
4. T. Costello and J. McCarthy. Useful counterfactuals. *ETAI (Electronic Transactions on Artificial Intelligence)*, 3:Section A, 1999.
5. G. Crocco and L. Fariñas del Cerro. Structure, consequence relation and logic, volume 4. In *D. M. Gabbay (ed.), What is a Logical System, Oxford University Press*, pages 239–259, 1995.
6. G. Crocco and P. Lamarre. On the connection between non-monotonic inference systems and conditional logics. In *B. Nebel and E. Sandewall editors, Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference*, pages 565–571, 1992.

7. H. C. M. de Swart. A gentzen-or beth-type system, a practical decision procedure and a constructive completeness proof for the counterfactual logics vc and vcs . *Journal of Symbolic Logic*, 48(1):1–20, 1983.
8. J. P. Delgrande. A first-order conditional logic for prototypical properties. *Artificial Intelligence*, 33(1):105–130, 1987.
9. J. P. Delgrande and C. Groeneboer. A general approach for determining the validity of commonsense assertions using conditional logics. *International Journal of Intelligent Systems*, 5(5):505–520, 1990.
10. N. Friedman and J. Y. Halpern. Plausibility measures and default reasoning. *Journal of the ACM*, 48(4):648–685, 2001.
11. D. M. Gabbay. Labelled deductive systems (vol i). *Oxford Logic Guides*, Oxford University Press, 1996.
12. D. M. Gabbay, L. Giordano, A. Martelli, N. Olivetti, and M. L. Sapino. Conditional reasoning in logic programming. *Journal of Logic Programming*, 44(1-3):37–74, 2000.
13. Dov M. Gabbay and Nicola Olivetti. *Goal-directed Proof Theory*. Kluwer Academic Publishers, 2000.
14. I. P. Gent. A sequent or tableaux-style system for lewis’s counterfactual logic vc . *Notre Dame Journal of Formal Logic*, 33(3):369–382, 1992.
15. L. Giordano, V. Gliozzi, and N. Olivetti. Iterated belief revision and conditional logic. *Studia Logica*, 70(1):23–47, 2002.
16. L. Giordano, V. Gliozzi, and N. Olivetti. Weak agm postulates and strong ramsey test: a logical formalization. *Artificial Intelligence*, 168(1-2):1–37, 2005.
17. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Analytic Tableaux for KLM Preferential and Cumulative Logics. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proceedings of LPAR 2005 (12th Conference on Logic for Programming, Artificial Intelligence, and Reasoning)*, volume 3835 of *LNAI*, pages 666–681, Montego Bay, Jamaica, December 2005. Springer-Verlag.
18. L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato. Analytic Tableaux Calculi for KLM Rational Logic R. In *Proceedings of JELIA 2006*, volume 4160 of *LNAI*, pages 190–202. Springer-Verlag, September 2006.
19. L. Giordano, V. Gliozzi, N. Olivetti, and C. Schwind. Tableau calculi for preference-based conditional logics. In *Proceedings of TABLEAUX 2003 (Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 2796 of *LNAI*, Springer, pages 81–101, 2003.
20. L. Giordano, V. Gliozzi, N. Olivetti, and C.B. Schwind. Extensions of tableau calculi for preference-based conditional logics. In *Proceedings of M4M-4*, pages 220–234. Informatik-Bericht 194, December 2005.
21. L. Giordano, V. Gliozzi, and G. L. Pozzato. KLMLean 2.0: a Theorem Prover for KLM Logics of Nonmonotonic Reasoning. In *Proceedings of TABLEAUX 2007*, volume to appear of *LNAI*. Springer-Verlag, 2007.
22. L. Giordano and C. Schwind. Conditional logic of actions and causation. *Artificial Intelligence*, 157(1-2):239–279, 2004.
23. G. Grahne. Updates and counterfactuals. *Journal of Logic and Computation*, 8(1):87–117, 1998.
24. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
25. P. Lamarre. A tableaux prover for conditional logics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 4th International Conference*, pages 572–580, 1993.

26. D. Lewis. Counterfactuals. *Basil Blackwell Ltd*, 1973.
27. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. In *Annal of Pure and Applied Logic*, 51(1-2):125–157, 1991.
28. D. Nute. Topics in conditional logic. *Reidel, Dordrecht*, 1980.
29. N. Obeid. Model-based diagnosis and conditional logic. *Applied Intelligence*, 14:213–230, 2001.
30. N. Olivetti and G. L. Pozzato. CondLean: A Theorem Prover for Conditional Logics. In *Proceedings of TABLEAUX 2003*, volume 2796 of *LNAI*, pages 264–270. Springer, September 2003.
31. N. Olivetti and G. L. Pozzato. CondLean 3.0: Improving Condlean for Stronger Conditional Logics. In *Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 328–332. Springer-Verlag, September 2005.
32. N. Olivetti and G. L. Pozzato. KLMLean 1.0: a Theorem Prover for Logics of Default Reasoning. In *Proceedings of M4M-4*, pages 235–245. Informatik-Bericht 194, December 2005.
33. N. Olivetti, G. L. Pozzato, and C. B. Schwind. A Sequent Calculus and a Theorem Prover for Standard Conditional Logics. *ACM Transactions on Computational Logics (TOCL)*, to appear, 2007.
34. G. L. Pozzato. *Proof Methods for Conditional and Preferential Logics*. PhD thesis, 2007.
35. C. B. Schwind. Causality in action theories. *Electronic Transactions on Artificial Intelligence (ETAI)*, 3(A):27–50, 1999.
36. R. Stalnaker. A theory of conditionals. In *N. Rescher (ed.), Studies in Logical Theory, American Philosophical Quarterly, Monograph Series no.2, Blackwell, Oxford*, pages 98–112, 1968.
37. L. Viganò. Labelled non-classical logics. *Kluwer Academic Publishers, Dordrecht*, 2000.

Indexing Techniques for the DLV Instantiator

Gelsomina Catalano, Nicola Leone, and Simona Perri

Department of Mathematics, University of Calabria
I-87030 Rende (CS), Italy
{catalano,leone,perri}@mat.unical.it

Abstract. In Answer Set Programming (ASP) systems, the computation consists of two main phases: (1) the input program is first simplified and instantiated, generating a ground (i.e., variable free) program, (2) propositional algorithms are then applied on the ground program to generate the answer sets. The instantiation phase is much more than a simple variables-elimination; it performs the evaluation of relevant programs fragments, producing a ground program which has precisely the same answer sets as the theoretical instantiation, but it is sensibly smaller in size. For instance, if the input program is disjunction-free and stratified, then its evaluation is completely done by the instantiator (there is no second phase at all). The instantiation phase may be computationally expensive; in fact, it has been recognized to be the key issue for solving real-world problems by using Answer Set Programming.

In this paper, we propose to employ main-memory indexing techniques for enhancing the performances of the instantiation procedure of the ASP system **DLV**. In particular, in order to check the impact that indexing may have on the performances of the system, we implemented a first argument indexing strategy for the **DLV** instantiator, and we carried out an experimentation activity on a collection of benchmark problems, including also real-world instances. The results of experiments are very positive and confirm the intuition that indexing can be very useful for improving the efficiency of the instantiation process.

1 Introduction

Answer Set Programming (ASP) is a fully declarative programming style – proposed in the area of logic programming and nonmonotonic reasoning [1–3] – which has been recognized as a promising approach for dealing with problems requiring advanced modeling capabilities for problem representation.

After many years of theoretical research and some years of considerable efforts on developing effective implementations, there are nowadays a number of systems supporting this programming style, like **DLV**[4], **Smodels** [5], **GnT** [6], and **Cmodels** [7], that can be utilized as advanced tools for solving real-world problems in a highly declarative way.

The computation of the answer sets in ASP systems consists of two main phases: the input program is first simplified and instantiated, generating a ground (i.e., variable free) program, and then propositional algorithms are applied on the ground program to generate the answer sets. The instantiation phase is much more than a simple variables-elimination; it performs the evaluation of relevant programs fragments, producing a ground program which has precisely the same answer sets as the theoretical instantiation, but it is sensibly smaller in size. For instance, if the input program is disjunction-free and stratified, then its evaluation is completely

done by the instantiation procedure (also called instantiator) which computes the single answer set without producing any instantiation.

Since the instantiation phase may be computationally expensive, having a good instantiator is crucial for the efficiency of the entire ASP system. The recent application of ASP systems in the areas of Knowledge Management, Security, and Information Integration [8, 9], has made evident the need of improving ASP instantiators.

This paper focuses on the instantiator of **DLV** which is widely recognized to be a very strong point of the **DLV** system. It already incorporates a number of optimization techniques [10–12] but, since ASP applications grow in size, the **DLV** instantiator needs to efficiently handle larger and larger amount of data.

A critical issue for the efficiency of the **DLV** instantiator is the retrieval of ground instances from the extensions of the predicates. Indeed, the size of the extensions may be very large, and thus, in the absence of techniques for speeding-up the retrieval, the time spent in identifying candidate instances can dramatically affect the performances of the instantiator.

In this paper, we face this issue and, in this respect, we propose the use of indexing techniques, that is techniques for the design and the implementation of data structures that allow to efficiently access to large datasets.

Main-memory indexing methods have been already profitably used in the logic programming area [13, 14]. Indeed, effective indexing has become an integral component of high performance declarative programming systems. Almost all the Prolog implementations support indexing on the main functor symbol of the first argument of predicates, and some of them, like XSB [15], SWI-Prolog [16], support more sophisticated indexing schemata.

In this work, in order to assess the impact of indexing on the performances of the **DLV** instantiator, we implemented a kind of first argument indexing strategy for the instantiation procedure of **DLV**, and we carried out an experimentation activity on a collection of benchmark programs. The results of the experiments are very positive; we have obtained non-trivial speed-ups, ranging from a few percent to orders of magnitude, across a wide range of applications. Given these results, we believe that it is of utmost importance the design of a technique for indexing any argument, that, hence, can be exploited also in those case in which first argument indexing is not applicable or the first argument of a predicate is not the most appropriate for indexing.

2 The Language of Answer Set Programming

2.1 Syntax

A variable or a constant is a *term*. An *atom* is $a(t_1, \dots, t_n)$, where a is a *predicate* of arity n and t_1, \dots, t_n are terms. A *literal* is either a *positive literal* p or a *negative literal* $\text{not } p$, where p is an atom.¹ A (*disjunctive*) *rule* r has the following form:

$$a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m, \\ n \geq 1, m \geq k \geq 0$$

where $a_1, \dots, a_n, b_1, \dots, b_m$ are atoms. The disjunction $a_1 \vee \dots \vee a_n$ is the *head* of r , while the conjunction $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$ is the *body* of r .

We denote by $H(r)$ the set $\{a_1, \dots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the

¹ Without loss of generality, in this paper we do not consider strong negation, which is irrelevant for the instantiation process; the symbol ‘*not*’ denotes default negation here.

set of atoms occurring positively (resp., negatively) in $B(r)$. For a literal L , $var(L)$ denotes the set of variables occurring in L . For a conjunction (or a set) of literals C , $var(C)$ denotes the set of variables occurring in the literals in C , and, for a rule r , $var(r) = var(H(r)) \cup var(B(r))$. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r , i.e., $var(r) = var(B^+(r))$.

An *ASP program* \mathcal{P} is a finite set of safe rules. A *not*-free (resp., \vee -free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it.

A predicate occurring only in *facts* (rules of the form $a \leftarrow$), is referred to as an *EDB* predicate, all others as *IDB* predicates. The set of facts in which *EDB* predicates occur, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

2.2 Semantics

Let \mathcal{P} be a program. The *Herbrand Universe* and the *Herbrand Base* of \mathcal{P} are defined in the standard way and denoted by $U_{\mathcal{P}}$ and $B_{\mathcal{P}}$, respectively.

Given a rule r , a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where $\sigma : var(r) \mapsto U_{\mathcal{P}}$ is a substitution mapping the variables occurring in r to constants in $U_{\mathcal{P}}$. We denote by $ground(\mathcal{P})$ the set of all the ground instances of the rules occurring in \mathcal{P} .

An *interpretation* for \mathcal{P} is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal *not* A is *true* w.r.t. I if A is false w.r.t. I ; otherwise *not* A is false w.r.t. I .

Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I .

A *model* for \mathcal{P} is an interpretation M for \mathcal{P} such that every rule $r \in ground(\mathcal{P})$ is true w.r.t. M . A model M for \mathcal{P} is *minimal* if no model N for \mathcal{P} exists such that N is a proper subset of M . The set of all minimal models for \mathcal{P} is denoted by $MM(\mathcal{P})$.

Given a program \mathcal{P} and an interpretation I , the *Gelfond-Lifschitz (GL) transformation* of \mathcal{P} w.r.t. I , denoted \mathcal{P}^I , is the set of positive rules

$$\mathcal{P}^I = \{ a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_k \mid a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_k, \neg b_{k+1}, \dots, \neg b_m \\ \text{is in } ground(\mathcal{P}) \text{ and } b_i \notin I, \text{ for all } k < i \leq m \}$$

Definition 1. [17, 1] Let I be an interpretation for a program \mathcal{P} . I is an *answer set* for \mathcal{P} if $I \in MM(\mathcal{P}^I)$ (i.e., I is a minimal model for the positive program \mathcal{P}^I). \square

3 Instantiation of Answer Set Programs: DLV's Strategy

Using advanced database techniques ([10, 11]) and a clever backjumping algorithm [12], the **DLV** instantiator efficiently generates a ground instantiation of the input that has the same answer sets as the full program instantiation, but is much smaller in general. For example, if the input program is normal and stratified, the instantiator is able to compute the single answer set of the program, namely the set of the facts and the atoms derived by the instantiation procedure.

In this section, we provide a short description of the overall instantiation process of the **DLV** system, and focus on the “heart” procedure of this process which produces the ground instances of a given rule.

In Figure 1 we have depicted the general structure of the instantiator.

An input program \mathcal{P} is first analyzed by the parser, which also builds the extensional database from the facts in the program, and encodes the rules in the intensional database in a suitable way. Then, a rewriting procedure (see [10]), optimizes the rules in order to get an equivalent program \mathcal{P}' that can be instantiated more efficiently and that can lead to a smaller ground program. At this point, another module of the instantiator computes the dependency graph of \mathcal{P}' , its connected components, and a topological ordering of these components. Finally, \mathcal{P}' is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component, according to the dependency graph. During the instantiation of each component, for the evaluation of recursive rules, an improved version of the generalized semi-naive technique [18] is used.

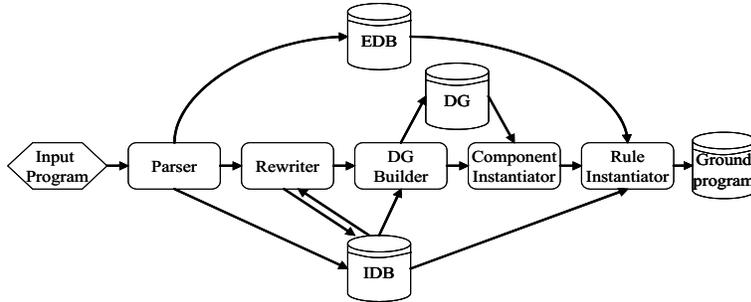


Fig. 1. Architecture of DLV's Instantiator

3.1 Rule Instantiation

In this section, we describe the process of rule instantiation – the heart of the **DLV** instantiator.

The algorithm *Instantiate*, shown in Figure 2, generates the possible instantiations for a rule r of a program \mathcal{P} . When this procedure is called, for each predicate p occurring in the body of r we are given its extension, as a set I_p containing all its ground instances. We say that the mapping $\theta : var(r) \rightarrow U_{\mathcal{P}}$ is a valid substitution for r if it is valid for every literal occurring in its body, i.e., if for every positive literal L (resp., negative literal *not* L) in $B(r)$, $\theta L \in I_L$ ² (resp. $\theta L \notin I_L$) holds. *Instantiate* outputs such valid substitutions for r .

Note that, since the rule is safe, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal. For the sake of presentation, we assume that the body is ordered in a way that any negative literal always follows the positive atoms containing its variables. Actually, **DLV** has a specialized module that computes a clever ordering of the body [11] satisfying this assumption.

Instantiate first stores the body literals L_1, \dots, L_n into an ordered list $B = (null, L_1, \dots, L_n, last)$. Then, it starts the computation of the substitutions for r . To this end, it maintains a variable θ , initially set to \emptyset , representing, at each step, a partial substitution for $var(r)$.

Now, the computation proceeds as follows: For each literal L_i , we denote by $PreviousVars(L_i)$ the set of variables occurring in any literal that precedes L_i in

² Meaning the extension of the predicate occurring in L .

Algorithm *Instantiate*

Input R : Rule, I_{L_1}, \dots, I_{L_n} : SetOfInstances;
Output S : SetOfTotalSubstitutions;
var L : Literal, B : ListOfAtoms, θ : Substitution;
begin
 $\theta = \emptyset$;
 (* return the ordered list of the body literals ($null, L_1, \dots, L_n, last$) *)
 $B := \text{BodyToList}(R)$;
 $L := L_1$; $S := \emptyset$;
while $L \neq null$
if $\text{Match}(L, I_L, \theta)$ **then**
if ($L \neq last$) **then**
 $L := \text{NextLiteral}(L)$;
else (* θ is a total substitution for the variables of R *)
 $S := S \cup \{\theta\}$;
 $L := \text{PreviousLiteral}(L)$; (* look for another solution *)
 $\theta := \theta \upharpoonright_{\text{PreviousVars}(L)}$;
else
 $L := \text{PreviousLiteral}(L)$;
 $\theta := \theta \upharpoonright_{\text{PreviousVars}(L)}$;
output S ;
end;

Function $\text{Match}(L:\text{Literal}, I_L:\text{SetOfInstances}, \text{var } \theta:\text{Substitution}):\text{Boolean}$
begin
 (* take a ground instance G from I_L , if any *)
while $\text{getInstance}(I_L, G)$
 (* if G is consistent with the current substitution θ , extend θ and exit *)
if $\text{extend}(G, \theta)$ **then**
return true;
return false;
end;

Fig. 2. Computing the instantiations of a rule

the list B , and by $\text{FreeVars}(L_i)$ the set of variables that occurs for the first time in L_i , i.e., $\text{FreeVars}(L_i) = \text{var}(L_i) - \text{PreviousVars}(L_i)$.

At each iteration of the **while** loop, by using function Match , we try to find a match for a literal L_i with respect to θ . More precisely, we look in I_{L_i} for a ground instance G which is consistent with the variables in $\text{PreviousVars}(L_i)$, and then use G in order to extend θ to the variables in $\text{FreeVars}(L_i)$. If there is no such a substitution, then we backtrack to the previous literal in the list, or else we consider two cases: if there are further literals to be evaluated, then we continue with the next literal in the list; otherwise, θ encodes a (total) valid substitution and is thus added to the output set S . Even in this case, we backtrack for finding another solution.

Note that, for the sake of clarity, we described a simplified version of the instantiation procedure currently implemented in **DLV**. Indeed, the algorithm *Instantiate* reported above is based on a classical chronological backtracking schema while the actual **DLV** instantiation procedure exploits a more efficient backjumping technique (see, [12]).

4 An Indexing Technique for ASP Programs Instantiation

A critical issue for the efficiency of the **DLV** instantiator is the task accomplished by function *Match* shown in Figure 2. This function takes as input a literal L , its extension I_L and a partial substitution θ and tries to find a ground instance in I_L matching θ .

This task, in the absence of techniques for speeding-up the retrieval of candidate instances, may be very expensive. Indeed, the size of I_L can be very large and thus, a simple approach based on linear search through I_L leads to a drop in performance of the instantiator (also because *Match* is invoked very frequently during the instantiation).

In this Section, we describe a strategy for matching in a more efficient way literals whose first term is already instantiated. That is, this technique allows to efficiently match a literal L whose first term is either a constant or a variable X such that $X \in \text{PreviousVars}(L)$ and thus θ already contains a substitution for X .

The strategy relies on the exploitation of suitable hash index structures that boost the retrieval of ground instances according to values of their first term.

For each predicate p , its extension I_p is implemented by means of a list storing, according to lexicographical order, the ground instances of p . We associate to each extension I_p a hash map representing an index to I_p . More in detail, for each predicate p , let I_p be the extension of p and $C \subseteq U_{\mathcal{P}}$ be the set of all the distinct constants appearing as first argument of some instance in I_p . An index to I_p is a hash map M_p that associates to each $c \in C$ (the key of the map) a pointer pt to I_p . In particular, pt identifies the first ground instance in I_p having c as first argument and thus, due to the lexicographical order of I_p , facilitates the retrieval of all ground instances in I_p with the same characteristic.

By using these structures, the match of a literal L whose first term is instantiated with a constant c can be performed as follows: first of all, we access to the index corresponding to L using c as key. Then, we simply follow the pointer associated to c in order to retrieve all the ground instances having c as first term and try to extend θ by using, one after the other, such instances. Note that, since the index is implemented by means of a hash map, looking up the pointer pt by its key is efficient. In particular, the average case complexity of this operation is constant time.

For the implementation we used a STL `hash_map`[19] and we simply needed to provide it with a hash function. Since in **DLV** all the constants (both numbers and string) are properly stored in a unique data structure, we could choose as hash function the one that assigns to each constant the respective index in the structure. This made the function both efficient to be computed and almost free from collisions. Moreover, the size of the map is also properly limited because of the way the STLs implement it[19].

We remark that, in the original implementation of **DLV** there is only one case in which a literal is instantiated quite efficiently, i.e. when $\text{FreeVars}=\emptyset$ and thus θL is a ground atom. Indeed, in this case, its match simply requires to look for θL in the extension I_L and, since I_L is lexicographically ordered, this can be performed by a binary search. However, by means of the indexes, we can get speed improvements also in this particular case, because indexes allow us to perform binary search on a subset of I_L .

Note that, at present, **DLV** does not support indexes for recursive predicates. This is due to the fact that at the time of the evaluation of such predicates, their extensions are not completely determined, rather, they are continuously updated. Consequently, keeping an index to the extension of a recursive predicate, would

require that every time a new instance is added to the extension, a new entry may be added also to its associated index. Thus, intuitively, if on the one hand indexes speed-up retrievals, on the other hand, they slow-down updates. We are currently pondering about the possibility of using indexes also for recursive predicates, and in particular we are evaluating, with the help of an experimental analysis, the measure in which slowdowns overshadow speedups.

As already stated in the previous Section, the **DLV** instantiator is endowed with a specialized module that, before the instantiation of a rule, computes a clever ordering of the body [11]. Roughly, the reordering algorithm works as follows: at each step $i > 1$, we have already established which are the literals that have to be placed in the first $i - 1$ positions in the final ordering of the body, and we make a greedy choice to select the literal that has to be placed in the i -th position. This literal, say it L , is chosen if it is minimal with respect to a selectivity criterion. In particular, the selectivity criterion that we exploit combines two factors: one is a measure of how much the choice of L reduces the search space for possible substitutions and the other takes into account the binding of the variables of L (by preferring literals with already bound variables, we may quickly detect possible inconsistencies). Obviously, for maximizing the exploitation of this indexing, we could modify the reordering criterion in order to maximize the number of literals placed so that their first argument is bound. However, this may considerably affect the overall ordering of the body, and since the instantiation time of a rule strongly depends on the order of evaluation of literals, it is not clear if implementing this change can be really convenient. Perhaps, this could be discovered by means of a proper experimental analysis, but we believe that major investigation should be preferably directed at the development of a technique for indexing any argument, rather than at allowing a wider usage of the first argument indexing. Indeed, there are many cases in which, whatever the reordering criterion, the first argument indexing technique is not applicable (for instance, when the first argument of a literal is a variable occurring solely in it) or is not effective (for instance, when the number of different constants appearing as first argument of the ground instances of a literal L is small w.r.t the size of the extension of L).

5 Experimental Results

5.1 Benchmark Programs

In order to check the impact of the indexing technique on the **DLV** instantiator, we carried out an experimentation activity on a collection of benchmark problems, taken from different domains. For space limitation, we do not include the code of benchmark programs; however they can be retrieved, together with the binaries used for the experiments, from our web page: <http://www.mat.unical.it/indexing/indexing.tar.gz>. Moreover, we give below a very short description of the problems. In particular, the first three ones have been taken from Asparagus, the environment for benchmarking of ASP systems [20], and we considered, for each of them, three different instances. GrammarBasedIE and Reachability have also been used for *The First Answer Set Programming System Competition*.

GrammarBasedIE The original name is *GrammarBasedInformationExtraction*. It is a problem of information extraction from unstructured documents. The instances we have used are in_001.asp, in_020.asp, in_082.asp.

Reachability The classical reachability problem. Instances: AL_12_2_E_I (a tree with 12 levels and 2 children), AL_14_2_E_I (14 levels and 2 children) and AL_15_2_E_I

Program	DLV	DLV+Indexing
GrammarBasedIE_1	12.04	9.10
GrammarBasedIE_2	6.35	4.88
GrammarBasedIE_3	10.35	7.26
Reachability_12	14.99	2.60
Reachability_14	331.16	38.76
Reachability_15	–	165.89
Samegen_40	21.43	0.36
Samegen_50	66.41	0.75
Samegen_60	165.64	1.33
Scheduling	60.98	18.74
Cristal	3.75	2.23
3COL-simplex	–	23.25
3COL-ladder	871.42	3.20
K-Decomp	6.37	6.07
Ramsey(3,7) \neq 23	20.94	18.56
Timetabling_U	9.58	4.66
Timetabling_S	36.62	35.56
InsuranceWorkflow1	72.60	58.65
InsuranceWorkflow2	169.40	60.25
DocClass	–	14.06
DataIntegration	7.03	0.22

Table 1. Execution Times

(15 levels and 2 children).

Samegen Given the parent relationship, find all pairs of persons belonging to the same generation. Instances: 40 generations, 50 generations and 60 generations.

Scheduling A scheduling program for determining shift rotation of employees.

Cristal A deductive databases application that involves complex knowledge manipulations on databases, developed at CERN in Switzerland [21].

3COL-simplex The classical 3-colorability problem on a simplex graph with 540900 edges and 180901 nodes.

3COL-ladder 3-colorability on a ladder graph with 59848 edges and 79800 nodes.

K-Decomp Decide if exists a hypertree decomposition [22] of a conjunctive query having width $\leq K$.

Ramsey(3,7) \neq 23 Prove that 23 is not the Ramsey number Ramsey(3,7) [23].

Timetabling_U A timetable problem for the first year of the faculty of Science of the University of Calabria.

Timetabling_S An instance of the high-school timetabling problem.

InsuranceWorkflow An ASP program emulating the execution of a workflow in which each step is a transformation to be applied to some insurance data (in order to query for and/or extract implicit knowledge). We considered the encoding of two different steps. Problems provided by the company EXEURA s.r.l. [24].

DocClass A problem of document classification provided by EXEURA s.r.l. [24].

DataIntegration Given some tables containing discording data, find a repair where some key constraints are satisfied. The problem was originally defined within the EU project INFOMIX [8].

5.2 Results

We implemented the indexing technique described in Section 4 in the Instantiator module of **DLV** and we compared our prototype with the official **DLV** release (July 14th 2006) [25] by using the above benchmark problems. All binaries were produced by using the GNU compiler GCC 4.1.2, and the experiments were performed on a Intel CORE SOLO 1.8 GHz with 512 Mbytes of main memory.

Table 1 shows the results of our tests. Each test has been repeated three times in order to obtain more trustworthy information. For each benchmark program P described in column 1, columns 2 and 3 report the times employed to instantiate P by using the standard **DLV** instantiator, and the instantiator **DLV+Indexing**, respectively. All running times are expressed in seconds. The symbol ‘-’ means that the instantiator did not terminate within 30 minutes.

The results confirm the intuition that the indexing technique can be very useful for improving the efficiency of the **DLV** instantiator. Indeed, it is clear from the table that, in general, **DLV+Indexing** outperforms the standard **DLV** instantiator. In particular, there are some cases in which the speed-up is really impressive. Consider, for instance, `DocClass`. Here, the standard **DLV** instantiator does not terminate within 30 minutes, while **DLV+Indexing** takes 14.06s. A similar behavior can be noted in `3COL-simplex` and `Reachability_15`. We have notable improvements also for `Reachability_12`, `Reachability_14`, `Samegen_40`, `Scheduling`, and `DataIntegration`, where times differ by one order of magnitude, for `Samegen_50` and `Samegen_60` where the difference is of two orders of magnitude and for `3COL-ladder` where the difference is even of three orders of magnitude. However, also the other benchmarks show some improvements even if not so high. Clearly, the speed-up reflects how intensively indexing is used for a given benchmark. For instance, `K-Decomp` and `Timetabling_S` exhibit only a little performance gain because their encodings are such that first term indexing is exploitable only for the match of few literals. For this kind of programs, **DLV** could take advantage of more sophisticated indexing techniques allowing the indexing of arguments other than the first.

6 Conclusions And Ongoing Work

In this paper, we present a first attempt in the direction of exploiting main-memory indexing strategies for the instantiation of ASP programs. In particular, we implemented in **DLV** a first argument indexing technique and performed several experiments in order to check the impact of the indexes on the instantiator of **DLV**.

The results of the experiments are very positive and encourage further investigation in this respect. Currently, we are pondering about the possibility to use indexes also for recursive predicates. Moreover, we are working on the design of a technique, specialized for the **DLV** instantiator, allowing the indexing of any argument, and thus exploitable also in those case in which first argument indexing is not applicable or the first argument of a predicate is not the most appropriate for indexing.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. *ACM Transactions on Database Systems* **22**(3) (1997) 364–418
3. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers (2000) 79–103
4. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* **7**(3) (2006) 499–562
5. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A System for Answer Set Programming. In Baral, C., Truszczyński, M., eds.: *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR’2000)*, Breckenridge, Colorado, USA (2000)

6. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). Volume 2923 of Lecture Notes in AI (LNAI), Fort Lauderdale, Florida, USA, Springer (2004) 331–335
7. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
8. 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: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
9. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Non-monotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints (2005)
10. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, ed.: Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDL’99), Prolog Association of Japan (1999) 135–139
11. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In Eiter, T., Faber, W., Truszczyński, M., eds.: Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR’01, Vienna, Austria, September 2001, Proceedings. Volume 2173 of Lecture Notes in AI (LNAI), Springer Verlag (2001)
12. Leone, N., Perri, S., Scarcello, F.: BackJumping Techniques for Rules Instantiation in the DLV System. In: Proceedings of the 10th International Workshop on Nonmonotonic Reasoning (NMR 2004), Whistler, BC, Canada. (2004) 258–266
13. Demoen, B., Mariën, A., Callebaut, A.: Indexing in Prolog. In: Proceedings of the North American Conference on Logic Programming, MIT Press (1989) 1001–1012
14. Carlsson, M.: Freeze, Indexing, and other implementation issues in the WAM. In: Proceedings of the Fourth International Conference on Logic Programming, MIT Press (1987) 40–58
15. Rao, P., Sagonas, K.F., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR’97). Volume 1265 of Lecture Notes in AI (LNAI), Dagstuhl, Germany, Springer Verlag (1997) 2–17
16. Wielemaker, J.: SWI-Prolog 5.1: Reference Manual, University of Amsterdam (1997-2003)
17. Przymusiński, T.C.: Stable Semantics for Disjunctive Programs. *New Generation Computing* **9** (1991) 401–424
18. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
19. Standard Template Library Programmer’s Guide <http://www.sgi.com/tech/stl/>.
20. Asparagus, Homepage <http://asparagus.cs.uni-potsdam.de/>.
21. CRISTAL project, homepage <http://proj-cristal.web.cern.ch/>.
22. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions and Tractable Queries. In: Proceedings of the 18th ACM Symposium on Principles of Database Systems – PODS’99. (1999) 21–32 Full paper in *Journal of Computer and System Sciences*.
23. Radziszowski, S.P.: Small Ramsey Numbers. *The Electronic Journal of Combinatorics* **1** (1999)
24. Exeura s.r.l., Homepage <http://www.exeura.it/>.
25. Faber, W., Pfeifer, G.: DLV homepage (since 1996) <http://www.dlvsystem.com/>.

Combination Methods for Model-Checking of Infinite-State Systems ^{*}

Silvio Ghilardi¹, Enrica Nicolini², Silvio Ranise², and Daniele Zucchelli^{1,2}

¹ Dipartimento di Informatica, Università degli Studi di Milano (Italia)

² LORIA & INRIA-Lorraine, Nancy (France)

Abstract. Manna and Pnueli have extensively shown how a mixture of first-order logic (FOL) and discrete Linear time Temporal Logic (LTL) is sufficient to precisely state verification problems for the class of reactive systems. Theories in FOL model the (possibly infinite) data structures used by a reactive system while LTL specifies its (dynamic) behavior. In this paper, we give a decidability result for the model-checking of safety properties by lifting combination methods for (non-disjoint) theories in FOL. The proof suggests how decision procedures for the constraint satisfiability problem of theories in FOL and the exploration of a safety graph (associated to the system) can be integrated. This paves the way to employ efficient Satisfiability Modulo Theories solvers in the model-checking of infinite state systems. We illustrate our technique on two examples.

1 Introduction

In [10] and many other writings, Manna and Pnueli have extensively shown how a mixture of first-order logic (FOL) and discrete Linear time Temporal Logic (LTL) is sufficient to precisely state verification problems for the class of reactive systems. Theories in FOL model the (possibly infinite) data structures used by a reactive system while LTL specifies its (dynamic) behavior. The combination of LTL and FOL allows one to specify infinite state systems and the subtle ways in which their data flow influences the control flow. Indeed, the capability of automatically solving model-checking problems is of paramount importance for supporting the automation of verification techniques using this framework. Our approach is to reduce this to a first-order combination problem over non-disjoint theories.

Preliminarily, we describe our framework for *integrating LTL operators with theories in FOL* (Section 2.1): we fix a theory T in a first-order signature Σ and consider as a temporal model a sequence $\mathcal{M}_1, \mathcal{M}_2, \dots$ of standard (first-order) models of T and assume such models to share the same carrier (or, equivalently, the domain of the temporal model is ‘constant’). Following [12], we consider

^{*} An extended version of this paper entitled “*Combination Methods for Satisfiability and Model-Checking of Infinite-State Systems*” will appear in the proceedings of CADE 2007.

symbols from a subsignature Σ_r of Σ to be *rigid*, i.e. in a temporal model $\mathcal{M}_1, \mathcal{M}_2, \dots$, the Σ_r -restrictions of the \mathcal{M}_i 's must coincide. The symbols in $\Sigma \setminus \Sigma_r$ are called ‘flexible’ and their interpretation is allowed to change over time (free variables are similarly divided into ‘rigid’ and ‘flexible’). For model-checking, the *initial states* and the *transition relation* are represented by formulae, whose role is to non-deterministically restricting the temporal evolution of the model (Section 3).

In this framework, the safety model-checking problem turns out to be undecidable even if the constraint satisfiability problem for the underlying theory is decidable (this can be obtained by a standard reduction to the reachability problem for Minsky machines [11]). The main *contribution* (Theorem 3.1 in Section 3) of the paper is to identify suitable hypotheses (i.e. T_r -compatibility and local finiteness [5]) to obtain the decidability of the related model-checking problem for quantifier-free safety properties. The proof of this result suggests how decision procedures for the constraint satisfiability problem of theories in FOL and algorithms for exploring a safety graph associated to the system can be integrated. This paves the way to employ efficient *Satisfiability Modulo Theories* (SMT) solvers in the model-checking of infinite state systems, as previous proposals have suggested their use for bounded model-checking [2]. Finally, we illustrate our techniques on two examples (Example 3.1 and 3.2). As this contribution is meant for a presentation-only track, details and proof sketches have been omitted. The reader can find more information in [6] and full technical details in [7].

2 Background

We assume the usual syntactic and semantic model-theoretic notions of first-order logic (see [7]). A Σ -theory T is a set of sentences in the signature Σ ; the sentences in T are also called *axioms*. A theory is *universal* iff it has universal closures of open formulas as axioms. We also assume the usual first-order notions of interpretation, satisfiability, validity, and logical consequence. The equality symbol ‘=’ is interpreted as the identity. 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 Σ -structure \mathcal{M} is a *model* of a Σ -theory T (in symbols, $\mathcal{M} \models T$) iff all the sentences of T are true in \mathcal{M} . A Σ -theory T admits *elimination of quantifiers* iff for every formula $\varphi(\underline{x})$ there is a quantifier-free formula $\varphi'(\underline{x})$ such that $T \models \varphi(\underline{x}) \leftrightarrow \varphi'(\underline{x})$. Standard versions of Linear Arithmetics, Real Arithmetics, acyclic lists, and any theory axiomatizing enumerated datatypes admit elimination of quantifiers. The (*constraint*) *satisfiability problem* for the theory T is the problem of deciding whether a Σ -sentence (Σ -constraint, resp.) is satisfiable in a model of T . We will use free constants instead of variables in constraint satisfiability problems, so that we (equivalently) redefine a constraint satisfiability problem for the theory T as the problem of *establishing the satisfiability of $T \cup \Gamma$* (or, equivalently, the T -satisfiability of Γ) *for a finite set Γ of ground Σ^a -literals*

(where $\Sigma^{\underline{a}} := \Sigma \cup \{\underline{a}\}$, for a finite set of new constants \underline{a}). For the same reason, from now on, by a ‘ Σ -constraint’ we mean a ‘ground $\Sigma^{\underline{a}}$ -constraint’, where the free constants \underline{a} should be clear from the context.

A Σ -embedding (or, simply, an embedding) between two Σ -structures $\mathcal{M} = (M, \mathcal{I})$ and $\mathcal{N} = (N, \mathcal{J})$ is any mapping $\mu : M \rightarrow N$ among the corresponding support sets satisfying the condition

$$(*) \quad \mathcal{M} \models \varphi \text{ iff } \mathcal{N} \models \varphi,$$

for all Σ^M -atoms φ (here \mathcal{M} is regarded as a Σ^M -structure, by interpreting each additional constant $a \in M$ into itself and \mathcal{N} is regarded as a Σ^M -structure by interpreting each additional constant $a \in M$ into $\mu(a)$).

The T_0 -compatibility notion is crucial for the completeness of combination schemas [5].

Definition 2.1 (T_0 -compatibility [5]). *Let T be a theory in the signature Σ and 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 (1) $T_0 \subseteq T_0^*$; (2) T_0^* has quantifier elimination; (3) every model of T_0 can be embedded into a model of T_0^* ; and (4) every model of T can be embedded into a model of $T \cup T_0^*$.*

Local finiteness yields termination of combination schemas [5].

Definition 2.2 (Local Finiteness [5]). *A Σ_0 -theory T_0 is locally finite iff Σ_0 is finite and, for every finite set of free constants \underline{a} , there are finitely many ground $\Sigma_0^{\underline{a}}$ -terms $t_1, \dots, t_{k_{\underline{a}}}$ such that for every further ground $\Sigma_0^{\underline{a}}$ -term u , we have that $T_0 \models u = t_i$ (for some $i \in \{1, \dots, k_{\underline{a}}\}$). If such $t_1, \dots, t_{k_{\underline{a}}}$ are effectively computable from \underline{a} (and t_i is computable from u), then T_0 is effectively locally finite.*

If T_0 is effectively locally finite, for any finite set of free constants \underline{a} it is possible to compute finitely many $\Sigma_0^{\underline{a}}$ -atoms $\psi_1(\underline{a}), \dots, \psi_m(\underline{a})$ such that for any $\Sigma_0^{\underline{a}}$ -atom $\psi(\underline{a})$, there is some i such that $T_0 \models \psi_i(\underline{a}) \leftrightarrow \psi(\underline{a})$. These atoms $\psi_1(\underline{a}), \dots, \psi_m(\underline{a})$ are the *representatives* (modulo T_0 -equivalence) and they can replace arbitrary $\Sigma_0^{\underline{a}}$ -atoms for computational purposes. For example, any theory in a purely relational signature is locally finite.

2.1 Temporal Logic

We assume the standard syntactic and semantic notions concerning Propositional LTL (PLTL), such as PLTL-formula and PLTL-Kripke model. Following [10], we fix a first-order signature Σ and we consider formulae obtained by applying temporal and Boolean operators (but no quantifiers) to first-order Σ -formulae.

Definition 2.3 (LTL($\Sigma^{\underline{a}}$)-Sentences). *Let Σ be a signature and \underline{a} be a (possibly infinite) set of free constants. The set of LTL($\Sigma^{\underline{a}}$)-sentences is inductively defined as follows: (i) if φ is a first-order $\Sigma^{\underline{a}}$ -sentence, then φ is an LTL($\Sigma^{\underline{a}}$)-sentence and (ii) if ψ_1, ψ_2 are LTL($\Sigma^{\underline{a}}$)-sentences, so are $\psi_1 \wedge \psi_2$, $\neg\psi_1$, $X\psi_1$, $\psi_1 U \psi_2$.*

We abbreviate $\neg(\neg\psi_1 \wedge \neg\psi_2)$, $\top U\psi$, $\neg\Diamond\neg\psi$ as $\psi_1 \vee \psi_2$, $\Diamond\psi$ and $\Box\psi$, respectively. Notice that free constants are allowed in the definition of a $LTL(\Sigma^a)$ -sentence.

Definition 2.4. *Given a signature Σ and a set \underline{a} of free constants, an $LTL(\Sigma^a)$ -structure (or simply a structure) is a sequence $\mathcal{M} = \{\mathcal{M}_n = (M, \mathcal{I}_n)\}_{n \in \mathbb{N}}$ of Σ^a -structures. The set M is called the domain (or the universe) and \mathcal{I}_n is called the n -th level interpretation function of the $LTL(\Sigma^a)$ -structure.*

So, an $LTL(\Sigma^a)$ -structure is a family of Σ^a -structures indexed over the naturals. When considering a background Σ -theory T , these structures will also be models of T .

Definition 2.5. *Given an $LTL(\Sigma^a)$ -sentence φ and $t \in \mathbb{N}$, the notion of “ φ being true in the $LTL(\Sigma^a)$ -structure $\mathcal{M} = \{\mathcal{M}_n = (M, \mathcal{I}_n)\}_{n \in \mathbb{N}}$ at the instant t ” (in symbols $\mathcal{M} \models_t \varphi$) is inductively defined as follows:*

- if φ is a first-order Σ^a -sentence, $\mathcal{M} \models_t \varphi$ iff $\mathcal{M}_t \models \varphi$;
- $\mathcal{M} \models_t \neg\varphi$ iff $\mathcal{M} \not\models_t \varphi$;
- $\mathcal{M} \models_t \varphi \wedge \psi$ iff $\mathcal{M} \models_t \varphi$ and $\mathcal{M} \models_t \psi$;
- $\mathcal{M} \models_t X\varphi$ iff $\mathcal{M} \models_{t+1} \varphi$;
- $\mathcal{M} \models_t \varphi U \psi$ iff there exists $t' \geq t$ such that $\mathcal{M} \models_{t'} \psi$ and for each t'' , $t \leq t'' < t' \Rightarrow \mathcal{M} \models_{t''} \varphi$.

We say that φ is true in \mathcal{M} or, equivalently, that \mathcal{M} satisfies φ (in symbols $\mathcal{M} \models \varphi$) iff $\mathcal{M} \models_0 \varphi$.

Below, following [12], we partition the signature into a *rigid* (i.e. time independent) and a *flexible* (i.e. time dependent) part. There are various reasons supporting this choice. The most important is that our framework allows us more flexibility in solving certain problems: actions from the environment on a reactive systems are somewhat unpredictable and can be better modelled by flexible function symbols, as demonstrated by the following Example.

Example 2.1. Suppose we want to model a water level controller. To this aim, we need two functions symbols *in(flow)/out(flow)* expressing the water level variations induced by the environment and by the opening action of the valve, respectively: these functions depend both on the current water level and on the time instant, thus the natural choice is to model them by just *unary* function symbols, which are then *flexible* because the time dependency becomes in this way implicit. On the other hand, the constants expressing the alarm and the overflow level should not depend on the time instant, hence they are modeled as *rigid* constants; for obvious reasons, the arithmetical binary comparison symbol $<$ is also time-independent, hence *rigid* too. Having chosen these (flexible and rigid) symbols, we can express constraints on the behavior of our system by introducing a suitable theory (see Example 3.1 below for details).

There is also a more technical (but still crucial) reason underlying our distinction between rigid and flexible symbols: we can avoid some undecidability problems by carefully choosing problematic function or predicates to be flexible. In fact, if we succeed to keep the rigid part relatively simple (e.g., a locally finite theory), then we usually do not lose decidability.

Definition 2.6. An LTL-theory is a 5-tuple $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ where Σ is a signature, T is a Σ -theory (called the underlying theory of \mathcal{T}), Σ_r is a subsignature of Σ , and $\underline{a}, \underline{c}$ are sets of free constants.

Σ_r is the *rigid subsignature* of the LTL-theory; the constants \underline{c} will be rigidly interpreted, whereas the constants \underline{a} will be interpreted in a time-dependent way. The constants \underline{a} are also (improperly) called the *system variables* of the LTL-theory, and the constants \underline{c} are called its *system parameters*. The equality symbol will always be considered as rigid. An LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ is *totally flexible* iff Σ_r is empty and is *totally rigid* iff $\Sigma_r = \Sigma$. For the semantic side, we introduce the following

Definition 2.7. An LTL($\Sigma^{\underline{a}, \underline{c}}$)-structure $\mathcal{M} = \{\mathcal{M}_n = (M, \mathcal{I}_n)\}_{n \in \mathbb{N}}$ is appropriate for an LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ iff we have

$$\mathcal{M}_n \models T, \quad \mathcal{I}_n(f) = \mathcal{I}_m(f), \quad \mathcal{I}_n(P) = \mathcal{I}_m(P), \quad \mathcal{I}_n(c) = \mathcal{I}_m(c).$$

for all $m, n \in \mathbb{N}$, for each function symbol $f \in \Sigma_r$, for each relational symbol $P \in \Sigma_r$, and for all constant $c \in \underline{c}$.

Because of completeness issues, the following class of *locally finite compatible* LTL-theories is introduced:

Definition 2.8. An LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ is *locally finite compatible* iff there is a universal and effectively locally finite Σ_r -theory T_r such that T is T_r -compatible and the constraint satisfiability problem for T is decidable.

3 The Model-Checking Problem

Given two signatures Σ_r and Σ such that $\Sigma_r \subseteq \Sigma$, we define the *one-step signature* as $\Sigma \oplus_{\Sigma_r} \Sigma := ((\Sigma \setminus \Sigma_r) \uplus (\Sigma \setminus \Sigma_r)) \cup \Sigma_r$, where \uplus denotes disjoint union. In order to build the one-step signature $\Sigma \oplus_{\Sigma_r} \Sigma$, we first consider two copies of the symbols in $\Sigma \setminus \Sigma_r$; the two copies of $r \in \Sigma \setminus \Sigma_r$ are denoted by r^0 and r^1 , respectively. Notice that the symbols in Σ_r are not renamed. Also, arities in the one-step signature $\Sigma \oplus_{\Sigma_r} \Sigma$ are defined in the obvious way: the arities of the symbols in Σ_r are unchanged and if n is the arity of $r \in \Sigma \setminus \Sigma_r$, then n is the arity of both r^0 and r^1 . The one-step signature $\Sigma \oplus_{\Sigma_r} \Sigma$ will be also written as $\bigoplus_{\Sigma_r}^2 \Sigma$; similarly, we can define the n -step signature $\bigoplus_{\Sigma_r}^{n+1} \Sigma$ for $n > 1$ (our notation for the copies of $(\Sigma \setminus \Sigma_r)$ -symbols extends in the obvious way, that is we denote by r^0, r^1, \dots, r^n the $n + 1$ copies of r).

Definition 3.1. Given two signatures Σ_r and Σ such that $\Sigma_r \subseteq \Sigma$, two Σ -structures $\mathcal{M}_0 = \langle M, \mathcal{I}_0 \rangle$ and $\mathcal{M}_1 = \langle M, \mathcal{I}_1 \rangle$ whose Σ_r -reducts are the same, the one-step $(\Sigma \oplus_{\Sigma_r} \Sigma)$ -structure $\mathcal{M}_0 \oplus_{\Sigma_r} \mathcal{M}_1 = \langle M, \mathcal{I}_0 \oplus_{\Sigma_r} \mathcal{I}_1 \rangle$ is defined as follows: (a) for each function or predicate symbol $s \in \Sigma \setminus \Sigma_r$, $(\mathcal{I}_0 \oplus_{\Sigma_r} \mathcal{I}_1)(s^0) := \mathcal{I}_0(s)$ and $(\mathcal{I}_0 \oplus_{\Sigma_r} \mathcal{I}_1)(s^1) := \mathcal{I}_1(s)$ and (b) for each function or predicate symbol $r \in \Sigma_r$, $(\mathcal{I}_0 \oplus_{\Sigma_r} \mathcal{I}_1)(r) := \mathcal{I}_0(r)$.

If φ is a Σ -formula, the $\Sigma \oplus_{\Sigma_r} \Sigma$ formulae φ^0, φ^1 are obtained from φ by replacing each symbol $r \in \Sigma \setminus \Sigma_r$ by r^0 and r^1 , respectively. The one-step theory $T \oplus_{\Sigma_r} T$ is taken to be the combination of the theory T with a partially renamed copy of itself: Given two signatures Σ_r and Σ such that $\Sigma_r \subseteq \Sigma$, the $(\Sigma \oplus_{\Sigma_r} \Sigma)$ -theory $T \oplus_{\Sigma_r} T$ is defined as $\{\varphi^0 \wedge \varphi^1 \mid \varphi \in T\}$. We will write $\bigoplus_{\Sigma_r}^2 T$ instead of $T \oplus_{\Sigma_r} T$; the n -step theories $\bigoplus_{\Sigma_r}^{n+1} T$ (for $n > 1$) are similarly defined.

Let now $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ be an LTL-theory with *finitely* many parameters and system variables. A *transition relation* for the LTL-theory \mathcal{T} is a $(\Sigma^{\underline{a}, \underline{c}} \oplus_{\Sigma_r^c} \Sigma^{\underline{a}, \underline{c}})$ -sentence δ : we write such formula as $\delta(\underline{a}^0, \underline{a}^1)$ to emphasize that it contains the two copies of the system variables \underline{a} (on the other hand, the system parameters \underline{c} are not duplicated and will never be displayed). An *initial state description* for the LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ is simply a $\Sigma^{\underline{a}, \underline{c}}$ -sentence $\iota(\underline{a})$ (again, the system parameters \underline{c} will not be displayed).

Definition 3.2 (LTL-System Specification and Model-Checking). *An LTL-system specification is an LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ (with finitely many system variables and parameters) endowed with a transition relation $\delta(\underline{a}^0, \underline{a}^1)$ and with an initial state description $\iota(\underline{a})$. An LTL($\Sigma^{\underline{a}, \underline{c}}$)-structure $\mathcal{M} = \{\mathcal{M}_n = (M, \mathcal{I}_n)\}_{n \in \mathbb{N}}$ is a run for such an LTL-system specification iff it is appropriate for \mathcal{T} and moreover it obeys the initial state description ι and the transition δ , i.e. (1) $\mathcal{M}_0 \models \iota(\underline{a})$, and (2) $\mathcal{M}_n \oplus_{\Sigma_r^c} \mathcal{M}_{n+1} \models \delta(\underline{a}^0, \underline{a}^1)$, for every $n \geq 0$. The model-checking problem for the system specification $(\mathcal{T}, \delta, \iota)$ is the following: given an LTL($\Sigma^{\underline{a}, \underline{c}}$)-sentence φ , decide whether there is a run \mathcal{M} for $(\mathcal{T}, \delta, \iota)$ such that $\mathcal{M} \models \varphi$.³ The ground model-checking problem for $(\mathcal{T}, \delta, \iota)$ is similarly defined for a ground φ .*

The (syntactic) *safety model-checking problem* is the model-checking problem for formulae of the form $\Diamond v$, where v is a $\Sigma^{\underline{a}, \underline{c}}$ -sentence. Since v is intended to describe the set of *unsafe* states, we say that the system specification $(\mathcal{T}, \delta, \iota)$ is *safe for v* iff the model-checking problem for $\Diamond v$ has a negative solution. This implies that $\Box \neg v$ is true for all runs of $(\mathcal{T}, \delta, \iota)$.

In the literature about model-checking (especially, for finite-state systems), it is usually assumed the seriality of the transition relation: every state of the system must have at least one successor state (see, e.g., [1] for more details).

Definition 3.3. *An LTL-system specification $(\mathcal{T}, \delta, \iota)$, based on the LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$, is said to be serial iff for every $\Sigma^{\underline{a}, \underline{c}}$ -structure $\mathcal{M}_0 = (M, \mathcal{I}_0)$ which is a model of T , there is another $\Sigma^{\underline{a}, \underline{c}}$ -structure $\mathcal{M}_1 = (M, \mathcal{I}_1)$ (still a model of T) such that $(\mathcal{M}_0)_{|\Sigma_r} = (\mathcal{M}_1)_{|\Sigma_r}$ and $\mathcal{M}_0 \oplus_{\Sigma_r^c} \mathcal{M}_1 \models \delta(\underline{a}^0, \underline{a}^1)$.*

Although the notion of seriality defined above is non-effective, there exist simple and effective conditions ensuring it. For example, if the transition relation δ consists of the conjunction of (possibly guarded) assignments of the form $P(\underline{a}^0) \rightarrow a^1 = t^0(\underline{a}^0)$ where $P(\underline{a}^0)$ is the condition under which the assignment is executed, then δ is serial. The standard trick [1] of ensuring seriality by a 0-ary predicate describing error states works in our framework too.

³ In the literature, the model-checking problem is the complement of ours, i.e. it is the problem of deciding whether a given sentence is true in all runs.

Definition 3.4. An LTL-system specification $(\mathcal{T}, \delta, \iota)$, based on the LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$, is locally finite compatible iff \mathcal{T} is.

In the rest of this paper, besides assuming the decidability of the constraint satisfiability problem of the theory underlying \mathcal{T} , we require that any LTL-system specification $(\mathcal{T}, \delta, \iota)$ is serial and both ι and δ are ground. Unfortunately, these hypotheses are not sufficient to guarantee decidability. In fact, it is possible to reduce the ground safety model-checking problem to the reachability problem of Minsky machines, which is known to be undecidable (see, e.g., [7]).

Fortunately, the safety model-checking problem is decidable for locally finite compatible LTL-system specifications. In the rest of this Section, let $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ be a locally finite compatible LTL-theory, $(\mathcal{T}, \delta, \iota)$ be an LTL-system specification based on \mathcal{T} , and $v(\underline{a})$ be a ground $\Sigma^{\underline{a}, \underline{c}}$ -sentence. The related safety model-checking problem amounts to checking whether there exists a run $\mathcal{M} = \{\mathcal{M}_n\}_{n \in \mathbb{N}}$ for $(\mathcal{T}, \delta, \iota)$ such that $\mathcal{M} \models_n v(\underline{a})$ for some $n \geq 0$: if this is the case, we say that the system is *unsafe* since there is a *bad run of length n*.

We can ignore bad runs of length $n = 0$, because the existence of such runs can be preliminarily decided by checking the ground sentence $\iota(\underline{a}) \wedge v(\underline{a})$ for T -satisfiability. So, for $n \geq 1$, taking into account the seriality of the transition, a bad run of length $n + 1$ exists iff the ground $(\bigoplus_{\Sigma_r^c}^{n+2} \Sigma^{\underline{a}, \underline{c}})$ -sentence

$$\iota^0(\underline{a}^0) \wedge \delta^{0,1}(\underline{a}^0, \underline{a}^1) \wedge \delta^{1,2}(\underline{a}^1, \underline{a}^2) \wedge \dots \wedge \delta^{n,n+1}(\underline{a}^n, \underline{a}^{n+1}) \wedge v^{n+1}(\underline{a}^{n+1}) \quad (1)$$

is $\bigoplus_{\Sigma_r^c}^{n+2} T$ -satisfiable, where $\iota^0(\underline{a}^0)$ is obtained by replacing each flexible symbol $r \in \Sigma \setminus \Sigma_r$ with r^0 in $\iota(\underline{a})$ (the system variables \underline{a} are similarly renamed as \underline{a}^0); $\delta^{i,i+1}(\underline{a}^i, \underline{a}^{i+1})$ is obtained by replacing in $\delta(\underline{a}^0, \underline{a}^1)$ the copy r^0 and r^1 of each flexible symbol $r \in \Sigma \setminus \Sigma_r$ with r^i and r^{i+1} respectively (the two copies $\underline{a}^0, \underline{a}^1$ of the system variables \underline{a} are similarly renamed as $\underline{a}^i, \underline{a}^{i+1}$); and $v^{n+1}(\underline{a}^{n+1})$ is obtained by replacing each flexible symbol $r \in \Sigma \setminus \Sigma_r$ with r^{n+1} in $v(\underline{a})$ (the system variables \underline{a} are similarly renamed as \underline{a}^{n+1}). For the sake of simplicity, we will write formula (1) by omitting the superscripts of ι , δ , and v (but we maintain those of the system variables \underline{a}).

Now, for a given $n + 1$, an iterated application of the main combination result in [5] and the fact that T_0 -compatibility is a modular property (see again [5]) yield the decidability of the satisfiability of formula (1). Unfortunately, this is not sufficient to solve the model-checking problem for LTL-system specifications since the length of a bad run is not known apriori. To solve this problem, we reduce the existence of a satisfiable formula of the form (1) to a reachability problem in a safety graph (see Definition 3.7 below).

Definition 3.5. A ground $(\Sigma^{\underline{a}, \underline{c}} \oplus_{\Sigma_r^c} \Sigma^{\underline{a}, \underline{c}})$ -sentence δ is said to be purely left (purely right) iff for each symbol $r \in \Sigma \setminus \Sigma_r$, we have that r^1 (r^0 , resp.) does not occur in δ . We say that δ is pure iff it is a Boolean combination of purely left or purely right atoms.

Given a formula $\delta(\underline{a}^0, \underline{a}^1)$, it is always possible (see, e.g., [5]) to obtain an equisatisfiable formula $\delta(\underline{a}^0, \underline{a}^1, \underline{a}^0)$ which is pure by introducing “fresh” constants

that we call \underline{d}^0 (i.e., $\underline{d}^0 \cap (\underline{a}^0 \cup \underline{a}^1) = \emptyset$) to name “impure” subterms. Usually, $\tilde{\delta}$ is called the purification of δ . Let A_1, \dots, A_k be the atoms occurring in $\tilde{\delta}(\underline{a}^0, \underline{a}^1, \underline{d}^0)$. A $\tilde{\delta}$ -assignment is a conjunction $B_1 \wedge \dots \wedge B_k$ (where B_i is either A_i or $\neg A_i$, for $1 \leq i \leq k$), such that $B_1 \wedge \dots \wedge B_k \rightarrow \tilde{\delta}$ is a propositional tautology. Since $\tilde{\delta}$ is pure, we can represent a $\tilde{\delta}$ -assignment V in the form $V^l(\underline{a}^0, \underline{a}^1, \underline{d}^0) \wedge V^r(\underline{a}^0, \underline{a}^1, \underline{d}^0)$, where V^l is a purely left conjunction of literals and V^r is a purely right conjunction of literals. As a consequence, a bad run of length $n + 1$ exists iff the ground sentence

$$\iota(\underline{a}^0) \wedge \bigwedge_{i=0}^n (V_{i+1}^l(\underline{a}^i, \underline{a}^{i+1}, \underline{d}^i) \wedge V_{i+1}^r(\underline{a}^i, \underline{a}^{i+1}, \underline{d}^i)) \wedge v(\underline{a}^{n+1}) \quad (2)$$

is $\bigoplus_{\Sigma_r}^{n+2} T$ -satisfiable, where $\underline{d}^0, \underline{d}^1, \dots, \underline{d}^n$ are $n + 1$ copies of the fresh constants \underline{d}^0 and V_1, \dots, V_{n+1} range over the set of $\tilde{\delta}$ -assignments. Since T_r is locally finite, there are finitely many ground $\Sigma_r^{\underline{a}^0, \underline{a}^1, \underline{d}^0}$ -literals which are representative (modulo T_r -equivalence) of all $\Sigma_r^{\underline{a}^0, \underline{a}^1, \underline{d}^0}$ -literals.

Definition 3.6 (Guessing). Let Σ be a signature Σ and S be a finite set of Σ -atoms. An S -guessing \mathcal{G} is a Boolean assignment to members of S . We also view \mathcal{G} as the set $\{\varphi \mid \varphi \in S \text{ and } \mathcal{G}(\varphi) \text{ is assigned to true}\} \cup \{\neg\varphi \mid \varphi \in S \text{ and } \mathcal{G}(\varphi) \text{ is assigned to false}\}$.

A guessing $G(\underline{a}^0, \underline{a}^1, \underline{d}^0)$ over the finitely many representatives of the locally finite theory T_r will be called a *transition Σ_r -guessing*.

Definition 3.7. The safety graph associated to the LTL-system specification $(\mathcal{T}, \delta, \iota)$ based on the locally finite compatible LTL-theory \mathcal{T} is the directed graph defined as follows:

- the nodes are the pairs (V, G) where V is a $\tilde{\delta}$ -assignment and G is a transition Σ_r -guessing;
- there is an edge $(V, G) \rightarrow (W, H)$ iff the ground sentence

$$G(\underline{a}^0, \underline{a}^1, \underline{d}^0) \wedge V^r(\underline{a}^0, \underline{a}^1, \underline{d}^0) \wedge W^l(\underline{a}^1, \underline{a}^2, \underline{d}^1) \wedge H(\underline{a}^1, \underline{a}^2, \underline{d}^1) \quad (3)$$

is T -satisfiable.

The initial nodes of the safety graph are the nodes (V, G) such that $\iota(\underline{a}^0) \wedge V^l(\underline{a}^0, \underline{a}^1, \underline{d}^0) \wedge G(\underline{a}^0, \underline{a}^1, \underline{d}^0)$ is T -satisfiable; the terminal nodes of the safety graph are the nodes (V, G) such that $V^r(\underline{a}^0, \underline{a}^1, \underline{d}^0) \wedge v(\underline{a}^1) \wedge G(\underline{a}^0, \underline{a}^1, \underline{d}^0)$ is T -satisfiable.

In (3) we still follow our convention of writing only the system variable renamings (flexible symbols being renamed accordingly). More in detail: we make three copies r^0, r^1, r^2 of every flexible symbol $r \in \Sigma \setminus \Sigma_r$. Both V^r and W^l might contain in principle two copies r^0, r^1 of r : the two copies in V^r keep their original names, whereas the two copies in W^l are renamed as r^1, r^2 , respectively. However, V^r is a right formula (hence it does not contain r^0) and W^l is a left

formula (hence it does not contain r^1): the moral of all this is that only the copy r^1 of r occurs after renaming, which means that (3) is after all just a plain $\Sigma^{\underline{a}^0} \cdot \underline{a}^1 \cdot \underline{a}^2 \cdot \underline{a}^0 \cdot \underline{a}^1$ -sentence (thus, it makes sense to test it for T -satisfiable). Notice that the Skolem constants \underline{d}^0 of V^r are renamed as \underline{d}^1 in W^l .

The decision procedure for safety model-checking relies on the following fact.

Proposition 3.1. *The system is unsafe iff either $\iota(\underline{a}) \wedge v(\underline{a})$ is T -satisfiable or there is a path in the safety graph from an initial to a terminal node.*

The idea behind the proof is the following: by contradiction, assume there is a path from an initial to a terminal node and the system is safe. Repeatedly, compute Σ_r -ground interpolants of (2) between T and $\bigoplus_{\Sigma_r}^j T$ (such interpolants exist by T_r -compatibility), for $j = n + 1, \dots, 1$. This yields the T -unsatisfiability of the final node (formula) in the graph; a contradiction.

Theorem 3.1. *The ground safety model-checking problem for a locally finite compatible LTL-system specification is decidable.*

Example 3.1. Consider the water level controller in [14] where (i) changes in the water level by *in(flow)/out(flow)* depend on the water level l and on the time instant; (ii) if $l \geq l_{\text{alarm}}$ at a given state (where l_{alarm} is a fixed value), then a valve is opened and, at the next observable instant, $l' = \text{in}(\text{out}(l))$; and (iii) if $l < l_{\text{alarm}}$ then the valve is closed and, at the next observable instant, $l' = \text{in}(l)$.

Let us now consider the LTL-theory $\mathcal{T} = \langle \Sigma, T, \Sigma_r, \underline{a}, \underline{c} \rangle$ where l is the only system variable ($\underline{a} := \{l\}$) and there are no system parameters ($\underline{c} := \emptyset$); $\Sigma_r = \{l_{\text{alarm}}, l_{\text{overflow}}, <\}$, $l_{\text{alarm}}, l_{\text{overflow}}$ are two constant symbols and $<$ is a binary predicate symbol; $\Sigma := \Sigma_r \cup \{\text{in}, \text{out}\}$; T_r is the theory of dense linear orders without endpoints endowed with the additional axiom $l_{\text{alarm}} < l_{\text{overflow}}$; and

$$T := T_r \cup \left\{ \begin{array}{l} \forall x (x < l_{\text{alarm}} \rightarrow \text{in}(x) < l_{\text{overflow}}), \\ \forall x (x < l_{\text{overflow}} \rightarrow \text{out}(x) < l_{\text{alarm}}) \end{array} \right\}$$

It can be shown that the constraint satisfiability problem for T is decidable, T_r admits quantifier elimination, and T_r is effectively locally finite. From these, it follows that \mathcal{T} is a locally finitely compatible LTL-theory. We consider now the LTL-system specification $(\mathcal{T}, \delta, \iota)$ where $\iota := l < l_{\text{alarm}}$ and

$$\delta := (l_{\text{alarm}} \leq l^0 \rightarrow l^1 = \text{in}^0(\text{out}^0(l^0))) \wedge (l^0 < l_{\text{alarm}} \rightarrow l^1 = \text{in}^0(l^0)).$$

Notice that δ is a purely left $(\Sigma^{\underline{a}} \oplus_{\Sigma_r} \Sigma^{\underline{a}})$ -formula.

We consider the safety model-checking problem specified by the LTL-system above and whose unsafe states are described by $v := l_{\text{overflow}} < l$. Using the procedure suggested by Theorem 3.1 we can prove that the system is safe, i.e. that there is no run \mathcal{M} for $(\mathcal{T}, \delta, \iota)$ such that $\mathcal{M} \models \diamond v$. We can observe that the task in practice is not extremely hard computationally. It is sufficient to consider just 50 nodes (modulo T -equivalence) of the safety graph that are T -satisfiable (i.e. the nodes (V, G) such that $V \wedge G$ is T -satisfiable). Also, instead

of considering all the edges of the safety graph, it is sufficient to build just the paths starting from the initial nodes or ending in a terminal node (namely to apply a forward/backward search strategy). In the first case, only 26 nodes of the safety graph are reachable from an initial node. In the latter, just 12 nodes are backward reachable from a terminal node. Hence the problem is clearly amenable to automatic analysis by using a decision procedure for T .

Example 3.2. The aim of this example is to use our techniques to analyze the safety of the well-known Lamport’s mutual exclusion “Bakery” algorithm. If the number of involved processes is unknown, we can build for the problem an appropriate LTL-system specification \mathcal{T} which violates our assumptions because it has universal (instead of ground) transition relation and initial state description. More in detail, we use a language with two sorts, one for the individuals (i.e the involved processes), the other for the tickets. The tickets are ruled by the theory of dense total order with named distinct (rigid) endpoints 0 and 1; moreover, a (flexible) function for the ticket assignment is constrained by an “almost-injectivity” axiom (i.e., people cannot have the same ticket with the exception of the ticket 1 that means being out of the queue). Finally, a flexible constant models the current ticket bound and a flexible predicate captures the served individuals. The transition says the following: (i) the values of the current ticket bound are strictly increasing; (ii) every individual is removed from the queue immediately after being served; (iii) if an individual is in the queue and is not served, then its ticket is preserved; (iv) if an individual is not the first in the queue, it cannot be served; (v) if an individual is not in the queue, either remains out of the queue or takes a ticket lying in the interval between two consecutive values of the current ticket bound (without being immediately served). The initial state description says that no one is in the queue and the current ticket bound is set to 0, whereas the unsafe states are the ones in which at least two people are served at the same time.

By Skolemization and instantiation, we produce out of \mathcal{T} a locally finite compatible LTL-system specification \mathcal{T}' which is safe iff \mathcal{T} is safe. Safety of \mathcal{T}' can then be easily checked through our techniques (see [7] for details). We point out that the features of \mathcal{T} that make the whole construction to work are *purely syntactic* in nature: they basically consist of the finiteness of the set of terms of certain sorts in the skolemized Herbrand universe.

4 Related work

The literature on infinite state model-checking is extremely vast. For lack of space, we consider works which are closely related to ours. The paper [3] extensively reviews constrained LTL, which can be the basis for model checking of infinite state systems but it does not allow for flexible symbols (apart from system variables). Other results and techniques from [3] (and also from the recent [4]) should be taken into account for integration in our framework so to be able to handle richer underlying theories such as Linear Arithmetic.

An integration of classic tableaux and automated deduction techniques is presented in [13,9]. While using a similar approach, [13] only provides a uniform framework for such an integration with no guarantee of full automation, while [9] focuses on the decidability of the model-checking problem of particular classes of parametrized systems. Both works do not use combination techniques. The approach in [2] proposes the reduction of bounded model-checking problems to SMT problems. Theorem 3.1 identifies precise conditions under which our reduction yields a decision procedure: our safety graph is not just an approximation of the set of reachable states. With [2], we share the focus on using SMT solvers, which is also a common feature of the “abstract-check-refine” approach to infinite-state model-checking (see the seminal work in [8]).

References

1. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
2. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE 2002*, volume 2392 of *LNCS*, 2002.
3. S. Demri. Linear-time temporal logics with Presburger constraints: An overview. *Journal of Applied Non-Classical Logics*, 16(3-4), 2006.
4. S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen. Towards a model-checker for counter systems. In *Proc. of ATVA 2006*, volume 4218 of *LNCS*, 2006.
5. S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3-4), 2004.
6. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Combination methods for satisfiability and model-checking of infinite-state systems. Accepted for publication in CADE 2007.
7. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Combination Methods for Satisfiability and Model-Checking of Infinite-State Systems. Technical Report RI313-07, U. degli Studi di Milano, 2007. Available at <http://homes.dsi.unimi.it/~zucchelli/publications/techreport/GhiNiRaZu-RI313-07.pdf>.
8. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. of CAV 1997*, volume 1254 of *LNCS*. Springer, 1997.
9. M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In *Proc. of CAV 2001*, volume 2102 of *LNCS*, 2001.
10. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
11. M. L. Minsky. Recursive unsolvability of Post’s problem of “tag” and other topics in the theory of Turing machines. *Annals of Mathematics*, 74(3), 1961.
12. D. A. Plaisted. A decision procedure for combination of propositional temporal logic and other specialized theories. *Journal of Automated Reasoning*, 2(2), 1986.
13. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1), 1999.
14. V. Sofronie-Stokkermans. Interpolation in local theory extensions. In *Proc. of IJCAR 2006*, volume 4130 of *LNCS*, 2006.

Multivalued Action Languages with Constraints in CLP(FD) ^{*}

Agostino Dovier¹, Andrea Formisano², and Enrico Pontelli³

¹ Univ. di Udine, Dip. di Matematica e Informatica. dovier@dimi.uniud.it

² Univ. di Perugia, Dip. di Matematica e Informatica. formis@dipmat.unipg.it

³ New Mexico State University, Dept. Computer Science. epontell@cs.nmsu.edu

Abstract. Action description languages, such as \mathcal{A} and \mathcal{B} [6], are expressive instruments introduced for formalizing planning domains and problems. The paper starts by proposing a methodology to encode an action language (with conditional effects and static causal laws), a slight variation of \mathcal{B} , using *Constraint Logic Programming over Finite Domains*. The approach is then generalized to lift the use of constraints to the level of the action language itself. A prototype implementation has been developed, and the preliminary results are presented and discussed.

1 Introduction

The construction of intelligent agents that can be effective in real-world environments has been a goal of researchers from the very first days of Artificial Intelligence. It has long been recognized that such an agent must be able to *acquire*, *represent*, and *reason* with knowledge. As such, a *reasoning component* has been an inseparable part of most agent architectures in the literature.

Although the underlying representations and implementations may vary between agents, the reasoning component of an agent is often responsible for making decisions that are critical to its existence. Logic programming languages offer many attributes that make them suitable as knowledge representation languages. Their declarative nature allows the modular development of provably correct reasoning modules [2]. Recursive definitions can be easily expressed and reasoned upon. Control knowledge and heuristic information can be declaratively introduced in the reasoning process. Furthermore, many logic programming languages offer a natural support for nonmonotonic reasoning, which is considered essential for commonsense reasoning. These features, along with the presence of efficient solvers [1, 11, 15, 7], make logic programming an attractive paradigm for knowledge representation and reasoning.

In the context of knowledge representation and reasoning, a very important application of logic programming has been in the domain of reasoning about actions and change and planning [2]. Planning problems have been effectively encoded using *Answer Set Programming (ASP)* [2]—where distinct answer sets represent different trajectories leading to the desired goal. Other logic programming paradigms, e.g., *Constraint Logic Programming over Finite Domains (CLP(FD))*, have been used less frequently to handle problems in reasoning about actions (e.g., [14, 17]). Comparably more emphasis

^{*} This paper will appear in Proc. of the 23rd Int. Conference on Logic Programming (ICLP07).

has been placed in encoding planning problems as (non-logic programming) constraint satisfaction problems [10].

Recent 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*, e.g., the languages \mathcal{A} and \mathcal{B} [6]. 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.

The goal of this work is to explore the relevance of constraint solving and constraint logic programming [11, 1] in dealing with action languages and planning. The push towards this exploratory study came from a recent investigation [4, 5] aimed at comparing the practicality and efficiency of answer set programming versus constraint logic programming in solving various combinatorial and optimization problems. The study indicated that CLP offers a valid alternative, especially in terms of efficiency, to ASP when dealing with planning problems; furthermore, CLP offers the flexibility of programmer-developed search strategies and the ability to handle numerical constraints.

The first step, in this paper, is to demonstrate a scheme that directly processes an action description specification, in a language similar to \mathcal{B} , producing a CLP(FD) program that can be used to compute solutions to the planning problem. Our encoding has some similarities to the one presented in [10], although we rely on CLP instead of CSP, and our action language supports static causal laws and non-determinism—while the work of Lopez and Bacchus is restricted to STRIPS-like specifications.

While the first step relies on using constraints to compute solutions to a planning problem, the second step brings the expressive power of constraints to the level of the action language, by allowing multi-valued fluents and constraint-producing actions. The extended action language (named \mathcal{B}_{MV}^{FD}) can be as easily supported by the CLP(FD) framework, and it allows a declarative encoding of problems involving actions with resources, delayed effects, and maintenance goals. These ideas have been developed in a prototype, and some preliminary experiments are reported.

We believe that the use of CLP(FD) can greatly facilitate the transition of declarative extensions of action languages to concrete and effective implementations, overcoming some inherent limitations (e.g., efficiency and limited handling of numbers) of other logic-based systems (e.g., ASP).

2 An Action Language

“Action languages are formal models of parts of the natural language that are used for talking about the effect of actions” [6]. Action languages are used to define *action descriptions* that embed knowledge to formalize planning problems. In this section, we use a variant of the language \mathcal{B} , based on the syntax used in [16]. With a slight abuse of notation, we simply refer to this language as \mathcal{B} .

2.1 Syntax of \mathcal{B}

An action signature consists of a set \mathcal{F} of fluent names, a set \mathcal{A} of action names, and a set \mathcal{V} of values for fluents in \mathcal{F} . In this section, we consider Boolean fluents, hence $\mathcal{V} =$

$\{0, 1\}$.¹ A *fluent literal* is either a fluent f or its negation $\text{neg}(f)$. Fluents and actions are concretely represented by *ground* atomic formulae $p(t_1, \dots, t_n)$ from a logic language \mathcal{L} . For simplicity, we assume that the set of admissible terms is finite.

The language \mathcal{B} allows us to specify an *action description* \mathcal{D} , which relates actions, states, and fluents using predicates of the following forms:

- `executable(a, [list-of-conditions])` asserts that the given conditions have to be satisfied in the current state in order for the action a to be executable;
- `causes(a, l, [list-of-conditions])` encodes a dynamic causal law, describing the effect (the fluent literal l) of the execution of action a in a state satisfying the given conditions;
- `caused([list-of-conditions], l)` describes a static causal law—i.e., the fact that the fluent literal l is true in a state satisfying the given preconditions;

(where `[list-of-conditions]` denotes a list of fluent literals). An *action description* is a set of executability conditions, static, and dynamic laws. A specific *planning problem* contains an action description \mathcal{D} along with a description of the *initial state* and the *desired goal* (\mathcal{O}):

- `initially(l)` asserts that the fluent literal l is true in the initial state,
- `goal(f)` asserts that the goal requires the fluent literal f to be true in the final state.

Fig. 4 reports an encoding of the three barrels problem using this language.

2.2 Semantics of \mathcal{B}

If $f \in \mathcal{F}$ is a fluent, and S is a set of fluent literals, we say that $S \models f$ iff $f \in S$ and $S \models \text{neg}(f)$ iff $\text{neg}(f) \in S$. Lists of literals $[\ell_1, \dots, \ell_n]$ denote conjunctions of literals. We denote with $\neg S$ the set $\{f : \text{neg}(f) \in S\} \cup \{\text{neg}(f) : f \in S\}$. A set of fluent literals is *consistent* if there are no fluents f s.t. $S \models f$ and $S \models \text{neg}(f)$. If $S \cup \neg S \supseteq \mathcal{F}$ then S is *complete*. A set S of literals is *closed* under a set of static laws $\mathcal{S}\mathcal{L} = \{\text{caused}(C_1, \ell_1), \dots, \text{caused}(C_n, \ell_n)\}$, if for all $i \in \{1, \dots, n\}$ it holds that $S \models C_i \Rightarrow S \models \ell_i$. The set $\text{Clo}_{\mathcal{S}\mathcal{L}}(S)$ is defined as the smallest set of literals containing S and closed under $\mathcal{S}\mathcal{L}$. $\text{Clo}_{\mathcal{S}\mathcal{L}}(S)$ is uniquely determined and not necessarily consistent.

The semantics of an action language on the action signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{A} \rangle$ is given in terms of a transition system $\langle \mathcal{S}, v, R \rangle$ [6], consisting of a set \mathcal{S} of states, a total interpretation function $v : \mathcal{F} \times \mathcal{S} \rightarrow \mathcal{V}$ (in this section $\mathcal{V} = \{0, 1\}$), and a transition relation $R \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Given a transition system $\langle \mathcal{S}, v, R \rangle$ and a state $s \in \mathcal{S}$, let:

$$\text{Lit}(s) = \{f \in \mathcal{F} : v(f, s) = 1\} \cup \{\text{neg}(f) : f \in \mathcal{F}, v(f, s) = 0\}.$$

Observe that $\text{Lit}(s)$ is consistent and complete. Given a set of dynamic laws $\mathcal{D}\mathcal{L} = \{\text{causes}(a, \ell_1, C_1), \dots, \text{causes}(a, \ell_n, C_n)\}$ for the action $a \in \mathcal{A}$ and a state $s \in \mathcal{S}$, we define the *effect of a in s* as follows: $E(a, s) = \{\ell_i : 1 \leq i \leq n, \text{Lit}(s) \models C_i\}$.

Let \mathcal{D} be an action description defined on the action signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{A} \rangle$, composed of dynamic laws $\mathcal{D}\mathcal{L}$, executability conditions $\mathcal{E}\mathcal{L}$, and static causal laws $\mathcal{S}\mathcal{L}$. The transition system $\langle \mathcal{S}, v, R \rangle$ *described by \mathcal{D}* is a transition system such that:

¹ Consequently, we often say that a fluent is true (resp., false) if its value is 1 (resp., 0).

- If $s \in \mathcal{S}$, then $Lit(s)$ is closed under \mathcal{SL} ;
- R is the set of all triples $\langle s, a, s' \rangle$ such that

$$Lit(s') = \text{Clo}_{\mathcal{SL}}(E(a, s) \cup (Lit(s) \cap Lit(s'))) \quad (1)$$

and $Lit(s) \models C$ for at least one condition $\text{executable}(a, C)$ in \mathcal{EL} .

Let $\langle \mathcal{D}, \mathcal{O} \rangle$ be a planning problem instance, where $\Delta = \{\ell \mid \text{initially}(\ell) \in \mathcal{O}\}$ is a complete set of fluent literals. A *trajectory* is a sequence $s_0 a_1 s_1 a_2 \dots a_n s_n$ s.t. $\langle s_i, a_{i+1}, s_{i+1} \rangle \in R$ ($0 \leq i < n$). Given the corresponding transition system $\langle \mathcal{S}, v, R \rangle$, a sequence of actions a_1, \dots, a_n is a solution (a *plan*) to the planning problem $\langle \mathcal{D}, \mathcal{O} \rangle$ if there is a trajectory $s_0 a_1 s_1 \dots a_n s_n$ in $\langle \mathcal{S}, v, R \rangle$ s.t. $Lit(s_0) = \Delta$ and $Lit(s_n) \models \ell$ for each $\text{goal}(\ell) \in \mathcal{O}$. The plan is sequential and the desired length is given.

3 Modeling \mathcal{B} and planning problems in CLP(FD)

Let us describe how action theories are mapped to finite domain constraints. We will focus on how constraints can be used to model the possible transitions from each individual state of the transition system. If s_v and s_u are the starting and ending states of a transition, we assert constraints that relate the truth value of fluents in s_v and s_u .

A Boolean variable is introduced to describe the truth value of each fluent in a state. The value of a fluent f in s_v (resp., s_u) is represented by the variable IV_f^v (resp., EV_f^u). These variables can be used to build expressions IV_l^v (EV_l^u) that represent the truth value of each fluent literal l . In particular, if l is a fluent f , then $\text{IV}_l^v = \text{IV}_f^v$; if l is the literal $\text{neg}(f)$, then $\text{IV}_l^v = 1 - \text{IV}_f^v$. Similar equations can be set for EV_l^u . In a similar spirit, given a conjunction of literals $\alpha \equiv [l_1, \dots, l_n]$ we will denote with IV_α^v the expression $\text{IV}_{l_1}^v \wedge \dots \wedge \text{IV}_{l_n}^v$; an analogous definition is given for EV_α^u . We will also introduce, for each action a_i , a Boolean variable A_i^v , representing whether the action is executed or not in the transition from s_v to s_u under consideration.

Given a specific fluent f , we develop constraints that determine when EV_f^u is true and false. Let us consider the dynamic causal laws that have f as a consequence:

$$\text{causes}(a_{t_1}, f, \alpha_1) \quad \dots \quad \text{causes}(a_{t_m}, f, \alpha_m)$$

Analogously, we consider the static causal laws that assert $\text{neg}(f)$:

$$\text{causes}(a_{f_1}, \text{neg}(f), \beta_1) \quad \dots \quad \text{causes}(a_{f_n}, \text{neg}(f), \beta_n)$$

Let us also consider the static causal laws related to f

$$\begin{array}{lll} \text{caused}(\gamma_1, f) & \dots & \text{caused}(\gamma_h, f) \\ \text{caused}(\psi_1, \text{neg}(f)) & \dots & \text{caused}(\psi_\ell, \text{neg}(f)) \end{array}$$

Finally, for each action a_i we will have its executability conditions:

$$\text{executable}(a_i, \delta_1^i) \quad \dots \quad \text{executable}(a_i, \delta_p^i)$$

Figure 1 describes the Boolean constraints that can be used in encoding the relations that determine the truth value of the fluent f . We will denote with $C_f^{v,u}$ the conjunction of such constraints. Given an action specification over the set of fluents \mathcal{F} , the system of constraints $C_{\mathcal{F}}^{v,v+1}$ includes:

- the constraint $C_f^{v,v+1}$ for each $f \in \mathcal{F}$ and for each $0 \leq v < N$ where N is the chosen length of the plan;

$EV_f^u \leftrightarrow \text{Posfired}_f^{v,u} \vee (\neg \text{Negfired}_f^{v,u} \wedge IV_f^v)$	(2)
$\neg \text{Posfired}_f^{v,u} \vee \neg \text{Negfired}_f^{v,u}$	(3)
$\text{Posfired}_f^{v,u} \leftrightarrow \text{DynP}_f^v \vee \text{StatP}_f^u$	(4)
$\text{Negfired}_f^{v,u} \leftrightarrow \text{DynN}_f^v \vee \text{StatN}_f^u$	(5)
$\text{DynP}_f^v \leftrightarrow \bigvee_{i=1}^m (IV_{\alpha_i}^v \wedge A_{\alpha_i}^v)$	(6)
$\text{StatP}_f^u \leftrightarrow \bigvee_{i=1}^h EV_{\gamma_i}^u$	(7)
$\text{DynN}_f^v \leftrightarrow \bigvee_{i=1}^n (IV_{\beta_i}^v \wedge A_{\beta_i}^v)$	(8)
$\text{StatN}_f^u \leftrightarrow \bigvee_{i=1}^{\ell} EV_{\psi_i}^u$	(9)

Fig. 1. The constraint $C_f^{v,u}$ for the generic fluent f

- for each $f \in \mathcal{F}$ and $0 \leq v \leq N$, the constraints $IV_f^v = EV_f^v$
- for each $0 \leq v < N$, the constraint $\sum_{a_j \in \mathcal{A}} A_j^v = 1$
- for each $0 \leq v < N$ and for each action $a_i \in \mathcal{A}$, the constraints $A_i^v \rightarrow \bigvee_{j=1}^p IV_{\delta_j}^v$

This modeling has been translated into a concrete implementation using SICStus Prolog. In this translation constrained CLP variables directly reflect the Boolean variables modeling fluent and action's occurrences. Consequently, causal laws and executability conditions are directly translated into CLP constraints. The complete code and proofs can be found at www.dimi.uniud.it/dovier/CLPASP. Let us proceed with a sketch of the correctness proof of the constraint-based encoding w.r.t. the semantics.

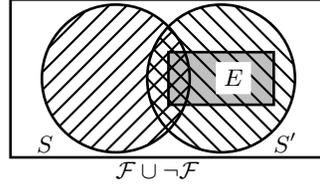


Fig. 2. Sets of fluents involved in a state transition

Let $S = Lit(s_v)$ (resp., $S' = Lit(s_{v+1})$) be the set of fluent literals that holds in s_v (resp., s_{v+1}). We denote by $E = E(a, s)$, and by $\text{Clo} = \text{Clo}_{S\mathcal{L}}$. Fig. 2 depicts the equation $S' = \text{Clo}(E \cup (S \cap S'))$. From any specific, known, S (resp., S'), we can obtain a consistent assignment σ_S (resp., $\sigma_{S'}$) of truth values for all the variables IV_f^v (resp., EV_f^{v+1}) of s_v (resp., s_{v+1}). Conversely, each truth assignment σ_S (resp., $\sigma_{S'}$) for all variables IV_f^v (resp., EV_f^{v+1}) corresponds to a consistent set of fluents S (resp., S').

Let σ_a be the assignment of truth values for such variables such that $\sigma_a(A_i^v) = 1$ if and only if a_i occurs in the state transition from s_v to s_{v+1} . Note that the domains of σ_S , $\sigma_{S'}$, and σ_a are disjoint, so we can safely denote by $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ the composition of the three assignments. Clearly, $E \subseteq S'$.

Theorem 1 states the completeness of the constraint-based modeling of the planning problem. For any given action description \mathcal{D} , if a triple $\langle s, a, s' \rangle$ belongs to the transition system described by \mathcal{D} , then the assignment $\sigma = \sigma_S \circ \sigma_{S'} \circ \sigma_a$ satisfies $C_{\mathcal{F}}^{v,v+1}$.

Theorem 1 (Completeness). *If $S' = \text{Cl}_{\mathcal{O}_{\mathcal{S}\mathcal{L}}}(E(a_i, s_v) \cup (S \cap S'))$ then $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ is a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$.*

Let us observe that the converse of the above theorem does not necessarily hold. The problem arises from the fact that the implicit minimality in the closure operation is not reflected in the computation of solutions to the constraint. Consider the action description where $\mathcal{F} = \{f, g, h\}$ and $\mathcal{A} = \{a\}$, with predicates:

`executable(a, []). causes(a, f, []). caused([g], h). caused([h], g).`

Setting $S = \{\text{neg}(f), \text{neg}(g), \text{neg}(h)\}$ and $S' = \{f, g, h\}$ determines a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$ with the execution of action a , but $\text{Cl}_{\mathcal{O}_{\mathcal{S}\mathcal{L}}}(E \cup (S \cap S')) = \{f\} \subset S'$. However, the following holds:

Theorem 2 (Weak Soundness). *Let $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ identify a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$. Then $\text{Cl}_{\mathcal{O}_{\mathcal{S}\mathcal{L}}}(E(a_i, s_v) \cup (S \cap S')) \subseteq S'$.*

Let us consider the set of static causal laws $\mathcal{S}\mathcal{L}$. $\mathcal{S}\mathcal{L}$ identifies a *definite propositional program* P as follows. For each positive fluent literal p , let $\varphi(p)$ be the (fresh) predicate symbol p , and for each negative fluent literal $\text{neg}(p)$ let $\varphi(\text{neg}(p))$ be the (fresh) predicate symbol \tilde{p} . The program P is the set of clauses of the form $\varphi(p) \leftarrow \varphi(l_1), \dots, \varphi(l_m)$, for each static causal law $\text{caused}([l_1, \dots, l_m], p)$. Notice that p and \tilde{p} are independent predicate symbols in P . From P one can extract the dependency graph $\mathcal{G}(P)$ in the usual way, and the following result can be stated.

Theorem 3 (Correctness). *Let $\sigma_S, \sigma_{S'}, \sigma_a$ be a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$. If the dependency graph of P is acyclic, then $\text{Cl}_{\mathcal{O}_{\mathcal{S}\mathcal{L}}}(E(a_i, s_v) \cup (S \cap S')) = S'$.*

If the program P meets the conditions of the previous theorem, then the following holds.

Theorem 4. *There is a trajectory $\langle s_0, a_1, s_1, a_2, \dots, a_n, s_n \rangle$ in the transition system if and only if there is a solution for the constraints $C_{\mathcal{F}}^{0,1} \wedge C_{\mathcal{F}}^{1,2} \wedge \dots \wedge C_{\mathcal{F}}^{n-1,n}$.*

4 The Action Language \mathcal{B}_{MV}^{FD}

Constraints represent a very declarative notation to express relationships between unknowns; as such, the ability to use them directly in the action theory would greatly enhance the declarative and expressive power of the action language, facilitating the encoding of complex action domains, such as those involving multivalued fluents.

Example 1 (Control Knowledge). Domain-specific control knowledge can be formalized as constraints that we expect to be satisfied by all the trajectories. For example, we may know that if a certain action occurs at a given time step (e.g., `ingest_poison`) then at the next time step we will always perform the same action (e.g., `call_doctor`). This could be encoded as `occ(ingest_poison) \Rightarrow occ(call_doctor)`¹ where `occ(a)` is a fluent describing the occurrence of the action a and f^1 indicates that the fluent f should hold at the next time step. The operator \Rightarrow is an implication constraint.

Example 2 (Delayed Effect). Let us assume that the action `a` (e.g., `req_reimbursement`) has a delayed effect (e.g., `bank_account` increased by \$50 after 30 time units). This could be expressed as a dynamic causal law:

$$\text{causes}(\text{request_reimbursement}, \text{incr}(\text{bank}, 50)^{30}, [])$$

where `incr` is a constraint introduced to deal with additive computations.

Example 3 (Maintenance Goals). It is not uncommon to encounter planning problems where along with the type of goals described earlier (known as *achievement* goals), there are also *maintenance* goals, representing properties that must persist throughout the trajectory. Constraints are a natural way of encoding maintenance properties, and can be introduced along with simple temporal operators. E.g., if the fluent `fuel` represents the amount of fuel available, then the maintenance goal which guarantees that we will not be left stranded could be encoded as: `always(fuel > 0)`.

Furthermore, the encoding of an action theory using multivalued fluents leads to more compact and more efficient representations, facilitating constraint propagation during planning (with pruning of the search space) and better exposing non-determinism (that could be exploited, for example, by a parallel planner).

4.1 Syntax of \mathcal{B}_{MV}^{FD}

Let us introduce the syntax of \mathcal{B}_{MV}^{FD} . As for \mathcal{B} , the action signature consists of a set \mathcal{F} of fluent names, a set \mathcal{A} of action names, and a set \mathcal{V} of values for fluents in \mathcal{F} .

In the definition of an action description, an assertion (*domain declaration*) of the kind `fluent(f, v_1, v_2)` or `fluent($f, \{v_1, \dots, v_k\}$)` declares that f is a fluent and that its set of values \mathcal{V} is the interval $[v_1, v_2]$ or the set $\{v_1, \dots, v_k\}$.² An *annotated fluent* (AF) is an expression f^a , where f is a fluent and $a \in \mathbb{N}^-$.³ Intuitively speaking, an annotated fluent f^a denotes the value the fluent f had in the past, $-a$ steps ago. Such fluents can be used in *fluent expressions* (FE), which are defined inductively as follows:

$$\text{FE} ::= n \mid \text{AF} \mid \text{abs}(\text{FE}) \mid \text{FE}_1 \oplus \text{FE}_2 \mid \text{rei}(\text{FC})$$

where $n \in \mathbb{Z}$, $\oplus \in \{+, -, *, /, \text{mod}\}$. `rei(FC)` is the reification of fluent constraint FC.

Fluent expressions can be used to build *fluent constraints* (FC), i.e., formulae of the form $\text{FE}_1 \text{ op } \text{FE}_2$, where FE_1 and FE_2 are fluent expressions and $\text{op} \in \{\text{eq}, \text{neq}, \text{geq}, \text{leq}, \text{lt}, \text{gt}\}$. The language \mathcal{B}_{MV}^{FD} allows one to specify an *action description*, which relates actions, states, and fluents using predicates of the following forms:

- axioms of the form `executable(a, C)` asserting that the fluent constraint C has to be entailed by the current state in order for the action a to be executable.
- axioms of the form `causes(a, C, C_1)` encode dynamic causal laws. The action a can be executed if the constraint C_1 is entailed by the current state; the state produced by the execution of the action is required to entail the constraint C .
- axioms of the form `caused(C_1, C_2)` describe static causal laws. If the fluent constraint C_1 is satisfied in a state, then the constraint C_2 must also hold in such state.

² Note that we could generalize the notion of domain to more complex and non-numeric sets.

³ With \mathbb{N}^- we denote the set $\{0, -1, -2, -3, \dots\}$. We will often denote f^0 simply by f .

An *action description* is a set of executability conditions, static and dynamic laws.

Observe that traditional action languages like \mathcal{B} are special cases of \mathcal{B}_{MV}^{FD} . For example, the dynamic causal law of \mathcal{B} :

$$\text{causes}(a, f, [\text{f}_1, \dots, \text{f}_k, \text{neg}(g_1), \dots, \text{neg}(g_h)])$$

can be encoded as

$$\text{causes}(a, \text{f}^0 \text{ eq } 1, [\text{f}_1^0 \text{ eq } 1, \dots, \text{f}_k^0 \text{ eq } 1, \text{g}_1^0 \text{ eq } 0, \dots, \text{g}_h^0 \text{ eq } 0])$$

A specific instance of a planning problem is a pair $\langle \mathcal{D}, \mathcal{O} \rangle$, where \mathcal{D} is an action theory, and \mathcal{O} contains any number of axioms of the form $\text{initially}(C)$ and $\text{goal}(C)$, where C is a fluent constraint.

4.2 Semantics of \mathcal{B}_{MV}^{FD}

Each fluent f is assigned uniquely to a domain $\text{dom}(f)$ in the following way:

- If $\text{fluent}(f, v_1, v_2) \in \mathcal{D}$ then $\text{dom}(f) = \{v_1, v_1 + 1, \dots, v_2\}$.
- If $\text{fluent}(f, \text{Set}) \in \mathcal{D}$, then $\text{dom}(f) = \text{Set}$.

A function $\nu : \mathcal{F} \rightarrow \mathbb{Z} \cup \{\perp\}$ is a *state* if $\nu(f) \in \text{dom}(f)$ for all $f \in \mathcal{F}$. For a number $n \geq 1$, we define a *state sequence* $\bar{\nu}$ as a tuple $\langle \nu_0, \dots, \nu_n \rangle$ where each ν_i is a state.

Let us consider a state sequence $\bar{\nu}$, a step $0 \leq i < |\bar{\nu}|$, a fluent expression φ , and let us define the concept of *value* of φ in $\bar{\nu}$ at step i (denoted by $\bar{\nu}(\varphi, i)$):

- $\bar{\nu}(x, i) = x$ if x is a number
- $\bar{\nu}(f^a, i) = \nu_{i-|a|}(f)$ if $|a| \leq i$, \perp otherwise
- $\bar{\nu}(\text{abs}(\varphi), i) = \text{abs}(\bar{\nu}(\varphi, i))$
- $\bar{\nu}(\varphi_1 \oplus \varphi_2, i) = \bar{\nu}(\varphi_1, i) \oplus \bar{\nu}(\varphi_2, i)$

We treat the interpretation of the various \oplus operations and relations as strict w.r.t. \perp .

Given a fluent constraint $\varphi_1 \text{ op } \varphi_2$, a state sequence $\bar{\nu}$ and a time $0 \leq i < |\bar{\nu}|$, the notion of satisfaction $\bar{\nu} \models_i \varphi_1 \text{ op } \varphi_2$ is defined as $\bar{\nu} \models_i \varphi_1 \text{ op } \varphi_2 \Leftrightarrow \bar{\nu}(\varphi_1, i) \text{ op } \bar{\nu}(\varphi_2, i)$.

If $\bar{\nu}(\varphi_1, i)$ or $\bar{\nu}(\varphi_2, i)$ is \perp , then $\bar{\nu} \not\models_i \varphi_1 \text{ op } \varphi_2$. \models_i can be generalized to the case of propositional combinations of fluent constraints. In particular, $\bar{\nu}(\text{rei}(C), i) = 1$ if $\bar{\nu} \models_i C$, else $\bar{\nu}(\text{rei}(C), i) = 0$. The operations \cap and \cup on states are defined next:

$$\nu_1 \cup \nu_2(f) = \begin{cases} \nu_1(f) & \text{if } \nu_1(f) = \nu_2(f) \\ \nu_1(f) & \text{if } \nu_2(f) = \perp \\ \nu_2(f) & \text{if } \nu_1(f) = \perp \end{cases} \quad \nu_1 \cap \nu_2(f) = \begin{cases} \nu_1(f) & \text{if } \nu_1(f) = \nu_2(f) \\ \perp & \text{otherwise} \end{cases}$$

Let $\bar{\nu}$ be a state sequence; we say that $\bar{\nu}$ is consistent if, for each $0 \leq i < |\bar{\nu}|$, and for each static causal law $\text{caused}(C_1, C_2)$ we have that: $\bar{\nu} \models_i C_1 \Rightarrow \bar{\nu} \models_i C_2$.

Let a be an action and $\bar{\nu}$ be a state sequence. The action a is executable in $\bar{\nu}$ at step i ($0 \leq i < |\bar{\nu}| - 1$) if there is an axiom $\text{executable}(a, C)$ such that $\bar{\nu} \models_i C$.

Let us denote with $\text{Dyn}(a)$ the set of dynamic causal law axioms for action a . The effects of executing a at step i in the state sequence $\bar{\nu}$, denoted by $\text{Eff}(a, \bar{\nu}, i)$, is

$$\text{Eff}(a, \bar{\nu}, i) = \bigwedge \{C \mid \text{causes}(a, C, C_1) \in \text{Dyn}(a), \bar{\nu} \models_i C_1\}$$

Furthermore, given a constraint C , a state sequence $\bar{\nu}$, and a step i , the reduct $\text{Red}(C, \bar{\nu}, i)$ is defined as the constraint

$$\text{Red}(C, \bar{\nu}, i) = C \wedge \bigwedge \{f^{-j} = \nu_{i-j}(f) \mid f \in \mathcal{F}, 1 \leq j \leq i\}$$

Let us denote with $Stat$ the set of static causal law axioms. We can define $Clo(\bar{\nu}, i)$ as

$$Clo(\bar{\nu}, i) = \bigwedge \{C_2 \mid \text{caused}(C_1, C_2) \in Stat, \bar{\nu} \models_i C_1\}$$

A state sequence $\bar{\nu}$ is a valid trajectory if the following conditions hold:

- for each axiom of the form $\text{initial}(C)$ in the action theory we have that $\bar{\nu} \models_0 C$
- for each axiom of the form $\text{goal}(C)$ we have that $\bar{\nu} \models_{|\bar{\nu}|-1} C$
- for each $0 \leq i < |\bar{\nu}| - 1$ there is an action a_i such that
 - action a_i is executable in $\bar{\nu}$ at step i
 - we have that $\nu_{i+1} = \sigma \cup (\nu_i \cap \nu_{i+1})$ (*) where σ is a solution of the constraint

$$Red(Eff(a_i, \bar{\nu}, i), \bar{\nu}, i+1) \wedge Clo(\bar{\nu}, i+1)$$

Let us conclude with some comments on the relationship between the semantics of \mathcal{B} and of \mathcal{B}_{MV}^{FD} . First of all, the lack of backward references in \mathcal{B} allows us to analyze each \mathcal{B} transition independently. Conversely, in \mathcal{B}_{MV}^{FD} we need to keep track of the complete trajectory—represented by a sequence of states.

In \mathcal{B} , the effect of a static or dynamic causal law is to set the truth value of a fluent. Hence, the sets E and Clo can be deterministically determined (even if they can be inconsistent) and the equation (1) takes care of these two sets and of the inertia. In \mathcal{B}_{MV}^{FD} , instead, less determined effects can be set. For instance, $\text{causes}(a, f \text{ gt } 1, [])$, if $\text{dom}(f) = \{0, 1, 2, 3\}$, admits two possible values for f in the next state. One could extend the semantics of Sect. 2, by working on sets of sets for E or Clo . Instead, we have chosen to encode the nondeterminism within the solutions of the introduced constraints. Equation (1) is therefore replaced by equation (*) above.

The concrete implementation of \mathcal{B}_{MV}^{FD} in SICStus Prolog is directly based on this semantics. We omit the detailed description, which is a fairly mechanical extension of the implementation of \mathcal{B} (in this case the main difference w.r.t. what done in Sect. 3, is that the set of the admitted values for multivalued fluents determines the domain of the CLP constrained variables), and relative proof of correctness due to lack of space.

4.3 Some Concrete Extensions

The language \mathcal{B}_{MV}^{FD} described above has been implemented using SICStus Prolog, as a fairly direct generalization of the encoding described for the Boolean case. In addition, we have introduced in the implementation some additional syntactic extensions, to facilitate the encoding of recurring problem features.

It is possible to add information about the *cost* of each action, fluent, and about the global cost of a plan. This can be done by writing rules of the form:

- `action_cost(action, VAL)` (if no information is given, the default cost is 1).
- `state_cost(FE)` (if no information is given, the default cost is 1) is the cost of a state, where FE is a fluent expression built on current fluents.
- `plan_cost(plan op n)` where n is a number, adds the information about the global cost admitted for the sequence of actions.
- `goal_cost(goal op NUM)` adds a constraint about the global cost admitted for the sequence of states.

- `minimize_action` to constrain the search to a plan with minimal global action cost.
- `minimize_state` which forces the search of a plan with minimal goal state cost.

The implementation of these cost-based constraints relies on the optimization features offered by SICStus’ labeling predicate: the labeling phase is guided by imposing an objective function to be optimized.

The language allows the definition of *absolute temporal constraints*, i.e., constraints that refer to specific time instances in the trajectory. We define a timed fluent as a pair `FLUENT @ TIME`. Timed fluents are used to build timed fluent expressions (TE) and timed primitive constraints (TC). E.g., `contains(5) @ 2 leq contains(5) @ 4` states that, at time 2, barrel 5 contains at most the same amount of water as at time 4. `contains(12) @ 2 eq 3` states that, at time 3, barrel 12 contains 3 liters of water. These constructs can be used in the following expressions:

- `cross_constraint(TC)` imposes the timed constraint TC to hold.
- `holds(FC, StateNumber)` It is a simplification of the above constraint. states that the primitive fluent constraint *FC* holds at the desired State Number (0 is the number of the initial state). It is therefore a generalization of the `initially` primitive. It allows to drive the plan search with some point information.
- `always(FC)` states that the fluent constraints FC holds in all the states. Current fluents must be used in order to avoid negative references.

The semantics can be easily extended by conjoining these new constraints to the formulae of the previous subsection.

4.4 An Extended \mathcal{B}_{MV}^{FD} Language: Looking into the Future

We propose to extend \mathcal{B}_{MV}^{FD} to allow constraints that reason about future steps of the trajectory (along lines similar to [3]).

Syntax: Let us generalize the syntax presented earlier as follows: an annotated fluent is of the form f^a , where $f \in \mathbb{Z}$. Constructing fluent formulae and fluent constraints now implies the ability of looking into the future steps of computation. In addition, we introduce another fluent constraint, that will help us encoding interesting problems: $incr(f^a, n)$ where f^a is an annotated fluent and n is a number. The *incr* constraint provides a simplified view of additive fluents [9].

Semantics: The definition of the semantics becomes a process of “validating” a sequence of states to verify their fitness to serve as a trajectory. Given a fluent formula/constraint φ and a time step i , we define the concept $Absolute(\varphi, i)$ as follows:

- if $\varphi \equiv n$ then $Absolute(\varphi, i) = n$
- if $\varphi \equiv f^a$ then $Absolute(\varphi, i) = f^{i+a}$
- if $\varphi \equiv \varphi_1 \oplus \varphi_2$ then $Absolute(\varphi, i) = Absolute(\varphi_1, i) \oplus Absolute(\varphi_2, i)$
- if $\varphi \equiv abs(\varphi_1)$ then $Absolute(\varphi, i) = abs(Absolute(\varphi_1, i))$
- if $\varphi \equiv rei(\varphi_1)$ then $Absolute(\varphi, i) = rei(Absolute(\varphi_1, i))$
- if $\varphi \equiv \varphi_1 \text{ op } \varphi_2$ then $Absolute(\varphi, i) = Absolute(\varphi_1, i) \text{ op } Absolute(\varphi_2, i)$
- if $\varphi \equiv incr(\varphi_1, n)$ then $Absolute(\varphi, i) = incr(Absolute(\varphi_1, i), n)$

For a sequence of states $\bar{\nu} = \langle \nu_0, \dots, \nu_n \rangle$ and actions $\bar{a} = \langle a_0, \dots, a_{n-1} \rangle$, let us define

$$Global(\bar{a}, \bar{\nu}) = \bigwedge_{i=0}^{n-1} Absolute(Eff(a_i, \bar{\nu}, i), i+1)$$

The sequence of states $\bar{\nu}$ is a trajectory if

- for each axiom of the form $initial(C)$ in the action theory we have that $\bar{\nu} \models_0 C$
- for each axiom of the form $goal(C)$ in the action theory we have that $\bar{\nu} \models_{|\bar{\nu}|-1} C$
- there is a sequence of actions $\bar{a} = \langle a_0, a_1, \dots, a_{n-1} \rangle$ with the following properties: for each $0 \leq i < n$ we have that a_i is executable at step i of $\bar{\nu}$ and $\nu_{i+1} = \sigma \cup (\nu_i \cap \nu_{i+1})$, where σ is a solution of $Red(Global(\bar{a}, \bar{\nu}), \bar{\nu}, i+1) \wedge Clo(\bar{\nu}, i+1)$.

In particular, if $Incr(C, i) = \{n \mid incr(f^i, n) \in C\}$ are all the *incr* constraints for an annotated fluent f^i , then θ is a solution of it w.r.t. a sequence of states $\bar{\nu}$ if $\nu_i(f) = \nu_{i-1}(f) + \sum_{n \in Incr(C, i)} n$.

5 Experimental Analysis

The prototype, implemented on an AMD Opteron 2.2GHz Linux machine, has been validated on a number of benchmarks. Extensive testing has been performed on the CLP encoding of \mathcal{B} , and additional results can be found at www.dimi.uniud.it/dovier/CLPASP. Here we concentrate on two representative examples.

5.1 Three-barrel Problem

We experimented with different encodings of the three-barrel problem. There are three barrels of capacity N (even number), $N/2 + 1$, and $N/2 - 1$, respectively. At the beginning the largest barrel is full of wine, the other two are empty. We wish to reach a state in which the two larger barrels contain the same amount of wine. The only permissible action is to pour wine from one barrel to another, until the latter is full or the former is empty. Figure 4 shows the encodings of the problem (for $N = 12$) in \mathcal{B} (where, it is also required that the smallest barrel is empty at the end) and \mathcal{B}_{MV}^{FD} . Table 1 provides the execution times (in seconds) for different values of N and different plan lengths. The \mathcal{B} encoding has been processed by our CLP(FD) implementation and by two ASP solvers (Smodels and Cmodels)—the encoding of a \mathcal{B} action description in ASP is natural (see [5]). The \mathcal{B}_{MV}^{FD} descriptions have been solved using SICStus Prolog.

5.2 2-Dimensional Protein Folding Problem

The problem we have encoded is a simplification of the protein structure folding problem. The input is a chain $a_1 a_2 \dots a_n$ with $a_i \in \{0, 1\}$, initially placed in a vertical position, as in Fig. 3-left. We will refer to each a_i as an *amino acid*. The permissible actions are the counterclockwise/clockwise *pivot moves*. Once one point i of the chain is selected, the points a_1, a_2, \dots, a_i will remain fixed, while the points a_{i+1}, \dots, a_n will perform a rigid counterclockwise/clockwise rotation. Each conformation must be

Barrels' capacities	Len.	Ans.	\mathcal{B}				\mathcal{B}_{MV}^{FD}	
			<i>l</i> parse	Smodels	Cmodels	CLP(FD)	unconstrained plan cost	constrained plan cost (in parentheses)
8-5-3	6	N	8.95	0.10	0.85	0.16+0.36	0.02+0.11	(70) 0.01+0.10
8-5-3	7	Y	8.94	0.28	1.34	0.19+0.47	0.02+0.13	(70) 0.03+0.13
8-5-3	8	Y	9.16	0.39	2.07	0.18+2.66	0.03+0.85	(70) 0.01+0.79
8-5-3	9	Y	9.22	0.39	8.11	0.22+1.05	0.02+0.28	(70) 0.05+0.28
12-7-5	10	N	35.63	18.31	325.28	0.45+26.86	0.05+7.79	(90) 0.03+6.78
12-7-5	11	Y	35.70	45.91	781.28	0.52+28.87	0.05+9.46	(90) 0.04+5.03
12-7-5	12	Y	35.58	81.12	4692.08	0.58+203.34	0.06+57.42	(100) 0.04+35.31
12-7-5	13	Y	35.67	18.87	1581.49	0.66+66.52	0.06+25.65	(100) 0.03+23.26
16-9-7	14	N	114.16	2018.65	–	1.28+2560.90	0.07+564.68	(170) 0.07+518.78
16-9-7	15	Y	113.53	2493.61	–	1.29+2833.97	0.07+688.84	(170) 0.07+520.14
16-9-7	16	Y	115.36	6801.36	–	1.37+17765.62	0.06+4282.86	(170) 0.04+1904.17
16-9-7	17	Y	114.04	2294.15	–	1.55+6289.06	0.06+1571.78	(200) 0.06+1389.27

Table 1. Experimental results with various instances of the three-barrels problem (For CLP(FD) and \mathcal{B}_{MV}^{FD} we reported the time required for the constrain and the labelling phases).

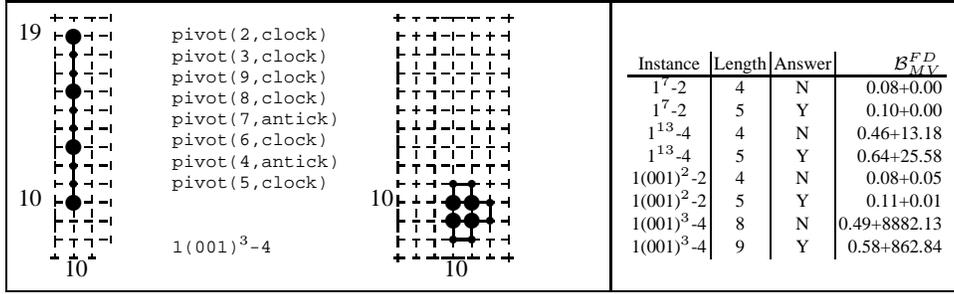


Fig. 3. On the left: Initial configuration, a plan, and final configuration with 4 contacts between 1-amino acids. On the right: some results for different sequences, energy levels, and plan lengths.

a *self-avoiding-walk*, i.e., no two amino acids are in the same position. Moreover, the chain cannot be broken—i.e., two consecutive amino acids are always at points at distance 1 (i.e., in contact). The goal is to perform a sequence of pivot moves leading to a configuration where at least k non-consecutive amino acids of value 1 are in contact. Fig. 3 shows a possible plan to reach a configuration with 4 contacts. The figure also reports some execution times. Fig. 5 reports the \mathcal{B}_{MV}^{FD} action description encoding this problem. Since the goal is based on the notion of cost of a given state, for which reified constraints are used extensively, a direct encoding in \mathcal{B} does not appear viable.

Let us consider the resolution of the problem starting from the input chain 1001001001. If $N = 10$, asking for a plan of 8 moves and for a solution with cost ≥ 4 , our planner finds the plan shown in Fig. 3-center in 862.84s. Notice that, by adding the pair of constraints $\text{holds}(x(3) \text{ eq } 11, 1)$ and $\text{holds}(y(3) \text{ eq } 11, 1)$ the time is reduced to 61.05s, and with the constraint $\text{holds}(x(4) \text{ eq } 11, 2)$. $\text{holds}(y(4) \text{ eq } 10, 2)$. the plan is found in only 5.11s. In this case, multivalued fluents and the ability to introduce domain knowledge allows \mathcal{B}_{MV}^{FD} to effectively converge to a solution.

```

(1) barrel(5). barrel(7). barrel(12).
(2) liter(0). liter(1). liter(2). ... liter(11). liter(12).
(3) fluent(cont(B,L):- barrel(B),liter(L),L =< B.
(4) action(fill(X,Y):- barrel(X),barrel(Y), neq(X,Y).
(5) causes(fill(X,Y),cont(X,0),[cont(X,LX),cont(Y,LY)]):-
(6)   action(fill(X,Y)), fluent(cont(X,LX)),
(7)   fluent(cont(Y,LY)), Y-LY >= LX.
(8) causes(fill(X,Y),cont(Y,LYnew),[cont(X,LX),cont(Y,LY)]):-
(9)   action(fill(X,Y)), fluent(cont(X,LX)),
(10)  fluent(cont(Y,LY)), Y-LY >= LX, LYnew is LX + LY.
(11) causes(fill(X,Y),cont(X,LXnew),[cont(X,LX),cont(Y,LY)]):-
(12)  action(fill(X,Y)), fluent(cont(X,LX)),
(13)  fluent(cont(Y,LY)), LX >= Y-LY, LXnew is LX-Y+LY.
(14) causes(fill(X,Y),cont(Y,Y),[cont(X,LX),cont(Y,LY)]):-
(15)  action(fill(X,Y)), fluent(cont(X,LX)),
(16)  fluent(cont(Y,LY)), LX >= Y-LY.
(17) executable(fill(X,Y),[cont(X,LX),cont(Y,LY)]) :-
(18)  action(fill(X,Y)), fluent(cont(X,LX)),
(19)  fluent(cont(Y,LY)), LX > 0, LY < Y.
(20) caused([ cont(X,LX) ], neg(cont(X,LY)) ) :-
(21)  fluent(cont(X,LX)), fluent(cont(X,LY)),
(22)  botte(X),liter(LX),liter(LY),neq(LX,LY).
(23) initially(cont(12,12)). initially(cont(7,0)). initially(cont(5,0)).
(24) goal(cont(12,6)). goal(cont(7,6)). goal(cont(5,0)).

(1) barrel(5). barrel(7). barrel(12).
(2) fluent(cont(B),0,B):- barrel(B).
(3) action(fill(X,Y):- barrel(X),barrel(Y), neq(X,Y).
(4) causes(fill(X,Y), cont(X) eq 0, [Y-cont(Y) geq cont(X)]):-
(5)   action(fill(X,Y)).
(6) causes(fill(X,Y), cont(Y) eq cont(Y)^(-1) + cont(X)^(-1),
(7)   [Y-cont(Y) geq cont(X) ]):- action(fill(X,Y)).
(8) causes(fill(X,Y), cont(Y) eq Y, [Y-cont(Y) lt cont(X)]):-
(9)   action(fill(X,Y)).
(10) causes(fill(X,Y), cont(X) eq cont(X)^(-1)-Y+cont(Y)^(-1),
(11)   [Y-cont(Y) lt cont(X)]):- action(fill(X,Y)).
(12) executable(fill(X,Y), [cont(X) gt 0, cont(Y) lt Y ]):-
(13)   action(fill(X,Y)).
(14) caused([], cont(12) eq 12-cont(5)-cont(7)).
(15) initially(cont(12) eq 12).
(16) goal(cont(12) eq cont(7)).

```

Fig. 4. \mathcal{B} description (above) and \mathcal{B}_{MV}^{FD} description (below) of the 12-7-5 barrels problem

5.3 Other Examples

We report results from two other planning problems. The first (3x3-puzzle) is an encoding of the 8-tile puzzle problem, where the goal is to find a sequence of moves to re-order the 8 tiles, starting from a random initial position. In the *Community-M* problem, there are M persons, identified by the numbers $1, 2, \dots, M$. At each time step, one of the persons, say j , provided (s)he owns more than j dollars, gives j dollars to someone else. The goal consists of reaching a state in which there are no two persons i and j such that the difference between what is owned by i and j is greater than 1. Table 2 lists some results for $M = 5$ and for two variants of the problem: The person i initially owns $i + 1$ dollars (*inst1*) or $2 * i$ dollars (*inst2*).

```

(1) length(10).
(2) amino(A) :- length(N), interval(A,1,N).
(3) direction(clock). direction(antick).
(4) fluent(x(A),1,M) :- length(N), M is 2*N, amino(A).
(5) fluent(y(A),1,M) :- length(N), M is 2*N, amino(A).
(6) fluent(type(A),0,1) :- amino(A).
(7) fluent(saw,0,1).
(8) action(pivot(A,D)):- length(N), amino(A), 1<A,A<N, direction(D).
(9) executable(pivot(A,D),[]) :- action(pivot(A,D)).
(10) causes(pivot(A,clock),x(B) eq x(A)^(-1)+y(B)^(-1)-y(A)^(-1),[]) :-
(11) action(pivot(A,clock)),amino(B),B > A.
(12) causes(pivot(A,clock),y(B) eq y(A)^(-1)+x(A)^(-1)-x(B)^(-1),[]) :-
(13) action(pivot(A,clock)),amino(B),B > A.
(14) causes(pivot(A,antick),x(B) eq x(A)^(-1)-y(B)^(-1)+y(A)^(-1),[]) :-
(15) action(pivot(A,antick)),amino(B),B > A.
(16) causes(pivot(A,antick),y(B) eq y(A)^(-1)-x(A)^(-1)+x(B)^(-1),[]) :-
(17) action(pivot(A,antick)),amino(B),B > A.
(18) caused([x(A) eq x(B), y(A) eq y(B)],saw eq 0) :-
(19) amino(A),amino(B),A<B.
(20) initially(saw eq 1).
(21) initially(x(A) eq N) :- length(N), amino(A).
(22) initially(y(A) eq Y) :- length(N), amino(A),Y is N+A-1.
(23) initially(type(X) eq 1) :- amino(X), X mod 3 =:= 1.
(24) initially(type(X) eq 0) :- amino(X), X mod 3 =\= 1.
(25) goal(saw gt 0).
(26) state_cost( FE ) :- length(N), auxc(1,4,N,FE).
(27) auxc(I,J,N,0) :- I > N - 3,!.
(28) auxc(I,J,N,FE) :- J > N,!,I1 is I+1,J1 is I1+3,auxc(I1,J1,N,FE).
(29) auxc(I,J,N,FE1+type(I)*type(J)*
(30) rei(abs(x(I)-x(J))+abs(y(I)-y(J)) eq 1)):-
(31) J1 is J + 2, auxc(I,J1,N,FE1).
(32) always(x(1) eq 10). always(y(1) eq 10).
(33) always(x(2) eq 10). always(y(2) eq 11).
(34) goal_cost(goal geq 4).

```

Fig. 5. \mathcal{B}_{MV}^{FD} Encoding of the HP-protein folding problem with pivot moves on input of the form 1001001001... starting from a vertical straight line.

6 Conclusions and Future Work

The objective of this paper was to initiate an investigation of using constraint logic programming techniques in handling action description languages and planning problems. In particular, we presented an implementation of the language \mathcal{B} using CLP(FD), and reported on its performance. We also presented the action language \mathcal{B}_{MV}^{FD} , which

Instance	Length	Answer	\mathcal{B}				\mathcal{B}_{MV}^{FD}
			<i>lparse</i>	Smodels	Cmodels	CLP(FD)	unconstrained plan cost
3x3-puzzle	10	N	45.18	0.83	1.96	0.68+9.23	0.32+11.95
3x3-puzzle	11	Y	45.59	1.85	2.35	0.70+24.10	0.34+27.34
Community-5 _{inst1}	6	N	208.39	96.71	3.55	2.70+73.91	0.02+34.86
Community-5 _{inst1}	7	Y	205.48	10.57	2.45	3.17+0.18	0.04+0.03
Community-5 _{inst2}	6	N	204.40	54.20	3.15	2.67+61.68	0.03+19.40
Community-5 _{inst2}	7	Y	208.48	3.69	1.07	3.17+0.17	0.04+0.02

Table 2. Excerpt of experimental results with different instances of various problems (For CLP(FD) and \mathcal{B}_{MV}^{FD} we reported the time required for the constrain and the labelling phases).

allows the use of multivalued fluents and the use of constraints as conditions and consequences of actions. We illustrated the application of \mathcal{B}_{MV}^{FD} to two planning problems. Both languages have been implemented using SICStus Prolog.

The encoding in CLP(FD) allow us to think of extensions in several directions, such as encoding of qualitative and quantitative preferences (a preliminary study has been presented in [12]), introduction of global action constraints for improving efficiency (e.g., alldifferent among states), and use of constraints to represent incomplete states—e.g., to determine most general conditions for the existence of a plan and to conduct conformant planning [13]. We also believe that significant improvements in efficiency could be achieved by delegating parts of the constraint solving process to a dedicated solver (e.g., encoded using a constraint platform such as GECODE).

Acknowledgments This work is partially supported by MIUR projects PRIN05-015491 and FIRB03-RBNE03B8KK, and by NSF grants 0220590, 0454066, and 0420407. The authors wish to thank Tran Cao Son for the useful discussions and suggestions.

References

- [1] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [2] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
- [3] C. Baral, T.C. Son, and L.C. Tuan. A Transition Function based Characterization of Actions with Delayed and Continuous Effects. *KRR*, Morgan Kaufmann, 2002.
- [4] A. Dovier, A. Formisano, and E. Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In Proc. of *ICLP05*, LNCS 3668, pp. 67–82, 2005.
- [5] A. Dovier, A. Formisano, and E. Pontelli. An Empirical Study of CLP and ASP Solutions of Combinatorial Problems. *J. of Experimental & Theoretical Artificial Intelligence*, 2007.
- [6] M. Gelfond and V. Lifschitz. Action Languages. *Elect. Trans. Artif. Intell.* 2:193–210, 1998.
- [7] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In Proc. of *AAAI'04*, pp. 61–66, AAAI/Mit Press, 2004.
- [8] A.K. Jonsson, P.H. Morris, N. Muscettola, K. Rajan, and B.D. Smith. Planning in Interplanetary Space: Theory and Practice. In *AIPS*, 2002.
- [9] J. Lee and V. Lifschitz. Describing Additive Fluents in Action Language C+. *IJCAI*, 2003.
- [10] A. Lopez and F. Bacchus. Generalizing GraphPlan by Formulating Planning as a CSP. In Proc. of *IJCAI*, 2003.
- [11] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
- [12] T. Phan, T.C. Son, E. Pontelli. CPP: a Constraint Logic Programming based Planner with Preferences. *LPNMR*, Springer Verlag, 2007.
- [13] T. Phan, T.C.Son, C. Baral. Planning with Sensing Actions, Incomplete Information, and Static Causal Laws using Logic Programming. *TPLP* (to appear).
- [14] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, 2001.
- [15] P. Simons. Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Report 58, Helsinki University of Technology, 2000.
- [16] T.C. Son, C. Baral, and S.A. McIlraith. Planning with different forms of domain-dependent control knowledge. *LPNMR*, Springer Verlag, pp. 226–239, 2001.
- [17] M. Thielscher. Reasoning about Actions with CHRs and Finite Domain Constraints. In Proc. of *ICLP02*, LNCS 2401, pp. 70–84, 2002.

SLDNF-Draw: a Visualisation Tool of Prolog Operational Semantics

Marco Gavanelli

Engineering Department, University of Ferrara,
Via Saragat 1,
44100 Ferrara, Italy.

<http://www.ing.unife.it/docenti/MarcoGavanelli/>

Abstract. Logic Programming is a programming paradigm widely used for teaching Artificial Intelligence in university courses. Also, learning it is propaedeutical for understanding formal specification languages, widely accepted tools in software engineering. Many widely used Logic Programming languages (such as Prolog) adopt the SLDNF resolution as operational semantics.

However, after a first phase in which students learn how specifications can be translated into executable code, a second phase is necessary for engineering the resulting program. In this second phase, having a clear picture of the execution model is crucial.

In this paper, SLDNF-Draw, a program that visualises the SLDNF operational semantics, is presented. It has been successfully used in university courses to support teaching of Prolog.

1 Introduction

Logic programming languages are one of the most successful examples of declarative programming paradigms; besides the notable number of practical applications, they are widely used in Artificial Intelligence, and are often used in university courses as a valid tool to support learning of AI techniques [4].

In fact, many applications in AI represent the knowledge of the system in logics, and tree-search is widely used to find a solution. Logic programming languages provide efficient tree-search algorithms, and have transparent backtracking mechanisms. Moreover, in many logic programming languages (such as Prolog), the user is not required to adopt the predefined search method, nor to blindly accept the other engineered choices adopted by the language developers: the language lets the user easily redefine the interpreter by means of meta-interpretation. Practical applications range from solving combinatorial problems (for instance, Constraint Logic Programming is one of Logic Programming ribs and is effectively used by various companies, like British Airways, Cisco Systems, Airfrance, just to name a few) to typical Artificial Intelligence applications, like expert systems or machine learning.

However, learning Logic Programming requires a change in the approach to programming: while in imperative programming the focus is on algorithms

(*Algorithms + Data Structures = Programs* [14]), in declarative programming the control is intrinsically embedded in the language, and the programmer has only to formally state the logics of the program (*Algorithm = Logic + Control* [7]). Thus, algorithms become less important (at least at a first view), while giving correct specifications becomes crucial. This is exactly the goal of rapid prototyping: make specification and implementation (almost) coincide.

At a second view, however, the program must often be engineered and be written taking into account the underlying algorithm, in order to avoid infinite loops, or frustrating inefficiencies.

The reasons are many. Many real-life logic programming languages (such as Prolog and its dialects) are devised to let the user define efficient programs, possibly at the expense of completeness. Other logic programming languages (such as Answer Set Programming languages [11,8]) rely on a grounding of the program, so they are limited to programs that handle data structures of pre-defined size.

For example, Prolog skips the inefficient breadth-first search strategies, and prefers an incomplete but fast depth-first search strategy, that is prone to infinite loops if the knowledge base is written naively, but can be almost as efficient as an equivalent program written in an imperative language if the program is well-engineered.

The story is not new: students have to face and learn the engineering art of choosing the right tradeoff, and exploit the available tools at their full capabilities.

The problem in teaching Prolog to engineering students is that they often adopt an imperative viewpoint; they are usually very skilled in writing programs in C, or C++, or Java. Thus, they often start imagining the operational behaviour when they write a program, without thinking very much about the logic behind it. They become quite skilled in optimising Prolog programs, before fully understanding the logic programming style itself.

For instance, typically students think that the clause should be identified univocally, and not simply non-deterministically selected. Thus they often ask the teacher if there is a language statement, or instruction, that the programmer should use to identify the intended clause. The answer is twofold. In a first phase, the professor should convince the students that such a syntactic item is *not necessary* for a correct specification of the program. In a second phase, the available syntactic tools (like mode declarations, indexing, and even the cut) can be described, in order to improve the efficiency.

The learning process can be seen composed of two steps:

- learning to program in logics, forgetting about algorithms
- figuring out the algorithm that has been developed, and improving it.

Hermenegildo [12] has a similar viewpoint, and suggested to use in the first step a logic language with a complete search strategy, and to introduce standard Prolog, with its efficiency issues, in a second phase.

Learning, in both of the two phases, relies on a good understanding of the operational semantics. The operational semantics of Prolog is formalised through

SLDNF-resolution [1], i.e., SLD (Selected Literal Definite clause) resolution together with Negation as Failure. In SLD resolution every state in the computation is a node of a tree; the tree will be typically explored by depth-first search with backtracking. It is important, for students, to visualise such trees, understand which clauses will be selected, which will be cut. The Prolog tracer could be used, but its use is indeed limited for learning purposes.

This paper describes a tool that visualises SLDNF trees. The tool, called SLDNF Draw, is extensively used in teaching Prolog, and a set of exercises are chosen to make the students try the trickiest parts of the operational semantics. By seeing wrong programs, and correcting them, the students learn more efficiently the art, and engineering, of programming in Prolog.

The rest of the paper is organised as follows. First the specifications and features of SLDNF Draw are given in Section 2. Some Prolog exercises that have been proposed to students and visualised through SLDNF Draw are in Section 3. Related work is described (Section 4). Future extensions are reported in Section 5. Conclusions follow.

2 SLDNF Draw

SLDNF Draw [5] is a program that visualises SLDNF trees given a knowledge base and a goal. SLDNF Draw is a tool intended for teaching Prolog in engineering classes. Thus, the following requirements have been taken into account

Open source meta-interpretation SLDNF Draw should be a software that students can download, and use freely. Moreover, teachers want students to be able to look at the source code and understand how the program works. Students are encouraged to modify and improve it, as looking at the source code can be useful to learn techniques of Prolog programming. For this reason, it was important to develop SLDNF Draw as a meta-interpreter. In this way, students get acquainted to the Prolog environment, can study a real-life example of meta-interpreter, and can propose or develop improvements for the next releases of the program.

Full Prolog Syntax Prolog is sometimes defined as SLD Resolution plus Negation as Failure. Actually, Prolog is a full programming language with built-ins that can handle mathematics (the `is` predicate, relational operators, etc.), meta-predicates for aggregates (`findall`, `setof`, etc.), and various extra-logic predicates (like `var`, `==`, `copy_term`, etc.). All these constructs are part of the ISO Prolog and of most Prolog dialects, and are used in almost every non trivial application. For these reasons, SLDNF Draw has been developed to handle built-in predicates and easily extensible for other, future predicates.

Cut One of the difficult concepts for students to understand is the *cut* symbol (written “!”). This extra-logic predicate operationally removes some branches of

the SLDNF tree. Of course, cuts should be used as rarely as possible. The trend is to exclude the cut from new logic programming languages (even very efficient ones, like Mercury [13]), and to prefer operators having a similar operational semantics but a more understandable behaviour (like, for example, the `->/3` meta-predicate).

However, the cut is still part of almost all Prolog implementations, and of ISO Prolog as well. The cut is still widely used in Prolog programming, so understanding it is definitely important to understand existing software; moreover, it can be useful to engineer and profile the execution of a program.

For these reasons, it is important for students to understand the operational semantics of cut, and visualise the nodes that will be explored and the ones that will be not.

Output In order to be useful, the output format should be easy to visualise, and to zoom in case of large trees. It is desirable for the output format to be understandable and easy to convert to other graphic formats. \LaTeX was the chosen output format: it contains packets that visualise trees given the declarative representation, and it is easily convertible to Postscript and most vector formats, as well as to bitmap formats useful for the web.

Thanks to the meta-interpretation facilities of Prolog, the implementation is rather simple: it is a classical meta-interpreter that executes a Prolog program and, during execution of a node, saves on a file (as imperative side-effects) the \LaTeX instructions that generate the tree. In Fig. 1 we give a sketch of the meta-interpreter, where `init_X`, `close_X` (where X can be `box`, `node` or `subtree`) write on a file the \LaTeX instructions necessary for drawing the element X .

Some care was necessary to retrieve the names of the variables, in order to visualize the bindings. We used the library `var_name` of *ECLⁱPS^e* [3], that was developed for debugging purposes.

3 How to exploit SLDNF-Draw for understanding Prolog

A set of exercises are prepared for the students. Teachers provide the knowledge base (the Prolog program) and ask the students to visualise the corresponding SLDNF tree. Then, students are asked to comment on the result: whether the program is correct, how to correct it if it is not, suggest improvements if it could be made more efficient, and so on. Some of the proposed exercises are shown in the rest of this section.

3.1 Reversibility

One of the first examples in many Prolog courses is about the `member` predicate and Prolog reversibility. In logic programming, the predicate of membership of a list

```

draw([],File):-
    print_resolvent(File,"true").
draw([not(A)|B],File):-
    init_box(File),
    draw(A,File),
    close_box(File),
    draw(B,File).
draw([A|B],File):-
    init_subtree(File),
    print_resolvent(File,[A|B]),
    clause(A,Body),
    append(Body,B,Resolvent),
    init_node(File),
    draw(Resolvent,File),
    close_node(File),
    (draw(B,File) ; close_subtree(File), fail).

```

Fig. 1. Sketch of the SLDNF-Draw meta-interpreter.

```

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

```

can be obviously used to check if an element belongs to a list (Figure 2 left), but

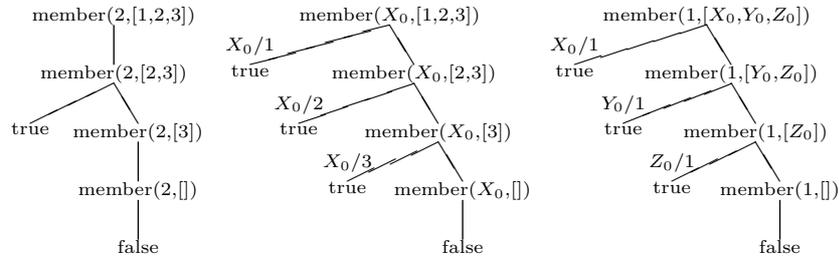


Fig. 2. `member` for checking membership, instantiating a variable, and building a list

can also be used to generate assignments (Figure 2 centre) for a variable (useful, for example, for implementing a generate-and-test pattern) or even for generating the elements of a list, or use lists as approximation of sets [10] (Figure 2 right).

3.2 Recursion and Last call optimisation

Most Prolog compilers are able to optimise tail-recursive predicates, with the so called *last call optimisation*. In this case, the SLD tree shows the difference

between a naively written predicate, like the following, that computes the length of a list (Figure 3 left):

```
len([],0).
len([H|T],N):- len(T,M), N is M+1.
```

and its tail recursive version (Figure 3 right):

```
lent([],N,N).
lent([H|T],Ni,No) :- Nt is Ni+1, lent(T,Nt,No).
```

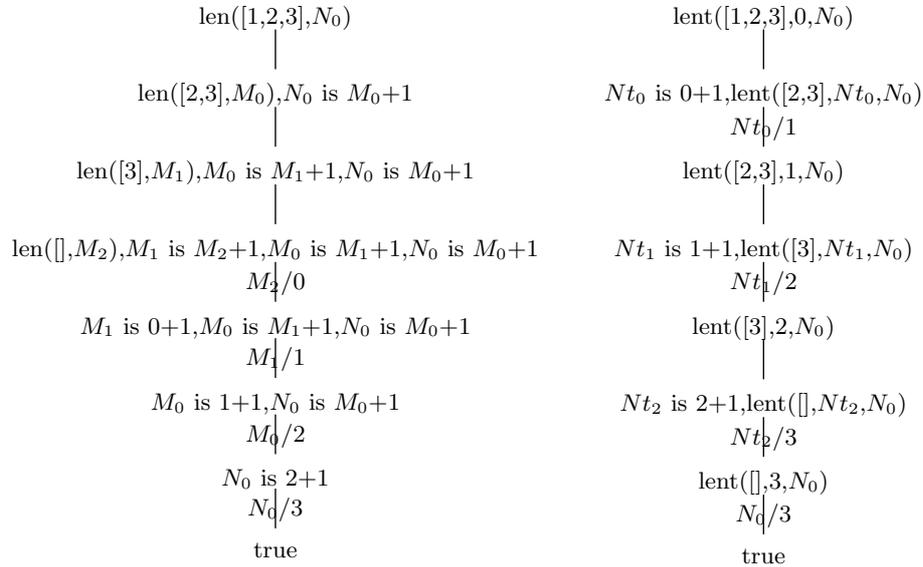


Fig. 3. Length of a list: recursive and tail recursive implementations

The differences are evident, and it is easy to convince students that the first implementation takes more memory by simply looking at the shapes of the trees: of course, the resolvent must be kept in memory in order to execute the program. Also, the resolvent shows the typical evolution of the stack in a recursive call: first all the activations are put in the stack, then the end condition is reached and the activation records are popped out; so the tree has a diamond-like shape. The tail-recursive implementation, instead, needs a constant number of activation records.

3.3 Cut

The cut is always difficult to understand fully, and even expert Prolog programmers can make mistakes. One of the proposed exercises gives this definition of the minimum of two numbers:

```

minw(A,B,A) :- A < B, !.
minw(A,B,B).

```

Such a definition seems perfectly reasonable at a first sight: the minimum is A if $A < B$ and it is B otherwise. But the SLD tree shows very clearly that the answers can be unsound: in Figure 4, the first tree shows how the answer to the question “*what is the minimum between 1 and 2*” is computed: there exists only one computed answer, which is 1. In the second tree of Figure 4, instead, the question is “*is 2 the minimum between 1 and 2?*”, and the answer, surprisingly, is *yes*.

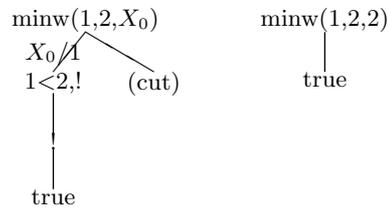


Fig. 4. Two invocations of the minimum predicate with cut give different (unsound) results

When facing this program, students are able to find a correct implementation of the `minw` predicate. Some of them think that the problem stands in the fact that the second clause has a wrong meaning, thus the solution should give a correct meaning to each of the clauses taken by themselves. In this way, the implementation is correct even without cut, and propose this solution:

```

minw(A,B,A) :- A < B.
minw(A,B,B) :- A >= B.

```

Others have a more imperative viewpoint, and suggest that the problem is in the first clause: the assignment on the output variable (that provides the minimum) is done too early, before the test $A < B$. Remembering the `if` instruction of imperative languages, they propose to postpone the assignment to the output variable after the test has been executed:

```

minw(A,B,M) :- A < B,!, M=A.
minw(A,B,B).

```

Both ideas are correct; students should be encouraged to propose alternative solutions, and select their preferred one, knowing the pros and cons of the various implementations.

3.4 Negation

In some cases the specifications of the program coincide exactly with the definition given in mathematics. Consider, for example, the minimum of a list. The

definition proposed to students in mathematics courses is the following: an element is the minimum of a set if it belongs to the set and there is no such element belonging to the set which is smaller.

$$\text{min}(S) = M \iff M \in S \wedge \neg(\exists X \in S, X < M).$$

The specifications are already executable:

```

minimumlist(M,L) :- member(M,L), not(smaller(L,M)).
smaller(L,M) :- member(X,L), X<M.

```

If the specifications are correct, the program is already correct; logics supports rapid prototyping. On the other hand, the execution model can be inefficient, as can be seen from the SLDNF tree (Fig. 5).

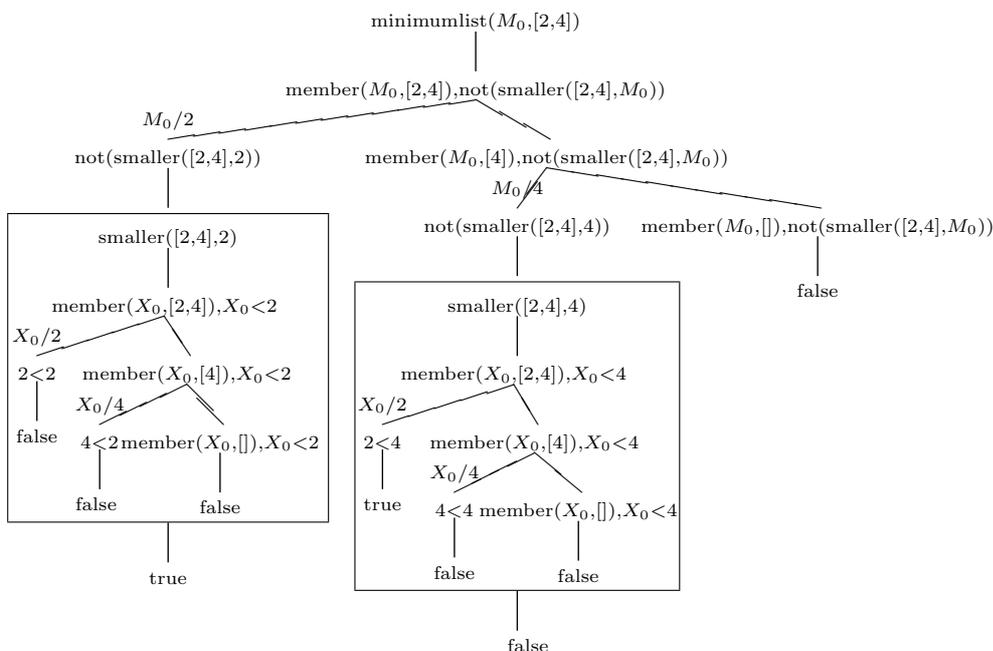


Fig. 5. Minimum of a list: implementation based on the definition.

Vast parts of the tree are evidently repeated, so one may think to optimise it by cutting some branches. Since the minimum value is unique, one can stop the search as soon as the minimum is found. Students are usually very keen in understanding which branches can be cut, and where the cut can be put in order not to change the semantics of the program (Figure 6).

As a second step, if the efficiency requirements are tight, it can be implemented with tail recursion, but this typically means figuring out an algorithm, not simply implementing specifications (Figure 7).

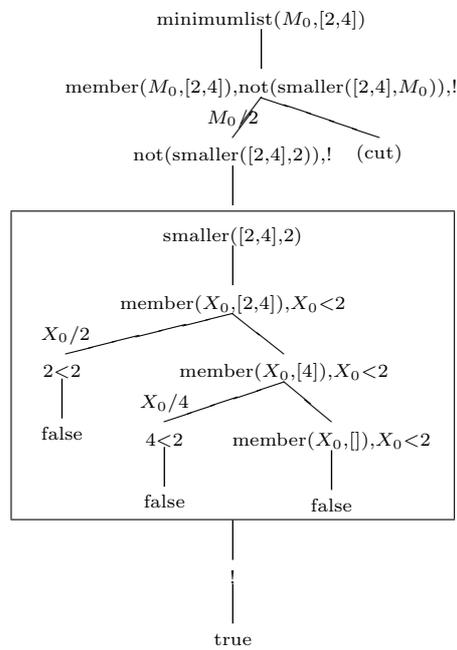


Fig. 6. Minimum of a list: implementation based on the definition, cutting redundant branches.

```

minimumlistt([],M,M).
minimumlistt([H|T],Mi,Mo):- H<Mi,!, minimumlistt(T,H,Mo).
minimumlistt([H|T],Mi,Mo):- H>=Mi, minimumlistt(T,Mi,Mo).

```

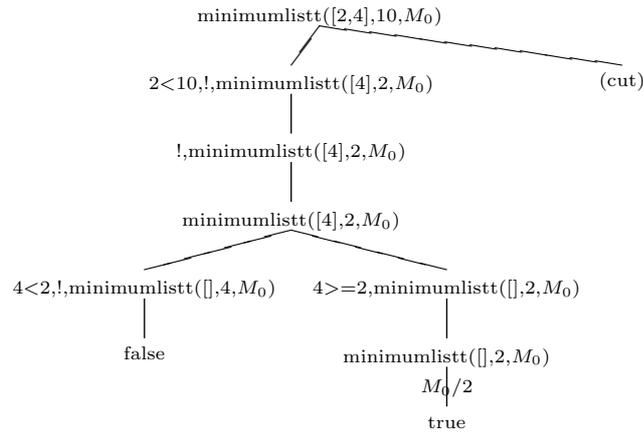


Fig. 7. Minimum of a list: tail recursive version.

3.5 Occur-check

Integer numbers can be defined in the very same way the students are taught in mathematics courses, i.e., from Peano axioms: an integer number is either zero (0) or a successor (s) of an integer number. Basic operations can be defined easily; for example the sum can be defined with two clauses, saying that $X + 0 = X$ and $s(A) + B = s(C)$ whenever it is known that $A + B = C$:

```

sum(0,X,X).
sum(s(A),B,s(C)):- sum(A,B,C).

```

Given such a definition, simple equations can be solved, as conjunction of goals. For example,

$$\begin{cases} X + Y = 3 \\ Y + 1 = X \end{cases}$$

is written as the goal `sum(X,Y,s(s(s(0))))`, `sum(Y,s(0),X)`, and the Prolog interpreter correctly provides $X = s(s(0))$ and $Y = s(0)$.

However, the interpreter can give wrong results if the occur-check is turned off, as the following example shows (Figure 8 left):

$$\begin{cases} 1 + Y = X \\ 0 + X = Y \end{cases}$$

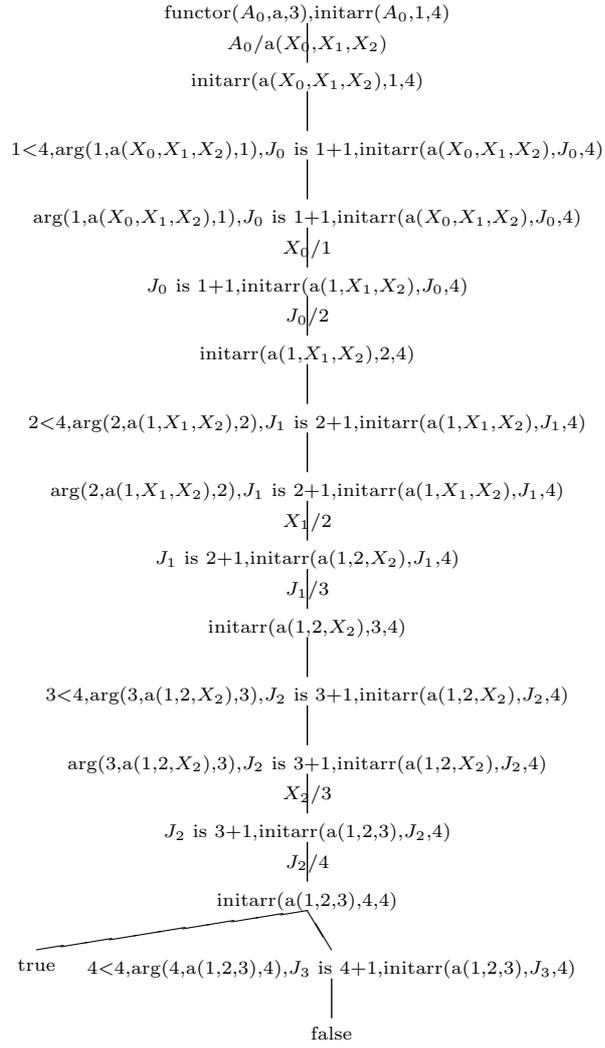


Fig. 9. Example with arrays

SLD Draw [9] draws on-screen the SLD tree of a goal. The source code is not provided; it does not handle negation, although it can show the cut. However, a good understanding of negation is fundamental in Prolog programming, so using a standard representation of negation in SLDNF trees is important for students. Moreover, having available the source code is useful in order to understand how the program works.

CI Space [2] is a set of tools for learning computational intelligence. Among others, it contains a Java applet for drawing SLD trees. It has a limited support for negation and no support for built-ins.

Showing the behaviour of the full Prolog language is important. As noted by Ducassé [12], especially for students who are not seeking academic careers, Prolog is viewed as a toy language (too small and too simple) and is not appreciated as a real programming language. So, focusing only on SLD resolution with Negation as Failure without considering the built-ins could be too restrictive. Built-ins are important to show the full capabilities of Prolog. In particular, engineering students should know the built-ins that can improve efficiency: e.g., handling arithmetics is a must, and arrays are very useful. A didactic program should show how these built-in predicates behave, and how they interact with the other predicates. Stated otherwise, a visualisation program should show the trees for a full Prolog syntax, not just limit itself to SLD resolution.

5 Future work

In future work, we aim at extending the visualisation capabilities of SLDNF Draw. Some improvements have been proposed by students, and include constraint propagation and branch-and-bound (in order to visualise Constraint Logic Programming [6] derivations).

Another interesting extension would concern providing a more dynamic output: the exploration of the tree could be given incrementally, interactively or through animations. This could be performed by invoking a visualisation software from Prolog, or the animation could be saved in a SVG or Flash format.

6 Conclusion

This work presented SLDNF Draw, a program that draws the operational semantics of Prolog. It is developed as a meta-interpreter, and it is available for the students to look at its source code. It can be freely downloaded from

<http://www.ing.unife.it/software/sldnfDraw/>

It has been used in university courses for teaching logic programming and, in particular, the features of Prolog. SLDNF Draw handles the full syntax of Prolog, it is not restricted to SLD resolution, but can handle negation, cut, built-in predicates (like arithmetic or aggregation predicates), which are necessary in real-life applications.

In our experience, students find the program useful to understand the operational semantics of Prolog: they use it for making experiments, for understanding which implementations are more efficient, and for visualising the Prolog execution.

Acknowledgements

This work has been partially supported by the MIUR PRIN 2005 project n. 2005-015491 *Constraints and preferences as a unifying formalism for system analysis and solution of real-life problems*.

References

1. K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
2. M. Cline, W. Coelho, K. O’Neill, M. Pavlin, J. R. Santos, S. Sueda, L. Tung, A. Yap, C. Conati, P. Gorniak, H. Hoos, A. Mackworth, and D. Poole. CIspace: Tools for learning computational intelligence. <http://www.cs.ubc.ca/labs/lci/CIspace/>, Nov. 2003.
3. ECRC and IC-Parc. *ECLⁱPS^e User Manual, Release 5.2*. IC-Parc, Imperial College, London, UK, 2001.
4. O. Garcia, R. Perez, B. Silverman, H. Austin, R. Baum, L. Brady, R. Cameron, S. Castaneda, J. Chen, P. Dey, G. DiCristina, A. Elmaghraby, R. Foster, C. Freeman, M. Kirch, A. Lawrence, A. Manesh, S. Manickam, C. Ramamoorthy, R. Rariden, U. Reichenbach, S. Rosenbaum, F. Saner, F. Severance, C. Torsonone, D. Valentine, H. V. Landingham, and R. Vasquez. On teaching AI and expert system courses. *IEEE Trans. Educ.*, 36(1):193–197, Feb. 1993.
5. M. Gavanelli. SLDNF Draw web page. <http://www.ing.unife.it/software/sldnfDraw/>, 2004.
6. J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.
7. R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.
8. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
9. F. G. López. SLD draw. <http://polaris.lcc.uma.es/~pacog/sldDraw/>, Oct. 2003.
10. T. Munakata. Notes on implementing sets in Prolog. *Communications of the ACM*, 35(3):112–120, Mar. 1992.
11. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR’97*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.
12. E. Pontelli. Teaching (constraint) logic programming panel at ICLP 2003. *ALP Newsletter*, 17(1), Feb. 2004.

13. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, Oct.-Dec. 1996.
14. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

Decision algorithms for fragments of real analysis. II. A theory of differentiable functions with convexity and concavity predicates^{*}

Domenico Cantone and Gianluca Cincotti

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

Abstract. We address the decision problem for a fragment of real analysis involving differentiable functions with continuous derivative. The proposed theory, besides the operators of Tarski's theory of reals, includes predicates for comparisons, monotonicity, convexity and derivative of functions over bounded closed intervals or unbounded intervals. Our decision algorithm is obtained by showing that satisfiable formulae of our theory admit canonical models in which functional variables are interpreted as piecewise exponential functions. These can be implicitly described within the decidable Tarski's theory of reals. Our satisfiability test generalizes previous decidability results not involving derivative operators.

Key words: Automated theorem proving, decision procedures, elementary real analysis.

1 Introduction

Verification of floating point hardware and hybrid systems has given an important impulse to the formalization in computerized environments of the theory of reals and some of its extensions. In this connection, among others, we cite the work on theorem proving with the real numbers using a version of the HOL theorem prover [12], the mechanization of real analysis in Isabelle/HOL [9], in PVS [10], and in the interactive proof system IMPS [11], the ongoing efforts with the Mizar system [1, 15], the attempt to formalize Cauchy's integral theorem and integral formula in the EtnaNova proof-verifier [20, 5, 16], and so on.

To keep within reasonable limits the amount of details that a user must provide to a verification system in proof sessions, it is necessary that the verifier has a rich endowment of decision procedures, capable to formalize "obvious" deduction steps. Thus, a proof verifier for real analysis should include in its inferential kernel a decision procedure for Tarski's elementary theory of reals [22] as well as efficient decision tests for more specialized subtheories such as

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

the existential theory of reals [13], the theory of bounded and stable quantified constraints [18], and other even more specific classes of constraints.

In some situations, one may also need to reason about real functions, represented in the language as interpreted or uninterpreted function symbols.¹ However, one must be aware that the existential theory of reals extended with the interpreted symbols $\log 2$, π , e^x , and $\sin x$ is undecidable [19]. On the other hand, it has been shown in [14] that the first-order theory of the real numbers extended with the exponential function e^x is decidable, provided that Schanuel’s conjecture in transcendental number theory holds [6, Chapter 3, pp. 145-176].

The existential theory of reals has been extended in [4] with uninterpreted continuous function symbols, function sum, function point evaluation, and with predicates expressing comparison, monotonicity (strict and non-strict) and non-strict convexity of functions. Such decidability result has been further extended in [3], where a decision algorithm for the sublanguage RMCF^+ (augmented theory of Reals with Monotone and Convex Functions) of elementary analysis has been given. The theory RMCF^+ consists of the propositional combination of predicates related to certain properties of continuous functions which must hold in bounded or unbounded intervals. More precisely, the arguments of RMCF^+ ’s predicates are numerical and functional terms. Numerical variables are interpreted by real numbers, whereas functional variables are interpreted by everywhere defined continuous real functions of one real variable. Furthermore, numerical terms are obtained by composing numerical variables through the basic arithmetic operations, whereas functional terms are obtained by composing functional variables through the addition (or subtraction) operator over functions.

In this paper, we consider a new theory, RDF (theory of Reals and Differentiable Functions), which in part extends the RMCF^+ theory as it allows continuous and differentiable (with continuous derivative) real functions as interpretation of functional variables. The RDF theory contains all RMCF^+ ’s predicates plus other predicates concerning first order derivative of functions.

The arguments of RDF’s predicates are numerical terms and functional variables (this is the only restriction with respect to the theory RMCF^+ , since additive functional terms are not currently allowed in RDF).

We will show that the RDF theory is decidable by generalizing the proof techniques used in [3, 4]. In particular, the decision algorithm is obtained by exploiting the fact that any satisfiable RDF-formula admits a model in which functional variables are interpreted by parametric piecewise exponential functions. Since such functions can be implicitly described by existential formulae of Tarski’s theory of reals, the decidability of RDF follows.

¹ Interpreted function symbols have a predefined interpretation (e.g., the exponential and the sinus functions, e^x and $\sin x$, respectively), whereas uninterpreted function symbols have no predefined meaning attached to them and therefore they can be interpreted freely (e.g. the “generic” function symbols f and g).

The paper is organized as follows. In Section 2 we present the syntax and semantics of RDF-formulae. Then in Section 3 we review the normalization process for unquantified formulae of first order theory with equality and introduce the notion of ordered RDF-formulae. Section 4 reports a decision algorithm for RDF theory and a sketch proof of its correctness. Final remarks and open problems are given in the last section.

2 The RDF theory

In this section we introduce the language of the theory RDF (Reals with Differentiable Functions) and give its intended semantics.

Two different kinds of variables can occur in RDF. *Numerical variables*, denoted with x, y, \dots , are used to represent real numbers; whereas, *functional variables*, denoted with f, g, \dots , are used to represent continuous and differentiable (with continuous derivative) real functions. RDF-formulae can also involve the following constant symbols:

- $0, 1$, which are interpreted as the real numbers $0, 1$, respectively;
- $\mathbf{0}, \mathbf{1}$, which are interpreted as the null function and the 1-constant function, respectively.

The language RDF includes also two distinguished symbols: $-\infty, +\infty$; they cannot occur everywhere in RDF formulae but only as “range defining” parameters as it will be clear from the following definitions.

Definition 1. Numerical terms *are recursively defined by:*

1. every numerical variable x, y, \dots or constant $0, 1$ is a numerical term;
2. if t_1, t_2 are numerical terms, then $(t_1 + t_2), (t_1 - t_2), (t_1 * t_2)$, and (t_1/t_2) are numerical terms;
3. if t is a numerical term and f is a functional variable, then $f(t)$ and $D[f](t)$ are numerical terms;
4. an expression is a numerical term only if it can be shown to be so on the basis of 1, 2, 3 above.

In the following we will use the term “functional variable” to refer either to a functional variable or to a functional constant.

Definition 2. An extended numerical variable (*resp. term*) is a numerical variable (*resp. term*) or one of the symbols $-\infty$ and $+\infty$.

Definition 3. An RDF-atom is an expression having one of the following forms:

$$\begin{aligned}
& t_1 = t_2, \quad t_1 > t_2, \\
& (f = g)_{[s_1, s_2]}, \quad (f > g)_{[t_1, t_2]}, \\
& (D[f] = t)_{[s_1, s_2]}, \quad (D[f] > t)_{[s_1, s_2]}, \quad (D[f] \geq t)_{[s_1, s_2]}, \\
& (D[f] < t)_{[s_1, s_2]}, \quad (D[f] \leq t)_{[s_1, s_2]}, \\
& \text{Up}(f)_{[s_1, s_2]}, \quad \text{Strict_Up}(f)_{[s_1, s_2]}, \\
& \text{Down}(f)_{[s_1, s_2]}, \quad \text{Strict_Down}(f)_{[s_1, s_2]}, \\
& \text{Convex}(f)_{[s_1, s_2]}, \quad \text{Strict_Convex}(f)_{[s_1, s_2]}, \\
& \text{Concave}(f)_{[s_1, s_2]}, \quad \text{Strict_Concave}(f)_{[s_1, s_2]},
\end{aligned}$$

where t, t_1, t_2 are numerical terms, f, g are functional variables, and s_1, s_2 are extended numerical terms such that $s_1 \neq +\infty$ and $s_2 \neq -\infty$.

Definition 4. An RDF-formula is any propositional combination of RDF-atoms.

The semantics of RDF-formulae is defined as follows.

Definition 5. A real model for an RDF-formula is an interpretation M such that:

1. For any numerical variable x , the value Mx is a real number.
2. For any functional variable f , (Mf) is an everywhere defined differentiable real function of one real variable with continuous derivative.
3. For any composite numerical term $t_1 \otimes t_2$, $M(t_1 \otimes t_2)$ is the real number $Mt_1 \otimes Mt_2$, where $\otimes \in \{+, -, *, /\}$.
4. For any numerical term $f(t)$, $M(f(t))$ is the real number $(Mf)(Mt)$.
5. For any numerical term $D[f](t)$, $M(D[f](t))$ is the real number $D[(Mf)](Mt)$.
6. Let t, t_1, t_2 be numerical terms, let f, g be functional variables and assume that $Mt, Mt_1, Mt_2, (Mf), (Mg)$ are respectively their interpretations by M . Let s_1, s_2 be extended numerical terms and let Ms_1, Ms_2 be their associated interpretations, where $Ms_i, i = 1, 2$ is a real number as above if s_i is a numerical term; otherwise Ms_i is the symbol $-\infty$ (resp. $+\infty$) if $s_i = -\infty$ (resp. $+\infty$).

RDF-atoms are interpreted by a given real model M according to the following rules:

- (a) $t_1 = t_2$ (resp. $t_1 > t_2$) is true if and only if $Mt_1 = Mt_2$ (resp. $Mt_1 > Mt_2$);
- (b) $(f = g)_{[s_1, s_2]}$ is true² if and only if $Ms_1 > Ms_2$, or $Ms_1 \leq Ms_2$ and $(Mf)(x) = (Mg)(x)$ for every $x \in [Ms_1, Ms_2]$;
- (c) $(f > g)_{[t_1, t_2]}$ is true if and only if $Mt_1 > Mt_2$, or $Mt_1 \leq Mt_2$ and $(Mf)(x) > (Mg)(x)$ for every $x \in [Mt_1, Mt_2]$;
- (d) $(D[f] \bowtie t)_{[s_1, s_2]}$, with $\bowtie \in \{=, >, \geq, <, \leq\}$, is true if and only if $Ms_1 > Ms_2$, or $Ms_1 \leq Ms_2$ and $D[(Mf)](x) \bowtie Mt$ for every $x \in [Ms_1, Ms_2]$;

² With some abuse of notation, bounded and unbounded intervals are represented with the same formalism $[a, b]$. Furthermore, we assume $-\infty < +\infty$.

- (e) $Up(f)_{[s_1, s_2]}$ (resp. *Strict_Up*) is true if and only if $Ms_1 \geq Ms_2$, or $Ms_1 < Ms_2$ and the function (Mf) is monotone nondecreasing (resp. strictly increasing) in the interval $[Ms_1, Ms_2]$;
- (f) $Convex(f)_{[s_1, s_2]}$ (resp. *Strict_Convex*) is true if and only if $Ms_1 \geq Ms_2$, or $Ms_1 < Ms_2$ and the function (Mf) is convex (resp. strictly convex) in the interval $[Ms_1, Ms_2]$;
- (g) the truth value of $Down(f)_{[s_1, s_2]}$ (resp. *Strict_Down*) and $Concave(f)_{[s_1, s_2]}$ (resp. *Strict_Concave*) are defined in a manner completely analogous to the above definitions (e) and (f).

One can not expect that any deep theorem of real analysis can be directly expressed by an RDF-formula, and therefore automatically verified. Indeed, our decidability result is to be regarded as just one more step towards the mechanization of the “obvious”, which is basic for the realization of powerful interactive proof verifiers in which the user assumes control only for the more challenging deduction steps (such as, for instance, the instantiation of quantified variables), otherwise leaving the burden of the verification of small details to the system.

We give next a few examples of statements which could be verified automatically by our proposed decision test for RDF.

Example 1 Let f be a real differential function in a closed interval $[a, b]$ with continuous derivative such that $f(a) = f(b)$, $f'(a) \neq 0$, and $f'(b) \neq 0$. Then there exists some $a < c < b$ such that $f'(c) = 0$. (This is a weakened version of Rolle’s theorem.)

A possible formalization of the above statement is given by the universal closure of the formula:

$$(a < b \wedge f(a) = f(b) \wedge D[f](a) \neq 0 \wedge D[f](b) \neq 0) \longrightarrow (\exists c)(a < c < b \wedge D[f](a) = 0),$$

whose theoremhood can be tested by showing that the following RDF-formula is unsatisfiable:

$$(a < b \wedge f(a) = f(b) \wedge D[f](a) \neq 0 \wedge D[f](b) \neq 0) \wedge ((D[f] > 0)_{[a, b]} \vee (D[f] < 0)_{[a, b]}).$$

□

Example 2 Let f be a real differential function in a closed interval $[a, b]$ with constant derivative in $[a, b]$. Then the function f is linear in $[a, b]$.

A possible formalization of the above statement is given by the universal closure of the formula:

$$(D[f] = t)_{[a, b]} \longrightarrow \left(Convex(f)_{[a, b]} \wedge Concave(f)_{[a, b]} \right),$$

whose theoremhood can be tested by showing that the following RDF-formula is unsatisfiable:

$$(D[f] = t)_{[a,b]} \wedge (\neg \text{Convex}(f)_{[a,b]} \vee \neg \text{Concave}(f)_{[a,b]}) .$$

□

In the rest of the paper we will present a satisfiability test for RDF.

3 The normalization process

Our decidability test makes use of the following general normalization process (cf. [4]). Let T be an unquantified first order theory, with equality $=$, variables x_1, x_2, \dots , function symbols f_1, f_2, \dots and predicate symbols P_1, P_2, \dots .

Definition 6. A formula φ of T is in normal form if:

1. every term occurring in φ is a variable or has the form $f(x_1, x_2, \dots, x_n)$, where x_1, x_2, \dots, x_n are variables and f is a function symbol;
2. every atom in φ is in the form $x = t$ where x is a variable and t is a term, or in the form $P(x_1, x_2, \dots, x_n)$, where x_1, x_2, \dots, x_n are variables and P is a predicate symbol.

Lemma 1. There is an effective procedure to transform any formula φ in T into an equisatisfiable formula ψ in normal form.

Proof. See [4, Lemma 2.2].

□

Definition 7. A normal form formula φ of T is in standard normal form if it is a conjunction of literals of the kinds:

$$\begin{aligned} x = y, \quad x = f(x_1, \dots, x_n), \quad x \neq y \\ P(x_1, x_2, \dots, x_n), \quad \neg P(x_1, x_2, \dots, x_n). \end{aligned}$$

where $x, y, x_1, x_2, \dots, x_n$ are variables, f is a function symbol, and P is a predicate symbol.

Let S be the class of all formulae of T in standard normal form. Then we have:

Lemma 2. T is decidable if and only if S is decidable.

Proof. See [4, Lemma 2.4].

□

Next, we describe the standard normal form for the RDF theory.

First of all, we observe that the following equivalences hold:

$$\begin{aligned}
t_1 = t_2 - t_3 &\equiv t_2 = t_1 + t_3, \\
t_1 = t_2/t_3 &\equiv (t_3 \neq 0) \wedge (t_2 = t_1 * t_3), \\
t_1 \neq t_2 &\equiv (t_2 > t_1) \vee (t_1 > t_2), \\
t_1 \not> t_2 &\equiv (t_1 = t_2) \vee (t_2 > t_1), \\
t_1 > t_2 &\equiv (t_1 = t_2 + v) \wedge (v > 0),
\end{aligned}$$

where t_1, t_2, t_3 stand for numerical terms and v stands for a new numerical variable. Moreover, by recalling that function symbols are modeled by differentiable functions (with continuous derivative), the following equivalences hold:

$$\begin{aligned}
\text{Up}(f)_{[s_1, s_2]} &\equiv (D[f] \geq 0)_{[s_1, s_2]}, \\
\text{Down}(f)_{[s_1, s_2]} &\equiv (D[f] \leq 0)_{[s_1, s_2]},
\end{aligned}$$

where s_1, s_2 are extended numerical terms.³

By applying the elementary normalization process hinted to above, we can consider, w.l.o.g., only formulae in *standard normal form* whose literals are of the following kinds:

$$\begin{aligned}
&x = y + w, \quad x = y * w, \quad x > 0, \quad y = f(x), \quad y = D[f](x), \\
&(f = g)_{[z_1, z_2]}, \quad (f \neq g)_{[z_1, z_2]}, \\
&(f > g)_{[w_1, w_2]}, \quad (f \not> g)_{[w_1, w_2]}, \\
&(D[f] \bowtie y)_{[z_1, z_2]}, \quad (D[f] \not\bowtie y)_{[z_1, z_2]}, \quad \text{where } \bowtie \in \{=, >, \geq, <, \leq\}, \\
\text{Strict_Up}(f)_{[z_1, z_2]}, \quad \neg \text{Strict_Up}(f)_{[z_1, z_2]}, \quad (\text{resp. Strict_Down}), \\
\text{Convex}(f)_{[z_1, z_2]}, \quad \neg \text{Convex}(f)_{[z_1, z_2]}, \quad (\text{resp. Strict_Convex}), \\
\text{Concave}(f)_{[z_1, z_2]}, \quad \neg \text{Concave}(f)_{[z_1, z_2]}, \quad (\text{resp. Strict_Concave}),
\end{aligned}$$

where x, y, w, w_1, w_2 are numerical variables, z_1, z_2 are extended numerical variables,⁴ and f, g are functional variables.

The decision algorithm to be presented in Section 4 requires one more preparatory step.

Let φ be a formula of the RDF theory.

Definition 8. A domain variable for a formula φ is a numerical variable such that it is either the argument of some functional variable (e.g., x in $f(x)$ or in $D[f](x)$) or it is the argument of the range defining parameters of some interval mentioned in φ (e.g., x and y in $\text{Convex}(f)_{[x, y]}$).

We consider a *strict linear ordering* of the domain variables:

³ Observe that strict-monotonicity predicates can not be treated in the same way.

⁴ By Definition 3, we have $z_1 \neq +\infty$ and $z_2 \neq -\infty$.

Definition 9. Let $D = \{x_1, x_2, \dots, x_n\}$ be the set of domain variables of φ . A formula φ is said to be ordered if:

$$\varphi \text{ is satisfiable} \iff \varphi \wedge \bigwedge_{i=1}^{n-1} (x_i < x_{i+1}) \text{ is satisfiable.}$$

The family RDF_{ord} of all the ordered formulae in RDF is a proper subset of RDF. It is straightforward to prove that:

Lemma 3. RDF is decidable if and only if RDF_{ord} is decidable. \square

In the rest of the paper, formulae of RDF will always be assumed to be ordered and in standard normal form.

4 The decision algorithm

In this section we present a decision algorithm which solves the satisfiability problem for RDF-formulae. The algorithm takes as input an ordered RDF-formula φ in standard normal form and reduces it, through a sequence of effective and satisfiability preserving transformations $\varphi \rightsquigarrow \varphi_1 \rightsquigarrow \varphi_2 \rightsquigarrow \varphi_3$, into a formula $\psi = \varphi_3$ of the existential Tarski's theory of reals, i.e., an existentially quantified formula involving only real variables, the arithmetic operators $+$, $*$, and the predicates $=$, $<$.

From the decidability of Tarski's theory of reals (see [22]; see also [7]), the decidability of RDF follows immediately.

In the following, w_i denotes a numerical variable, whereas z_i denotes an extended numerical variable.

In view of the results mentioned in the previous section, without loss of generality we can assume that our input formula φ is in standard normal form and ordered. The sequence of transformations needed to go from φ to ψ is given by the following:

Reduction algorithm for RDF.

- **Input** : an ordered formula φ of RDF in standard normal form.
- **Output** : a formula φ_3 of the Tarski's theory of reals which is equisatisfiable with φ .

The algorithm involves the following three fundamental steps:

1. $\varphi \rightsquigarrow \varphi_1$: *Negative clauses removal.*

Given an ordered formula φ of RDF in standard normal form, we construct an equisatisfiable formula φ_1 involving only positive predicates. The general idea applied in this step is to substitute every negative clause involving a functional symbol with an implicit existential assertion.

For the sake of simplicity, in the following we use the relation $x \preceq y$ as a shorthand for

$$x \preceq y \equiv_{\text{Def}} \begin{cases} x \leq y & \text{if } x \text{ and } y \text{ are both numerical variables} \\ \mathbf{true} & \text{if } x = -\infty \text{ or } y = +\infty \\ \mathbf{false} & \text{if } x = +\infty \text{ and } y = -\infty. \end{cases}$$

- (a) For any literal of type $(f \neq g)_{[z_1, z_2]}$ occurring in φ , introduce three new numerical variables x, y_1, y_2 and replace $(f \neq g)_{[z_1, z_2]}$ by the formula:

$$\Gamma \wedge y_1 \neq y_2,$$

where

$$\Gamma \equiv (z_1 \preceq x \preceq z_2) \wedge y_1 = f(x) \wedge y_2 = g(x).$$

- (b) Replace any literal of type $(f \not\leq g)_{[w_1, w_2]}$ occurring in φ by the formula:

$$\Gamma \wedge y_1 \leq y_2,$$

where

$$\Gamma \equiv (w_1 \leq x \leq w_2) \wedge y_1 = f(x) \wedge y_2 = g(x).$$

and x, y_1, y_2 are new numerical variables.

- (c) Replace any literal of type $(D[f] \neq y)_{[z_1, z_2]}$ (resp. $\not\leq, \not\geq, \not<, \not>$) occurring in φ by the formula:

$$\Gamma \wedge y_1 \neq y \quad (\text{resp. } \leq, <, \geq, >),$$

where

$$\Gamma \equiv (z_1 \preceq x \preceq z_2) \wedge y_1 = D[f](x),$$

and x, y_1 are new numerical variables.

- (d) Replace any literal of type $\neg\text{Up}(f)_{[z_1, z_2]}$ (resp. $\neg\text{Strict_Up}(f)_{[z_1, z_2]}$) occurring in φ by the formula:

$$\Gamma \wedge y_1 > y_2 \quad (\text{resp. } \Gamma \wedge y_1 \geq y_2),$$

where

$$\Gamma \equiv (z_1 \preceq x_1 < x_2 \preceq z_2) \wedge \bigwedge_{i=1}^2 y_i = f(x_i),$$

and x_1, x_2, y_1, y_2 are new numerical variables.

The case of literals of the form $\neg\text{Down}(f)_{[z_1, z_2]}$ (resp. $\neg\text{Strict_Down}(f)_{[z_1, z_2]}$) can be handled similarly.

- (e) Replace any literal of type $\neg\text{Convex}(f)_{[z_1, z_2]}$ (resp. $\neg\text{Strict_Convex}(f)_{[z_1, z_2]}$) occurring in φ by the formula:⁵

$$\Gamma \wedge (y_2 - y_1)(x_3 - x_1) > (x_2 - x_1)(y_3 - y_1)$$

⁵ The formula asserts the existence of three points x_1, x_2, x_3 , such that $(x_2, f(x_2))$ lies above (resp. $(x_2, f(x_2))$ does not lie below) the straight line joining the two points $(x_1, f(x_1))$ and $(x_3, f(x_3))$.

$$\text{(resp. } \Gamma \wedge (y_2 - y_1)(x_3 - x_1) \geq (x_2 - x_1)(y_3 - y_1)\text{),}$$

where,

$$\Gamma \equiv (z_1 \preceq x_1 < x_2 < x_3 \preceq z_2) \wedge \bigwedge_{i=1}^3 y_i = f(x_i)$$

and $x_1, x_2, x_3, y_1, y_2, y_3$ are new numerical variables.

The case of literals of the form $\neg\text{Concave}(f)_{[z_1, z_2]}$ (resp. $\neg\text{Strict_Concave}(f)_{[z_1, z_2]}$) can be handled similarly.

It is straightforward to prove that the formulae φ and φ_1 are equisatisfiable.

At this point, by normalizing φ_1 and in view of Lemma 2, we can assume without loss of generality that φ_1 is in standard normal form, i.e., it is a conjunction of literals of the following types:

$$\begin{array}{llll} x = y + w, & x = y * w, & x > 0, & y = f(x), \quad y = D[f](x), \\ (f = g)_{[z_1, z_2]}, & (f > g)_{[w_1, w_2]}, & & \\ (D[f] \bowtie y)_{[z_1, z_2]}, & \text{where } \bowtie \in \{=, >, \geq, <, \leq\}, & & \\ \text{Strict_Up}(f)_{[z_1, z_2]}, & \text{Strict_Down}(f)_{[z_1, z_2]}, & & \\ \text{Convex}(f)_{[z_1, z_2]}, & \text{Strict_Convex}(f)_{[z_1, z_2]}, & & \\ \text{Concave}(f)_{[z_1, z_2]}, & \text{Strict_Concave}(f)_{[z_1, z_2]}. & & \end{array}$$

Furthermore, in view of Lemma 3, we may further assume that φ_1 is ordered with domain variables v_1, v_2, \dots, v_r .

2. $\varphi_1 \rightsquigarrow \varphi_2$: *Explicit evaluation of functions over domain variables.*

This step is in preparation of the elimination of functional clauses.

For every domain variable v_j and for every functional variable f occurring in φ_1 , introduce two new numerical variables y_j^f, t_j^f and add the literals $y_j^f = f(v_j)$ and $t_j^f = D[f](v_j)$ to φ_1 . Moreover, for each literal $x = f(v_j)$ already occurring in φ_1 , add the literal $x = y_j^f$; likewise, for each literal $x = D[f](v_j)$ already occurring in φ_1 , add the literal $x = t_j^f$.

In this way, we obtain a new formula φ_2 which is clearly equisatisfiable to φ_1 .

3. $\varphi_2 \rightsquigarrow \varphi_3$: *Functional variables removal.*

In this step all literals containing functional variables are eliminated.

Let $V = \{v_1, v_2, \dots, v_r\}$ be the collection of the domain variables of φ_2 with their implicit ordering, and let the index function $ind : V \cup \{-\infty, +\infty\} \mapsto \{1, 2, \dots, r\}$ be defined as follows:

$$ind(x) =_{\text{Def}} \begin{cases} 1 & \text{if } x = -\infty, \\ l & \text{if } x = v_l, \text{ for some } l \in \{1, 2, \dots, r\}, \\ r & \text{if } x = +\infty. \end{cases}$$

For each functional symbol f occurring in φ_2 , let us introduce the new numerical variables γ_0^f, γ_r^f and proceed as follows:

- (a) For each literal of type $(f = g)_{[z_1, z_2]}$ occurring in φ_2 , add the literals:

$$y_i^f = y_i^g, \quad t_i^f = t_i^g,$$

for $i \in \{ind(z_1), \dots, ind(z_2)\}$; moreover, if $z_1 = -\infty$, add the literal:

$$\gamma_0^f = \gamma_0^g;$$

likewise, if $z_2 = +\infty$, add the literal:

$$\gamma_r^f = \gamma_r^g.$$

- (b) For each literal of type $(f > g)_{[w_1, w_2]}$ occurring in φ_2 , add the literal:

$$y_i^f > y_i^g,$$

for $i \in \{ind(w_1), \dots, ind(w_2)\}$.

- (c) For each literal of type $(D[f] \bowtie y)_{[z_1, z_2]}$ occurring in φ_2 , where $\bowtie \in \{=, <, \leq, >, \geq\}$, add the formulae:

$$t_i^f \bowtie y,$$

$$\frac{y_{j+1}^f - y_j^f}{v_{j+1} - v_j} \bowtie y,$$

for $i, j \in \{ind(z_1), \dots, ind(z_2)\}$, $j \neq ind(z_2)$. Additionally, if $\bowtie \in \{\leq, \geq\}$ add also the formulae:

$$\left(\frac{y_{j+1}^f - y_j^f}{v_{j+1} - v_j} = y \right) \longrightarrow (t_j^f = y \wedge t_{j+1}^f = y);$$

moreover, if $z_1 = -\infty$, add the formula:

$$\gamma_0^f \bowtie y,$$

and if $z_2 = +\infty$, add the formula:

$$\gamma_r^f \bowtie y.$$

- (d) For each literal of type $\text{Strict_Up}(f)_{[z_1, z_2]}$ (resp. $\text{Strict_Down}(f)_{[z_1, z_2]}$) occurring in φ_2 , add the formulae:

$$t_i^f \geq 0 \quad (\text{resp. } \leq),$$

$$y_{j+1}^f > y_j^f \quad (\text{resp. } <),$$

for $i, j \in \{ind(z_1), \dots, ind(z_2)\}$, $j \neq ind(z_2)$. Moreover, if $z_1 = -\infty$, add the formula:

$$\gamma_0^f > 0 \quad (\text{resp. } <),$$

and if $z_2 = +\infty$, add the formula:

$$\gamma_r^f > 0 \quad (\text{resp. } <).$$

- (e) For each literal of type $\text{Convex}(f)_{[z_1, z_2]}$ (resp. $\text{Concave}(f)_{[z_1, z_2]}$) occurring in φ_2 , add the following formulae:⁶

$$t_i^f \leq \frac{y_{i+1}^f - y_i^f}{v_{i+1} - v_i} \leq t_{i+1}^f \quad (\text{resp. } \geq),$$

$$\left(\frac{y_{i+1}^f - y_i^f}{v_{i+1} - v_i} = t_i^f \vee \frac{y_{i+1}^f - y_i^f}{v_{i+1} - v_i} = t_{i+1}^f \right) \longrightarrow (t_i^f = t_{i+1}^f),$$

for $i \in \{\text{ind}(z_1), \dots, \text{ind}(z_2) - 1\}$; moreover, if $z_1 = -\infty$, add the formula:

$$\gamma_0^f \leq t_1^f \quad (\text{resp. } \geq),$$

and if $z_2 = +\infty$, add the formula:

$$\gamma_r^f \geq t_r^f \quad (\text{resp. } \leq).$$

- (f) For each literal of type $\text{Strict_Convex}(f)_{[z_1, z_2]}$ (resp. $\text{Strict_Concave}(f)_{[z_1, z_2]}$) occurring in φ_2 , add the following formulae:

$$t_i^f < \frac{y_{i+1}^f - y_i^f}{v_{i+1} - v_i} < t_{i+1}^f \quad (\text{resp. } >),$$

for $i \in \{\text{ind}(z_1), \dots, \text{ind}(z_2) - 1\}$; moreover, if $z_1 = -\infty$, add the formula:

$$\gamma_0^f < t_1^f \quad (\text{resp. } >),$$

and if $z_2 = +\infty$, add the formula:

$$\gamma_r^f > t_r^f \quad (\text{resp. } <).$$

- (g) Drop all literals involving some functional variable.
Let φ_3 be the resulting formula.

Plainly, the formula φ_3 involves only literals of the following types:

$$t_1 \leq t_2, \quad t_1 < t_2, \quad t_1 = t_2,$$

where t_1 and t_2 are terms involving only real variables, the real constants 0 and 1, and the arithmetic operators $+$ and $*$ (and their duals $-$ and $/$), so that the formula φ_3 belongs to the decidable (existential) Tarski's theory of reals. Thus, to show that our theory RDF has a solvable satisfiability problem, it is enough to show that even the latter transformation step preserves satisfiability. Such result, though not particularly deep, requires a quite technical and lengthy proof, which will be provided in an extended version of the present paper.

Therefore we can conclude with our main result:

Theorem 1. *The theory RDF has a decidable satisfiability problem.* □

⁶ Observe that this group of formulae implicitly forces the relations $\frac{y_j^f - y_{j-1}^f}{v_j - v_{j-1}} \leq \frac{y_{j+1}^f - y_j^f}{v_{j+1} - v_j}$ for each $j \in \{\text{ind}(z_1) + 1, \dots, \text{ind}(z_2) - 1\}$. Geometrically, the point of coordinates (v_j, y_j^f) does not lie above (resp. lies below) the straight line joining the two points (v_{j-1}, y_{j-1}^f) and (v_{j+1}, y_{j+1}^f) .

5 Conclusions

In this paper we have shown that the satisfiability problem for the fragment RDF of the theory of reals with differential functions and various predicates is solvable.

The result has been obtained through a sequence of effective reductions which transforms any RDF-formula into an equisatisfiable formula of the decidable Tarski's theory of reals. Our decidability result is based on the existence of canonical models of satisfiable RDF-formulae in which functional variables are interpreted by parametric piecewise exponential functions.

By further generalizing canonical models, we expect that more elaborate constructs (such as, for instance, function addition and derivation predicate over open intervals) can be allowed without disrupting decidability.

References

1. E. Bonarska. *An Introduction to PC Mizar*, Fondation Philippe le Hodey, Brussels, 1990.
2. D. Cantone, A. Ferro, and E. G. Omodeo. *Computable set theory*, volume no.6 Oxford Science Publications of *International Series of Monographs on Computer Science*. Clarendon Press, 1989.
3. D. Cantone, G. Cincotti, G. Gallo. Decision algorithms for some fragments of real analysis: A theory of continuous functions with strict-convexity constructs. *Journal of Symbolic Computation*, Vol. 41(7), pp. 763–789, 2006.
4. D. Cantone, A. Ferro, E. Omodeo, J.T. Schwartz. Decision algorithms for some fragments of analysis and related areas. *Communications on Pure and Applied Mathematics*, Vol.40, pp. 281-300, 1987.
5. D. Cantone, E.G. Omodeo, J.T. Schwartz, and P. Ursino. Notes from the Logbook of a Proof-Checker's Project. In N. Dershowitz, editor, *Proc. of the International Symposium on Verification: Theory and Practice*, LNCS 2772, pp. 182–207, Springer-Verlag, 2003.
6. G.V. Chudnovsky. *Contributions to the Theory of Transcendental Numbers*. Amer. Math. Soc. Providence, RI, 1984.
7. G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Second GI Conference on Automata Theory and Formal Languages*, LNCS Vol. 33, Springer-Verlag, Berlin, 1975.
8. A. Ferro, E.G. Omodeo, J.T. Schwartz. Decision procedures for elementary sublanguages of set theory, I. Multi-level syllogistic and some extensions. *Communications on Pure and Applied Mathematics*, Vol. 33, pp. 599–608, 1980.
9. J.D. Fleuriot. On the Mechanization of Real Analysis in Isabelle/HOL. In *Proc. of the 13th International Conference on Theorem Proving in Higher Order*, LNCS 1869, pp. 145–161, Springer-Verlag, London, UK, 2000.
10. H. Gottlieb. *Automated Theorem Proving for Mathematics: Real Analysis in PVS*. PhD thesis, University of St. Andrews, 2001.
11. J.D. Guttman and F.J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213-248, 1993.
12. J. Harrison. *Theorem Proving with the Real Numbers*, Springer-Verlag, 1998.

13. J. Heintz, M.-F. Roy, and P. Solernó. On the theoretical and practical complexity of the existential theory of reals. *The Computer Journal*, Vol. 36 (5), pp. 427-431, 1993.
14. A.J. Macintyre and A.J. Wilkie, On the decidability of the real exponential field. In P. Odifreddi, editor, *Kreiseliana*, pp. 441-467, A.K. Peters, Wellesley, MA, 1996.
15. M. Muzalewski. *An Outline of PC Mizar*, Fondation Philippe le Hodey, Brussels, 1993.
16. E.G. Omodeo and J.T. Schwartz. A ‘theory’ mechanism for a proof-verifier based on first-order set theory. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, LNCS 2408, pp. 214-230, Springer-Verlag, 2002.
17. W.V. Quine. *Methods of logic*. Henry Holt, New York, 1950.
18. S. Ratschan. Efficient solving of quantified inequality constraints over the real numbers. To appear in *ACM Transactions on Computational Logic*. A pre-print is available at the arXiv.org e-Print archive at <http://arxiv.org/>.
19. D. Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, Vol. 33, pp. 514-520, 1968.
20. J.T. Schwartz, D. Cantone, and E.G. Omodeo. *Computational logic and set theory: The systematic application of formalized logic to the foundations of analysis*. Book in preparation, to appear in the series Texts in Computer Science, Springer-Verlag, 2008. A first draft is available at <http://www.settheory.com/intro.html> .
21. G. Sutcliffe and C.B. Suttner, The TPTP Problem Library: CNF Release v1.2.1, *Journal of Automated Reasoning*, Vol. 21(2), pp. 177-203, 1998. (Available online at <http://www.tptp.org>)
22. A. Tarski. *A Decision method for elementary algebra and geometry* (2nd ed. rev.). University of California Press, Berkeley, 1951.

On the satisfiability problem for a 3-level quantified syllogistic^{*}

Domenico Cantone and Marianna Nicolosi Asmundo

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

Abstract. We present a fragment of multi-sorted set-theoretic formulae, called *3LQS*, that admits a restricted form of quantification over individual and set variables, and show that it has a solvable satisfiability problem.

The proof is carried out by showing that the theory *3LQS* enjoys a small model property, i.e., any satisfiable *3LQS*-formula ψ has a finite model whose size depends solely on the size of ψ itself.

1 Introduction

Computable set theory is a research field active since the late seventies. The principal results achieved during the years, mainly regarding the study of the satisfiability problem for several fragments of set theory, have been collected in [7, 8]. The most efficient decision procedures have been implemented in the proof verifier *EtnaNova* [10, 11] and within the system *STeP* [2].

The basic set theoretic language is *MLS* (Multi-Level Syllogistic), extended in several ways by the introduction of operators, predicates and quantifiers.

Most of the decidability results regard one-sorted multi-level syllogistics, that is languages with variables of one kind only, ranging over the Von Neumann universe of sets. On the other hand, few decidability results exist for multi-sorted stratified syllogistics, where variables of several kinds are allowed. This, despite of the fact that the need of proving theorems containing variables of different sorts often arises in many fields of mathematics and computer science.

In [9] an efficient decision procedure for the satisfiability of the language *2LS*, a version of *MLS* with variables of two sorts (individuals, sets), is presented. Subsequently, in [4], the extension of *2LS* with the singleton operator and the cartesian product operator is proved decidable. The result is obtained by embedding *2LS* in the class of purely universal formulae of the elementary theory of relations. In [6] Tarski's and Presburger's arithmetics extended with sets have been studied. In [5] the language *3LSSPU* is proved decidable. *3LSSPU* is a multi-sorted stratified syllogistic allowing variables of three kinds (individuals, sets, sets of sets), and involving the singleton, powerset, and general union operators in addition to the operators and predicates in *2LS*.

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

In this paper we present a decidability result for the satisfiability problem of the set-theoretic language $3LQS$ (3-level quantified syllogistic).

$3LQS$ is a multi-sorted quantified syllogistic, whose language allows variables of different kinds. In particular, analogously to $3LSSPU$, it contains *individual variables*, varying over the elements of a given nonempty universe D , *set variables*, ranging over subsets of D , and *collection variables*, varying over collections of subsets of D .

The language of $3LQS$ admits a restricted form of quantification over individual and set variables. Its vocabulary contains only the predicate symbols $=$ and \in . In spite of that, $3LQS$ allows to express several constructs of set theory. Among them, the most comprehensive one is the set former, which in turn allows to express other operators like the powerset operator, the singleton operator, and so on.

We will show that $3LQS$ enjoys a small model property by presenting a procedure which allows to extract, out of a model satisfying a $3LQS$ -formula ψ , a finite model which also satisfies ψ . The procedure of construction of the finite model is inspired to the algorithms described in [4] and [5]. In particular it extends the approach presented in [4] to handle quantifiers over set variables and it is simpler and more efficient than the procedure introduced in [5].

The paper is organized as follows. In Section 2, we introduce the syntax and the semantics of the $3LQS$ language and we illustrate its expressiveness. Subsequently, in Section 3 the machinery needed to prove the decidability result is provided. In particular, a general definition of a restricted $3LQS$ -interpretation (relativized $3LQS$ -interpretation) is introduced, together with some useful technical results. In Section 4, a decision procedure for the satisfiability problem of $3LQS$ is presented. We use notions and results introduced in Section 3 to construct a small $3LQS$ model and to prove the correctness of the decision procedure. Finally, in Section 5, we draw our conclusions.

2 The language $3LQS$

2.1 Syntax

The language $3LQS$ (3-level quantified syllogistic) of our concern has

- (i) a collection \mathcal{V}_0 of *individual* or *level 0* variables, denoted by lower case letters x, y, z, \dots ;
- (ii) a collection \mathcal{V}_1 of *set* or *level 1* variables, denoted by final upper case letters X, Y, Z, \dots ;
- (iii) a collection \mathcal{V}_2 of *collection* or *level 2* variables, denoted by initial upper case letters A, B, C, \dots .

The formulae of $3LQS$ are defined as follows:

- (1) level 0 atomic formulae:
 - $x = y$, for $x, y \in \mathcal{V}_0$;
 - $x \in X$, for $x \in \mathcal{V}_0, X \in \mathcal{V}_1$;
- (2) level 1 atomic formulae:
 - $X = Y$, for $X, Y \in \mathcal{V}_1$;
 - $X \in A$, for $X \in \mathcal{V}_1, A \in \mathcal{V}_2$;
 - $(\forall z_1) \dots (\forall z_n)\varphi_0$, with φ_0 a propositional combination of level 0 atoms;
- (3) level 2 atomic formulae:
 - $(\forall Z_1) \dots (\forall Z_m)\varphi_1$, where φ_1 is a propositional combination of level 0 and level 1 atoms
- (4) the formulae of $3LQS$ are the propositional combinations of atoms of level 0, 1, and 2.

2.2 Semantics

A $3LQS$ -interpretation is a pair $\mathcal{M} = (D, M)$, where

- D is any nonempty collection of objects, called the *domain* or *universe* of \mathcal{M} , and
- M is an assignment to variables of $3LQS$ such that
 - $Mx \in D$, for each individual variable $x \in \mathcal{V}_0$;
 - $MX \in \text{pow}(D)$, for each set variable $X \in \mathcal{V}_1$;
 - $MA \in \text{pow}(\text{pow}(D))$, for all collection variables $A \in \mathcal{V}_2$.¹

Let

- $\mathcal{M} = (D, M)$, a $3LQS$ -interpretation
- $x_1, \dots, x_l \in \mathcal{V}_0$,
- $X_1, \dots, X_m \in \mathcal{V}_1$,
- $A_1, \dots, A_n \in \mathcal{V}_2$,
- $u_1, \dots, u_l \in D$,
- $U_1, \dots, U_m \in \text{pow}(D)$,
- $\mathcal{A}_1, \dots, \mathcal{A}_n \in \text{pow}(\text{pow}(D))$.

By

$$\mathcal{M}[x_1/u_1, \dots, x_l/u_l, X_1/U_1, \dots, X_m/U_m, A_1/\mathcal{A}_1, \dots, A_n/\mathcal{A}_n],$$

we denote the interpretation $\mathcal{M}' = (D, M')$ such that

$$\begin{aligned} M'x_i &= u_i, \text{ for } i = 1, \dots, l \\ M'X_j &= U_j, \text{ for } j = 1, \dots, m \\ M'A_k &= \mathcal{A}_k, \text{ for } k = 1, \dots, n \end{aligned}$$

and which otherwise coincides with \mathcal{M} on all remaining variables. Throughout the paper we use the abbreviations: \mathcal{M}^z for $\mathcal{M}[z_1/u_1, \dots, z_n/u_n]$ and \mathcal{M}^Z for $\mathcal{M}[Z_1/U_1, \dots, Z_m/U_m]$.

¹ We recall that, for any set s , $\text{pow}(s)$ denotes the *powerset* of s , i.e., the collection of all subsets of s .

Definition 1. Let φ be a 3LQS-formula and let $\mathcal{M} = (D, M)$ be a 3LQS-interpretation. We define the notion of satisfiability of φ with respect to \mathcal{M} (denoted by $\mathcal{M} \models \varphi$) inductively as follows

1. $\mathcal{M} \models x = y$ iff $Mx = My$;
2. $\mathcal{M} \models x \in X$ iff $Mx \in MX$;
3. $\mathcal{M} \models X = Y$ iff $MX = MY$;
4. $\mathcal{M} \models X \in A$ iff $MX \in MA$;
5. $\mathcal{M} \models (\forall z_1) \dots (\forall z_n) \varphi_0$ iff $\mathcal{M}[z_1/u_1, \dots, z_n/u_n] \models \varphi_0$, for all $u_1, \dots, u_n \in D$;
6. $\mathcal{M} \models (\forall Z_1) \dots (\forall Z_m) \varphi_1$ iff $\mathcal{M}[Z_1/U_1, \dots, Z_m/U_m] \models \varphi_1$, for all $U_1, \dots, U_m \in \text{pow}(D)$.

Propositional connectives are interpreted in the standard way:

7. $\mathcal{M} \models \varphi_1 \wedge \varphi_2$ iff $\mathcal{M} \models \varphi_1$ and $\mathcal{M} \models \varphi_2$;
8. $\mathcal{M} \models \varphi_1 \vee \varphi_2$ iff $\mathcal{M} \models \varphi_1$ or $\mathcal{M} \models \varphi_2$;
8. $\mathcal{M} \models \neg \varphi$ iff $\mathcal{M} \not\models \varphi$. □

Let ψ be a 3LQS-formula, if $\mathcal{M} \models \psi$, i.e. \mathcal{M} satisfies ψ , then \mathcal{M} is said to be a 3LQS-model for ψ . A 3LQS-formula is said to be *satisfiable* if it has a 3LQS-model. A 3LQS-formula is *valid* if it is satisfied by all 3LQS-interpretations.

2.3 Restriction over quantifiers

To guarantee the correctness of the decision procedure of Section 4.2, quantifiers over individual variables nested within quantifiers over set variables have to be restricted.

Let $(\forall Z_1) \dots (\forall Z_m) \varphi_1$ be an atomic formula of level 2. We require that

$$\neg \varphi_0 \rightarrow \bigwedge_{j=1}^m \bigwedge_{i=1}^n z_i \in Z_j$$

is valid for every level 1 atomic formula of the form $(\forall z_1) \dots (\forall z_n) \varphi_0$ present in φ_1 : in this case we say that the atom $(\forall z_1) \dots (\forall z_n) \varphi_0$ is *linked* to the variables Z_1, \dots, Z_m .

The condition above guarantees that, if a given interpretation assigns to z_1, \dots, z_n elements of the domain that make φ_0 false, such values are contained in the subsets of the domain assigned to Z_1, \dots, Z_m . By the construction of the relativized model presented in Sections 3 (definition of relativized interpretation) and 4.2 (definition of the finite domain), this is enough to make sure that satisfiability is preserved in the finite model.

Such constraint is not particularly restrictive as the examples in Section 2.4 illustrate: when quantifiers over individual variables are nested within quantifiers over set variables, they are used to describe a property of the quantified set variables (see for example the 3LQS version of the powerset operation).

2.4 Expressiveness of the language $3LQS$

Several constructs of elementary set theory are easily expressible within the language $3LQS$. The most comprehensive one is the set former, which in turn allows to express other significant operators such as the powerset operator and its variants, the singleton operator, binary union, intersection, and set difference, etc.

Atomic formulae of type $X = \{z : \varphi(z)\}$ can be expressed in $3LQS$ by the formula

$$(\forall z)(z \in X \leftrightarrow \varphi(z)) \quad (1)$$

provided that after transformation into prenex normal form one obtains a formula satisfying the syntactic constraints of $3LQS$. In particular this is achieved whenever $\varphi(z)$ is any unquantified formula of $3LQS$. Therefore the following constructs can be expressed in $3LQS$:

- $X_1 = X_2 \cup X_3$ [in $3LQS$: $(\forall z)(z \in X_1 \leftrightarrow z \in X_2 \vee z \in X_3)$];
- $X_1 = X_2 \cap X_3$ [in $3LQS$: $(\forall z)(z \in X_1 \leftrightarrow z \in X_2 \wedge z \in X_3)$];
- $X_1 = X_2 \setminus X_3$ [in $3LQS$: $(\forall z)(z \in X_1 \leftrightarrow z \in X_2 \wedge z \notin X_3)$];
- $X \subseteq Y$ [in $3LQS$: $(\forall z)(z \in X \rightarrow z \in Y)$];
- $X = \{x\}$ [in $3LQS$: $(\forall z)(z \in X \leftrightarrow z = x)$];
- etc.

The same remark applies also to atomic formulae of type $A = \{Z : \varphi(Z)\}$. In this case, in order that a prenex normal form of

$$(\forall Z)(Z \in A \leftrightarrow \varphi(Z)) \quad (2)$$

be in the language $3LQS$, it is enough that

- (a) $\varphi(Z)$ does not contain any quantifier over level 1 variables, and
- (b) all quantified variables of level 0 in $\varphi(Z)$ are linked to the variable Z .

Therefore one can express the following operators:

- $A_1 = A_2 \cup A_3$ [in $3LQS$: $(\forall Z)(Z \in A_1 \leftrightarrow Z \in A_2 \vee Z \in A_3)$];
- $A_1 = A_2 \cap A_3$ [in $3LQS$: $(\forall Z)(Z \in A_1 \leftrightarrow Z \in A_2 \wedge Z \in A_3)$];
- $A_1 = A_2 \setminus A_3$ [in $3LQS$: $(\forall Z)(Z \in A_1 \leftrightarrow Z \in A_2 \wedge Z \notin A_3)$];
- $A_1 \subseteq A_2$ [in $3LQS$: $(\forall Z)(Z \in A_1 \rightarrow Z \in A_2)$];
- $A = \{X\}$ [in $3LQS$: $(\forall Z)(Z \in A \leftrightarrow Z = X)$];
- $A = \text{pow}(X)$ [in $3LQS$: $(\forall Z)(Z \in A \leftrightarrow (\forall z)(z \in Z \rightarrow z \in X))$];
- $A = \text{pow}_{=h}(X)$, where $\text{pow}_{=h}(X) = \{Z : Z \subseteq X \text{ and } |Z| = h\}$ with h a positive integer constant;
- $A = \text{pow}_{\leq h}(X)$, where $\text{pow}_{\leq h}(X) = \{Z : Z \subseteq X \text{ and } |Z| \leq h\}$ with h a positive integer constant;

- $A = \text{pow}^*(X_1, \dots, X_n)$, where $\text{pow}^*(X_1, \dots, X_n) = \{Z : Z \subseteq \bigcup_{i=1}^n X_i \text{ and } Z \cap X_i \neq \emptyset, \text{ for all } 1 \leq i \leq n\}$ [in 3LQS: $(\forall Z)(Z \in A \leftrightarrow ((\forall z)(z \in Z \rightarrow z \in \bigvee_{i=1}^n X_i) \wedge \bigwedge_{i=1}^n (\exists z)(z \in Z \wedge z \in X_i))$];
- $A = X_1 \otimes \dots \otimes X_n$, where $X_1 \otimes \dots \otimes X_n = \{\{x_1, \dots, x_n\} : x_i \in X_i, \text{ for all } 1 \leq i \leq n\}$;
- etc.

3 Relativized interpretations

We introduce the notion of relativized interpretation, to be used together with the decision procedure of Section 4.2 to construct out of a model $\mathcal{M} = (D, M)$ for a 3LQS-formula ψ , a finite interpretation $\mathcal{M}^* = (D^*, M^*)$ satisfying ψ as well.

Definition 2. Let $\mathcal{M} = (D, M)$ be a 3LQS-interpretation. Let $D^* \subseteq D$, $d^* \in D^*$, and $\mathcal{V}'_1 \subseteq \mathcal{V}_1$. We define the relativized interpretation $\text{Rel}(\mathcal{M}, D^*, d^*, \mathcal{V}'_1)$ of \mathcal{M} with respect to D^* , d^* , and \mathcal{V}'_1 to be the interpretation (D^*, M^*) , where

$$M^*x = \begin{cases} Mx, & \text{if } Mx \in D^* \\ d^*, & \text{otherwise} \end{cases}$$

$$M^*X = MX \cap D^*$$

$$M^*A = ((MA \cap \text{pow}(D^*)) \setminus \{M^*X : X \in \mathcal{V}'_1\}) \cup \{M^*X : X \in \mathcal{V}'_1, MX \in MA\}.$$

□

We spend some words on the intuition behind the definition of M^*A . Analogously to M^*X , M^*A is obtained from the intersection of the interpretation of A in \mathcal{M} with the power set of the finite domain D^* . However, such operation may leave in $MA \cap \text{pow}(D^*)$ some sets J such that $J = M^*X$ with $MX \notin MA$. Such J s have to be removed from the restricted interpretation of A in order to guarantee that satisfiability of ψ is preserved (this justifies the $\setminus \{M^*X : X \in \mathcal{V}'_1\}$ in Definition 2). Further, there also may be some $MX \in MA$ such that $M^*X \notin MA \cap \text{pow}(D^*)$. Again, to make the restricted model preserve satisfiability of ψ , such M^*X have to be added to the interpretation of A in the restricted model (this justifies the $\cup \{M^*X : X \in \mathcal{V}'_1, MX \in MA\}$ in Definition 2).

For ease of notation, we will often omit the reference to the element $d^* \in D^*$ and write simply $\text{Rel}(\mathcal{M}, D^*, \mathcal{V}'_1)$ in place of $\text{Rel}(\mathcal{M}, D^*, d^*, \mathcal{V}'_1)$.

The definition of relativized interpretation given above is inspired by the construction of the finite model described in [5]. However, in our case the construction of the finite model results simpler: we don't have to close the formula with respect to the powerset operation and the general union operation, and don't have to construct a normalized model.

The following satisfiability result holds for unquantified atomic formulae.

Lemma 1. *Let $\mathcal{M} = (D, M)$ be a 3LQS-interpretation. Also, let $D^* \subseteq D$, $d^* \in D^*$, and $\mathcal{V}'_1 \subseteq \mathcal{V}_1$ be given. Let us put $\mathcal{M}^* = \text{Rel}(\mathcal{M}, D^*, d^*, \mathcal{V}'_1)$. Then the following holds.*

- (a) $\mathcal{M}^* \models x = y$ iff $\mathcal{M} \models x = y$, for all $x, y \in \mathcal{V}_0$ such that $Mx, My \in D^*$;
- (b) $\mathcal{M}^* \models x \in X$ iff $\mathcal{M} \models x \in X$, for all $X \in \mathcal{V}_1$ and $x \in \mathcal{V}_0$ such that $Mx \in D^*$;
- (c) $\mathcal{M}^* \models X = Y$ iff $\mathcal{M} \models X = Y$, for all $X, Y \in \mathcal{V}_1$ such that if $MX \neq MY$ then $(MX \Delta MY) \cap D^* \neq \emptyset$;
- (d) if for all $X, Y \in \mathcal{V}'_1$ such that $MX \neq MY$ we have $(MX \Delta MY) \cap D^* \neq \emptyset$, then $\mathcal{M}^* \models X \in A$ iff $\mathcal{M} \models X \in A$, for all $X \in \mathcal{V}'_1$, $A \in \mathcal{V}_2$.²

Proof.

Cases (a), (b) and (c) are easily verified. We prove only case (d).

- (d) Assume that for all $X, Y \in \mathcal{V}'_1$ such that $MX \neq MY$ we have $(MX \Delta MY) \cap D^* \neq \emptyset$. Let $X \in \mathcal{V}'_1$ and $A \in \mathcal{V}_2$. If $MX \in MA$, then obviously $M^*X \in M^*A$. On the other hand, if $MX \notin MA$, but $M^*X \in M^*A$, then necessarily we must have $M^*X = M^*Z$, for some $Z \in \mathcal{V}'_1$ such that $MZ \in MA$. But then, as $MX \neq MZ$, we would have by our hypothesis that $M^*X \neq M^*Z$, which is a contradiction. ■

3.1 Relativized interpretations and quantified atomic formulae

Satisfiability results for quantified atomic formulae are treated as shown in the following. Let us put

$$\begin{aligned} \mathcal{M}^{z,*} &= \text{Rel}(\mathcal{M}^z, D^*, \mathcal{V}'_1) \\ \mathcal{M}^{*,z} &= \mathcal{M}^*[z_1/u_1, \dots, z_n/u_n] \\ \mathcal{M}^{Z,*} &= \text{Rel}(\mathcal{M}^Z, D^*, \mathcal{V}'_1 \cup \{Z_1, \dots, Z_m\}) \\ \mathcal{M}^{*,Z} &= \mathcal{M}^*[Z_1/U_1, \dots, Z_m/U_m]. \end{aligned}$$

The lemmas in the following provide useful technical results to be employed in the proof of Theorem 1. In particular Lemmas 2 and 3 are used to prove Lemma 4.

Lemma 2. *Let $u_1, \dots, u_n \in D^*$ and let $z_1, \dots, z_n \in \mathcal{V}_0$. Then, for every $x \in \mathcal{V}_0$ and $X \in \mathcal{V}_1$ we have:*

- (i) $M^{*,z}x = M^{z,*}x$,

² We recall that Δ denotes the symmetric difference operator defined by $s\Delta t = (s \setminus t) \cup (t \setminus s)$.

(ii) $M^{z,*}X = M^{*,z}X$

Lemma 3. *Let $\mathcal{M} = (D, M)$ be a 3LQS-interpretation, $D^* \subseteq D$, $\mathcal{V}'_1 \subseteq \mathcal{V}_1$, $Z_1, \dots, Z_m \in \mathcal{V}_1 \setminus \mathcal{V}'_1$, $U_1, \dots, U_m \in \text{pow}(D^*) \setminus \{M^*X : X \in \mathcal{V}'_1\}$.*

Then the 3LQS-interpretations $\mathcal{M}^{,Z}$ and $\mathcal{M}^{Z,*}$ coincide.*

Lemma 4. *Let $\mathcal{M} = (D, M)$ be a 3LQS-interpretation. Let $D^* \subseteq D$, $d^* \in D^*$, $\mathcal{V}'_1 \subseteq \mathcal{V}_1$ be given, and let $\mathcal{M}^* = \text{Rel}(\mathcal{M}, D^*, d^*, \mathcal{V}'_1)$. Further, let $(\forall z_1) \dots (\forall z_n)\varphi_0$ and $(\forall Z_1) \dots (\forall Z_m)\varphi_1$ be two atomic formulae of level 1 and 2, respectively, such that $Mx \in D^*$, for every $x \in \mathcal{V}_0$ occurring in φ_0 or in φ_1 . We have*

- (i) *if $\mathcal{M} \models (\forall z_1) \dots (\forall z_n)\varphi_0$, then $\mathcal{M}^* \models (\forall z_1) \dots (\forall z_n)\varphi_0$;*
- (ii) *if $\mathcal{M} \models (\forall Z_1) \dots (\forall Z_m)\varphi_1$, then $\mathcal{M}^* \models (\forall Z_1) \dots (\forall Z_m)\varphi_1$, provided that $(MX\Delta MY) \cap D^* \neq \emptyset$, for every $X, Y \in \mathcal{V}_1$ such that $MX \neq MY$.*

4 The satisfiability problem for 3LQS-formulae

In this section we solve the satisfiability problem for 3LQS, i.e. the problem of establishing for any given formula of 3LQS whether it is satisfiable or not, by proving a small model property for 3LQS-formulae.

We begin by showing how the general satisfiability problem for 3LQS can be reduced to a simpler instance of it.

4.1 Normalized 3LQS-conjunctions

Let ψ be a formula of 3LQS and let ψ_{DNF} be a disjunctive normal form of ψ . Then ψ is satisfiable if and only if at least one of the disjuncts of ψ_{DNF} is satisfiable. We recall that the disjuncts of ψ_{DNF} are conjunctions of level 0, 1, and 2 literals, i.e. level 0, 1, and 2 atoms or their negation. In view of the previous observations, without loss of generality, we can suppose that our formula ψ is a conjunction of level 0, 1, and 2 literals. In addition, we can also assume that no bound variable in ψ can occur in more than one quantifier or can occur also free.

For decidability purposes, negative quantified conjuncts occurring in ψ can be eliminated as explained below. Let $\mathcal{M} = (D, M)$ be a model for ψ . Then $\mathcal{M} \models \neg(\forall z_1) \dots (\forall z_n)\varphi_0$ if and only if $\mathcal{M}[z_1/u_1, \dots, z_n/u_n] \models \neg\varphi_0$, for some $u_1, \dots, u_n \in D$, and $\mathcal{M} \models \neg(\forall Z_1) \dots (\forall Z_m)\varphi_1$ if and only if $\mathcal{M}[Z_1/U_1, \dots, Z_m/U_m] \models \neg\varphi_1$, for some $U_1, \dots, U_m \in \text{pow}(D)$. Thus, each negative literal of type $\neg(\forall z_1) \dots (\forall z_n)\varphi_0$ can be replaced by $\neg(\varphi_0)_{z'_1, \dots, z'_n}^{z_1, \dots, z_n}$, where z'_1, \dots, z'_n are newly introduced variables of level 0. Likewise, each negative literal of type $\neg(\forall Z_1) \dots (\forall Z_m)\varphi_1$ can be replaced by $\neg(\varphi_1)_{Z'_1, \dots, Z'_m}^{Z_1, \dots, Z_m}$, where Z'_1, \dots, Z'_m are newly introduced variables of level 1.

Hence, we can further assume that ψ is a conjunction of literals of the following types:

- (1) $x = y, x \neq y, x \in X, x \notin X, X = Y, X \neq Y, X \in A, X \notin A$;
- (2) $(\forall z_1) \dots (\forall z_n)\varphi_0$, where $n > 0$ and φ_0 is a propositional combination of level 0 atoms;
- (3) $(\forall Z_1) \dots (\forall Z_m)\varphi_1$, where $m > 0$ and φ_1 is a propositional combination of level 0 and level 1 atoms, where atoms of type $(\forall z_1) \dots (\forall z_n)\varphi_0$ in φ_1 are linked to the bound variables Z_1, \dots, Z_m .

We call such formulae *normalized 3LQS-conjunctions*.

4.2 A decision procedure

In view of the preceding discussion we can limit ourselves to consider the satisfiability problem for normalized 3LQS-conjunctions only.

Thus, let ψ be a normalized 3LQS-conjunction and assume that $\mathcal{M} = (D, M)$ is a model for ψ .

We show how to construct, out of \mathcal{M} , a finite 3LQS-interpretation $\mathcal{M}^* = (D^*, M^*)$ which is a model of ψ . We proceed as follows. First we outline a procedure to build a nonempty finite universe $D^* \subseteq D$ whose size depends solely on ψ and can be computed *a priori*. Then, a finite 3LQS-interpretation $\mathcal{M}^* = (D^*, M^*)$ is constructed according to Definition 2. Finally \mathcal{M}^* is proved to be a model of ψ .

Construction of the universe D^* . Let us denote by $\mathcal{W}_0, \mathcal{W}_1$, and \mathcal{W}_2 the collections of the variables of level 0, 1, and 2 present in ψ , respectively. Then we compute D^* by means of the procedure *SmallDomain*(ψ, \mathcal{M}) illustrated in Figure 1 as follows.

Let ψ_1, \dots, ψ_k be the conjuncts of ψ . To each conjunct ψ_i of the form $(\forall Z_{i1}) \dots (\forall Z_{im_i})\varphi_i$ we associate the collection $\varphi_{i1}, \dots, \varphi_{il_i}$ of atomic formulae of type (2) present in the matrix of ψ_i and call the variables Z_{i1}, \dots, Z_{im_i} the *arguments of* $\varphi_{i1}, \dots, \varphi_{il_i}$. Then we put

$$\Phi = \{\varphi_{ij} : 1 \leq i \leq k \text{ and } 1 \leq j \leq l_i\}.$$

For every pair of variables X, Y in \mathcal{W}_1 such that $MX \neq MY$ let $u_{X,Y}$ be any element in the symmetric difference of MX and MY and put $\Delta_1 = \{u_{X,Y} : X, Y \text{ in } \mathcal{W}_1 \text{ and } MX \neq MY\}$. We initialize D^* with the set $\{Mx : x \text{ in } \mathcal{W}_0\} \cup \Delta_1$. Then, for each $\varphi \in \Phi$ of the form $(\forall z_1) \dots (\forall z_n)\varphi_0$ having Z_1, \dots, Z_m as arguments and for each ordered m -tuple $(X_{i_1}, \dots, X_{i_m})$ of variables in \mathcal{W}_1 , if $\mathcal{M}\varphi_{X_{i_1}, \dots, X_{i_m}}^{Z_1, \dots, Z_m} = \mathbf{false}$ we insert in D^* elements $u_1, \dots, u_n \in D$ such that

$$\mathcal{M}[z_1/u_1, \dots, z_n/u_n]\varphi_{0X_{i_1}, \dots, X_{i_m}}^{Z_1, \dots, Z_m} = \mathbf{false},$$

otherwise we leave D^* unchanged.

It is clear that after that a formula $\varphi \in \Phi$ of the form $(\forall z_1) \dots (\forall z_n) \varphi_0$ with arguments Z_1, \dots, Z_m has been treated, the size of D^* can increase of at most $n|\mathcal{V}_1|^m$ elements. This gives a bound of $\mathcal{O}(|\psi|^{q+2})$ on the final size of D^* , where q is the maximal length of quantifiers prefixes over level 1 variables present in ψ .

```

procedure SmallDomain( $\psi, \mathcal{M}$ )
1.  $\Phi := \emptyset$ ;
2. for  $i = 1, \dots, k$  do
3.   if  $\psi_i$  is of kind  $(\forall Z_{i1}), \dots, (\forall Z_{im_i}) \varphi_i$  then
4.     for  $j = 1, \dots, il_i$  do
5.       //  $\varphi_{ij}$  are the atomic formulae of type (2) present
6.       // in the matrix of  $\psi_i$ 
7.        $\Phi := \Phi \cup \{\varphi_{ij}\}$ ;
8.     endfor;
9.   endif;
10. endfor;
11.  $\Delta_1 := \emptyset$ ;
12. for  $X, Y \in \mathcal{W}_1(\psi)$  such that  $MX \neq MY$  do
13.   if  $u_{XY} \in MX \Delta MY$  then
14.     // just take one  $u_{XY}$ 
15.      $\Delta_1 := \Delta_1 \cup \{u_{XY}\}$ ;
16.   endif;
17. endfor;
18.  $D^* := \{Mx : x \in \mathcal{W}_0\} \cup \Delta_1$ ;
19. for every  $\varphi \in \Phi$  do
20.   if  $\varphi$  is of kind  $(\forall z_1) \dots (\forall z_n) \varphi_0$ 
21.     with arguments  $Z_1, \dots, Z_m$  then
22.     for every  $(X_{i1}, \dots, X_{im}) \in \mathcal{W}_1^m$  do
23.       if  $\mathcal{M} \varphi_{X_{i1}, \dots, X_{im}}^{Z_1, \dots, Z_m} = \mathbf{false}$  then
24.         //  $u_1, \dots, u_n$  are elements of  $D$  such that
25.         //  $\mathcal{M}[z_1/u_1, \dots, z_n/u_n] \varphi_{X_{i1}, \dots, X_{im}}^{Z_1, \dots, Z_m} = \mathbf{false}$ .
26.          $D^* := D^* \cup \{u_1, \dots, u_n\}$ ;
27.       endif;
28.     endfor;
29.   endif;
30. endfor;
31. return  $D^*$ ;
end procedure

```

Fig. 1. Procedure of construction of D^* .

Correctness of the procedure. Now we put $\mathcal{M}^* = \text{Rel}(\mathcal{M}, D^*, \mathcal{W}_1)$. We have to show that, if $\mathcal{M} \models \psi$, then $\mathcal{M}^* \models \psi$.

Theorem 1. *Let \mathcal{M} be a 3LQS-interpretation satisfying a normalized 3LQS-conjunction ψ . Further, let $\mathcal{M}^* = \text{Rel}(\mathcal{M}, D^*, \mathcal{W}_1)$ be the 3LQS-interpretation defined according to Definition 2, where D^* is constructed as described in the procedure depicted in Figure 1, and let \mathcal{W}_1 be defined as above. Then $\mathcal{M}^* \models \psi$.*

Proof.

We have to prove that $\mathcal{M}^* \models \psi'$ for every literal ψ' in ψ . Each ψ' is of one of the three kinds introduced in Section 4.1. By applying Lemma 1 or 4 to every ψ' in ψ (according to the kind of ψ') we obtain the thesis.

Notice that the hypotheses of Lemma 1 and of Lemma 4 are fulfilled by the construction described by the procedure *SmallDomain* in Figure 1:

- $Mx \in D^*$, for every variable $x \in \mathcal{V}_0$. Furthermore, $(MX \Delta MY) \cap D^* \neq \emptyset$ for every $X, Y \in \mathcal{V}_1$ such that $MX \neq MY$ (see line 15 of the procedure *SmallDomain*, where the generic set of individual variables \mathcal{V}_0 is substituted with \mathcal{W}_0 and \mathcal{V}_1 with \mathcal{W}_1);
- the constraint introduced in Section 2.1 for atomic formulae of level 2, concerning links between quantified variables of level 0 and 1, is satisfied (lines 16 – 24). ■

5 Conclusions and future work

We have presented the multi-sorted stratified set-theoretic language 3LQS and have provided a decision procedure for its satisfiability problem.

Our next step will be to exploit the procedure for the construction of the finite model (in particular of the universe D^* and of the interpretation \mathcal{M}^*) to build a tableau-based decision procedure for 3LQS in the flavour of [3, 1], in order to obtain a more efficient decision procedure that could be embedded in automated theorem provers and proof verifiers.

References

1. B. Beckert and U. Hartmer. A Tableau Calculus for Quantifier-free Set Theoretic Formulae. In *TABLEAUX '98: Proceedings of the International Conference on Theorem Proving with Analytic Tableaux and Related Methods*, LNCS 1397, pages 93–107, Oisterwijk, The Netherlands, 1998. Springer.
2. N. S. Bjørner, A. Browne, E. S. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. Intl. Conference on Computer Aided Verification*, volume 1102 of LNCS, pages 415–418. Springer, 1996.
3. D. Cantone. A fast saturation strategy for set-theoretic tableaux. In *TABLEAUX '97: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 122–137, London, UK, 1997. Springer-Verlag.
4. D. Cantone and V. Cutello. A decidable fragment of the elementary theory of relations and some applications. In *ISSAC '90: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 24–29, New York, NY, USA, 1990. ACM Press.

5. D. Cantone and V. Cutello. Decision procedures for stratified set-theoretic syllogistics. In Manuel Bronstein, editor, *Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation, ISSAC'93 (Kiev, Ukraine, July 6-8, 1993)*, pages 105–110, New York, 1993. ACM Press.
6. D. Cantone, V. Cutello, and J. T. Schwartz. Decision problems for Tarski and Presburger arithmetics extended with sets. In *CSL '90: Proceedings of the 4th Workshop on Computer Science Logic*, pages 95–109, London, UK, 1991. Springer-Verlag.
7. D. Cantone, A. Ferro, and E. Omodeo. *Computable set theory*. Clarendon Press, New York, NY, USA, 1989.
8. D. Cantone, E. Omodeo, and A. Policriti. *Set Theory for Computing - From decision procedures to declarative programming with sets*. Springer-Verlag, Texts and Monographs in Computer Science, 2001.
9. A. Ferro and E.G. Omodeo. *An efficient validity test for formulae in extensional two-level syllogistic*. *Le Matematiche*, 33:130–137, 1978.
10. E. G. Omodeo, D. Cantone, A. Policriti, and J. T. Schwartz. A Computerized Referee. In *Reasoning, Action and Interaction in AI Theories and Systems*, Lecture Notes in Artificial Intelligence, vol. 4155, pages 117–139, 2006.
11. J. Schwartz, D. Cantone, and E. Omodeo. *Computational Logic and Set Theory (Texts in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. In preparation.

Generalizzazione di Clausole Basata su un Nuovo Criterio di Similarità

S. Ferilli, T.M.A. Basile, N. Di Mauro, M. Biba, and F. Esposito

Dipartimento di Informatica
Università di Bari
via E. Orabona, 4 - 70125 Bari - Italia
{ferilli, basile, ndm, biba, esposito}@di.uniba.it

Abstract. Pochi lavori in letteratura sono stati dedicati alla definizione di criteri di similarità tra formule della Logica del Prim'Ordine, dove la presenza di relazioni fa sì che diverse porzioni di una descrizione possano essere mappate in molti modi differenti su un'altra descrizione. Quindi, è importante individuare dei criteri generali che siano in grado di supportare il confronto tra formule. Questo potrebbe avere molte applicazioni; questo lavoro tratta il caso di due descrizioni (ad es., una definizione ed una osservazione) che devono essere generalizzate, dove il criterio di similarità potrebbe aiutare a concentrarsi sulle sottoparti delle descrizioni che sono più simili e quindi più probabilmente corrispondenti tra loro, basandosi sulla loro struttura sintattica. Sperimentazioni su dataset reali dimostrano l'efficacia dell'approccio proposto e l'efficienza di una implementazione in una procedura di generalizzazione.

1 Introduzione

La Logica del Prim'Ordine è un potente formalismo in grado di esprimere relazioni tra oggetti, il che gli consente di superare le limitazioni delle rappresentazioni proposizionali o attributo-valore. Tuttavia, la presenza di relazioni fa sì che varie porzioni di una descrizione possano essere mappate in molti possibili modi differenti su un'altra descrizione. Questo pone un importante problema di sforzo computazionale quando due descrizioni devono essere confrontate tra di loro. D'altra parte, le tecniche per il confronto tra due descrizioni del prim'ordine potrebbero avere molte applicazioni, in particolare in Intelligenza Artificiale: per esempio, aiutare una procedura di sussunzione a convergere velocemente; valutare il grado di similarità tra due formule; implementare una procedura di matching flessibile; supportare tecniche di classificazione del tipo instance-based oppure il clustering concettuale.

Per quanto riguarda l'apprendimento supervisionato, molti sistemi si basano sulla generalizzazione delle definizioni rispetto alle osservazioni, ed una funzione di similarità potrebbe aiutare la procedura a concentrarsi sulle componenti che sono più simili e che quindi è più probabile che corrispondano. Ovviamente, questo riguarda gli aspetti semantici del dominio e quindi non esiste un modo

preciso (algoritmico) per riconoscere le (sotto-)formule giuste. Quindi, il problema deve essere affrontato euristicamente, sviluppando qualche metodo che può ipotizzare quali parti di una descrizione si riferiscono a quali parti di un'altra descrizione, basandosi esclusivamente sulla loro struttura sintattica. Per questo motivo, è necessario cercare similarità parziali tra le componenti delle descrizioni.

In particolare, molti sistemi di Apprendimento Automatico nel Prim'Ordine inducono teorie sotto forma di Programmi Logici, una restrizione della Logica del Prim'Ordine a insiemi di *Clausole di Horn*, ossia formule logiche della forma $l_1 \wedge \dots \wedge l_n \Rightarrow l_0$ dove gli l_i sono *atomi*, spesso rappresentate in stile Prolog come $l_0 :- l_1, \dots, l_n$ da interpretarsi come " l_0 (detto *testa* della clausola) è vera, se l_1 e ... e l_n (detto *corpo* della clausola) sono tutti veri". Senza perdita di generalità [8], nel seguito si tratterà il caso di clausole Datalog connesse.

Nei paragrafi seguenti verranno presentati alcuni criteri ed una formula sui quali basare le considerazioni sulla similarità tra clausole del prim'ordine. Dopo aver effettuato una panoramica dei lavori connessi nel paragrafo 4, il 5 mostra come la formula proposta è in grado di guidare in maniera efficace una procedura di generalizzazione. Infine, il paragrafo 6 conclude l'articolo presentando spunti per lavori futuri.

2 La Formula di Similarità

Intuitivamente, la valutazione della similarità tra due entità i' ed i'' potrebbe essere basata sia sulla presenza di caratteristiche comuni, che dovrebbero incidere positivamente sulla valutazione della similarità, sia sulle caratteristiche di ciascuna entità non possedute dall'altra (definiamo questo come *residuo* della prima entità rispetto alla seconda), che dovrebbe incidere negativamente sul valore complessivo della funzione di similarità [5]. Quindi, possibili parametri di similarità sono:

- n , il numero delle catteristiche possedute da i' ma non da i'' (*residuo* di i' rispetto a i'');
- l , il numero delle catteristiche possedute sia da i' che da i'' ;
- m ,il numero delle catteristiche possedute da i'' ma non da i' (*residuo* di i'' rispetto a i').

Si è quindi sviluppata una nuova funzione che esprime il grado di similarità tra i' ed i'' basato su tali parametri:

$$sf(i', i'') = sf(n, l, m) = 0.5 \frac{l+1}{l+n+2} + 0.5 \frac{l+1}{l+m+2} \quad (1)$$

La funzione restituisce valori in $]0, 1[$, il che richiama la teoria della probabilità e quindi può aiutare gli esseri umani ad interpretare il valore risultante. Una sovrapposizione completa del modello sull'osservazione tende al limite di 1 al crescere del numero delle caratteristiche in comune. Il valore di similarità totale 1 non viene mai raggiunto, il che è coerente con l'intuizione che l'unico caso in cui questo deve succedere è l'esatta identità tra le due entità, ossia $i' = i''$

(in seguito, assumeremo $i' \neq i''$). Al contrario, nel caso di non-sovrapposizione, la funzione tende a 0 al crescere del numero delle caratteristiche non condivise. Questo è coerente con l'intuizione che non esiste un limite al numero di caratteristiche differenti in due descrizioni che contribuiscono a renderle diverse. Inoltre, nel caso in cui non ci siano caratteristiche che descrivono le due entità ($n = l = m = 0$, cioè il confronto di due oggetti che non hanno nessuna caratteristica) la funzione assume il valore di $1/2$, che può essere considerato intuitivamente corretto per rappresentare il caso di massima incertezza. Per esempio, un tale caso si verifica quando un modello include un oggetto privo di proprietà da soddisfare: confrontandolo con un altro oggetto osservato, privo anch'esso di proprietà, non si è in grado di capire se la sovrapposizione è totale perchè le due descrizioni non hanno realmente nessuna proprietà da soddisfare, oppure perchè precedenti generalizzazioni hanno rimosso dal modello tutte le caratteristiche di cui prima disponeva. Va notato che ciascuno dei due termini si riferisce in particolare ad una delle due entità che si stanno confrontando, e quindi si potrebbe introdurre un peso per attribuire diversa importanza a ciascuna di esse (questo potrebbe essere tipicamente necessario quando il confronto riguarda un modello rispetto ad una osservazione); questo, comunque, travalica gli scopi di questo articolo.

3 Criteri di Similarità

Nelle formule del prim'ordine, i termini rappresentano oggetti specifici; i predicati unari in generale rappresentano proprietà dei, mentre i predicati n -ari esprimono relazioni fra termini. Quindi, si possono definire due livelli di similarità per coppie di descrizioni del prim'ordine: il livello degli *oggetti*, relativo alla similarità tra termini nelle descrizioni, e il livello *strutturale*, che si riferisce a come i grafi delle relazioni nelle descrizioni si sovrappongono:

Example 1. Consideriamo come esempio illustrativo nel resto dell'articolo, le seguenti due clausole (in questo caso una regola C ed una osservazione classificata E):

$$\begin{aligned}
 C : h(X) &:- p(X, Y), p(X, Z), p(W, X), r(Y, U), o(Y, Z), q(W, W), s(U, V), \\
 &\quad \pi(X), \phi(X), \rho(X), \pi(Y), \sigma(Y), \tau(Y), \phi(Z), \sigma(W), \tau(W), \pi(U), \phi(U). \\
 E : h(a) &:- p(a, b), p(a, c), p(d, a), r(b, f), o(b, c), q(d, e), t(f, g), \\
 &\quad \pi(a), \phi(a), \sigma(a), \tau(a), \sigma(b), \tau(b), \phi(b), \tau(d), \rho(d), \pi(f), \phi(f), \sigma(f).
 \end{aligned}$$

3.1 Similarità tra Oggetti

Consideriamo due clausole C' e C'' ; chiamiamo $A' = \{a'_1, \dots, a'_n\}$ l'insieme dei termini in C' , e $A'' = \{a''_1, \dots, a''_m\}$ l'insieme dei termini in C'' . Quando si confrontano una coppia $(a', a'') \in A' \times A''$, ossia un oggetto preso da C' e uno da C'' , rispettivamente, si possono distinguere due tipi di caratteristiche: le proprietà espresse dai predicati unari (*attributi caratteristici*), e i modi in cui questi sono relazionati ad altri oggetti in base ai predicati n -ari (*attributi relazionali*).

Più precisamente, gli attributi relazionali sono definiti dalla posizione che gli oggetti occupano tra gli argomenti dei predicati n -ari, poichè posizioni diverse si riferiscono a ruoli diversi svolti dagli oggetti. In seguito, intenderemo un *ruolo* come una coppia $R = (\text{predicato}, \text{posizione})$ (scritto in maniera compatta come $R = \text{predicato}/\text{arietà.posizione}$). Per esempio, attributi caratteristici possono essere **maschio**(X) o **alto**(X), mentre attributi relazionali possono essere espressi da predicati come **genitore**(X, Y), dove in particolare l'argomento in prima posizione identifica il ruolo di genitore (*genitore/2.1*), e il secondo rappresenta il ruolo di figlio (*genitore/2.2*).

Due valori di similarità possono quindi essere associati ad a' e a'' : una *similarità caratteristica*, dove (1) è applicata ai valori relativi agli attributi caratterizzanti, e una *similarità relazionale*, basata sul numero di volte che due oggetti svolgono lo stesso ruolo nei predicati n -ari.

La similarità caratteristica tra a' e a'' , $\text{sf}_c(a', a'')$, può essere calcolata considerando l'insieme P' delle proprietà relative ad a' e l'insieme P'' delle proprietà relative ad a'' come $\text{sf}(n_c, l_c, m_c)$ per i seguenti parametri:

$n_c = |P' \setminus P''|$ è il numero delle proprietà possedute dall'oggetto rappresentato dal termine a' in C' ma non dall'oggetto rappresentato dal termine a'' in C'' (*residuo caratteristico* di a' rispetto ad a'');

$l_c = |P' \cap P''|$ è il numero di proprietà in comune tra l'oggetto rappresentato dal termine a' in C' e l'oggetto rappresentato dal termine a'' in C'' ;

$m_c = |P'' \setminus P'|$ è il numero di proprietà possedute dall'oggetto rappresentato dal termine a'' in C'' ma non dall'oggetto rappresentato dal termine a' in C' (*residuo caratteristico* di a'' rispetto ad a').

Una tecnica simile si può usare per calcolare la similarità relazionale tra a' ed a'' . In questo caso, poichè un oggetto può svolgere più volte lo stesso ruolo in relazioni differenti (ad esempio, un genitore di molti figli), bisogna considerare i *multiinsiemi* R' e R'' dei ruoli svolti da a' ed a'' , rispettivamente. Quindi, la similarità relazionale tra a' ed a'' , $\text{sf}_r(a', a'')$, può essere calcolata come $\text{sf}(n_r, l_r, m_r)$, dove

$n_r = |R' \setminus R''|$ esprime quante volte a' svolge in C' ruoli che a'' non svolge in C'' , (*residuo relazionale* di a' rispetto ad a'');

$l_r = |R' \cap R''|$ è il numero di volte che sia a' in C' che a'' in C'' svolgono gli stessi ruoli;

$m_r = |R'' \setminus R'|$ esprime quante volte a'' svolge in C'' ruoli che a' non svolge in C' , (*residuo relazionale* di a'' rispetto ad a').

Complessivamente, la *similarità di oggetti* tra due termini si può definire come $\text{sf}_o(a', a'') = \text{sf}_c(a', a'') + \text{sf}_r(a', a'')$.

Example 2. Le proprietà e ruoli per lo stesso termine in C e E , e il confronto per alcune delle possibili coppie, sono riportate in Tabella 1.

Table 1. Similarità di Oggetti

C			E		
t'	P'	R'	t''	P''	R''
X	$\{\pi, \phi, \rho\}$	$\{p/2.1, p/2.1, p/2.2\}$	a	$\{\pi, \phi, \sigma, \tau\}$	$\{p/2.1, p/2.1, p/2.2\}$
Y	$\{\pi, \sigma, \tau\}$	$\{p/2.2, r/2.1, o/2.1\}$	b	$\{\sigma, \tau\}$	$\{p/2.2, r/2.1, o/2.1\}$
Z	$\{\phi\}$	$\{p/2.2, o/2.2\}$	c	$\{\phi\}$	$\{p/2.2, o/2.2\}$
W	$\{\sigma, \tau\}$	$\{p/2.1, p/2.1, p/2.2\}$	d	$\{\tau, \rho\}$	$\{p/2.1, p/2.1\}$
U	$\{\pi, \phi\}$	$\{r/2.2, s/2.1\}$	f	$\{\pi, \phi, \sigma\}$	$\{r/2.2, t/2.1\}$

Alcuni confronti:

t'/t''	$(P' \setminus P''), (P' \cap P''), (P'' \setminus P')$	$(R' \setminus R''), (R' \cap R''), (R'' \setminus R')$
X/a	$\{\rho\}, \{\pi, \phi\}, \{\sigma, \tau\}$	$\emptyset, \{p/2.1, p/2.1, p/2.2\}, \emptyset$
Y/b	$\{\pi\}, \{\sigma, \tau\}, \emptyset$	$\emptyset, \{p/2.2, r/2.1, o/2.1\}, \emptyset$
Y/c	$\{\pi, \sigma, \tau\}, \emptyset, \{\phi\}$	$\{r/2.1, o/2.1\}, \{p/2.2\}, \{o/2.2\}$
Z/b	$\{\phi\}, \emptyset, \{\sigma, \tau\}$	$\{o/2.2\}, \{p/2.2\}, \{r/2.1, o/2.1\}$
Z/c	$\emptyset, \{\phi\}, \emptyset$	$\emptyset, \{p/2.2, o/2.2\}, \emptyset$
W/d	$\{\sigma\}, \{\tau\}, \{\rho\}$	$\{p/2.2\}, \{p/2.1, p/2.1\}, \emptyset$
U/f	$\emptyset, \{\pi, \phi\}, \{\sigma\}$	$\{s/2.1\}, \{r/2.2\}, \{t/2.1\}$

Corrispondenti valutazioni di similarità:

t'/t''	(n_c, l_c, m_c)	$\text{sf}_c(t', t'')$	(n_r, l_r, m_r)	$\text{sf}_r(t', t'')$	$\text{sf}_o(t', t'')$
X/a	(1, 2, 2)	0.55	(0, 3, 0)	0.80	1.35
Y/b	(1, 2, 0)	0.75	(0, 4, 0)	0.71	1.46
Y/c	(3, 0, 1)	0.45	(2, 1, 1)	0.27	0.72
Z/b	(1, 0, 2)	0.29	(1, 1, 2)	0.45	0.74
Z/c	(0, 1, 0)	0.67	(0, 2, 0)	0.75	1.42
W/d	(1, 1, 1)	0.50	(1, 2, 0)	0.68	1.18
U/f	(0, 2, 1)	0.68	(1, 1, 1)	0.50	1.18

3.2 Similarità Strutturale

Quando si cerca di valutare la similarità strutturale tra due formule, possono essere coinvolti molti oggetti, le cui relazioni reciproche rappresentano quindi un vincolo su come ognuno di quelli nella prima formula possa essere associato ad un altro nella seconda. Diversamente dal caso degli oggetti, quello che definisce la struttura di una formula è l'insieme dei predicati n -ari, ed in particolare il modo in cui questi sono applicati ai vari oggetti per metterli in relazione (un predicato applicato ad un numero di termini uguale alla sua arietà viene chiamato *atomo*). Questa è la parte più difficile, poichè le relazioni sono proprio la componente del paradigma del prim'ordine che causa indeterminazione nel mappare (parti di) una formula su (parti di) un'altra formula. D'ora in poi, chiameremo *compatibili* due (sotto-)formule del prim'ordine, che possono essere mappate tra di loro senza produrre incoerenze nell'associazione dei termini (in altre parole, un termine in una formula non può corrispondere a termini differenti in un'altra formula).

Dato un letterale n -ario, definiamo la sua *stella* come il multiset di predicati n -ari corrispondenti ai letterali ad esso connessi tramite qualche termine in comune (infatti, un predicato può apparire in più istanze multiple di questi letterali). Intuitivamente, la stella di un letterale è una vista in ampiezza di come il letterale è legato al resto della formula. La *similarità di stella* $\text{sf}_s(l', l'')$ tra

due letterali n -ari compatibili l' e l'' che hanno stelle S' ed S'' , rispettivamente, può essere calcolata come $\text{sf}(n_s, l_s, m_s)$ per i seguenti parametri:

$n_s = |S' \setminus S''|$ che esprime quante relazioni l' ha in più in C' rispetto a quelle che l'' ha in C'' (*residuo di stella* di l' rispetto a l'')
 $l_s = |S' \cap S''|$ che è il numero delle relazioni che hanno in comune l' in C' ed l'' in C'' ;
 $m_s = |S'' \setminus S'|$ che esprime quante relazioni l'' ha in più in C'' rispetto a quelle che l' ha in C' (*residuo di stella* di l'' rispetto a l').

Complessivamente, una valutazione più adeguata della similarità tra l' e l'' può essere ottenuta aggiungendo a questo risultato la similarità caratteristica e strutturale per tutte le coppie dei loro argomenti in posizioni corrispondenti:

$$\text{sf}_s(l', l'') = \text{sf}(n_s, l_s, m_s) + \sum_{t'/t'' \in \theta} \text{sf}_o(t', t'')$$

dove θ è l'insieme delle associazioni dei termini che mappano l' su l'' .

Ogni formula del prim'ordine può essere rappresentata come un grafo in cui gli atomi sono i nodi e gli archi connettono due nodi se e solo se questi sono relazionati in qualche modo. Ne consegue che un confronto tra due formule per valutare la loro similarità strutturale corrisponde al calcolo di un omomorfismo fra (sotto-)grafi, un problema noto essere *NP*-difficile in generale a causa della possibilità di mappare un (sotto-)grafo su un altro in molti modi differenti. Di conseguenza, si è interessati ad euristiche che possano dare suggerimenti significativi sulla sovrapposizione della struttura tra due formule con basso sforzo computazionale. In realtà, poggiando sul fatto che le clausole contengono un unico atomo nella testa ed una congiunzione di atomi nel corpo, si può sfruttare una rappresentazione a grafo più semplice rispetto a quella per formule generali, come descritto nel seguito. In particolare, si tratteranno solo clausole *connesse* (cioè il cui grafo associato è connesso), e si costruirà il grafo sulla base di una caratteristica semplice (per quanto riguarda i dettagli che esprime circa una formula), ma al tempo stesso potente (per quanto riguarda l'informazione che essa convoglia), che è la condivisione dei termini tra coppie di atomi.

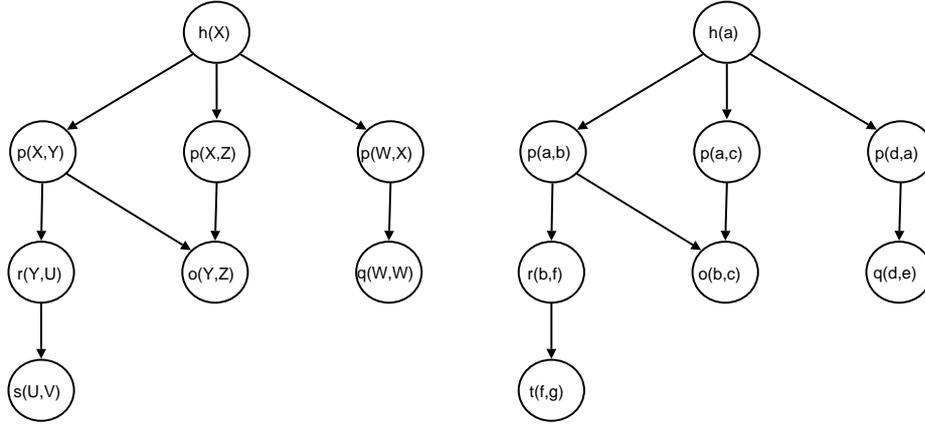
Data una clausola C , definiamo il suo *grafo associato* come $G_C = (V, E)$ con

- $V = \{l_0\} \cup \{l_i | i \in \{1, \dots, n\}, l_i \text{ costruito su predicati } k\text{-ari, } k > 1\}$ e
- $E \subseteq \{(a_1, a_2) \in V \times V \mid \text{terms}(a_1) \cap \text{terms}(a_2) \neq \emptyset\}$

dove $\text{terms}(a)$ denota l'insieme dei termini che appaiono come argomenti dell'atomo a . La strategia di selezione degli archi da rappresentare, schematizzata nell'Algoritmo 1, si basa sulla presenza di un unico atomo nella testa, che fornisce sia un punto di partenza che precise direzioni per attraversare il grafo, al fine di scegliere, tra le tante possibili, una prospettiva unica e ben definita sulla struttura della clausola. Più precisamente, costruiamo un grafo orientato aciclico, *stratificato* (cioè con l'insieme dei nodi partizionato) in modo tale che la testa sia l'unico nodo al livello 0 (primo elemento della partizione) ed ogni livello (elemento della partizione) successivo sia composto da nuovi nodi (non ancora

Algorithm 1 Costruzione del grafo associato a C

Require: $C = l_0 : -l_1, \dots, l_n$: Clause $i \leftarrow 0$; $Level_0 \leftarrow \{l_0\}$; $E \leftarrow \emptyset$; $Atoms \leftarrow \{l_1, \dots, l_n\}$ **while** $Atoms \neq \emptyset$ **do** $i \leftarrow i + 1$ $Level_i \leftarrow \{l \in Atoms \mid \exists l' \in Level_{i-1} \text{ s.t. } terms(l) \cap terms(l') \neq \emptyset\}$ $E \leftarrow E \cup \{(l', l'') \mid l' \in Level_{i-1}, l'' \in Level_i, terms(l') \cap terms(l'') \neq \emptyset\}$ $Atoms \leftarrow Atoms \setminus Level_i$ **end while**return $G = (\bigcup_i Level_i, E)$: graph associated to C

**Fig. 1.** Grafi associati alle clausole C (a sinistra) ed E (a destra)

raggiunti dagli archi) che hanno almeno un termine in comune con nodi del livello precedente. In particolare, ogni nodo nel nuovo livello è connesso tramite un arco entrante ad ogni nodo al livello precedente che ha tra i suoi argomenti almeno un termine in comune con esso.

Example 3. Costruiamo il grafo G_C , riportato in Figura 1. La testa rappresenta il livello 0 della stratificazione. Quindi, degli archi orientati possono essere introdotti da $h(X)$ a $p(X, Y)$, $p(X, Z)$ e $p(W, X)$, che sono i soli atomi ad avere X come argomento, il che produce il livello 1 della stratificazione dei termini. Adesso, il prossimo livello può essere costruito, aggiungendo archi orientati che vanno da atomi al livello 1 ad atomi non ancora considerati che condividono una variabile con loro: $r(Y, U)$ – fine di un arco che parte a $p(X, Y)$ –, $o(Y, Z)$ – fine di archi che partono da $p(X, Y)$ e $p(X, Z)$ – e $q(W, W)$ – fine di un arco che parte da $p(W, X)$. Il terzo (e ultimo) livello del grafo include l'unico atomo rimasto, $s(U, V)$ – avente un arco entrante da $r(Y, U)$.

Adesso, tutti i possibili cammini che partono dalla testa e raggiungono nodi *foglia* (quelli senza archi uscenti) possono essere interpretati come le componenti

base della struttura complessiva della clausola. Essendo questi cammini univocamente determinati, si riduce la quantità di indeterminazione nel confronto. Intuitivamente, un cammino descrive in profondità una porzione delle relazioni descritte nella clausola.

Date due clausole, C' e C'' , definiamo *intersezione* tra due cammini $p' = \langle l'_1, \dots, l'_{n'} \rangle$ in $G_{C'}$ e $p'' = \langle l''_1, \dots, l''_{n''} \rangle$ in $G_{C''}$ la coppia delle più lunghe sottosequenze iniziali compatibili di p' e p'' :

$p' \cap p'' = (p_1, p_2) = (\langle l'_1, \dots, l'_k \rangle, \langle l''_1, \dots, l''_k \rangle)$ s.t.
 $\forall i = 1, \dots, k : l'_1, \dots, l'_i$ compatibile con $l''_1, \dots, l''_i \wedge$
 $(k = n' \vee k = n'' \vee l'_1, \dots, l'_{k+1}$ incompatibile con $l''_1, \dots, l''_{k+1})$
e i due residui come le parti incompatibili restanti.

$p' \setminus p'' = \langle l'_{k+1}, \dots, l'_{n'} \rangle, p'' \setminus p' = \langle l''_{k+1}, \dots, l''_{n''} \rangle$

Quindi, la *similarità di cammini* tra p' e p'' , $\text{sf}_s(p', p'')$, può essere calcolata applicando (1) ai seguenti parametri:

$n_p = |p' \setminus p''| = n' - k$ è la lunghezza della sequenza finale incompatibile di p' rispetto a p'' (*residuo di cammino* di p' rispetto a p'');
 $l_p = |p_1| = |p_2| = k$ è la lunghezza della massima sequenza iniziale compatibile di p' e p'' ;
 $m_p = |p'' \setminus p'| = n'' - k$ è la lunghezza della sequenza finale incompatibile di p'' rispetto a p' (*residuo di cammino* di p'' rispetto a p').

più la similarità di stella di tutte le coppie di letterali nelle sequenze iniziali:

$$\text{sf}_p(p', p'') = \text{sf}(n_p, l_p, m_p) + \sum_{i=1, \dots, k} \text{sf}_s(l'_i, l''_i)$$

Example 4. Poichè la testa è unica (e quindi può essere associata univocamente), in seguito tratteremo solo i letterali del corpo per mostrare i criteri strutturali. La Tabella 2 riporta i confronti delle stelle per un campione di letterali in C ed E , mentre la Tabella 3 mostra alcuni confronti tra cammini.

Va notato che nessun criterio è di per sé nettamente discriminante, ma la loro cooperazione riesce con successo a distribuire i valori di similarità e a rendere le differenze sempre più chiare man mano che ciascuno viene composto con quelli precedenti.

Una generalizzazione può essere ora calcolata considerando le intersezioni dei cammini in base alla similarità decrescente, e aggiungendo alla generalizzazione parziale generata finora i letterali in comune per ogni coppia ogni volta che questi sono compatibili (vedere l'Algoritmo 2). Ulteriori generalizzazioni possono essere ottenute tramite backtracking. Questo permette anche, nel caso lo si desideri, di tagliare la generalizzazione quando si raggiunge una certa soglia di lunghezza, assicurando che solo le porzioni con similarità meno significative vengano rimosse.

Example 5. L'intersezione dei cammini con il più alto valore di similarità è C.3/E.3, e quindi la prima generalizzazione parziale risulta $\{p(X, Z), o(Y, Z)\}$,

Table 2. Similarità di stelle

C		E	
l'	S'	l''	S''
$p(X, Y)$	$\{p/2, p/2, r/2, o/2\}$	$p(a, b)$	$\{p/2, p/2, r/2, o/2\}$
$p(X, Z)$	$\{p/2, p/2, o/2\}$	$p(a, c)$	$\{p/2, p/2, o/2\}$
$r(Y, U)$	$\{p/2, o/2, s/2\}$	$r(b, f)$	$\{p/2, o/2, t/2\}$

Alcuni confronti:

l'	l''	$(S' \setminus S''), (S' \cap S''), (S'' \setminus S')$
$p(X, Y)$	$p(a, b)$	$\emptyset, \{p/2, p/2, r/2, o/2\}, \emptyset$
$p(X, Y)$	$p(a, c)$	$\{r/2\}, \{p/2, p/2, o/2\}, \emptyset$
$p(X, Z)$	$p(a, c)$	$\emptyset, \{p/2, p/2, o/2\}, \emptyset$
$p(X, Z)$	$p(a, b)$	$\emptyset, \{p/2, p/2, o/2\}, \{r/2\}$
$r(Y, U)$	$r(b, f)$	$\{s/2\}, \{p/2, o/2\}, \{t/2\}$

Corrispondenti valutazioni di similarità:

l'	l''	(n_s, l_s, m_s)	$\text{sf}(n_s, l_s, m_s)$	sf_o		$\text{sf}_s(l', l'')$
$p(X, Y)$	$p(a, b)$	(0, 4, 0)	0.71	$\text{sf}_o(X, a) = 1.35$	$\text{sf}_o(Y, b) = 1.46$	3.52
$p(X, Y)$	$p(a, c)$	(1, 3, 0)	0.73	$\text{sf}_o(X, a) = 1.35$	$\text{sf}_o(Y, c) = 0.72$	2.80
$p(X, Z)$	$p(a, c)$	(0, 3, 0)	0.80	$\text{sf}_o(X, a) = 1.35$	$\text{sf}_o(Z, c) = 1.42$	3.57
$p(X, Z)$	$p(a, b)$	(0, 3, 1)	0.73	$\text{sf}_o(X, a) = 1.35$	$\text{sf}_o(Z, b) = 0.74$	2.82
$r(Y, U)$	$r(b, f)$	(1, 2, 1)	0.60	$\text{sf}_o(Y, b) = 1.46$	$\text{sf}_o(U, f) = 1.18$	3.24

con associazioni $\{X/a, Y/b, Z/c\}$. Subito dopo si considera C.2/E.2, le cui associazioni sono compatibili con quelle attuali, cosicchè esso contribuisce con $\{p(X, Y)\}$ alla generalizzazione (non ci sono nuove associazioni). Poi viene C.1/E.1, che essendo compatibile estende la generalizzazione aggiungendo $\{r(Y, U)\}$ e le associazioni con $\{U/f\}$. A questo punto tocca a C.1/E.2 e quindi a C.2/E.1, che sono compatibili ma ridondanti, e quindi non aggiungono niente alla generalizzazione attuale (né alle associazioni). In seguito viene considerato C.4/E.4, che è compatibile ed estende la generalizzazione e le associazioni attuali, rispettivamente, con $\{p(W, X)\}$ e $\{W/d\}$. Infine, vengono considerati C.3/E.2, C.2/E.3, C.3/E.1 e C.1/E.3, che però vengono scartati perché le relative associazioni sono incompatibili.

4 Lavori correlati

Molte misure di distanza sono state sviluppate per rappresentazioni attributo-valore, ma pochi lavori hanno affrontato la definizione di misure di similarità o di distanza per descrizioni del prim'ordine. In [4], viene proposta una misura di distanza basata sulla teoria delle probabilità applicata alle componenti della formula. Rispetto a questa misura, la nostra funzione non richiede assunzioni o ipotesi semplificative (come per esempio indipendenza statistica, mutua esclusione), il che facilita la gestione delle probabilità, né alcuna conoscenza *a priori* del linguaggio di rappresentazione (come per esempio i tipi del dominio). Inoltre, il nostro approccio non necessita che l'utente imponga dei pesi sull'importanza dei predicati e non è basato sulla presenza di relazioni obbligatorie, come per la sottoclausola $G1$ in [4].

Table 3. Similarità di cammini

Path No.	C	E
1.	$\langle p(X, Y), r(Y, U), s(U, V) \rangle$	$\langle p(a, b), r(b, f), t(f, g) \rangle$
2.	$\langle p(X, Y), o(Y, Z) \rangle$	$\langle p(a, b), o(b, c) \rangle$
3.	$\langle p(X, Z), o(Y, Z) \rangle$	$\langle p(a, c), o(b, c) \rangle$
4.	$\langle p(W, X), q(W, W) \rangle$	$\langle p(d, a), q(d, e) \rangle$

p' p''	$p' \cap p''$	$p' \setminus p''$ $p'' \setminus p'$	$\theta_{p' \cap p''}$	$(n, l, m)_p$ $\text{sf}_p(p', p'')$
C.1	$\langle p(X, Y), r(Y, U) \rangle$	$\langle s(U, V) \rangle$	$\{X/a, Y/b, U/f\}$	(1, 2, 1)
E.1	$\langle p(a, b), r(b, f) \rangle$	$\langle t(f, g) \rangle$		7.36
C.1	$\langle p(X, Y) \rangle$	$\langle r(Y, U), s(U, V) \rangle$	$\{X/a, Y/b\}$	(2, 1, 1)
E.2	$\langle p(a, b) \rangle$	$\langle o(b, c) \rangle$		3.97
C.2	$\langle p(X, Y) \rangle$	$\langle o(Y, Z) \rangle$	$\{X/a, Y/b\}$	(1, 1, 2)
E.1	$\langle p(a, b) \rangle$	$\langle r(b, f), t(f, g) \rangle$		3.97
C.2	$\langle p(X, Y), o(Y, Z) \rangle$	$\langle \rangle$	$\{X/a, Y/b, Z/c\}$	(0, 2, 0)
E.2	$\langle p(a, b), o(b, c) \rangle$	$\langle \rangle$		7.95

Algorithm 2 Generalizzazione basata sulla similarità

Require: C : Rule; E : Example

$P_C \leftarrow \text{paths}(C)$; $P_E \leftarrow \text{paths}(E)$;

$P \leftarrow \{(p_C, p_E) \in P_C \times P_E \mid p_C \cap p_E \neq (\langle \rangle, \langle \rangle)\}$;

$G \leftarrow \emptyset$; $\theta \leftarrow \emptyset$

while $P \neq \emptyset$ **do**

$(\bar{p}_C, \bar{p}_E) \leftarrow \text{argmax}_{(p_C, p_E) \in P} (\text{sf}(p_C, p_E))$

$P \leftarrow P \setminus \{(\bar{p}_C, \bar{p}_E)\}$

$(\bar{q}_C, \bar{q}_E) \leftarrow \bar{p}_C \cap \bar{p}_E$

$\theta_q \leftarrow \text{substitution s.t. } q_C = q_E$

if θ_q compatible with θ **then**

$G \leftarrow G \cup q_C$; $\theta \leftarrow \theta \cup \theta_q$

end if

end while

return G : generalization between C and E

Molti sistemi di apprendimento supervisionato dimostrano l'importanza di una misura di similarità. Per esempio, *KGB* [1] usa una funzione di similarità, parametrizzata dall'utente, per guidare la generalizzazione; le nostre idee di similarità caratteristica e relazionale sono molto simili a questa, ma il calcolo della similarità è più immediato. Mentre *KGB* non può gestire informazione negativa nelle clausole, il nostro approccio può essere facilmente esteso a questo scopo. Il classificatore k-Nearest Neighbor *RIBL* [2] è basato su una versione modificata della funzione proposta in [1]. L'idea di base è che la similarità di oggetti dipende dalla similarità dei valori dei loro attributi (la similarità della loro dimensione), e, ricorsivamente, dalla similarità degli oggetti a questi connessi. Tale propagazione pone il problema della indeterminazione (incertezza) nelle associazioni, che il nostro approccio evita grazie al diverso approccio strutturale.

[9] presenta un approccio per l'induzione di distanze su esempi in logica del prim'ordine, che dipende dal pattern che discrimina i concetti-obiettivo. Si selezionano k clausole e i valori di verità che esprimono se la clausola copre o meno l'esempio sono utilizzati come k componenti di una distanza tra gli esempi nello spazio $\{0, 1\}^k$. [6] organizza i termini in una gerarchia in base all'importanza e propone una distanza tra termini basata su interpretazioni ed una funzione che mappa ogni espressione semplice in un numero naturale. [7] presenta una funzione di distanza tra atomi basata sulla differenza rispetto alla loro lgg e usa la funzione per calcolare distanze tra clausole. La funzione consiste in una coppia: la prima componente estende la distanza in [6] ed è basata sulle differenze tra i funtori nei due termini, mentre la seconda componente è basata sulle differenze nelle occorrenze delle variabili e permette di differenziare casi che la prima componente non può discriminare.

5 Sperimentazioni

All scopo di verificare se i criteri proposti siano in grado di produrre indizi significativi sulla similarità tra due strutture, la procedura di generalizzazione guidata dalla similarità è stata confrontata con una versione precedente (non guidata) utilizzata nel sistema INTHELEX [3]. Il sistema è stato impostato in maniera tale che, quando la prima generalizzazione prodotta dalla procedura guidata non era coerente con tutti gli esempi negativi, il sistema poteva cercare generalizzazioni più specifiche in backtracking¹. Per valutare l'accuratezza predittiva è stata usata la 10-fold cross-validation.

Un primo confronto ha riguardato il classico dataset ILP della Mutagenesi, sul quale il sistema ha raggiunto un'accuratezza predittiva leggermente migliore (87%) rispetto alla versione non guidata (86%) sfruttando solo il 30% del tempo di esecuzione (4035 secondi anzichè 13720). E' importante notare che la prima generalizzazione trovata era sempre corretta, e quindi non è stato mai necessario fare backtracking per cercare altre alternative, mentre la versione non guidata ha dovuto calcolare 5815 ulteriori generalizzazioni per gestire quei casi in cui la prima generalizzazione trovata era incoerente con gli esempi pregressi. Questo conferma che la misura di similarità è in grado di guidare la corretta identificazione delle sotto-parti corrispondenti nelle descrizioni chimiche.

Altre sperimentazioni sono state condotte su un dataset² contenente descrizioni del layout della prima pagina di 122 articoli scientifici appartenenti a 4 classi diverse. Da questi sono stati ricavati 488 esempi positivi/negativi per apprendere regole di classificazione dei documenti, e 1488 esempi per apprendere regole che identificano 12 tipi di (etichette per le) componenti logiche dei

¹ Per evitare che il sistema impiegasse troppo tempo in generalizzazioni difficili, è stato impostato in maniera da abbandonare la generalizzazione attuale e cercare di generalizzare un'altra clausola (se presente) non appena venivano trovate in backtracking 500 (sotto-)generalizzazioni ridondanti oppure ne venivano calcolate 50 nuove ma incoerenti

² <http://lacam.di.uniba.it:8000/systems/inthelex/index.htm#datasets>

Table 4. Risultati sperimentali

		Ratio	Time (sec.)	Cl	Gen	Exc ⁺	Spec ⁺	Spec ⁻	Exc ⁻	Acc
Classificazione	SF	90.52	579	8	47(+0)	0	2	0	0	0.94
	I	70.22	137	7	33(+100)	0	1	1	1	0.97
	S80	73.63	206	7	33(+13)	0	0	1	1	0.97
Etichettatura	SF	91.09	22220	36	180(+0)	0	8	3	3	0.89
	I	68.85	33060	39	137(+5808)	0	15	11	12	0.93
	S80	71.75	15941	54	172(+220)	0	14	8	2	0.93

documenti (per esempio, titolo, autore, abstract etc). I risultati sono riportati nella Tabella 4.

La prima questione è se la funzione di similarità proposta fosse in grado di guidare la procedura di ricerca verso l'identificazione delle sotto-parti appropriate da mettere in corrispondenza nelle due descrizioni che si stavano confrontando. Poiché l'associazione corretta non è nota, questo può essere valutato solo indirettamente. Un modo per far ciò è valutare il fattore di compressione della generalizzazione guidata, ossia la porzione di letterali nelle clausole da generalizzare che è preservata dalla generalizzazione. Infatti, poiché ogni generalizzazione in INTHELEX deve essere un sottoinsieme di ciascuna delle clausole da generalizzare (a causa della assunzione di Object Identity), più letterali vengono preservati da queste clausole dalla generalizzazione, meno generale sarà la generalizzazione. Più formalmente, valutiamo la compressione come il rapporto tra la lunghezza della generalizzazione e quella della clausola più corta da generalizzare: più grande è questo valore, più sicuri si può essere che le associazioni corrette siano state trovate grazie ai criteri e alla formula di similarità. Ovviamente, più grande è la differenza in lunghezza tra due clausole da generalizzare, più grande sarà l'incertezza presente, e quindi più difficile sarà identificare appropriatamente le parti corrispondenti tra loro. È interessante notare che sul dataset dei documenti la generalizzazione guidata dalla similarità (SF) ha preservato in media più del 90% dei letterali della clausola più corta, con un massimo di 99,48% (193 letterali su 194, contro un esempio da 247) e solo 0,006 di varianza. Di conseguenza, ci si può aspettare che le generalizzazioni prodotte siano effettivamente le meno generali (o le approssimino molto). Per verificarlo, invece di calcolare la vera generalizzazione meno generale per ogni generalizzazione effettuata e confrontarla con quella prodotta dalla procedura guidata, il che sarebbe stato computazionalmente pesante e avrebbe richiesto di modificare il comportamento del sistema, si è calcolato quanti backtracking il sistema doveva calcolare per raggiungerne una più specifica. Il risultato è stato che, entro il limite dato, non è stata trovata nessuna altra generalizzazione, il che suggerisce che la prima generalizzazione trovata è probabilmente molto vicina a (ed in effetti molto spesso è) la generalizzazione meno generale. Da notare che applicando della formula di similarità di Tverski [10] (che rappresenta lo stato dell'arte nell'attuale letteratura) allo stesso problema, si sono ottenute sempre generalizzazioni più corte di quelle ottenute utilizzando la nostra formula.

Comunque, essendo tali generalizzazioni molto specifiche, la loro accuratezza predittiva risulta più bassa di quelle ottenute dall'algoritmo INTHELEX non guidato (I), il che probabilmente è dovuto alla necessità di un maggior numero di esempi per convergere a definizioni predittive oppure all'overfitting. Per questo motivo, la procedura di generalizzazione guidata dalla similarità è stata impostata per scartare almeno il 20% dei letterali della più corta clausola originale ($S80$): questo ha portato alla stessa accuratezza predittiva di I , ed ha ridotto drasticamente il tempo di esecuzione rispetto alla versione senza vincoli (ed anche rispetto ad I nel caso del compito di etichettatura). Anche il numero delle generalizzazioni decresce, sebbene al costo di uno sforzo maggiore di specializzazione (anche per mezzo di letterali negativi che sono gestiti come spiegato nella sezione precedente), che è in ogni caso più efficace rispetto a I (in particolare per quanto riguarda le eccezioni negative). Nel compito di classificazione il numero delle clausole decresce leggermente, mentre in quello di etichettatura il numero cresce di 15, ma viene equilibrato da 10 eccezioni negative in meno. È importante notare che le generalizzazioni trovate sono ancora molto aderenti agli esempi, come suggerisce il fatto che nelle sperimentazioni di classificazione sono state trovate in backtracking (e testate per verificarne la correttezza) solo 13 generalizzazioni più specifiche (delle quali solo 1 corretta), mentre I ne ha trovate 100, di cui 6 corrette. Nel task di etichettatura, $S80$ ha trovato 220 ulteriori generalizzazioni candidate (di cui 6 corrette) contro 5808 (di cui 39 corrette) di I . Questo dimostra che si potrebbe anche evitare il backtracking con poca perdita in $S80$, ma non in I .

Un altro importante parametro per valutare l'utilità della funzione proposta e della relativa strategia è il tempo di esecuzione. La Tabella 4 mostra che, nel task di etichettatura, utilizzando la funzione di similarità si raggiunge un risparmio che va da $1/3$ fino a $1/2$ del tempo totale, dell'ordine di ore. Le prestazioni nel task di classificazione sono in ogni caso confrontabili (differenze dell'ordine di decimi di secondo) il che si può probabilmente spiegare col fatto che tale task è molto più facile, per cui resta poco spazio per miglioramenti e il tempo necessario per calcolare la formula annulla il risparmio.

6 Conclusioni

Le relazioni, in Logica del Prim'Ordine, portano ad indeterminazione nel mappare porzioni di una descrizione su un'altra descrizione. In questo articolo si sono identificati un insieme di criteri ed una formula per il confronto tra clausole, e sono stati utilizzati per guidare una procedura di generalizzazione indicando le sottoparti delle descrizioni che più verosimilmente corrispondono l'una all'altra, basandosi solo sulla loro struttura sintattica.

Stando ai risultati sperimentali, la generalizzazione basata sulla similarità è in grado di catturare le associazioni corrette e può produrre la stessa accuratezza predittiva della versione non guidata, ma producendo teorie più 'pulite' e riducendo drammaticamente la quantità di tempo necessario per convergere (il risparmio di tempo cresce al crescere della complessità del problema).

Come lavoro futuro si potrebbe raffinare la metodologia di calcolo della similarità. Inoltre, la nuova misura di similarità potrebbe essere messa alla prova con altre applicazioni, quali il matching flessibile, il clustering concettuale oppure l'instance-based learning.

References

- [1] G. Bisson. Learning in FOL with a similarity measure. In W.R. Swartout, editor, *Proc. of AAAI-92*, pages 82–87, 1992.
- [2] W. Emde and D. Wettschereck. Relational instance based learning. In L. Saitta, editor, *Proc. of ICML-96*, pages 122–130, 1996.
- [3] F. Esposito, S. Ferilli, N. Fanizzi, T. Basile, and N. Di Mauro. Incremental multi-strategy learning for document processing. *Applied Artificial Intelligence Journal*, 17(8/9):859–883, 2003.
- [4] F. Esposito, D. Malerba, and G. Semeraro. Classification in noisy environments using a distance measure between structural symbolic descriptions. *IEEE Transactions on PAMI*, 14(3):390–402, 1992.
- [5] Dekang Lin. An information-theoretic definition of similarity. In *Proc. 15th International Conf. on Machine Learning*, pages 296–304. Morgan Kaufmann, San Francisco, CA, 1998.
- [6] S. Nienhuys-Cheng. Distances and limits on herbrand interpretations. In D. Page, editor, *Proc. of ILP-98*, volume 1446 of *LNAI*, pages 250–260. Springer, 1998.
- [7] J. Ramon. *Clustering and instance based learning in first order logic*. PhD thesis, Dept. of Computer Science, K.U.Leuven, Belgium, 2002.
- [8] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In *Inductive Logic Programming*, pages 64–90. Academic Press, 1992.
- [9] M. Sebag. Distance induction in first order logic. In N. Lavrač and S. Džeroski, editors, *Proc. of ILP-97*, volume 1297 of *LNAI*, pages 264–272. Springer, 1997.
- [10] A. Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.

Constraint-based simulation of biological systems described by Molecular Interaction Maps

Luca Bortolussi¹, Simone Fonda⁴, and Alberto Policriti^{2,3}

¹ Dept. of Mathematics and Computer Science, University of Trieste, Italy.

² Dept. of Mathematics and Computer Science, University of Udine, Italy

³ Istituto di Genomica Applicata, Udine, Italy

⁴ Dept. of Computer Science, University of Pisa, Italy.

Abstract. We present an application of stochastic Concurrent Constraint Programming, defining an encoding of biochemical networks described by the graphical notation of Molecular Interaction Maps. Such maps are compact, as they represent implicitly a wide set of reactions, and therefore not easy to simulate with standard tools. With our encoding, we are able to stochastically simulate these maps implicitly, without generating the full list of reactions.

1 Introduction

The aim of this work is the simulation of biological regulatory networks described by the graphical notation of Molecular Interaction Maps (MIM) [19]. In order to achieve this goal, we define an encoding of such maps in stochastic Concurrent Constraint Programming (sCCP) [2], a stochastic, concurrent, and constraint-based programming language.

In scientific literature, many mathematical tools have been proposed to describe, model, and simulate the behaviors of biological systems, all sharing the common goal of organizing and analyzing the available knowledge and understanding of such systems [7, 14, 15]. As happens with computer programming languages, one formalism could be better suited than another for a specific purpose, depending on the features it provides to its users. When choosing a formalism many parameters must be taken into account: continuous or discrete representation of the entities, expressivity, availability of mathematical analysis techniques, possibility of a computer aided simulation and visualization, and so on. Other important properties of a modeling language concern the process of writing and maintaining a model: compositional languages are usually preferable for systems composed of many interacting parts, like biological ones [23]. Formalisms can also differ in the size of produced models; generally, the more compact the better. Moreover, the same system can be modeled at different levels of detail; for example, a cell can be described as a functional black box or specifying its internal behaviors and mechanisms in detail.

As the focus of modeling in biology is to understand dynamical properties of biological systems, most of the modeling languages have a semantic based on ordinary differential equations [29] or stochastic processes [30]. Stochastic processes, usually continuous-time Markov Chains [21], have been used in biochemistry and biology since a long time, especially after the publishing of a simple and efficient simulation algorithm by Gillespie [11, 12]. These models are generally more realistic than the ones based on Ordinary Differential Equations (ODEs), as they represent molecules as discrete quantities (compared to the continuous approximation of ODEs) and, moreover, they have a noisy evolution (compared to the determinism of ODEs), an important issue in biology [20, 30].

Generally, both stochastic and differential models can be obtained in a canonical way when the list of reactions of the system is fully specified [8, 30] (i.e., we need to specify all the possible ways of reacting of all the possible reactants). A big problem with this approach is the *cost* of the notation: in some cases, the effort required for a complete specification can be overwhelming; this is true especially for bio-regulatory networks, due to the central role played by multi-molecular complexes and protein modifications [16, 17, 19, 9, 1]. In fact, even a small set of proteins can potentially generate a big number of different complexes, of which only a few different kinds may be present at a certain time. Manually listing all these complexes can be a very hard task.

In order to tackle this problem, we need a formalism that can work at an higher level of abstraction, representing entities and behaviors in an implicit way. One possibility is offered by the Molecular Interaction Maps [17, 19]: they are a compact graphical notation proposed by Kohn, with the goal to be clear, easy to use, compact, unambiguous, and (hopefully) widespread. In the author's expectations, the equivalent of electronic circuit diagrams for biologists.

The contribution of this work is the definition of an encoding of MIM in sCCP, a stochastic extension of Concurrent Constraint Programming [2], already used to model biological systems at different degrees of complexity [4, 3]. The crucial ingredient of such an encoding will be the *implicit* representation of complexes and reactions. Essentially, molecular complexes will be represented by graphs, modified locally by reactions. The idea of this approach originates from the work of Danos and Laneve [6] and from the attempt of encoding their κ -calculus in sCCP [3]. Before presenting the encoding we discuss Molecular Interaction Maps in more detail (Section 2) and address some issues related to their semantic (Section 3). Then, after briefly recalling sCCP (Section 4), we present the main ideas of the implicit simulation of MIMs in sCCP (Section 5). This is the core of the paper where we try to maintain in the encoding—and, consequently, in the implementation—the feature characterizing MIMs (namely their implicitness) providing, at the same time, a stochastic based engine for their simulation. Preliminary experimental results on application of our technique (Section 6) and a comparison of our approach with some related work (Section 7) are then presented. Finally, conclusions and future directions of work can be found in Section 8.

2 Molecular Interaction Maps

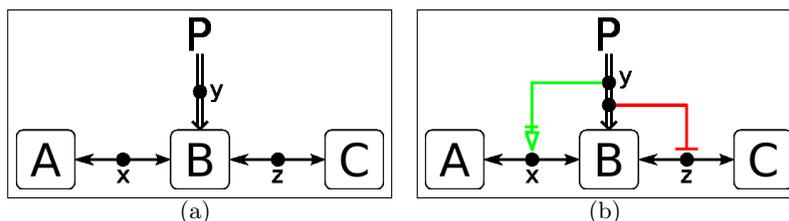


Fig. 1. Two very simple MIMs

We briefly introduce the MIM notation; the interested reader is referred to [19] for a detailed presentation. A MIM is essentially a graph, where nodes correspond to different (basic) molecular species, linked by different kinds of connecting lines, see Figure 1 for an example. The notation follows few general principles: to keep the diagram compact an *elementary molecular species*, like A , B or C in Figure 1(a), generally occur in *only one place* on a map. Different interactions between molecular species, instead, are distinguished by different lines and arrowheads; for instance, the double line in Figure 1(a) represents a covalent modification of B (in this case a *phosphorylation*, i.e. the attachment of a phosphate group at a specific place in the protein), while the single double-barbed arrows denote complexation operations. *Complex molecular species*, or simply complexes, are created as a consequence of interactions and are indicated by small circles on the corresponding interaction line; e.g. in Figure 1(a) x represents the complex $A : B$, the result of a complexation between A and B , while y represents the phosphorylated B molecule (denoted hereafter with pB).

In general, there are two types of interaction lines: *reactions* and *contingencies*. The former operate on molecular species, the latter on reactions or other contingencies. The former operate on molecular species, the latter on reactions or other contingencies. The red line with a T-shaped end of Figure 1(b) is an *inhibition* line; it states that phosphorylated B (indicated by y) cannot bind to C , (in fact the line terminates on the barbed arrow connecting B and C). Another contingency symbol of Figure 1(b) is the arrow with a bar preceding its empty arrowhead, terminating in x . This line represents a *requirement*: B must be phosphorylated in order to bind to A . Note that multiple nodes on an interaction line represent exactly the same molecular species. In Figure 2 we list a subset of arrows, taken from [19].

In order to fully explain the differences between Figure 1(a) and Figure 1(b), we need to introduce the *interpretations* of MIMs. The MIM notation, in fact, can be equipped with two different interpretations: *explicit* and *combinatorial*. In the explicit interpretation, an interaction line applies only to the molecular species directly connected to it. Looking again at Figure 1(a), we can say that

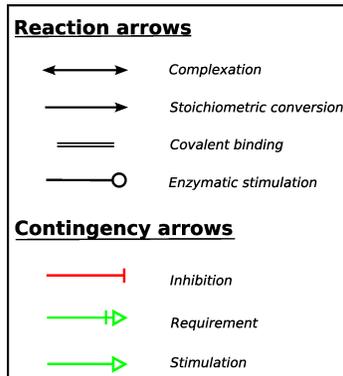


Fig. 2. A restricted list of reaction and contingency arrows of MIMs.

B can bind to A (forming the complex $A : B$) or to C (forming the complex $B : C$) but there is no way the complex $A : B : C$ will be formed. Moreover, if B is phosphorylated it cannot bind neither to A nor to C .

In the combinatorial interpretation, an interaction line represents a functional connection between domains or sites that (unless otherwise indicated) is independent of the modifications or bindings of the directly interacting species. In this way the map of Figure 1(a) states that A can bind to B , independently of the state of B . For instance, B could be phosphorylated and thus forming the $A : pB$ complex, or it could be bound to C , resulting in the $A : B : C$ complex. The first interpretation needs every interaction to be explicitly defined, in a “what is depicted is what happens” fashion, and this requires the use of many reaction lines. On the other hand, the combinatorial interpretation takes the opposite point of view, i.e. “what is not explicitly forbidden does happen”. In this case the behavior can be specialized by means of contingency lines.⁵

In Table 1 we summarize the differences between the two interpretations. While the two interpretations of the map of Figure 1(a) are different, the map of Figure 1(b) has exactly the same behavior under both interpretation, thanks to the use of inhibition and requirement symbols. Moreover, the combinatorial interpretation represents implicitly a large number of molecules: a single complexation arrow usually handles, on both sides, a big set of reactants. Consider, for example, the arrow connecting A to B in Figure 1(a); in the combinatorial interpretation it represents four reactions, namely the complexation of A with B , pB , $B : C$ and $pB : C$.

In principle, it is always possible to create an explicit MIM with the same behaviors of a combinatorial MIM, introducing more reaction arrows and con-

⁵ There is also a third layer of interpretation, called *heuristic* in [18], used mainly to structure and organize available knowledge of biological systems. In this interpretation, all interactions possible in the combinatorial case and forbidden in the explicit one are left unspecified, thus representing incomplete knowledge.

Complexes	Figure 1(a)		Figure 1(b)	
	Expl.	Comb.	Expl.	Comb.
A:B	✓	✓		
B:C	✓	✓	✓	✓
A:pB		✓	✓	✓
pB:C		✓		
A:B:C		✓		
A:pB:C		✓		

Table 1. Different interpretations of MIMs of Figure 1

tingencies. Explicit MIMs can be easily translated into a set of ODEs or into an explicit stochastic model for computer simulation, see [16]. Unfortunately, expliciting a combinatorial MIM is a non-trivial job, due to the combinatorial explosion of reaction arrows and contingencies needed in the map. In addition, an explicit ODE-based (or stochastic) simulation, like the one described in [16], requires to consider all possible molecular complexes that can be generated in the system as reactants or products of some reaction. However, at a given time, usually only a small subset of these complexes is present, hence the effort needed to generate explicitly this large number of complexes is largely unmotivated.

Our goal is precisely to define a simulation operating directly at the level of the combinatorial interpretation of MIMs, hence this is the interpretation we will consider in the rest of the paper. The advantages of this choice are clear: during each stage of a simulation, we need to represent only the complexes present in the system at that time. Moreover, models can be defined using the compact notation of combinatorial MIMs, hence the resulting map is usually smaller, thereby easier to build and understand.

3 Removing ambiguity from MIMs

The biggest obstacle towards the definition of an implicit simulation is, unfortunately, in the MIM notation itself. In fact, *the combinatorial interpretation is far from being a rigorous semantic*: combinatorial MIMs are intrinsically ambiguous, as there are many cases in which the exact behavior of the map is not defined. Moreover, some arrows in the MIM notation are just “syntactic sugar”, as their behavior can be defined in terms of other arrows. To tackle these problems, we have defined a set of *graph rewriting rules*, disambiguating some cases in what we deem a biologically plausible way and removing redundant arrows. This can be seen as the first step towards the definition of a formal semantic for MIMs. We discuss some examples in the following.

The MIM of Figure 3(a) represents a situation in which A can bind to B , forming a complex $(A : B)$ that can bind to C . Writing this in the standard, self-explanatory, chemical notation, we obtain $A + B \rightarrow A : B$ and $(A : B) + C \rightarrow A : B : C$. Biologically, every complex formed this way can, in principle, be

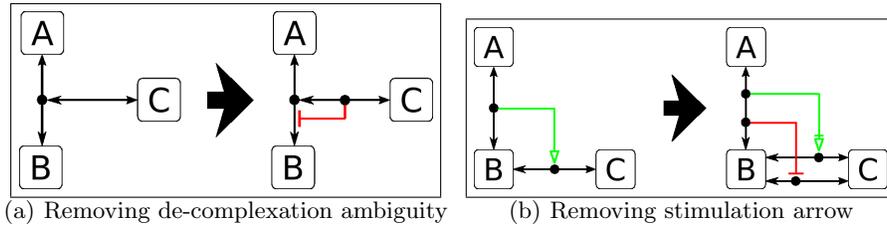


Fig. 3. An example of graph rewriting rules for MIMS

de-complexed, meaning that the map of Figure 3(a) describes also the inverse reactions $A : B : C \rightarrow (A : B) + C$ and $A : B \rightarrow A + B$. It is not clear, however, if the link between A and B in $A : B : C$ could be broken or not. We assume that this is not possible, because the map states that C can bind only to $A : B$ and not to A nor B alone, thus it plausibly describes a biological mechanism in which C binds to a modified structure containing A and B , a structure that cannot disappear while bound to C .⁶

In order to make this choice explicit, we add an inhibition symbol from the node representing the $A : B : C$ complex to the $A : B$ complexation arrow, thus preventing de-complexation of $A : B$ when C is bound. This is an example of a very simple graph rewriting rule in order to eliminate ambiguity.

Figure 3(b) left, shows a MIM where the $A : B$ complex *stimulates* the $B : C$ bond (stimulation is represented with an arrow with empty arrowhead). The intended meaning of stimulation is that B is more likely to bind to C when it is complexed with A . We can see this mechanism in an alternative way, using two complexation arrows between B and C , one representing the slower reaction and the other one representing the faster reaction. These arrows need to be mutually exclusive, see Figure 3(b) right. This second rewriting rule reduces the number of symbols of maps, simplifying the encoding in sCCP. Therefore, the two examples given are representative of two classes of rewriting rules: the first tackling ambiguity, the second simplifying the notation.

4 Stochastic Concurrent Constraint Programming

Concurrent Constraint Programming (CCP [27]) is a process algebra having two distinct entities: agents and constraints. Constraints are interpreted first-order logical formulae, stating relationships among variables (e.g. $X = 10$ or $X + Y < 7$). Agents in CCP, instead, have the capability of adding constraints (`tell`) into a “container” (the *constraint store*) and checking if certain relations

⁶ Note that different behaviors can be obtained changing the topology of the map. For instance, if C binds only to A instead of binding to $A : B$, the complex $A : B : C$ can be broken also in $(A : C) + B$.

$Program = D.A$
$D = \varepsilon \mid D.D \mid p(\mathbf{x}) : -A$
$A = \mathbf{0} \mid \text{tell}_\infty(c).A \mid M \mid \mathbf{local} \ x.A \mid A \parallel A$
$M = \pi.G \mid M + M$
$\pi = \text{tell}_\lambda(c) \mid \text{ask}_\lambda(c)$
$G = \mathbf{0} \mid \text{tell}_\infty(c).G \mid p(\mathbf{y}) \mid M \mid \mathbf{local} \ x.G \mid G \parallel G$

Table 2. Syntax of of sCCP.

are entailed by the current configuration of the constraint store (**ask**). The communication mechanism among agents is therefore asynchronous, as information is exchanged through global variables. In addition to **ask** and **tell**, the language has all the basic constructs of process algebras: non-deterministic choice, parallel composition, procedure call, plus the declaration of local variables. Moreover, constraints of the store can be defined using the computational machinery of Prolog [28, 27].

The stochastic version of CCP (sCCP [2, 4]) is obtained by adding a stochastic duration to all instructions interacting with the constraint store \mathcal{C} , i.e. **ask**, **tell**. Each instruction has an associated random variable, exponentially distributed with rate given by a function associating a real number to each configuration of the constraint store: $\lambda : \mathcal{C} \rightarrow \mathbb{R}^+$. Its syntax is defined by the grammar of Table 2.

The underlying semantic model of the language (defined via structural operational semantic, cf. [2]) is a Continuous Time Markov Chain [21] (CTMC), i.e. a stochastic process whose temporal evolution is a sequence of discrete jumps among states in continuous time. States of the CTMC correspond to configurations of the sCCP-system, consisting in the current set of processes and in the current configuration of the constraint store. The next state is determined by a race condition between all active instructions such that the fastest one is executed, like in stochastic π -calculus [22] or PEPA [13]. More details on sCCP and on its operational semantics can be found in [3].

Time-varying quantities, an important ingredient to deal with biological systems [4], are modeled as *stream variables*, i.e. growing lists with an unbounded tail.

In [3, 4] we argued that sCCP can be conveniently used for modeling a wide range of biological systems, like biochemical reactions, genetic regulatory networks, the formation of protein complexes, and the process of folding of a protein. In fact, while maintaining the compositionality of process algebras, the presence of a customizable constraint store and of variable rates gives a great flexibility to the modeler.

5 Encoding MIMs in sCCP

In this section we present the basic ideas for the direct simulation of MIM in sCCP. The starting point is the definition of a suitable representation of molecules and complexes. Actually, with respect to other process algebras like π -calculus [22], sCCP offers a crucial ingredient in this direction, namely the presence of the constraint store. In fact, the store is customizable and every kind of information can be represented by the use of suitable constraints, i.e. logical predicates. Manipulating and reasoning on such information can be performed in a logic programming style [27]. These ingredients make the constraint store an extremely flexible tool that can be naturally used to represent the data structures needed to operate on MIMs.

The idea to simulate MIMs is simple: we operate directly on the graphical representation. Of course, a MIM contains all possible interactions of the system, hence the graphical representation used in the simulation must be specialized to single molecules and complexes. Complexes, in particular, can be seen as *graphs*, with *nodes representing the basic molecules* (i.e. the proteins) of the complex, and with *edges representing the chemical bonds* tying them together. Such graphs will be referred in the following as *complex-graphs*.

Recalling Figure 1 and the combinatorial interpretation of Section 2, we can easily convince ourselves that molecular species are defined by the collection of their interaction sites. In our encoding, these sites are represented by *ports*, having the following properties:

- each port (or better, *port-type*) is the terminating point of one single arrow in the MIM⁷;
- each port-type is characterized by a unique identifier, called *port-type-id*, and by a boolean variable, INH_p , storing the state of the port. In fact, each port can be active ($INH_p = false$), meaning that it can take part to the corresponding reaction, or inhibited ($INH_p = true$) by biological mechanisms specified in the map;

Molecular-types correspond to nodes of the MIM or to points in the middle of an arrow (i.e., terminating points of reaction arrows). They consist of a unique identifier, *molecular-type-id*, of a list of port-types, implicitly determining all the possible reactions the molecule can be involved into, and of a list of contingencies starting from it (we present the treatment of contingencies at the end of the section). For instance, in Figure 1(a), A and x nodes define two distinct molecular-types.

Each graph of a complex can contain, in principle, several instances of the same molecular-type, just think of the case in which two copies of the same molecule are bound together (the so-called homodimers). In order to distinguish among these different copies, we introduce an unambiguous naming system inside each complex, enumerating each node of a complex-graph with an integer, local

⁷ This condition can always be made true by the application of suitable rewriting rules to the map.

<pre> molecular_type(molecular_type_id,port_list,contingency_list) node(molecular_type_id, mol_id) edge([mol_id1, port_type_id1],[mol_id2, port_type_id2]) complex_type(complex_type_id, node_list,edge_list, contingency_list) complex_number(complex_type_id, X) port_number(port_type_id, X) </pre>

Table 3. Predicates describing MIM structures and counting their occurrences.

to that complex, called *mol_id*. Moreover, each different complex graph that can be constructed according to the prescription of the MIM, identifies a *complex-type*; complex-types are also given an unique id, *complex_type_id*, assigned at run-time whenever a new complex-type is created.

Complex-graphs and molecular-types can be easily represented in sCCP. In fact, we just need to store all the characterizing information in suitable logical predicates, listed at the beginning of Table 3. Note that each molecule in a complex-type is unambiguously identified by the pair (*complex_type_id*, *mol_id*).⁸ Such pairs are called *coordinates* of the molecule.

Another important class of predicates, crucial for the run-time engine, are those counting the number of objects of a certain type. Specifically, at run-time we need to count how many complexes we have for each different complex-type (predicate **complex_number**), and how many active ports we have, for any port-type (predicate **port_number**). The variable *X* used in these predicates is a stream variable, defined in Section 4.

The reason for updating the number of ports or complexes at run-time, lies in the definition of the stochastic model for the simulation of MIMs. We adopt a classical approach, defining the speed of a reaction according to the *principle of mass action* [12]: the speed of each reaction is proportional to the quantity of each reactant. In our encoding, each reaction involves one or two port-types, hence its speed will be proportional to the number of active instances of such port-types.

The simulation algorithm of such stochastic model is based on the celebrated Gillespie algorithm [11, 12], extended in order to manage all the additional information of graphs and complexes. Essentially, we can see it as a loop composed of 4 basic steps:

1. choose the next reaction to execute;
2. choose the reactants;
3. create the products;

⁸ *molecular_type_id* can be recovered from *mol_id* and the predicate `node(molecular_type_id, mol_id)`.

```

complexation(port_id1,port_id2, rate) :-
  local complex_id1,complex_id2,mol_id1,mol_id2,nodes,
      edges,contingencies,new_complex_id.
  ask[rate·#port_id1·#port_id2](
    are_greater_than_zero(port_id1,port_id2)
    ∧ are_reactions_unlocked).
  tell∞(lock_reactions).
  tell∞(activate_port_manager(port_id1)).
  askF(is_port_manager_done(port_id1)).
  tell∞(activate_port_manager(port_id2)).
  askF(is_port_manager_done(port_id2)).
  tell∞(get_chosen_complex(port_id1,complex_id1,mol_id1)).
  tell∞(get_chosen_complex(port_id2,complex_id2,mol_id2)).
  tell∞(build_complex(complex_id1,mol_id1,complex_id2,
    mol_id2,nodes,edges,contingencies)).
  tell∞(add_complex(nodes,edges,contingencies,
    new_complex_id)).
  tell∞(update_numbers(complex_id1,complex_id2,
    new_complex_id,port_id1,port_id2)).
  tell∞(apply_contingencies(new_complex_id)).
  tell∞(unlock_reactions).
  complexation(port_id1,port_id2, rate)

port_manager(port_id) :-
  local complex_id,mol_id.
  askF(is_port_manager_active(port_id)).
  tell∞(choose_complex(port_id,complex_id,mol_id)).
  tell∞(complex_chosen(port_id,complex_id,mol_id)).
  tell∞(port_manager_done(port_id)).
  port_manager(port_id)

```

Table 4. Two sCCP agents used in the simulation of MIMs.

4. apply contingency rules to products.

We give now some details of its sCCP implementation. The choice of the next reaction can be seen as a stochastic race among all the enabled reactions. In sCCP, this effect is obtained associating an agent to each reaction arrow of the molecular interaction map, called *reaction agent*. For instance, in Table 4 we show the agent dealing with complexation. This agent tries to execute at a rate defined according to the principle of mass action ($\text{ask}_{[rate \cdot \#port_{id1} \cdot \#port_{id2}]}$). If the agent wins the competition, it gets control of the whole system (**lock_reactions**) and then it activates two auxiliary agents in order to identify the reactants actually involved (**activate_port_manager**). In fact, a reaction involves two complexes with available ports of the required port-type. However, as different complex-types can have such ports available, we must identify the ones really involved in the reaction. Essentially, we need to pick, with uniform probabil-

ity, one complex among all those having an active port of the required type. This operation is performed by *port managers* (see again Table 4): there is one agent for each port-type, keeping an updated list of all complex types containing active ports of its type and choosing one of them upon request from a reaction agent (**choose_complex**). Port agents keep the information about active ports in a list stored in a dedicated predicate, whose elements are pairs (*complex_type_id, mol_id*) identifying the molecule in a complex-type containing the port (we omitted these details from the code of Table 4).

When reactants have been chosen by port agents, the reaction agent generates the products of the reaction (**build_complex**). For instance, in case of a complexation reaction, the agent has to merge two complexes into one, adding nodes and edges to the complex description and marking as bound the ports involved in the reaction (i.e. removing them from lists of the corresponding port agents).

If, after these operations, a new (i.e. not present in the system) complex-type is obtained, then all the necessary predicates are added to the constraint store. Otherwise, the value’s variable counting the number of complexes of the generated type is increased, while those of the two reactants are decreased (**add_complex, update_numbers**). Checking if a complex-type is already present is, in fact, a problem of graph isomorphism. For the subclass of complex-graphs, this problem is quadratic in their description, thanks to the fact that ports are unique for each edge, see [26] for further details.

The last point of the simulation algorithm consists in the application of contingencies (**apply_contingencies**). These are the inhibition and requirement arrows, briefly introduced in Section 2. To grasp the rationale behind their implementation, consider again Figure 1(b). The inhibition arrow from y (the head of the contingency rule) to A - B complexation line (the tail) was used to forbid complexation between phosphorylated B and A . This essentially means that, if B is phosphorylated (i.e., the corresponding edge e is in the complex description), then B cannot bind to A , and so the port p_{B-A} connecting B to A , must become inactive.

This example suggests that contingencies are nothing but logical implications of the form:

IF (a set of edges E is in the complex)
THEN (some ports must become active/inactive)

Rules of this kind are stored in each molecular-type descriptor, so that each complex-type has associated a set of contingency rules potentially applicable. When a new complex type is created in a reaction, then the reaction agent checks what rules among those listed in the complex can be applied, and it modifies accordingly the value of INH_p of ports and their associated global counters, like the predicate **port_number** and the lists used by port agents.

6 Experimental Results

In this section we present some preliminary tests of the framework just presented. Before discussing a simple example in detail, we briefly comment on the implementation. At the moment, we have a first prototype implementation of sCCP, based on SICStus prolog [10]. Essentially, we have a meta-interpreter for the language, coded using SICStus’s Constraint Logic Programming libraries to manage the constraint store, cf. [3] for further details.

The interpreter accepts arbitrary sCCP programs, and it can also be extended with external libraries, defining sCCP agents and constraints. Hence, to test the encoding of MIMs in sCCP, we simply defined the templates of all agents and predicates needed; in this way, a MIM can be described simply by the collection of agents associated to reactions and ports and by the description of molecular types.

Unfortunately, the interpreter based on Prolog lacks computational efficiency. In the future, we plan to write a more efficient implementation of the whole language.

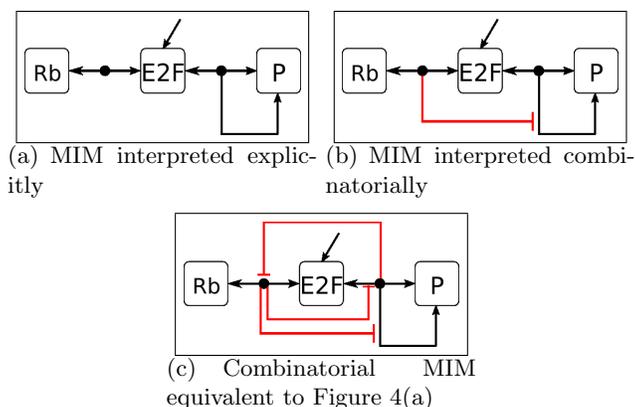


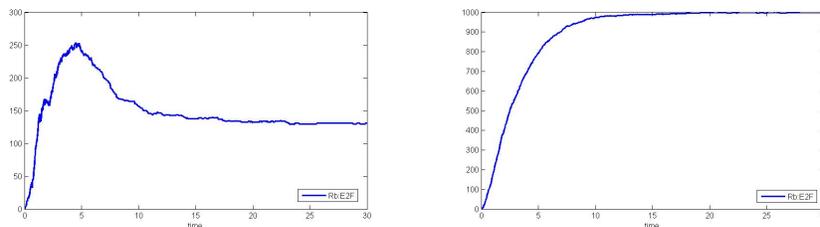
Fig. 4. Molecular Interaction Maps for a simple subnetwork of the G1/S phase transition network.

The example we consider is a very simple MIM taken from [16], where Kohn first introduced the MIM notation in order to study the mammalian G1/S cell-cycle phase transition. In that paper, Kohn studied this system using an evolutionary perspective: he started from a small subnetwork and extended it to include further details at each step. Here we consider the simplest system, composed by three proteins. One is the transcription factor *E2F*, involved in the regulation of several genes active in the replication of DNA (the S phase of the cell cycle). The second is a protein *Rb*, belonging to the Retinoblastoma family, inhibiting *E2F* activity when complexed to it. The third is a Protease *P* involved

in the degradation of $E2F$. The corresponding map, as taken from [16], is shown in Figure 4(a). As we can see, there is also an external feeding of $E2K$. The problem with this map is that in [16] Kohn interpreted it explicitly. This means that the bindings of $E2F$ are exclusive. In the combinatorial interpretation, instead, these two bindings can coexist in the same complex. However, in both maps the degradation of $E2F$ can happen when $E2F$ is bound only to protease P . This is a choice that we make explicit introducing a suitable inhibition arrow, see Figure 4(b). If we want to obtain the same behavior of the explicit interpretation, we need to make the bindings exclusive: this can be obtained simply adding two inhibition symbols in the map, ending up with Figure 4(c).

The differences between the two interpretations are not confined to the topology of the maps, but they obviously propagate to the dynamical behavior. In Figure 5, we show the simulation of the map of Figure 4(b) (Figure 5(a)) and of that of Figure 4(c) (Figure 5(b)). As we can see, the dynamics and the stable values of the complex $Rb:E2F$ are different. In fact, in the less restrictive system of Figure 4(b), part of the $Rb : E2F$ complex is also bound to protease P , hence the stationary value of pure $Rb : E2F$ is lower.

This example, despite its simplicity, shows two things: the feasibility of our encoding, which is able to simulate the maps implicitly, and the fact that defining possible interaction in combinatorial MIMs is a matter of forbidding unwanted or unrealistic behaviors.



(a) sCCP Simulation of MIM of Figure 4(b) (b) sCCP Simulation of MIM of Figure 4(c)

Fig. 5. sCCP simulation of MIMs of Figures 4(b) and 4(c). Parameters are as in [16].

7 Related Work

We briefly review some related work, focused on the simulation of MIMs. First of all, Kohn himself in [16] uses the explicit version of MIMs to generate the associated mass action ODEs. Of course, this requires all possible reagents to be defined, giving rise to the combinatorial explosion discussed in Section 2.

Other approaches, closer to ours, are based on the κ -calculus [6] and on β -binders [5]. κ -calculus is a process algebra with an operational semantic based on rewriting rules, having complexation and activation (i.e. phosphorylation) as basic primitives. However, κ -calculus does not have a stochastic semantics, and the operation of de-complexation is not supported. It follows that MIMs can only be partially described and they cannot be simulated.

β -binders [24], instead, are a modification of stochastic π -calculus [22] in which π -processes are contained into boxes, having interaction capabilities exposed in their surface (the so called binders). Processes in different boxes can communicate through typed channels having an high affinity provided by an external function. Boxes can also be split and merged at run-time. MIMs can be described as β -binders [5] using the same basic assumptions we made in Section 5: molecules are represented as boxes and characterized by the collection of their interaction sites (i.e. of their binders). However, this encoding differs from ours as it follows the principles put forward in [25]: each basic molecule present in the system will be described by a dedicated box. The description of complexes, instead, is stored in the external environment. Clearly, different basic molecules have different descriptions in terms of boxes (they differ in the inner π -processes and in their binders), hence the addition of new interaction capabilities requires the modification of all boxes involved. Our encoding, instead, associates an agent to each reaction arrow of the map (plus an agent to each port type); moreover, adding arrows requires just to put in parallel an agent of the corresponding type (plus the port managers for the new ports). The addition of new basic molecular species requires the addition in the store of the predicates describing them. Another substantial difference with β -binders is that we are not developing a new language, but we are simply programming an existing one, without even exploiting all its features. For instance, the functional rates can be used, as in [4, 3], to encode chemical kinetics different from mass action, like Michaelis-Menten one, resulting in a simplification of models (we need one reaction instead of three).

8 Conclusions

In this paper we presented an implementation in stochastic concurrent constraint programming of an algorithm to simulate biologically regulatory networks defined by means of Molecular Interaction Maps. This notation is implicit, as each edge in such a diagram represents a set of reactions potentially very large. Our sCCP-simulation, instead of generating the full list of reactions, is able to simulate the map implicitly, generating only those complexes that are actually present in the system at run-time. This is achieved using a graph-based representation of complexes, so that new complexes are dynamically constructed merging and splitting other complex-graphs. A prototype implementation has been written in SICStus prolog [10] and used to perform some preliminary tests.

The choice of sCCP as a language to describe (implicitly) such maps is motivated by different reasons. First of all, the details of the stochastic evolution

are dealt automatically by its (stochastic) semantics. In addition, the power of constraints allows to represent and reason directly on the graph-based representation of complexes, separating this description from the definition of agents performing the simulation. Another important motivation is that the model built in such way is compositional w.r.t. the addition of new edges (and nodes) in a MIM, see the discussion of the previous section. Finally, the sCCP description of a MIM is proportional in size to the number of symbols needed to write the map itself, thus taming the combinatorial explosion of possible reactions.

Currently, we are planning to realize a more efficient implementation, interfacing it with graphical tools to design MIMs, in order to analyze big bio-regulatory networks.

References

1. M.L. Blinov, J. Yang, J.R. Faeder, and W.S. Hlavacek. Graph theory for rule-based modeling of biochemical networks. *Transactions of Computational Systems Biology*, 2007.
2. L. Bortolussi. Stochastic concurrent constraint programming. In *Proceedings of 4th International Workshop on Quantitative Aspects of Programming Languages, QAPL 2006, ENTCS*, volume 164, pages 65–80, 2006.
3. L. Bortolussi. *Constraint-based approaches to stochastic dynamics of biological systems*. PhD thesis, PhD in Computer Science, University of Udine, 2007. Available at <http://www.dmi.units.it/~bortolu/files/reps/Bortolussi-PhDThesis.pdf>.
4. L. Bortolussi and A. Policriti. Modeling biological systems in concurrent constraint programming. In *Proceedings of Second International Workshop on Constraint-based Methods in Bioinformatics, WCB 2006*, 2006.
5. F. Ciocchetta, C. Priami, and P. Quaglia. Modeling kohn interaction maps with beta-binders: an example. *Transactions on computational systems biology*, LNBI 3737:33–48, 2005.
6. V. Danos and C. Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.
7. H. De Jong. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9(1):67–103, 2002.
8. L. Edelstein-Keshet. *Mathematical Models in Biology*. SIAM, 2005.
9. J.R. Faeder, M.L. Blinov, B. Goldstein, and W.S. Hlavacek. Rule-based modeling of biochemical networks. *Complexity*, 10:22–41, 2005.
10. Swedish Institute for Computer Science. Sicstus prolog home page.
11. D.T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. of Computational Physics*, 22, 1976.
12. D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. of Physical Chemistry*, 81(25), 1977.
13. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
14. H. Kitano. *Foundations of Systems Biology*. MIT Press, 2001.
15. H. Kitano. Computational systems biology. *Nature*, 420:206–210, 2002.
16. K. W. Kohn. Functional capabilities of molecular network components controlling the mammalian g1/s cell cycle phase transition. *Oncogene*, 16:1065–1075, 1998.
17. K. W. Kohn. Molecular interaction map of the mammalian cell cycle control and dna repair systems. *Molecular Biology of the Cell*, 10:2703–2734, August 1999.

18. K. W. Kohn, M. I. Aladjem, S. Kim, J. N. Weinstein, and Y. Pommier. Depicting combinatorial complexity with the molecular interaction map notation. *Molecular Systems Biology*, 2(51), 2006.
19. K. W. Kohn, M. I. Aladjem, J. N. Weinstein, and Y. Pommier. Molecular interaction maps of bioregulatory networks: A general rubric for systems biology. *Molecular Biology of the Cell*, 17(1):1–13, 2006.
20. H. H. Mcadams and A. Arkin. Stochastic mechanisms in gene expression. *PNAS USA*, 94:814–819, 1997.
21. J. R. Norris. *Markov Chains*. Cambridge University Press, 1997.
22. C. Priami. Stochastic π -calculus. *The Computer Journal*, 38(6):578–589, 1995.
23. C. Priami and P. Quaglia. Modelling the dynamics of biosystems. *Briefings in Bioinformatics*, 5(3):259–269, 2004.
24. C. Priami and P. Quaglia. Beta binders for biological interactions. In *Proceedings of Computational methods in system biology (CMSB04)*, 2005.
25. A. Regev and E. Shapiro. Cellular abstractions: Cells as computation. *Nature*, 419, 2002.
26. A. Romanel, L. Demattè, and C. Priami. The beta workbench. Technical Report TR-03-2007, Center for Computational and Systems Biology, Trento, 2007.
27. V. A. Saraswat. *Concurrent Constraint Programming*. MIT press, 1993.
28. L. Shapiro and E. Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, 1994.
29. S. H. Strogatz. *Non-Linear Dynamics and Chaos, with Applications to Physics, Biology, Chemistry and Engeneering*. Perseus books, 1994.
30. D. J. Wilkinson. *Stochastic Modelling for Systems Biology*. Chapman & Hall, 2006.

Interoperability Mechanisms for Ontology Management Systems*

Lorenzo Gallucci, Giovanni Grasso, Nicola Leone, Francesco Ricca

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{gallucci|grasso|leone|ricca}@mat.unical.it

Abstract. The Ontology Web Language (OWL) is a W3C recommendation which has been conceived to enrich Web pages with machine-understandable descriptions of the semantics of the presented contents (i.e. Web Ontologies). OWL is based on the Open World Assumption (OWA), and indeed, it is suitable for describing and sharing Web information. However, OWL is unpractical in some cases (for example in applications dealing with Enterprise Ontologies) where different assumptions, like the Closed World Assumption (CWA), are better suited. Conversely, the OntoDLP language, an extension of Disjunctive Logic Programming with the most important constructs of ontology specification languages, is based on the CWA and can be fruitfully exploited in such cases. Nevertheless, it may happen that enterprise systems have to share or obtain information from the Web. This means that suitable interoperability tools are needed.

In this paper we present a pragmatic approach to the interoperability between OWL and OntoDLP. Basically, we provide a couple of syntactic transformations that allow one to import an OWL ontology in a correspondent one specified in OntoDLP and vice versa. The proposed technique is based on an intuitive translation and preserves the semantics of the original specification when applied to interesting fragments of the two languages.

1 Introduction

The OWL (Ontology Web Language) [1] is a recommendation of the World Wide Web Consortium (W3C) which is playing an important role in the field of Semantic Web. Indeed, OWL has been conceived to enrich Web pages, now presenting information for humans, with machine-readable content. Thus, it aims at providing a common way for specifying the semantic content of Web information (i.e. *Web Ontologies*). OWL is built on-top of other preexisting Web languages, such as XML [2] and RDF(S) [3], and, its semantics is based on expressive Description Logics (DL)[4]. Essentially, the idea behind all the DLs (and, consequently, behind OWL) is the one of representing relationships among concepts

* 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.”

of a domain. Thus, the basic constructs of DLs allows one to specify restrictions on relationships, like e.g. $\forall R.C$ denotes the class of all individuals that are in the relationship R with only individuals belonging to the concept C . The semantics of DLs are usually given by exploiting a set-theoretic interpretation of concepts, i.e. a concept is interpreted as a set of individuals and roles (which are binary relationships among individuals) are interpreted as sets of pairs of individuals. The non-finiteness of the domain of interpretation and the Open World Assumption (OWA) are distinguishing features of Description Logics with respect to other modeling languages which have been developed in the study of non-monotonic reasoning, logic programming, and databases. These characteristics makes DLs, and thus OWL, particularly well suited for defining and sharing data in the Web, i.e. for defining *Web Ontologies*. However, there are many other application domains in which the data belongs to more conventional and structured sources (such as relational databases), and in which the specification of the knowledge must be considered complete. For example, this happens very often when enterprises share data for business purpose; a typical scenario is the one of banks or insurance companies sharing information about behavior and/or reliability of customers. In this scenario, *Enterprise Ontologies* (i.e ontologies containing specifications of terms and definitions relevant to business enterprises), are often used for sharing information stored in relational databases. It is straightforward to see that, in this settings, (i) the nature of the domain makes the CWA an essential tool; and, (ii) rule-based formalisms (permitting sophisticated forms of reasoning and querying) can be fruitfully applied. Thus, OWL may be unpractical to use and, conversely, OntoDLP [5] a novel ontology representation language based on Answer Set Programming (ASP) [6, 7], which presents both the above-mentioned features, can be fruitfully applied. Indeed, OntoDLP adds to an expressive logic programming language the most common ontology specification constructs (e.g. classes, relations, inheritance, axioms etc.); and more generally, OntoDLV can be suitably used when one has to deal with domains requiring the CWA, and/or the the reasoning capabilities of an expressive rule-based language.

It is worth noting that, even if the world of enterprise ontologies and the Web have very different requirements (e.g. open vs closed world assumption) it may happen that the enterprise applications need to exploit information made available in a Web site and vice versa. In this scenario, it is important to provide *interoperability* tools which make the systems able to simultaneously deal with both OWL and OntoDLP ontologies, or, at least, they must be able to *share* and/or *exchange* the information expressed in one of the two formalism with the other and vice versa.

In this paper, we propose a *pragmatic* approach to the problem of interoperability between a OntoDLP and OWL. In particular, we designed a general strategy (i) for “importing” an OWL ontology in OntoDLP; and (ii) for “exporting” an OntoDLP specification in an OWL one. We obtained an *import* and an *export* transformations which tries to “translate” each construct of the original language in an *intuitively* equivalent one on the destination language (for some

constructs the obtained behavior is only intuitively approximated). Moreover, we studied the theoretical properties of the above-mentioned transformations and we identified some fragments of the languages for which the *semantic equivalence* between the original OWL (resp. OntoDLP) ontology and the obtained OntoDLP (resp. OWL) ontology is guaranteed (i.e. the consequences entailed by the original specification under the source language semantics are the same of the ones entailed by the obtained specification under the destination language semantics).

The rest of the paper is organized as follows: Section 2 briefly describes the OntoDLP language; Section 3 presents the import and export transformations; Finally, in Section 5 we draw our conclusions.

2 The OntoDLP Language

In this section we informally describe the OntoDLP language, a knowledge representation and reasoning language which allows one to define and to reason on ontologies. For a better understanding, we will describe each construct in a separate paragraph and we will exploit an example (the *living being ontology*), which will be built throughout the whole section, thus illustrating the features of the language.

OntoDLP is actually an extension of (disjunctive) Answer Set Programming under the stable model semantics, and hereafter we assume the reader to be familiar with ASP syntax and semantics (for further details refer to [7]).

Classes. A (base) *class*¹ can be thought of as a collection of individuals that belong together because they share some properties.

Classes can be defined in OntoDLP by using the keyword **class** followed by its name, and class attributes can be specified by means of pairs (*attribute-name* : *attribute-type*), where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

Suppose we want to model the *living being* domain, and we have identified four classes of individuals: *persons*, *animals*, *food*, and *places*.

For instance, we can define the class *person* having the attributes name, age, father, mother, and birthplace, as follows:

```
class person(name:string, age:integer, father:person, mother:person,
             birthplace:place).
```

Note that, this definition is “recursive” (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e. objects made of other objects. Moreover, many properties can be represented by using alphanumeric strings

¹ For simplicity, we often refer to *base classes* by omitting the *base* adjective, which has the sole purpose of distinguishing this construct of the language from another one called *collection class* that will be described later in this section.

and numbers by exploiting the built-in classes *string* and *integer* (respectively representing the class of all alphanumeric strings and the class of non-negative numbers).

In the same way, we could specify the other above mentioned classes in our domain as follows:

```
class place(name:string).
class food(name:string, origin:place).
class animal(name:string, age:integer, speed:integer).
```

Objects. Domains contain individuals which are called *objects* or *instances*.

Each individual in OntoDLP belongs to a class and is univocally identified by using a constant called *object identifier* (oid) or *surrogate*.

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, with the following two facts

```
rome : place(name:"Rome").
john:person(name:"John", age:34, father:jack, mother:ann, birthplace:rome).
```

we declare that “Rome” and “John” are instances of the class *place* and *person*, respectively. Note that, when we declare an instance, we immediately give an oid to the instance (e.g. *rome* identifies a place named “Rome”), which may be used to fill an attribute of another object. In the example above, the attribute *birthplace* is filled with the oid *rome* modeling the fact that “John” was born in Rome; in the same way, “*jack*” and “*ann*” are suitable oids respectively filling the attributes *father*, *mother* (both of type *person*).

The language semantics (and our implementation) guarantees the referential integrity, both *jack*, *ann* and *rome* have to exist when *john* is declared.

Relations. *Relations* are declared like classes: the keyword **relation** (instead of **class**) precedes a list of attributes, and model relationships among objects. As an example, the relation *friend*, which models the friendship between two persons, can be declared as follows:

```
relation friend(pers1:person, pers2:person).
```

In particular, to assert that two persons, say “john” and “bill” are friends (of each other), we write the following logic facts (that we call tuples):

```
friend(pers1:john, pers2:bill).    friend(pers1:bill, pers2:john).
```

Thus, tuples of a relation are specified similarly to class instances, that is, by asserting a set of facts (but tuples are not equipped with an oid).

Inheritance. OntoDLP allows one to model taxonomies of objects by using the well-known mechanism of inheritance.

Inheritance is supported by OntoDLP by using the special binary relation *isa*. For instance, one can exploit inheritance to represent some special categories of persons, like *students* and *employees*, having some extra attribute, like a school, a company etc. This can be done in OntoDLP as follows:

```

class student isa {person}{
  code:string,
  school:string,
  tutor:person).

class employee isa {person}{
  salary:integer,
  skill:string,
  company:string,
  tutor:employee).

```

In this case, we have that *person* is a more generic concept or *superclass* and both *student* and *employee* are a specialization (or *subclass*) of *person*. Moreover, an instance of *student* will have both the attributes: *code*, *school*, and *tutor*, which are defined locally, and the attributes: *name*, *age*, *father*, *mother*, and *birthplace*, which are defined in *person*. We say that the latter are “inherited” from the superclass *person*. An analogous consideration can be made for the attributes of *employee* which will be *name*, *age*, *father*, *mother*, *birthplace*, *salary*, *skill*, *company*, and *tutor*.

An important (and useful) consequence of this declaration is that each proper instance of both *employee* and *student* will also be automatically considered an instance of *person* (the opposite does not hold!).

For example, consider the following instance of *student*:

```

al:student(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome,
  code:"100", school:"Cambridge", tutor:hanna).

```

It is automatically considered also instance of *person* as follows:

```

al:person(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome).

```

Note that it is not necessary to assert the above instance.

In OntoDLP there is no limitation on the number of superclasses (i.e. multiple inheritance is allowed). Moreover, there are two more built-in classes in OntoDLP. The first one, called *individual*, is the superclass of all the user-defined classes; and the second one, called *object* (or \top), is the only (direct) superclass of *individual*, *string*, and *integer* (thus, *object* is the most general OntoDLP type).²

We complete the description of inheritance recalling that OntoDLP allows one to organize also relations in taxonomies. In this case, relation attributes and tuples are inherited following the same criteria defined above for classes. Clearly, the taxonomies of classes and relations are distinct (class and relations are different constructs).

Collection Classes and Intensional Relations. The notions of base class and base relation introduced above correspond, from a database point of view, to the *extensional* part of the OntoDLP language. However, there are many cases in which some property or some class of individuals can be “derived” (or inferred) from the information already stated in an ontology. In the database world, the *views* allows to specify this kind of knowledge, which is usually called “intensional”. In OntoDLP there are two different “intensional” constructs: *collection classes* and *intensional relations*.

As an example, suppose we want to define the class of peoples which are less than 21 years old and have less than two friends (we name this class *youngAndShy*). Note that, this information is implicitly present in the ontology, and the “intensional” class *youngAndShy* can be defined as follows:

² For a formal description of inheritance we refer the reader to [5].

collection class *youngAndShy*(*friendsNumber*: *integer*) {
 $X : \text{youngAndShy}(\text{friendsNumber} : N) :- X : \text{person}(\text{age} : \text{Age}),$
 $\text{Age} < 21, \#count\{F : \text{friend}(\text{pers1} : X, \text{pers2} : F)\} < 2. \}$

Note that in this case the instances of the class *youngAndShy* are “borrowed” from the (base) class *person*, and are inferred by using a logic rule. Basically, this class *collects* instances defined in another class (i.e. *person*) and performs a re-classification based on some information which is already present in the ontology. Thus, in general, the collection classes neither have proper instances nor proper oid’s while they “collect” already defined objects.

In an analogous way we specify “*intensional relations*” by using the key words **intensional relation** followed by a logic program defining its tuples. It is worth noting that the programs which define collection classes and intensional relations must be normal and stratified ([7]).

Thus, in general, *collection classes* and *intensional relations* are both more natural and more expressive than relational database views, because they exploit a powerful language that allows recursion and negation as failure.

It is important to say that both collection classes and intensional relations can be organized in taxonomies by using the *isa* relation. Importantly, the inheritance hierarchy of collection classes (resp. intensional relations) and the one of base classes (resp. relations) are distinct (i.e a collection class cannot be superclass or subclass of a base class and vice versa).

Axioms and Consistency. An *axiom* is a consistency-control construct modeling sentences that are always true (at least, if everything we specified is correct). Axioms can be used for several purposes, such as constraining the information contained in the ontology and verifying its correctness. As an example suppose we declared the relation *colleague*, which associates persons working together in a company, as follows:

relation *colleague* (*emp1*:*employee*, *emp2*:*employee*).

It is clear that the information about the company of an employee (recall that there is an attribute *company* in the scheme of the class *employee*) must be consistent with the information contained in the tuples of the relation *colleague*. To enforce this property we assert the following axioms:

- (1) $:- \text{colleague}(\text{emp1} : X1, \text{emp2} : X2), \text{not } \text{colleague}(\text{emp1} : X2, \text{emp2} : X1)$
- (2) $:- \text{colleague}(\text{emp1} : X1, \text{emp2} : X2),$
 $X1 : \text{employee}(\text{company} : C), \text{not } X2 : \text{employee}(\text{company} : C).$

The above axioms states that, (1) the relation *colleague* is symmetric, and (2) if two persons are colleagues and the first one works for a company, then also the second one works for the same company.

If an axiom is violated, then we say that the ontology is inconsistent (that is, it contains information which is, somehow, contradictory or not compliant with the intended perception of the domain)³.

³ Note that, OntoDLP axioms are consistency control constructs (intended for *constraining* the ontology to the intended specification) and, thus they are radically different from OWL axioms.

Reasoning modules. Given an ontology, it can be very useful to reason about the data it describes. *Reasoning modules* are the language components endowing OntoDLP with powerful reasoning capabilities. Basically, a *reasoning module* is an ASP program conceived to reason about the data described in an ontology. Reasoning modules in OntoDLP are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints.

Syntactically, the name of the module is preceded by the keyword *module* while the logic rules are enclosed in curly brackets (this allows one to collect all the rules constituting the encoding of a problem).

As an example, suppose that the living being ontology contains a base relation that models the roads connecting neighbor places. The following module can be used to know whether there exists a route connecting two places.

```
module(connections){
    route(X,Y) :- road(start : X, end : Y).
    route(X,Y) :- road(start : X, end : Z), route(Z, Y). }
```

The above rules basically compute the transitive closure of the relation *road*; moreover, the “output” relation *route* has been used in this reasoning module, without the need of defining its scheme. We did not declare the (auxiliary) predicate *route* in the ontology, because it is related only to this computation (we simply used it). Note that, the information about routes is implicitly present in the ontology and the reasoning module just allows to make it explicit.

It is worth noting that, since reasoning modules have the full power of (disjunctive) Answer Set Programming, they can be also used to perform complex reasoning tasks on the information contained in an ontology. In practice, they allow one to solve problems which are complete for the second level of the polynomial hierarchy.

Querying. An important feature of the language is the possibility of asking queries in order to extract knowledge contained in the ontology, but not directly expressed. As in DLP a query can be expressed by a conjunction of atoms, which, in OntoDLP, can also contain complex terms.

As an example, we can ask for the list of persons having a father who is born in Rome as follows:

```
X:person(father:person(birthplace:place(name: "Rome")))?
```

Note that we are not obliged to specify all attributes; rather we can indicate only the relevant ones for querying. In general, we can use in a query both the predicates defined in the ontology and the auxiliary predicates in the reasoning modules. For instance, consider the reasoning module *connections* defined in the previous section, the query *route(rome,milan)?* asks whether there is a route connecting Rome and Milan.

It is worth noting that in presence of disjunction or unstratified negation in modules, we might obtain multiple answer sets; in this case the system supports both brave and cautious reasoning (see [5]).

3 Interoperability between OWL and OntoDLP

In this section we describe a pragmatic strategy that allows one to import an OWL-DL [1]⁴ ontology in OntoDLP, and to export an OntoDLP ontology in OWL. Hereafter, we assume the reader to be familiar with with both syntax and semantics of OWL[1] and Description Logics[4].

3.1 Importing OWL in OntoDLP

In the following we provide a description of the *import* strategy by exploiting some examples. Each group of OWL constructs is described in a separate paragraph.

OWL Thing (\top) and OWL Nothing (\perp). The OWL universal class *Thing* corresponds to the OntoDLP class *individual* (because both are the set of all individuals). Conversely, in OntoDLP we cannot directly express the empty class \perp , but we approximate it as follows:

```
class Nothing.    ::= X:Nothing().
```

Note the axiom imposes that the extension of *Nothing* is empty.

Atomic classes and class axioms ($C, C \sqsubseteq D$). Atomic classes are straightforwardly imported in OntoDLP. For example, we write: **class** *Person*() to import the specification of the atomic class *Person*.

Inclusion axioms directly correspond to the *isa* operator in OntoDLP. Thus, the statement $Student \sqsubseteq Person$ (asserting that student is a subclass of person) is imported by writing: **class** *Student* **isa** *Person*.

In OWL one can assert that two or more atomic classes are equivalent (i.e. they have the same extension) by using an equivalent class axiom (\equiv). OntoDLP does not have a similar construct, but we can obtain the same behavior by using collection classes and writing suitable rules to enforce the equivalence. For example, $USPresident \equiv PrincipalResidentOfWhiteHouse$ is imported as follows:

```
collection class USPresident {
  X:USPresident() :- X:PrincipalResidentOfWhiteHouse(). }
collection class PrincipalResidentOfWhiteHouse {
  X:PrincipalResidentOfWhiteHouse() :- X:USPresident(). }
```

Another class axiom provided by OWL, called *disjointWith*, asserts that two classes are disjoint. We approximate this behavior by using an axiom in OntoDLP. For example:

```
 $Man \sqcap Woman \sqsubseteq \perp$ 
```

⁴ OWL-DL is the largest *decidable* fragment of OWL which directly corresponds to a powerful Description Logics.

in represented in OntoDLP using the axiom:

$$::-\ X:Man(), X:Woman().$$

which asserts that an individual cannot belong to both class *Man* and class *Woman*.

Enumeration classes $\{a_1, \dots, a_n\}$. A class can be defined in OWL by exhaustively enumerating its instances (no individuals exist outside the enumeration).

For example, if we model the RGB color model as follows:

$$RGB \equiv red, green, blue$$

we will import it in OntoDLP by using a collection class in this way:

$$\text{collection class } RGB \{ \text{green} : RGB(). \text{red} : RGB(). \\ \text{green} : RGB(). \text{blue} : RGB(). \}$$

and we also add to the resulting ontology, the axiom $::-\ \#count\{ X: X:RGB() \} > 3$. in order to correctly fix the number of admissible instances of the class.

Properties and Restrictions $(\forall, \exists, \leq, \geq nR)$. One of the main features of OWL (and, originally of Description Logics) is the possibility to express restriction on relationships. Mainly, relationships are represented in OWL by means of properties (which are binary relations among individuals) and, three kinds of restrictions are supported: $\exists R.C$ (called some values from), $\forall R.C$ (called all values from) and restrictions on cardinality $\leq nR$. While properties are naturally “imported” in OntoDLP by exploiting relations, the restrictions on properties are simulated by exploiting logic rules.

We start considering $\exists R.C$, and for example, we define the class *Parent* as follows: $Parent \supseteq \exists hasChild.Person$, which means that parent contains the class of all individuals which are child of some instance of person. Importing this fragment of OWL in OntoDLP we obtain:

$$\text{collection class } Parent \{ \\ X:Parent() :- hasChild(X,Y), Y:Person(). \}$$

The rule allows one to infer all individuals having at least one child.

Also for the $\forall R.C$ property restriction we use a simple example, in which we define the concept *HappyFather* as follows:

$$HappyFather \sqsubseteq \forall hasChild.RichPerson$$

In practice, an individual is a happy father if all its children are rich. The above statement can be imported in OntoDLP in the following way:

$$\text{collection class } RichPerson \{ \\ Y:RichPerson() :- hasChild(X,Y), X:HappyFather(). \}$$

Similarly, we import the property restriction $\exists R.\{o\}$. For example we can describe the class of persons which are born in Africa as follows:

X

$$African \equiv \exists bornIn.africa$$

where *africa* is a specific individual representing the mentioned continent. To import it in OntoDLP, we write:

```
collection class Afrincan {  
  X:Afrincan() :- bornIn(X,africa). }  
intensional relation bornIn (domain:object, range:object).{  
  bornIn(X,africa) :- X : Afrincan(). }
```

Note that, in this case the import strategy is more precise than the one used of $\exists R.C$; in fact, we could also “fill” the *bornIn* (intensional) relation with exactly all the individuals belonging to class *African*.

We now consider the cardinality constraints that allow one to specify for a certain property either an exact number of fillers ($= nR.C$), or at least n / at most n different fillers (respectively $\geq nR.C$ and $\leq nR.C$). In order to describe the way how $\leq nR.C$ is imported, we define the class *ShyPerson* as a *Person* having at most five friends:

$$ShyPerson \equiv \leq 5hasFriend$$

To import it in OntoDLP we write:

```
collection class ShyPerson {  
  X:ShyPerson() :- hasFriend(X,-),  
  #count{Y:hasFriend(X,Y)}<= 5. }
```

Note that, the aggregate function `#count`(see [5]) allows one to infer all the individuals having less than (or exactly) five friends.

The remaining cardinality constraints can be imported by only modifying the operator working on the result of the aggregate function (with \geq and $=$ for $\geq nR.C$ and $= nR.C$, respectively).

OWL also allows to specify domain and range of a property. As an example, consider the property *hasChild* which has domain *Parent* and range *Person*.

$$\top \sqsubseteq \forall hasChild^-.Parent \quad \top \sqsubseteq \forall hasChild.Person$$

when we import this in OntoDLP we obtain:

```
relation hasChild (domain:Parent, range:Person ).
```

It is worth noting that consistently with *rdfs:domain* and *rdfs:range* semantic, we can state that an individual that occurs as subject (resp. object) of the relation *hasChild*, also belongs to the *Parent* (resp. *Person*) class. To simulate this behavior, the definition of the collection classes *Parent* and *Person* is modified by introducing the following rules (the first for *Parent*, the second for *Person*):

```
X:Parent() :- hasChild (X,-).  
Y:Person() :- hasChild (-,Y).
```

Moreover, in OWL properties can be organized in hierarchies, can be defined equivalent (by using the *owl:equivalentProperty* construct), functional, transitive and symmetric. Property inheritance is easily imported by exploiting the corresponding OntoDLP relation inheritance, while the remaining characteristics of a property (like being inverse of another) are expressed in OntoDLP by using *intensional relations* with suitable rules. For example if the relation *hasChild* is declared inverse of *hasParent*, when we import it in OntoDLP we have:

intensional relation *hasChild* (domain: *Parent*, range: *Person*) {
hasChild(*X*, *Y*) :- *hasParent*(*Y*, *X*). }
intensional relation *hasParent* (domain: *Person*, range: *Parent*) {
hasParent(*X*, *Y*) :- *hasChild*(*Y*, *X*). }

Similarly, the transitive property *ancestor*, is imported in OntoDLP as:

intensional relation *ancestor* (domain: *Person*, range: *Person*) {
ancestor(*X*, *Z*) :- *ancestor*(*X*, *Y*), *ancestor*(*Y*, *Z*). }

A classic example of symmetric property is the property *marriedWith*. We can import such a property into OntoDLP as:

intensional relation *marriedWith* (domain: *Person*, range: *Person*) {
marriedWith(*X*, *Y*) :- *marriedWith*(*Y*, *X*). }

Moreover, OWL functional and inverse functional properties are encoded by using suitable OntoDLP axioms. For example, consider the functional property *hasFather* and its inverse functional property *childOf*; they are imported in OntoDLP as:

:- *hasFather*(*X*,_), #count{*Y:hasFather*(*X*,*Y*)}> 1.
 :- *childOf*(_,*Y*), #count{*X:childOf*(*X*,*Y*)}> 1.

Intersection, Union and Complement (\sqcap , \sqcup , \neg). In OWL we can define a class having exactly the instances which are common to two other classes. Consider, for example the class *Woman* which is equivalent to the intersection of the classes *Person* and *Female*; in OWL we write:

$Woman \equiv Person \sqcap Female$

This expression is imported in OntoDLP as:

collection class *Woman* **isa** { *Person*, *Female* } () {
X:Woman() :- *X:Female*(), *X:Person*(). }

Note that we use inheritance in OntoDLP in order to state that each instance of class *Woman* is both instance of *Person* and *Female*; and, conversely, the logic rule allows one to assert that each individual that is common to *Person* and *Female* is an instance of class *Woman*.

In a similar way we deal with the class union construct. For instance, if we want to model the *Parent* class as the union of *Mother* and *Father*, then in OWL we write:

$$Parent \equiv Mother \sqcup Father$$

and the following is the result of the import of this axiom in OntoDLP:

```
collection class Parent {
  X:Parent() :- X:Mother().
  X:Parent() :- X:Father(). }
```

Another interesting construct of OWL is called complement-of, and is analogous to logical negation. An example is the class *InedibleFood* defined as complement of the class *EdibleFood*, as follows:

$$InedibleFood \equiv \neg EdibleFood$$

and, we import it in OntoDLP by using *negation as failure* as follows:

```
collection class InedibleFood {
  X:InedibleFood() :- X:individual(), not X:EdibleFood(). }
```

Individuals and datatypes. The import of the ABox of a OWL ontology is straightforward; and actually, the A-Box assertions are directly imported in OntoDLP facts. For example, consider the following

```
Person(mike)    hasFather(mark,mike)
```

which are, thus imported in OntoDLP as:

```
mike:Person().    hasFather(mark,mike).
```

OWL makes use of the RDF(S) datatypes which exploit the XMLSchema data-type specifications[8].

OntoDLP does not natively support datatypes other than integer and string. To import the others OWL datatypes we encode each datatype property filler in an OntoDLP string that univocally represents its value.

3.2 Exporting OntoDLP in OWL

In this section, we informally describe how an OntoDLP ontology is exported in OWL by using some example.

Classes. Exporting (base) classes (with no attribute), and inheritance it is quite easy since they can be directly encoded in OWL. For instance:

```
class Student isa Person.    becomes simply:    Student  $\sqsubseteq$  Person
```

However, OntoDLP class attributes do not have a direct counterpart in OWL, and we represent them introducing suitable properties and restrictions. Suppose that the class *Student* has an attribute *advisor* of type *Professor*. To export it in OWL, we first create a the functional property *advisor*, with *Student* as domain and *Professor* as range; and, then we export the class Student as $Student \sqsubseteq \forall advisor.Professor$.⁵

⁵ If a class *C* has more than one attribute, we create a suitable property restrictions for each attribute of *C* and we impose that *C* is the the intersection of all the defined property restrictions.

Relations. We can easily export binary (base) relations and inheritance hierarchies in OWL, since the destination language natively supports them. In particular, *isa* statements are translated in inclusion axioms, and domain and range description allowed us to simulate the attributes. For relations having arity greater than two, we adopt the accepted techniques described in the W3C Working Group Note on n-ary Relations [9].⁶

Instances. As we have seen for the import phase, also instances exporting is straightforward. For instance, if we have:

john:Person(father : mike). friends(mark, john).

then we can export it in OWL as:

Person(john) person_father(john, mike) friends(mark, john)

Note the *person_father* property, created as explained above for class attributes.

Collection classes and intensional relations. These constructs, representing the "intensional" part of the OntoDLP language, do not have corresponding language feature in OWL. Moreover, collection classes and intensional relations are exploited in the import strategy to "simulate" the semantics of several OWL constructs. Since we want to preserve their meaning as much as possible in our translation, we implemented a sort of "rule pattern matching" technique that recognizes whether a set of rules in a collection class or in an intensional relation corresponds to (the "import" of) an OWL construct. For example, when we detect the following rule (within a intensional relation):

ancestor(X, Z) :- ancestor(X, Y), ancestor(Y, Z).

we can assert that the relation *ancestor* is a transitive property. This can be done for all the supported OWL feature, because the correspondence induced by the import strategy between OWL constructs and corresponding collection classes is direct and not ambiguous.

In case of rules that do not "correspond" to OWL features, we export them as strings (using an auxiliary property). In this way, we are able to totally rebuild a collection class (intensional relation) when (re)importing a previously exported OWL ontology.

Axioms and Reasoning Modules. OWL does not support rules, thus we decided to export axioms and reasoning modules only for storage and completeness reasons. To this end, we defined two OWL classes, namely: *OntoDLP Axiom* and *OntoDLP ReasoningModule*. Then, for each reasoning module (resp. axiom) we create an instance of the *OntoDLP ReasoningModule* (resp. *OntoDLP Axiom*) class representing it; and we link the textual encoding of the rules (resp. axioms) to the corresponding instances of the *OntoDLP ReasoningModule* (resp. *OntoDLP Axiom*) class.

⁶ Basically, to represent an n-ary relation we create a new auxiliary class having n new functional properties.

4 Theoretical Properties

In this section we show some important properties of our import/output strategies. In particular, we single out fragments of OWL DL and OntoDLP where *equivalence* between the input and the output of our interoperability strategies is guaranteed.

Syntactic Equivalence. Let $import(O_{owl})$ and $export(O_{dlp})$ denote, respectively, the result of the application of our import and export strategies to OWL ontology O_{owl} and OntoDLP ontology O_{dlp} .

Theorem 1. *Given a OWL DL ontology O_{owl} , and an OntoDLP ontology O_{dlp} without class attributes and n -ary relations, we have that:*

- (i) $export(import(O_{owl})) = O_{owl}$, and
- (ii) $import(export(O_{dlp})) = O_{dlp}$.

This means that if we import (resp. export) an ontology, we are able to syntactically reconstruct it by successively applying the export (resp. import) strategy. Intuitively, the property holds because we defined a bidirectional mapping between the primitives of the two languages (actually, there is no ambiguity since we use a syntactically different kind of rule for each construct).

Semantic Equivalence. We now single out a restricted fragment of OWL DL in which the import strategy preserves the semantics of the original ontology (i.e. the two specifications have equivalent semantics).

Theorem 2. *Let Γ , Γ^R and Γ^L be the following sets of class descriptors: $\Gamma = \{A, B \sqcap C, \exists R.o\}$, $\Gamma^R = \Gamma \cup \{\forall R.C\}$, $\Gamma^L = \Gamma \cup \{\exists R.C, \sqcup\}$, and, let O_{owl} be an ontology containing only:*

- class axioms $A \equiv B$ where $A, B \in \Gamma$;
- class axioms $C \sqsubseteq D$ where $C \in \Gamma^L$ and $D \in \Gamma^R$;
- property axioms: domain, range, inverse-of, symmetry and transitivity;
- ABox assertions

then $import(O_{owl})$ under the OntoDLP semantic entails precisely the same consequences as O_{owl} .

Intuitively, the equivalence property holds because⁷ the the First Order Theories equivalent to the admitted fragment of OWL only contains Horn equality-free formulae whose semantics corresponds to the one of the produced logic program.

⁷ According to the approach of Borgida [10], also used in [11].

5 Conclusion and Related Work

In this paper, we proposed a *pragmatic* approach to the problem of interoperability between OntoDLP and OWL. In particular, we designed and implemented in the OntoDLV system two transformations which are able to *import* an OWL ontology in OntoDLP, and *export* an OntoDLP specification in an OWL one.

Moreover, we studied the theoretical properties of the above-mentioned transformations and we obtained some interesting results. The first one regards a syntactic property of the two transformations. Basically, it is guaranteed that applying the *import* (resp. *export*) transformation to an OWL (resp. OntoDLP) ontology O , we can rebuild O by applying the “inverse” *export* (resp. *import*) transformation. Furthermore, we identified some fragments of OWL for which the *semantic equivalence* between the original OWL ontology and the obtained OntoDLP ontology is guaranteed.

Our approach is, somehow, connected with the effort of combining OWL with rules for the Semantic Web (see [12] for an excellent survey). Indeed, one might think to import an OWL ontology in OntoDLP in order to exploit the latter to perform reasoning on top of the original specification. This simple idea has the drawback (from the Semantic Web viewpoint) of relaxing important assumptions like both open-world and unique name assumption. Conversely, one of the major problems existing in the interaction of rules and description logics with strict semantic integration is retaining decidability (which is, instead, ensured in our framework) without losing easy of use and expressivity. For instance, the SWRL[13] approach is undecidable; while, in the so-called DL-safe rules [14] a very strict safety condition is imposed to retain decidability. Notably, this safety condition has been recently weakened in some works [15, 16] thus obtaining a more flexible environment. However, the goal of the above-mentioned approaches is different from the one achieved in this paper; where we introduced some interoperability mechanisms to combine OWL and OntoDLP ontologies. Conversely, the techniques exploited to obtain our import transformation are similar to (even if more pragmatic and less general than) the ones used for reducing description logics to logic programming (see [17, 18, 11, 19, 20]).

References

1. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003) <http://www.w3.org/TR/owl-guide/>.
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M.: Extensible Markup Language (XML) 1.0. W3C Recommendation (2000) <http://www.w3.org/TR/REC-xml/>.
3. Brickley, D., Guha, R.V.: Rdf vocabulary description language 1.0: Rdf schema. w3c recommendation (2004) <http://www.w3.org/TR/rdf-schema/>.
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. CUP (2003)
5. Ricca, F., Leone, N.: Disjunctive Logic Programming with Types and Objects: The DLV+ System. *Journal of Applied Logics* **5** (2007)

6. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *NGC* **9** (1991) 365–385
7. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* **7** (2006) 499–562
8. T.Paul, V. Biron, K.P., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition. (2004) <http://www.w3.org/TR/xmlschema-2/>.
9. Noy, N., Rector, A.: Defining N-ary Relations on the Semantic Web. W3C Working Group Note (2006) <http://www.w3.org/TR/swbp-n-aryRelations/>.
10. Borgida, A.: On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence* **82** (1996) 353–367
11. Grosz, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logics. In: Proc. of the International World Wide Web Conference, WWW2003, Budapest, Hungary. (2003) 48–57
12. Antoniou, G., Damsio, C.V., Grosz, B., Horrocks, I., Kifer, M., Maluszynski, J., Patel-Schneider, P.F.: Combining Rules and Ontologies. A survey. (2005)
13. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosz, B., Dean, M.: Swrl: A semantic web rule language combining owl and ruleml (2004) W3C Member Submission. <http://www.w3.org/Submission/SWRL/>.
14. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. *J. Web Sem.* **3** (2005) 41–60
15. Rosati, R.: Integrating ontologies and rules: Semantic and computational issues. In: Reasoning Web. (2006) 128–151
16. Rosati, R.: Dl+log: Tight integration of description logics and disjunctive datalog. In: KR. (2006) 68–78
17. Belleghem, K.V., Denecker, M., Schreye, D.D.: A strong correspondence between description logics and open logic programming. In: ICLP. (1997) 346–360
18. Swift, T.: Deduction in ontologies via asp. In: LPNMR. (2004) 275–288
19. Heymans, S., Vermeir, D.: Integrating semantic web reasoning and answer set programming. In: Answer Set Programming. (2003)
20. Hustadt, U., Motik, B., Sattler, U.: Reducing shiq-description logic to disjunctive datalog programs. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada. (2004) 152–162

Some remarks on Rice’s and Rice-Shapiro theorems and related problems^{*}

Domenico Cantone and Salvatore Cristofaro

Dipartimento di Matematica e Informatica
Università di Catania
Viale Andrea Doria N. 6, I-95125 Catania, Italy
{cantone,cristofaro}@dmi.unict.it

Abstract. We present some extensions of the well-known theorems by Rice and Rice-Shapiro, which are particularly useful for proving non-recursiveness and nonrecursive enumerability of wide classes of sets of natural numbers. Our extensions do not require that the sets be index sets, as the above cited theorems do, and, in most cases, they allow for simple and quick proofs of undecidability results. We also discuss some problems concerning sequences of classes of partial recursive functions.

Key words: Rice’s theorem, undecidable problems, theory of recursive functions.

1 Introduction

Rice’s theorem [1] is a very powerful tool for proving the nonrecursiveness of a wide class of sets of natural numbers. We recall the usual form in which Rice’s theorem is stated.

Theorem 1 (Rice’s theorem). *Let \mathcal{C} be a class of 1-ary partial recursive functions. Then, for any acceptable programming system ϕ , the set $\{x : \phi_x \in \mathcal{C}\}$ is recursive if and only if \mathcal{C} is trivial, namely, if and only if \mathcal{C} is empty or it contains all the 1-ary partial recursive functions.*¹

Thus, if A is a set of natural numbers, and the characteristic function of A is expressible in the form “ $\phi_x \in \mathcal{C}$ ”, for some class \mathcal{C} of 1-ary partial recursive functions, then, in order to show that A is nonrecursive, it is enough to “exhibit”

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

¹ We will use the symbols $\phi, \varphi, \psi, \dots$ to denote variables ranging over acceptable programming systems. See the Appendix for a brief explanation of some of the terms and notations used in the paper as well as for some basic results of Recursion Theory (such as the s-m-n Theorem, the Recursion Theorem, etc.) which are particularly relevant to the present paper. The reader is referred to textbooks such as [5], [4] and [6] for a comprehensive introduction to the subject of Recursion Theory. Our notations follow closely those in [5] and [6], with some minor modifications.

two 1-ary partial recursive functions f and g such that $f \in \mathcal{C}$ and $g \notin \mathcal{C}$.² In such a case f and g are *witnesses* of the fact that \mathcal{C} is nontrivial. However, although Rice's Theorem characterizes a wide class of nonrecursive subsets of \mathbb{N} , and is simple to apply, it has a limited range of application. Namely, it can be applied only if the set A under consideration is an *index set*, meaning that for no pair of natural numbers x and y it is the case that $x \in A$, $y \notin A$ and $\phi_x = \phi_y$. (This is, indeed, a necessary and sufficient condition for the characteristic function of A to be expressible in the form " $\phi_x \in \mathcal{C}$," for some class \mathcal{C} of 1-ary partial recursive functions.)

A more general case in which membership to classes of partial recursive functions is involved is that of sets whose characteristic functions are expressible in the form " $\phi_x \in \Gamma_x$," where $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is an enumeration of classes of 1-ary partial recursive functions.³ For instance, the characteristic function of the set $A = \{x : \phi_x(0) = x\}$ can be expressed as " $\phi_x \in \Gamma_x$," where Γ_x is the class of all 1-ary partial recursive functions f such that $f(0) = x$.⁴

We notice that the characteristic function of every subset A of \mathbb{N} is always expressible in the above form. Indeed, if one sets

$$\Gamma_x = \begin{cases} \{\phi_x\}, & \text{if } x \in A \\ \emptyset, & \text{otherwise,} \end{cases}$$

then it is immediate to see that $A = \{x : \phi_x \in \Gamma_x\}$. In general, we give the following definition:

Definition 1. Let $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ be an enumeration of classes of 1-ary partial recursive functions. The diagonal set of $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ relative to ϕ , or, briefly, the ϕ -diagset of $\Gamma_0, \Gamma_1, \Gamma_2, \dots$, is the set

$$\ulcorner \Gamma^\neg \phi = \{x : \phi_x \in \Gamma_x\}.$$

Thus, for any subset A of \mathbb{N} , there is an enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes of 1-ary partial recursive functions such that $A = \ulcorner \Gamma^\neg \phi$.

In the spirit of Rice's and Rice-Shapiro theorems, our aim is to single out a number of "easily verifiable" conditions on an enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes of 1-ary partial recursive functions, which are sufficient for its ϕ -diagset to be nonrecursive (or nonrecursively enumerable).

To begin with, let $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ be an enumeration of classes of 1-ary partial recursive functions. We notice that if the set $\ulcorner \Gamma^\neg \phi$ is nonrecursive, then plainly

² Notice that, if f and g satisfy the additional condition that $f \subseteq g$ or that $\theta \notin \mathcal{C}$, for every finite restriction θ of f , then, by Rice-Shapiro Theorem ([2], [3]), A is also nonrecursively enumerable.

³ Formally speaking, an enumeration of classes of partial recursive functions is a function Γ which maps every natural number x into a class $\Gamma(x)$ of partial recursive functions. The $(x+1)$ -st term in the enumeration Γ , i.e., the class $\Gamma(x)$, is denoted as Γ_x . In the sequel we will denote an enumeration Γ of classes of partial recursive functions as $\Gamma_0, \Gamma_1, \Gamma_2, \dots$.

⁴ Observe that A is not an index set.

there must exist an infinite subset B of \mathbb{N} such that Γ_x is nontrivial, for all $x \in B$. However, the converse is easily seen to be false. Indeed, if one sets $\Gamma_x = \{\phi_x\}$, for all x , then each Γ_x is nontrivial, but $\ulcorner \Gamma^\top \phi \urcorner = \mathbb{N}$. Therefore, for the nonrecursiveness of $\ulcorner \Gamma^\top \phi \urcorner$ we need stronger conditions, other than, for instance, the fact that infinitely many of the classes $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ are nontrivial. A possibility would be that of requiring that the enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is *uniformly nontrivial*, in the sense of the following definition:

Definition 2. Let $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ be an enumeration of classes of 1-ary partial recursive functions. We say that $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is uniformly nontrivial with respect to ϕ (abbreviated as ϕ -u.n.t.) if there are 1-ary recursive functions α and β such that, for almost all x ,⁵

- (1) $\phi_{\alpha(x)} \in \Gamma_x$, and
- (2) $\phi_{\beta(x)} \notin \Gamma_x$.

Thus, if $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -u.n.t. then, for almost all x , not only the class Γ_x is nontrivial, but we can effectively find (the indices of) two witnesses $\phi_{\alpha(x)}$ and $\phi_{\beta(x)}$ of this fact. As a simple corollary of Theorem 3 in Section 2 we get

Property 1. Let $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ be an enumeration of classes of 1-ary partial recursive functions. If $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -u.n.t. then $\ulcorner \Gamma^\top \phi \urcorner$ is nonrecursive.

Remark 1. Notice that, from Roger's Isomorphism Theorem it follows that the property of an enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes 1-ary partial recursive functions of being ϕ -u.n.t. does not depend on the particular acceptable programming system ϕ we have under consideration; i.e., if ψ is any other acceptable programming system, then $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -u.n.t. iff $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ψ -u.n.t. Due to this fact, in the sequel we will speak sometimes of enumerations of classes of 1-ary partial recursive functions which are u.n.t. without specifying the acceptable programming system.

The crucial property of a u.n.t. enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes of 1-ary partial recursive functions is that, given x one can effectively find (an index for) a partial recursive function which is different from ϕ_x , in either cases in which $x \in \ulcorner \Gamma^\top \phi \urcorner$ or $x \notin \ulcorner \Gamma^\top \phi \urcorner$. For, if α and β are as in Definition 2, then, for almost all x , we have that $x \in \ulcorner \Gamma^\top \phi \urcorner$ implies $\phi_x \neq \phi_{\beta(x)}$, and $x \notin \ulcorner \Gamma^\top \phi \urcorner$ implies $\phi_x \neq \phi_{\alpha(x)}$. As we shall see in Example 1 below, there are enumerations of classes of partial recursive functions which are not u.n.t. but which satisfy the property mentioned above. To include this case in our discussion, we give the following definition.

Definition 3. Let $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ be an enumeration of classes of 1-ary partial recursive functions. We say that $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is weakly uniformly nontrivial with respect to ϕ (abbreviated as ϕ -w.u.n.t.) if there are 1-ary recursive functions α and β such that, for almost all x ,

- (1) $x \in \ulcorner \Gamma^\top \phi \urcorner$ implies that $\phi_x \neq \phi_{\beta(x)}$, and

⁵ We recall that a property $P(x)$ holds for almost all x , if there is an n such that $P(x)$ holds for all $x \geq n$.

(2) $x \notin \ulcorner \Gamma^\neg \phi$ implies that $\phi_x \neq \phi_{\alpha(x)}$.

From Theorem 3 in Section 2, the following property follows:

Property 2. Let $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ be an enumeration of classes of 1-ary partial recursive functions. If $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -w.u.n.t. then $\ulcorner \Gamma^\neg \phi$ is nonrecursive.

In Section 2 we will see that, by “strengthening” the conditions (1) and (2) of Definition 3, we may derive more than just the nonrecursiveness of the diagonal set $\ulcorner \Gamma^\neg \phi$ (cf. Theorems 4 and 5).

The essential difference between the properties of an enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes of 1-ary partial recursive functions of being ϕ -u.n.t. and that of being ϕ -w.u.n.t. is that, while in the first case it is required that the condition $\phi_{\alpha(x)} \in \Gamma_x$ holds for almost all x (cf. (1) of Definition 2), in the latter case it is required that the same condition holds for all but finitely many x which are *fixpoints*⁶ of the function α (cf. (1) of Definition 3), and similarly for the condition $\phi_{\beta(x)} \notin \Gamma_x$. Since the set of fixpoints of a recursive function depends on the particular acceptable programming system one has under consideration,⁷ we expect that there are enumerations $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes of 1-ary partial recursive functions which are w.u.n.t. with respect to some given acceptable programming system, but not w.u.n.t. with respect to some other acceptable programming system, and, indeed, this is the case, as is illustrated in the following example.

Example 1. Let us suppose that the acceptable programming system ϕ is such that $\phi_{2x} = f_\emptyset$, for all x , and let Γ_x be the class of all 1-ary partial recursive functions f whose domain is a subset of the domain of ϕ_{x+1} . Notice that

$$\ulcorner \Gamma^\neg \phi = \{x : \text{Domain}(\phi_x) \subseteq \text{Domain}(\phi_{x+1})\}.$$

By implicit use of the s-m-n Theorem, let α and β be 1-ary recursive functions such that $\phi_{\alpha(x)} = f_\emptyset$ and $\phi_{\beta(x)} = \lambda y.1$, for all x . Then, it is easy to verify that, for all x , $x \in \ulcorner \Gamma^\neg \phi$ implies that $\phi_x \neq \phi_{\beta(x)}$, and $x \notin \ulcorner \Gamma^\neg \phi$ implies that $\phi_x \neq \phi_{\alpha(x)}$. Thus $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -w.u.n.t. However, $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ cannot be ϕ -u.n.t. since, otherwise, we would have that $\phi_{f(x)} \neq \phi_{x+1}$ for almost all x , where f is some recursive function. By the Recursion Theorem this is impossible.

We notice also that, although the enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -w.u.n.t. there is an acceptable programming system ψ with respect to which it is not w.u.n.t. In fact, if we set $\psi = \lambda xy.\phi(x+1, y)$, then ψ is an acceptable programming system (as it follows from Rogers’ Isomorphism Theorem), but now there cannot be a recursive function β such that $x \in \ulcorner \Gamma^\neg \psi$ implies $\psi_x \neq \psi_{\beta(x)}$, simply because the condition $x \in \ulcorner \Gamma^\neg \psi$ holds for all x , and thus the existence of such a function β would contradict the Recursion Theorem.

One may have noticed that the fact that the particular enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ in the previous example is ϕ -w.u.n.t. depends essentially on the assumption that

⁶ By a fixpoint of a recursive function f we mean an x such that $\phi_x = \phi_{f(x)}$.

⁷ In other words, for a given recursive function f and a given acceptable programming systems φ and ψ , one has, in general, $\{x : \varphi_x = \varphi_{f(x)}\} \neq \{x : \psi_x = \psi_{f(x)}\}$.

the acceptable programming system ϕ satisfies the condition that $\phi_{2x} = f_\emptyset$, for all x . It is not difficult to see that there is, indeed, one such ϕ which satisfies this condition. For, let φ be any acceptable programming system, and let p be a recursive, increasing function such that, for all x , $\varphi_{p(x)} = f_\emptyset$ (such a function p exists by the Padding Lemma), and let R be the range of p . Since R is recursive, its complement, $\mathbb{C}R$, is recursive too. Moreover, $\mathbb{C}R$ is also infinite, and so let q be a recursive, increasing function whose range is $\mathbb{C}R$. Let ρ be the function defined by

$$\rho(x) = \begin{cases} p(x/2), & \text{if } x \text{ is even} \\ q((x-1)/2), & \text{otherwise.} \end{cases}$$

Plainly, the function ρ is recursive and bijective. Therefore, by Rogers' Isomorphism Theorem, $\phi =_{Def} \lambda xy. \varphi(\rho(x), y)$ is an acceptable programming system and we have that $\phi_{2x} = f_\emptyset$, for each x . In a similar way, it can be shown that there is an acceptable programming system ϕ such that, for all x , $\phi_{3x} = f_\emptyset$ and $\phi_{3x+2} = \lambda y.1$. Then, if we assume that the ϕ in Example 1 satisfies these last two conditions, it is immediate to verify that the enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ can not be ϕ -w.u.n.t. since, in this case, we have that $\ulcorner \Gamma^\neg \phi = \{x : (\exists y \leq x)(3y \leq x \leq 3y+1)\}$, a recursive set. Thus, the fact that $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -w.u.n.t. (and, in general, the fact that the diagonal set $\ulcorner \Gamma^\neg \phi$ is nonrecursive), depends on the "nature" of the acceptable programming system ϕ , and indeed each class Γ_x is defined in terms of ϕ . However, we may observe that the *membership problem* for $\ulcorner \Gamma^\neg \phi$, namely the problem of determining, for any given x , whether $x \in \ulcorner \Gamma^\neg \phi$, can be considered as an instance of a more general membership problem in which enumerations of classes of 2-ary partial recursive functions are involved. Indeed, for $x \in \mathbb{N}$, let Ω_x be the class of all 2-ary partial recursive functions f such that $\text{Domain}(f_x) \subseteq \text{Domain}(f_{x+1})$. Then, $\phi \in \Omega_x$ iff $\text{Domain}(\phi_x) \subseteq \text{Domain}(\phi_{x+1})$, and thus $\ulcorner \Gamma^\neg \phi = \{x : \phi \in \Omega_x\}$. In general we give the following definition.

Definition 4. *Given an enumeration $\Phi_0, \Phi_1, \Phi_2, \dots$ of classes of 2-ary partial recursive functions and given a 2-ary partial function f , we put*

$$\langle \Phi \rangle^f = \{x : f \in \Phi_x\}.$$

Thus $\ulcorner \Gamma^\neg \phi = \langle \Omega \rangle^\phi$.

Now, so far we have shown that, for the particular enumeration $\Omega_0, \Omega_1, \Omega_2, \dots$ defined above, the recursiveness of the set $\langle \Omega \rangle^\phi$ depends on the nature of the acceptable programming system ϕ . Similarly, one can easily show that the same would also hold if Ω_x were, for instance, the class of all 2-ary partial recursive functions f such that $\text{Domain}(f_x) \subseteq \mathbb{C}\text{Domain}(f_{x+1})$, whose corresponding set is $\langle \Omega \rangle^\phi = \{x : \text{Domain}(\phi_x) \cap \text{Domain}(\phi_{x+1}) = \emptyset\}$, or also if Ω_x were such that $\langle \Omega \rangle^\phi = \{x : \text{Domain}(\phi_x) \cup \text{Domain}(\phi_{x+1}) = \{x\}\}$.

We observe that in all of the cases above, the membership problem for $\langle \Omega \rangle^\phi$ essentially involves comparing the domains of the two "consecutive" functions ϕ_x and ϕ_{x+1} , by means of elementary set-theoretic operations, and so it is natural

to ask whether the reason why the recursiveness of $\langle \Omega \rangle^\phi$ depends upon ϕ is a consequence of this fact. But the answer is easily seen to be negative, as the following example shows.

Example 2. Let $\Omega_0, \Omega_1, \Omega_2, \dots$ be such that

$$\langle \Omega \rangle^\phi = \{x : \text{Domain}(\phi_x) \cup \{x+1\} \subseteq \text{Domain}(\phi_{x+1})\}.$$

We claim that $\langle \Omega \rangle^\phi$ is nonrecursive. Indeed, for $x \in \mathbb{N}$, let Γ_x be the class of all 1-ary partial recursive functions f such that $\text{Domain}(\phi_{x-1}) \cup \{x\} \subseteq \text{Domain}(f)$. Observe that $x \in \langle \Omega \rangle^\phi$ iff $\phi_{x+1} \in \Gamma_{x+1}$. Since the enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is plainly u.n.t., its diagonal set, $\ulcorner \Gamma^\top \phi \urcorner$, is nonrecursive. Then, we have $x \in \langle \Omega \rangle^\phi$ iff $x+1 \in \ulcorner \Gamma^\top \phi \urcorner$, i.e., $\langle \Omega \rangle^\phi = \{x : x+1 \in \ulcorner \Gamma^\top \phi \urcorner\}$, and thus our claim is correct.

An interesting question would be that of characterizing families \mathfrak{F} of enumerations of classes of 2-ary partial recursive functions containing only enumerations $\Phi_0, \Phi_1, \Phi_2, \dots$ such that the recursiveness of the set $\langle \Phi \rangle^\phi$ does not depend upon the acceptable programming system ϕ .

This question can be made more precise by introducing the notion of *safeness* of an enumeration of classes of 2-ary partial recursive functions, as follows.

Definition 5. Let $\Phi_0, \Phi_1, \Phi_2, \dots$ be an enumeration of classes of 2-ary partial recursive functions. We say that $\Phi_0, \Phi_1, \Phi_2, \dots$ is safe with respect to a class \mathcal{C} of subsets of \mathbb{N} , if for any two acceptable programming systems φ and ψ ,

$$\langle \Phi \rangle^\varphi \in \mathcal{C} \text{ iff } \langle \Phi \rangle^\psi \in \mathcal{C}.$$

Then, the question posed above can be rephrased as the problem of characterizing families of enumerations of classes of 2-ary partial recursive functions in terms of safeness (with respect to the class of all recursive sets) of their members.

A (relatively) small family of enumerations of classes of 2-ary partial recursive functions for which the above question admits a simple partial solution is that involving enumerations $\Phi_0, \Phi_1, \Phi_2, \dots$ such that, for all ϕ , the membership problem for the set $\langle \Phi \rangle^\phi$ can be expressed as “ $\phi_{r(x)} \in \Gamma_x$ ”, where r is a recursive, increasing function and $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is a u.n.t. enumeration of classes of 1-ary partial recursive functions. For instance, if Φ_x is the class of the functions f such that $f(2x, 0) = x$, then $\langle \Phi \rangle^\phi = \{x : \phi_{2x}(0) = x\} = \{x : \phi_{2x} \in \Gamma_x\}$, where Γ_x is the class of the functions g such that $g(0) = x$.

For this case we have the following result.

Theorem 2. Let $\Phi_0, \Phi_1, \Phi_2, \dots, r$ and $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ as above. If the complement of the range of r is infinite then the enumeration $\Phi_0, \Phi_1, \Phi_2, \dots$ is not safe with respect to the class of all recursive sets.

Proof. Suppose that $\mathcal{C}\text{Range}(r)$ is infinite. Let us assume, without any loss of generality, that the enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ is ϕ -u.n.t. (see Remark 1), and let α and β be as in Definition 2. Specifically, let n be such that $\phi_{\alpha(x)} \in \Gamma_x$ and $\phi_{\beta(x)} \notin \Gamma_x$ hold for all $x \geq n$. Let p be a padding function for ϕ , and let the function q be defined by the following primitive recursive equations:

$$\begin{cases} q(0) = \alpha(0), \\ q(x+1) = p(\alpha(x+1), q(x)+1). \end{cases}$$

Notice that the function q is recursive and increasing, and moreover

- (1) the complement of the range of q is recursive and infinite, and
- (2) $\phi_{q(x)} = \phi_{\alpha(x)}$, for all x .

Thus, let \tilde{r} and \tilde{q} be recursive, increasing functions such that $\text{Range}(\tilde{r}) = \mathbb{C}\text{Range}(r)$ and $\text{Range}(\tilde{q}) = \mathbb{C}\text{Range}(q)$, and let δ and σ be the functions defined by

$$\delta(x) = \begin{cases} q(r^{-1}(x)), & \text{if } x \in \text{Range}(r) \\ \tilde{q}(\tilde{r}^{-1}(x)), & \text{otherwise,} \end{cases} \quad \text{and} \quad \sigma(x) = \begin{cases} \tilde{q}(r^{-1}(x)), & \text{if } x \in \text{Range}(r) \\ \tilde{q}(\tilde{r}^{-1}(x)), & \text{otherwise,} \end{cases}$$

where $r^{-1} = \lambda x.(\mu y \leq x)(r(y) = x)$, and similarly for \tilde{r}^{-1} .⁸

Plainly, the functions δ and σ are recursive and, moreover, they are also bijective. Therefore, by Rogers' Isomorphism Theorem, $\varphi =_{Def} \lambda xy.\phi(\delta(x), y)$ and $\psi =_{Def} \lambda xy.\phi(\sigma(x), y)$ are acceptable programming systems. We claim that the set $\langle \Phi \rangle^\varphi$ is recursive, whereas $\langle \Phi \rangle^\psi$ is not. To begin with, notice that, for all x , we have that $\varphi_{r(x)} = \phi_{\delta(r(x))} = \phi_{q(x)}$ and $\psi_{r(x)} = \phi_{\sigma(r(x))} = \phi_{\tilde{q}(x)}$, and thus $\langle \Phi \rangle^\varphi = \{x : \phi_{q(x)} \in \Gamma_x\}$ and $\langle \Phi \rangle^\psi = \{x : \phi_{\tilde{q}(x)} \in \Gamma_x\}$.

Since $\phi_{\alpha(x)} \in \Gamma_x$ for each $x \geq n$, from (2), it follows that $\langle \Phi \rangle^\varphi \supseteq \{x : x \geq n\}$, and therefore $\langle \Phi \rangle^\varphi$ is a recursive set. Concerning $\langle \Phi \rangle^\psi$, let us suppose, by the way of contradiction, that it is recursive. Let s be the function defined by

$$s(x) = \begin{cases} q^{-1}(x), & \text{if } x \in \text{Range}(q) \\ \tilde{q}^{-1}(x), & \text{otherwise,} \end{cases}$$

and let $\Omega_x = \Gamma_{s(x)}$, for each x . Then, it is easy to verify that

$$\ulcorner \Omega \urcorner^\phi = (\text{Range}(q) \cap s^{-1}(\langle \Phi \rangle^\varphi)) \cup (\text{Range}(\tilde{q}) \cap s^{-1}(\langle \Phi \rangle^\psi)),$$

where $s^{-1}(A) = \{x : s(x) \in A\}$, for any set A . Thus, since s is a recursive function, the set $\ulcorner \Omega \urcorner^\phi$ is recursive. But this is in contrast with Property 1 since the enumeration $\Omega_0, \Omega_1, \Omega_2, \dots$ is ϕ -u.n.t. as $\phi_{\alpha(s(x))} \in \Omega_x$ and $\phi_{\beta(s(x))} \notin \Omega_x$, for each $x \geq n$, with $\lambda x.\alpha(s(x))$ and $\lambda x.\beta(s(x))$ recursive functions. This concludes the proof of the Theorem

In the next section we present some results which can be used to obtain quick proofs of the nonrecursiveness or nonrecursive enumerability of certain sets of natural numbers, based on the idea of regarding a set of natural numbers as the diagonal set of an enumeration of classes of 1-ary partial recursive functions, as discussed before. However, for expository reasons, we shall not mention explicitly

⁸ In general, given a recursive function f , we put $f^{-1} = \lambda x.(\mu y \leq x)(f(y) = x)$, i.e.,

$$f^{-1}(x) = \begin{cases} \text{the least } y \leq x \text{ such that } f(y) = x, & \text{if one such } y \text{ exists} \\ x + 1, & \text{otherwise.} \end{cases}$$

Notice that if f is increasing, then $f^{-1}(f(x)) = x$, for all x , and additionally $f(f^{-1}(x)) = x$, provided that $x \in \text{Range}(f)$.

that the sets we have under consideration are to be regarded as diagonal sets of enumerations of classes of 1-ary partial recursive functions, although, in some cases, we shall emphasize this fact. Moreover, we introduce also the new and intermediate concept of FIP-reduction which turns out to be useful for simplifying proofs.

2 Technical Results

We present a technique to show that certain subsets of \mathbb{N} are nonrecursive (or nonrecursively enumerable), which make use, basically, of the Recursion Theorem with parameters. Our technique has a wide range of applications and is open to possible extensions. In addition it requires only very minimal assumptions.

In the sequel we assume that ϕ is a fixed acceptable programming system and, for $x \in \mathbb{N}$, we denote with W_x and E_x the domain and the range of the function ϕ_x , respectively. Moreover, we put $K = \{x : x \in W_x\}$. If f is a 2-ary partial function, we put $\text{FIP}(f) = \{x : \phi_x = f_x\}$. Given a r.e. set W , an index for W is an x such that $W_x = W$. Similarly, if f is a 1-ary partial recursive function, an index for f is an x such that $f = \phi_x$.

We begin with an example illustrating our main idea.

Example 3. Let $A = \{x : \phi_x(0) = x\}$. Let us show that A is nonrecursive. We proceed as follows. Let g and h be the 2-ary partial recursive functions such that $g(x, y) = x$ and $h(x, y) = x + 1$, for all x and y . Notice that $\text{FIP}(g) \subseteq A$ and $\text{FIP}(h) \subseteq \mathbb{C}A$. Now, suppose by way of contradiction, that A is recursive. Let the function f be defined as follows

$$f(x, y) = \begin{cases} g(x, y), & \text{if } x \in \mathbb{C}A \\ h(x, y), & \text{otherwise.} \end{cases}$$

Then, since A is recursive, the function f is partial recursive and we have that

$$\mathbb{C}A \cap \text{FIP}(f) \subseteq A \quad \text{and} \quad A \cap \text{FIP}(f) \subseteq \mathbb{C}A. \quad (1)$$

By applying the s-m-n Theorem we get a 1-ary recursive function k such that $\phi_{k(x)} = f_x$, for all x , and therefore, by the Recursion Theorem, there is a number e such that $\phi_e = f_e$, i.e., $e \in \text{FIP}(f)$. By (1) above we have that $e \in \mathbb{C}A$ iff $e \in A$, which is a contradiction.

We notice that in the proof that set A of the previous example is nonrecursive, the starting point was to find the 2-ary partial recursive functions g and h such that $\text{FIP}(g) \subseteq A$ and $\text{FIP}(h) \subseteq \mathbb{C}A$, and then a subsequent application of the s-m-n Theorem and the Recursion Theorem did the rest. A similar argument can be used to show that many other sets are nonrecursive. This is the case, for instance, of the set $B = \{x : x^2 \in (W_x \cap E_x)\}$, where one can choose the functions g and h such that $g = \lambda xy.x^2$ and $h = \lambda xy.(x^2 + 1)$. The above idea can be suitably generalized to obtain stronger undecidability results, by introducing parameters at the appropriate place. This point is illustrated in the following example.

Example 4. Let

$$\begin{aligned} A &= \{x : W_x = \{x\}\}, \\ B &= \{x : W_x = \mathbb{N} \text{ AND } E_x = \{x, x+1, x+2, \dots\}\}, \text{ and} \\ C &= (\mathbb{E} \cap A) \cup (\mathbb{O} \cap B), \end{aligned}$$

where \mathbb{E} is the set of the even numbers and \mathbb{O} the set of odd numbers. Consider the following proofs that $\mathfrak{C}K \leq_m A$, $\mathfrak{C}K \leq_m B$ and $\mathfrak{C}K \leq_m C$. Let f be the 3-ary partial function defined by

$$f(x, y, z) = \begin{cases} 1, & \text{if } (z \geq y) \text{ AND } (x \in K \text{ OR } z = y) \\ \uparrow, & \text{otherwise.} \end{cases}$$

Since f is plainly partial recursive, by the s-m-n Theorem there is a 2-ary recursive function s such that $\phi_{s(x,y)}(z) = f(x, y, z)$, for all x, y and z . Since s is recursive, by the Recursion Theorem with parameters there is a 1-ary recursive function k such that $\phi_{k(x)}(z) = \phi_{s(x,k(x))}(z)$, for all x and z , i.e., $\phi_{k(x)} = \phi_{s(x,k(x))}$. By the definition of the function f , it is immediate to verify that, for all x , $x \notin K$ iff $W_{k(x)} = \{k(x)\}$, i.e., $x \in \mathfrak{C}K$ iff $k(x) \in A$. Therefore $\mathfrak{C}K \leq_m A$. To show that $\mathfrak{C}K \leq_m B$, let

$$g(x, y, z) = \begin{cases} y + z, & \text{if } \neg STP(e, x, z) \\ \uparrow, & \text{otherwise,} \end{cases}$$

where e is an index for K . Then, as before, if we apply the s-m-n Theorem to g first, and then apply the Recursion Theorem with parameters, we get a 1-ary recursive function l such that $x \in \mathfrak{C}K$ iff $l(x) \in B$, for all x , showing that $\mathfrak{C}K \leq_m B$, as required. Finally, to show that $\mathfrak{C}K \leq_m C$, it is enough to consider the function

$$h(x, y, z) = \begin{cases} f(x, y, z), & \text{if } y \in \mathbb{E} \\ g(x, y, z), & \text{otherwise,} \end{cases}$$

and then to use similar arguments as before.

What do the three proofs in Example 4 have in common? Let us call the sets A , B and C of Example 4 the target sets, and let us denote them with T . A careful inspection reveals that in proving the generic reduction $\mathfrak{C}K \leq_m T$ we started by constructing a 3-ary partial recursive function φ ($\varphi = f$ in the case $T = A$, $\varphi = g$ in the case $T = B$ and $\varphi = h$ in the case $T = C$) such that, for all x ,

$$x \in \mathfrak{C}K \text{ implies that } \text{FIP}(\varphi_x) \subseteq T \quad \text{and} \quad x \notin \mathfrak{C}K \text{ implies that } \text{FIP}(\varphi_x) \subseteq \mathfrak{C}T.$$

After that, we have used the s-m-n Theorem and the Recursion Theorem with parameters to obtain a recursive function r such that $r(x) \in \text{FIP}(\varphi_x)$, for all x . Such function r provides the desired many-one reduction from $\mathfrak{C}K$ to T . Notice also that the function h has been constructed by combining the functions f and g , as suggested by the observation that, for all x ,

$$x \in \mathfrak{C}K \text{ implies } \text{FIP}(f_x) \subseteq \mathfrak{C}\mathbb{E} \cup C \quad \text{and} \quad x \notin \mathfrak{C}K \text{ implies } \text{FIP}(f_x) \subseteq \mathfrak{C}\mathbb{E} \cup \mathfrak{C}C,$$

and similarly for the function g .⁹

The above considerations are formalized in the following definition.

Definition 6. *Given sets $A, B, X \subseteq \mathbb{N}$, we say that A is FIP-reducible to B with respect to X , and write $A \triangleleft_X B$, if there is a 3-ary partial recursive function f such that, for all x ,*

- (1) if $x \in A$ then $\text{FIP}(f_x) \subseteq \mathbb{C}X \cup B$;
- (2) if $x \notin A$ then $\text{FIP}(f_x) \subseteq \mathbb{C}X \cup \mathbb{C}B$.

We write $A \triangleleft B$ to mean that $A \triangleleft_{\mathbb{N}} B$.

Before we go further, let us state the following elementary property which will be useful to simplify proofs.

Property 3. Let A, B and X be subsets of \mathbb{N} . Then, $A \triangleleft_X B$ iff there is a 2-ary recursive function r such that, for all x and y ,

- (a) if $x \in A$ and $\phi_y = \phi_{r(x,y)}$, then $y \in (\mathbb{C}X \cup B)$;
- (b) if $x \notin A$ and $\phi_y = \phi_{r(x,y)}$, then $y \in (\mathbb{C}X \cup \mathbb{C}B)$.

Proof. Simply note that if f satisfies conditions (1) and (2) of Definition 6 then, by the s-m-n Theorem, there is a recursive function r such that, for all x, y and z , we have that $\phi_{r(x,y)}(z) = f(x, y, z)$. Then, observe that $\phi_y = \phi_{r(x,y)}$ iff $y \in \text{FIP}(f_x)$.

Next, we give the following definition.

Definition 7. *A 2-ary recursive function r satisfying conditions (a) and (b) of Property 3 is called a FIP-reduction from A to B relative to X . We write $A \triangleleft_X^r B$ to mean that r is a FIP-reduction from A to B relative to X . We write also $A \triangleleft^r B$ to mean that $A \triangleleft_{\mathbb{N}}^r B$.*

In the following lemma we present some properties of the relation of FIP-reduction.

Lemma 1. *Let A, B, X and Y be subsets of \mathbb{N} . Then,*

- (1) if $A \triangleleft_X B$ and $Y \subseteq X$, then $A \triangleleft_Y B$;
- (2) if X is finite, then $A \triangleleft_X B$;
- (3) if $A \triangleleft_X B$, $A \triangleleft_Y B$ and X (or Y) is recursive, then $A \triangleleft_{(X \cup Y)} B$;
- (4) if $A \triangleleft B$ then $A \leq_m B$;
- (5) if $A \leq_m B$ and $B \triangleleft_X Y$, then $A \triangleleft_X Y$;
- (6) if $A \triangleleft_X B$ then $A \triangleleft_X (B \cap X)$;
- (7) if $A \triangleleft_X B$ and $Y \cap X = \emptyset$, then if $A \triangleleft_X (B \cup Y)$.

⁹ See the proofs of Theorems 4 and 5 to understand the idea underlying the construction of the functions f and g , whose definition may seem a bit mysterious at first.

Proof. Properties (1), (6) and (7) are immediate consequences of the definition of FIP-reduction, so we give only the proofs of (2), (3), (4) and (5).

(2): Assume that X is finite, and let n be such that $X \cap \{x \in \mathbb{N} : x > n\} = \emptyset$. Moreover, let $e \in \mathbb{N}$ be such that $\phi_e \neq \phi_x$, for all $x \leq n$, and let r be the 2-ary recursive function defined by $r(x, y) = e$, for all x and y . Then, if $\phi_y = \phi_{r(x, y)}$, we must have that $y \in \mathbb{C}X$. This implies that $A \triangleleft_X^r B$.

(3): Suppose that $A \triangleleft_X B$, $A \triangleleft_Y B$, and also suppose that X is recursive. Let r and s be 2-ary recursive functions such that $A \triangleleft_X^r B$ and $A \triangleleft_Y^s B$. Let $t(x, y)$ be the function defined as follows:

$$t(x, y) = \begin{cases} r(x, y), & \text{if } y \in X \\ s(x, y), & \text{otherwise.} \end{cases}$$

Plainly, t is a recursive function, and it is immediate to verify that $A \triangleleft_{(X \cup Y)}^t B$.

(4): Assume that $A \triangleleft B$, and let r be such that $A \triangleleft^r B$. Then, since r is recursive, by the Recursion Theorem with parameters, there is a 1-ary recursive function k such that, for all x , $\phi_{k(x)} = \phi_{r(x, k(x))}$. Plainly, $A = \{x : k(x) \in B\}$, and thus $A \leq_m B$.

(5): Let $A \leq_m B$ via a function k and let r be a FIP-reduction from B to Y relative to X . Then it is immediate to verify that the function $\lambda xy.r(k(x), y)$ is a FIP-reduction from A to Y relative to X , and thus $A \triangleleft_X Y$.

Definition 8. Let \mathcal{F} be a family of subsets of \mathbb{N} , and let A and X be subsets of \mathbb{N} . We say that \mathcal{F} is FIP-reducible to A with respect to X , and write $\mathcal{F} \lll_X A$, if $B \triangleleft_X A$, for all $B \in \mathcal{F}$. Moreover, we write $\mathcal{F} \lll A$ to mean that $\mathcal{F} \lll_{\mathbb{N}} A$.

Lemma 2. Let A and X be subsets of \mathbb{N} , and let \mathcal{F} be a family of subsets of \mathbb{N} . If X is co-finite and $\mathcal{F} \lll_X A$, then $\mathcal{F} \lll A$ and $\mathbb{C}A \notin \mathcal{F}$. Therefore, if \mathcal{F} is closed under complementation, then $A \notin \mathcal{F}$.

Proof. Plainly, if X is co-finite and $\mathcal{F} \lll_X A$, then, by (2) and (3) of Lemma 1, it follows that $\mathcal{F} \lll A$. Thus, let us assume that $\mathcal{F} \lll A$.

Suppose, by way of contradiction, that $\mathbb{C}A \in \mathcal{F}$. Then, $\mathbb{C}A \triangleleft A$. So, let r be a 2-ary recursive function such that $\mathbb{C}A \triangleleft^r A$. By the Recursion Theorem there is an n such that $\phi_n = \phi_{r(n, n)}$. Then, we have that $n \in \mathbb{C}A$ iff $n \in A$, which is a contradiction. Therefore $\mathbb{C}A$ cannot be a member of \mathcal{F} .

The following theorem generalizes the idea used in the proof of Example 3.

Theorem 3. Let A and X be subsets of \mathbb{N} . Suppose that there are 2-ary partial recursive functions f and g such that

- (1) $X \cap \text{FIP}(f) \subseteq A$;
- (2) $X \cap \text{FIP}(g) \subseteq \mathbb{C}A$.

Then, $\text{REC} \lll_X A$.

Moreover, if X is co-finite, then $\text{REC} \lll A$ and A is nonrecursive.

Proof. Given a recursive set B , we define the function

$$h(x, y, z) = \begin{cases} f(y, z), & \text{if } x \in B \\ g(y, z), & \text{otherwise.} \end{cases}$$

Plainly, $h(x, y, z)$ is partial recursive, and thus, by the s-m-n Theorem, there is a recursive function $r(x, y)$ such that $\phi_{r(x, y)}(z) = h(x, y, z)$, for all x, y and z . It is immediate to verify that r is a FIP-reduction from B to A relative to X , and thus $B \triangleleft_X A$. This proves the first part of the theorem. The second part follows at once from Lemma 2 and the fact that **REC** is closed under complementation.

Remark 2. Notice that if A and X are subsets of \mathbb{N} , with X co-finite, then the condition on the existence of two 2-ary partial recursive functions f and g satisfying (1) and (2) of Theorem 3, is equivalent to the condition that A equals the diagonal set of some ϕ -w.u.n.t. enumeration $\Gamma_0, \Gamma_1, \Gamma_2, \dots$ of classes of 1-ary partial recursive functions (see Section 1). To see that this is the case, it is enough to set, for all x ,

$$\Gamma_x = \begin{cases} \{\phi_x\}, & \text{if } x \in A \cap \complement X \\ \mathcal{P} \setminus \{g_x\}, & \text{if } x \in A \cap X \\ \emptyset, & \text{if } x \in \complement A \cap \complement X \\ \{f_x\}, & \text{otherwise.} \end{cases}$$

Then, observe that $A = \ulcorner \Gamma \urcorner^\phi$. Moreover, the s-m-n Theorem ensures that there are recursive functions α and β such that $\phi_{\alpha(x)} = f_x$ and $\phi_{\beta(x)} = g_x$, for all x . It is then a simple matter to show that conditions (1) and (2) of Definition 3 are satisfied for almost all x , namely for all $x \in X$.

The two theorems which follow generalize the proofs in Example 4.

Theorem 4. *Let A and X be subsets of \mathbb{N} . Suppose that there are 2-ary partial recursive functions f and g such that*

- (1) $X \cap \text{FIP}(f) \subseteq A$;
- (2) $X \cap \text{FIP}(g) \subseteq \complement A$;
- (3) $f_x \subseteq g_x$, for all $x \in X \cap \complement A$.

Then, $\text{CO-RE} \triangleleft_X A$. Moreover, if X is co-finite, then $\text{CO-RE} \triangleleft A$ and A is nonrecursively enumerable.

Proof. Let h be the 4-ary partial function defined as follows:

$$h(t, x, y, z) = \begin{cases} g(y, z), & \text{if } x \in W_t \vee f(y, z) \downarrow \\ \uparrow, & \text{otherwise.} \end{cases}$$

Plainly, h is partial recursive, and thus, by the s-m-n Theorem, there is a 3-ary recursive function s such that $h(t, x, y, z) = \phi_{s(t, x, y)}(z)$, for all t, x, y and z .

Notice that, if $x \in W_t$, then $\phi_{s(t,x,y)} = g_y$, for each y . Moreover, if $x \notin W_t$ and $y \in X \cap \mathbb{C}A$, then $\phi_{s(t,x,y)} = f_y$, since $f_y \subseteq g_y$, since $y \in X \cap \mathbb{C}A$.

Now, let W be a r.e. set and let e be an index for W , i.e., $W_e = W$. Let $r = \lambda xy.s(e, x, y)$. We claim that $\mathbb{C}W \triangleleft_X^r A$. Indeed, let x and y be such that $y \in X$ and $\phi_y = \phi_{r(x,y)}$. We have to show that:

- (a) $x \in \mathbb{C}W$ implies that $y \in A$, and
- (b) $x \in W$ implies that $y \in \mathbb{C}A$.

Assume first that $x \in \mathbb{C}W$ and suppose, by way of contradiction, that $y \notin A$, i.e., $y \in \mathbb{C}A$. Then, since $y \in X$, we have that $y \in X \cap \mathbb{C}A$. Moreover, $x \in \mathbb{C}W$ implies that $x \notin W_e$, and hence $\phi_{s(e,x,y)} = f_y$, i.e., $\phi_{r(x,y)} = f_y$. This implies that $y \in X \cap \text{FIP}(f)$, and thus, from (1), it follows that $y \in A$: a contradiction. Therefore (a) holds. Concerning (b), assume that $x \in W$. Then, $x \in W_e$. This implies that $\phi_{s(e,x,y)} = g_y$ (i.e., $\phi_{r(x,y)} = \phi_y$), and so $y \in X \cap \text{FIP}(g)$, which in turn implies that $y \in \mathbb{C}A$.

So far we have shown that $\mathbb{C}W \triangleleft_X A$. Since W is an arbitrary r.e. set, we conclude that $\text{CO-RE} \triangleleft_X A$. This proves the first part of the Theorem. The second part follows from Lemma 2.

Theorem 5. *Let A and X be subsets of \mathbb{N} . Suppose that there is a 2-ary partial recursive function f such that*

- (1) $X \cap \text{FIP}(f) \subseteq A$;
- (2) if $\phi_x \subseteq f_x$ and $x \in X$, then $x \in \mathbb{C}A$.

Then, $\text{CO-RE} \triangleleft_X A$ and $\text{CO-RE} \triangleleft_X \mathbb{C}A$. Moreover, if X is co-finite, then $\text{CO-RE} \triangleleft A$ and $\text{CO-RE} \triangleleft \mathbb{C}A$ and neither A nor $\mathbb{C}A$ are recursively enumerable.

Proof. Let g be the 4-ary partial function defined as follows:

$$g(t, x, y, z) = \begin{cases} f(y, z), & \text{if } \neg \text{STP}(t, x, z) \\ \uparrow, & \text{otherwise.} \end{cases}$$

Since g is plainly partial recursive, by the s-m-n Theorem there is a 3-ary recursive function s such that $\phi_{s(t,x,y)}(z) = g(t, x, y, z)$, for all t, x, y and z . Notice that, if $x \notin W_t$ then $\{z : \neg \text{STP}(t, x, z)\} = \mathbb{N}$, and thus $\phi_{s(t,x,y)} = f_y$. Moreover, if $x \in W_t$ then there is a z^* such that $\{z : \neg \text{STP}(t, x, z)\} \subseteq \{z : z < z^*\}$, and thus $\phi_{s(t,x,y)} \subseteq f_y$.

Now, let W be a r.e. set, and let e be an index for W , i.e., $W_e = W$. Let $r = \lambda xy.s(e, x, y)$. Moreover, let x and y be such that $y \in X$ and $\phi_y = \phi_{r(x,y)}$. Then, by using a reasoning similar to that in the proof of Theorem 4, it is easy to verify that $x \in \mathbb{C}W$ implies that $y \in A$, and $x \in W$ implies that $y \in \mathbb{C}A$. Thus r is a FIP-reduction from $\mathbb{C}W$ to A relative to X , and so $\mathbb{C}W \triangleleft_X A$. Since W was an arbitrary r.e. we conclude that $\text{CO-RE} \triangleleft_X A$.

To show that $\text{CO-RE} \triangleleft_X \mathbb{C}A$, we observe that if $g(x, y) = f_\emptyset$ then, from (2), it follows that $X \cap \text{FIP}(g) \subseteq \mathbb{C}A$. Moreover, $g_x \subseteq f_x$, for each x . From these facts and (1), by applying Theorem 4, we get the desired result. The first part of the theorem is thus proved. The second part follows again from Lemma 2.

The following example presents some applications of Theorems 4 and 5.

Example 5. Let

$$A = \{x : W_x = \{x\}\}, \text{ and} \\ B = \{x : W_x = \mathbb{N} \text{ AND } E_x = \{x, x + 1, x + 2, \dots\}\} \text{ (cf. Example 3).}$$

Let f , g and h be the partial recursive functions such that, for all x and y :

$$f(x, y) = \begin{cases} 1, & \text{if } y = x \\ \uparrow, & \text{otherwise,} \end{cases} \quad g(x, y) = \begin{cases} 1, & \text{if } y \geq x \\ \uparrow, & \text{otherwise,} \end{cases} \quad h(x, y) = y + x.$$

Then, it is immediate to verify that the following conditions hold:

- (1) $\text{FIP}(f) \subseteq A$;
- (2) $\text{FIP}(g) \subseteq \mathcal{C}A$;
- (3) $f_x \subseteq g_x$, for all x ;
- (4) $\text{FIP}(h) \subseteq B$;
- (5) if $\phi_x \sqsubseteq h_x$ then $x \notin B$.

Therefore, by (1), (2) and (3), from Theorem 4, it follows that $\text{CO-RE} \lll A$. Similarly, from (4) and (5), using Theorem 5, it follows that $\text{CO-RE} \lll B$. Thus, $\mathcal{C}K \leq_m A$ and $\mathcal{C}K \leq_m B$. Observe also, that $\phi_x = g_x$ implies that $x \notin B$ (i.e., $\text{FIP}(g) \subseteq \mathcal{C}B$), and thus, by Theorem 4, we get $\mathcal{C}K \leq_m (A \cup B)$. Furthermore, if X and Y are disjoint subsets of \mathbb{N} (e.g., $X = \mathbb{E}$ and $Y = \mathbb{O}$, see Example 3), then by (1), (6) and (7) of Lemma 1, we get

$$\text{CO-RE} \lll_X (A \cap X) \cup (B \cap Y) \quad \text{and} \quad \text{CO-RE} \lll_Y (A \cap X) \cup (B \cap Y).$$

Therefore, if X (or Y) is recursive and, in addition, $X \cup Y$ is co-finite, then by (3) and (2) of Lemma 1 it follows that $\text{CO-RE} \lll (A \cap X) \cup (B \cap Y)$, and thus $\mathcal{C}K \leq_m ((A \cap X) \cup (B \cap Y))$.

3 Conclusions

We have presented some extensions of Rice's and Rice-Shapiro theorems which turn out to be useful for proving the nonrecursiveness and nonrecursive enumerability of many sets of natural numbers. Our extensions are based on the idea of regarding a set of natural numbers as the diagonal set of an enumeration of classes of partial recursive functions, a new concept which we have introduced in the paper, which, in many cases, is perhaps the most natural way of regarding a set. By extending this idea, we have also considered problems concerning the membership of acceptable programming systems to classes of partial recursive functions, and we have introduced the related notion of safeness of an enumeration of classes of partial recursive functions. Moreover, we have raised the question of characterizing families of safe enumerations. A partial solution to this question has been provided for a restricted family of enumerations of classes of partial recursive functions. We plan to further investigate the above question (and other related ones) more deeply in the future and to provide extended solutions.

References

1. H. Gordon Rice: Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, Vol. 74(2), 358–366, 1953.
2. H. Gordon Rice: On Completely Recursively Enumerable Classes and Their Key Arrays. *Journal of Symbolic Logic*, Vol. 21(3), pp. 304–308, 1956.
3. Norman Shapiro: Degrees of Computability. *Transactions of the American Mathematical Society*, Vol. 82(2), pp. 281–299, 1956.
4. Hartley Rogers, Jr.: *Theory of recursive functions and effective computability*. 2nd edn. MIT Press, Cambridge, MA, 1987.
5. Nigel Cutland: *Computability: an introduction to recursive function theory*. Cambridge University Press, Cambridge, 1980.
6. Carl H. Smith: *A recursive introduction to the theory of computation*. Springer-Verlag, New York, Inc., Secaucus, NJ., 1994.

Appendix: Basic definitions

Let A be a subset of \mathbb{N} . We denote with $\complement A$ the complement of A , i.e., $\complement A = \mathbb{N} \setminus A$. The *characteristic* function of A is the 1-ary total function χ_A such that, for all x , $\chi_A(x) = 1$, if $x \in A$ and $\chi_A(x) = 0$, otherwise. We say that A is *recursive* if χ_A is a recursive function; A is *recursively enumerable* (abbreviated r.e.) if A is the range of a partial recursive function; A is *co-recursively enumerable* (abbreviated co-r.e.) if its complement is r.e. Additionally, we say that A is *co-finite* if its complement is a finite set. We denote with RE, REC and CO–RE the classes of all recursive sets, of all r.e. sets, and of all co-r.e. sets, respectively.

If A and B are subsets of \mathbb{N} , we say that A is *many-one reducible* to B , and write $A \leq_m B$, if there is a 1-ary recursive function f such that $x \in A$ iff $f(x) \in B$, for all x . In such a case, we say that A is *many-one reducible to B via f* , and write $A \leq_m^f B$.

Let f be an $(n + 1)$ -ary partial function, with $n > 0$. For $x \in \mathbb{N}$, we denote with f_x the n -ary partial function such that, for all y_1, \dots, y_n ,

$$f_x(y_1, \dots, y_n) = f(x, y_1, \dots, y_n).$$

The domain and the range of a partial function f are denoted with $\text{Domain}(f)$ and $\text{Range}(f)$, respectively. The *everywhere undefined function*, i.e., the function with empty domain, is denoted with f_\emptyset . The class of all 1-ary partial recursive functions is denoted with \mathcal{P} .

If f is an n -ary partial function and $x_1, \dots, x_n \in \mathbb{N}$, we write $f(x_1, \dots, x_n) \downarrow$ to mean that $f(x_1, \dots, x_n)$ is defined (i.e., $(x_1, \dots, x_n) \in \text{Domain}(f)$). Similarly, we write $f(x_1, \dots, x_n) \uparrow$ to mean that $f(x_1, \dots, x_n)$ is undefined (i.e., $(x_1, \dots, x_n) \notin \text{Domain}(f)$).

If f and g are partial functions we write $f \subseteq g$ to mean that f is a restriction of g , and $f \sqsubseteq g$ to mean that f is a finite restriction of g .

An *acceptable programming system* is a 2-ary partial recursive function ϕ satisfying the following conditions:

- (1) for any 1-ary partial recursive function f there is an x such that $\phi_x = f$;
- (2) there is a 2-ary recursive function c such that $\phi_{c(x,y)} = \lambda z. \phi_x(\phi_y(z))$, for all x and y .

Next, we recall some basic results in the Theory of Recursive Functions.

Theorem 6. (*Step-counter Theorem*) *If ϕ is an acceptable programming system, then there is a 3-ary recursive predicate STP such that, for all x, y and u ,*

- (1) $\phi_x(y) \downarrow$ iff there exists a z such that $STP(x, y, z)$ holds;
- (2) if $STP(x, y, u)$ holds then $STP(x, y, z)$ holds for all $z \geq u$.

Theorem 7. (*The s-m-n Theorem*) *Let ϕ be an acceptable programming system. If f is an $(n+1)$ -ary partial recursive function, then there exists an n -ary recursive function s such that, for all x_1, \dots, x_n and y ,*

$$f(x_1, \dots, x_n, y) = \phi_{s(x_1, \dots, x_n)}(y).$$

Theorem 8. (*Recursion Theorem*) *Let ϕ be an acceptable programming system. If f is a 1-ary recursive function, there are infinitely many numbers n such that*

$$\phi_{f(n)} = \phi_n.$$

Theorem 9. (*The Recursion Theorem with parameters*) *Let ϕ be an acceptable programming system. If f is a 2-ary recursive function, then there is a 1-ary recursive function k such that, for all x ,*

$$\phi_{f(x, k(x))} = \phi_{k(x)}.$$

Theorem 10. (*Rogers' Isomorphism Theorem*) *Let ϕ be a fixed acceptable programming system, and let ψ be a 2-ary partial recursive function. Then, the following conditions are equivalent:*

- (1) ψ is an acceptable programming system;
- (2) there are recursive functions τ_1 and τ_2 such that, for all x ,

$$\phi_x = \psi_{\tau_1(x)} \quad \text{and} \quad \psi_x = \phi_{\tau_2(x)};$$

- (3) there is a recursive, bijective function ρ such that, for all x ,

$$\phi_x = \psi_{\rho(x)}.$$

Theorem 11. (*Padding Lemma*) *Let ϕ be an acceptable programming system. Then, there is a 2-ary recursive function p , called a padding function for ϕ , such that, for all x and y , $\phi_{p(x,y)} = \phi_x$ and $p(x, y) > y$.*