Properties of Stable Model Semantics Extensions

Mário Abrantes

Departamento de Matemática, Escola Superior de Tecnologia e de Gestão,
Instituto Politécnico de Bragança, 5300-253 Bragança, Portugal

Luís Moniz Pereira

Centro de Inteligência Artificial (CENTRIA), Departamento de Informática,
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The stable model (SM) semantics lacks the properties of existence, relevance and cumulativity. If we prospectively consider the class of conservative extensions of the SM semantics (i.e., semantics that for each normal logic program P retrieve a superset of the set of stable models of P), one may wander how do the semantics of this class behave in what concerns the aforementioned properties. That is the type of issue dealt with in this paper. We define a large class of conservative extensions of the SM semantics, dubbed affix stable model semantics (ASM), and study the above referred properties into two non-disjoint subfamilies of the class ASM, here dubbed ASM^h and ASM^m. From this study a number of results stem which facilitate the assessment of semantics in the class $ASM^h \cup ASM^m$ with respect to the properties of existence, relevance and cumulativity, whilst unveiling relations among these properties. As a result of the approach taken in our work, light is shed on the characterization of the SM semantics, as we show that the properties of (lack of) existence and (lack of) cautious monotony are equivalent, which opposes statements on this issue that may be found in the literature. We also characterize the relevance failure of SM semantics in a more clear way than usually stated in the literature.

KEYWORDS: Stable model semantics, Conservative extensions to stable model semantics, Existence, Relevance, Cumulativity, Defectivity, Excessiveness, Irregularity, 2-valued semantics for logic programs

1 Introduction

The *SM* semantics (Gelfond and Lifschitz 1988) is generally accepted by the scientific community working on logic programs semantics as the *de facto* standard 2-valued semantics. Nevertheless there are some advantageous properties the SM semantics lacks such as (1) model existence for every normal logic program, (2) relevance, and (3) cumulativity (Pinto and Pereira 2011). Model *existence* guarantees that every normal logic program has a semantics. This is important to allow arbitrary updates and/or merges involving Knowledge Bases, possibly from different authors or sources (Pinto and Pereira 2011). *Relevance* allows for top-down query solving without the need to always compute complete models, but just the sub-models that sustain the answer to a query, though guaranteed extendable to whole ones (Pinto and Pereira 2011). As for *cumulativity*, it allows the programmer to take advantage of tabling techniques (Swift 1999) for speeding up computations (Pinto and Pereira 2011). Independently of the motivations that underlay the design of a semantics for logic programs, one may ask if it is easy to guarantee some or all of

the above properties, or even if it is easy to assess the profile of the resulting semantics in what concerns these properties. In this work we define a family of 2-valued conservative extensions of the *SM* semantics, the *affix stable model* semantics family, *ASM*. We then take two subclasses, $ASM^h \subset ASM$ and $ASM^m \subset ASM$, and present a number of results that simplify the task of assessing the semantics in $ASM^h \cup ASM^m$ on the properties of existence, relevance and cumulativity. The semantics in these two classes bear resemblance with the already known *SM* and *MH* semantics (see section 3), and this stands for the motivation to consider them. The following results, obtained in this work, should be emphasized: (1) We present a refined definition of cumulativity for semantics in the class $ASM^h \cup ASM^m$, which turns into an easier job the dismissal of this property by resorting to counter-examples; (2) We divide the sets of rules of normal logic programs into layers, and use the decomposition of models into that layered structure to define three new (structural) properties, *defectivity*, *excessiveness* and *irregularity*, which allow to state a number of relations between the properties of existence, relevance and cumulativity for semantics of the $ASM^h \cup ASM^m$ class, and at the same time facilitate the assessment of semantics in this class with respect to those properties; (3) As a result of the approach in our work light is shed on the characterization of *SM* semantics, as we show that the properties of (lack of) existence and (lack of) cautious monotony are equivalent, which opposes statements on this issue that may be found in the literature; we also characterize the relevance failure of *SM* semantics in a more clear way than usually stated in the literature. It should be stressed that this study is on the properties of a class of 2-valued semantics, under a prospection motivation. The weighing of such semantics rationales under an 'intuitive' point of view (or any other equivalently non-objective concept) is beyond the reach of our study. The results presented in this paper are enounced for the universe of finite ground normal logic programs, and are either proved in (Abrantes 2013), or immediate consequences of results there contained.

The remainder of the paper proceeds as follows. In section 2 we define the language of normal logic programs and the terminology to be used in the sequel. In section 3 the families *ASM*, ASM^h and ASM^m are defined. In section 4 we characterize the property of cumulativity for the families ASM^h and ASM^m, whilst in section 5 the properties of defectivity, excessiveness and irregularity are defined. Some relations among existence, relevance and cumulativity, which are revealed by means of those properties, are stated. Section 6 is dedicated to final remarks.

2 Language and Terminology of Logic Programs

A *normal logic program* defined over a language $\mathscr{L}$ is a set of *normal* rules, each of the form

$$b_0 \leftarrow b_1, \cdots, b_m, not\ c_1, \cdots, not\ c_n \tag{1}$$

where m,n are non-negative integers and b_j, c_k are *atoms* of $\mathscr{L}$; b_i and $not\ c_k$ are generically designated *literals*, $not\ c_k$ being specifically designated *default literal*. The operator ',' stands for the conjunctive connective, the operator '*not*' stands for negation by default and the operator '$\leftarrow$' stands for a *dependency* operator that establishes a dependence of b_0 on the conjunction on the right side of '$\leftarrow$'. b_0 is the *head* of the rule and $b_1, \cdots, b_m, not\ c_1, \cdots, not\ c_n$ is the *body* of the rule. A rule is a *fact* if $m = n = 0$. A literal (or a program) is *ground* if it does not contain variables. The set of all ground atoms of a normal logic program is called *Herbrand base* of P,

H_P. A program is *finite* if it has a finite number of rules[1]. Given a program P, program Q is a *subprogram* of P if $Q \subseteq P$, where Q, P are envisaged as sets of rules.

For ease of exposition we henceforth use the following abbreviations: $Atoms(E)$, is the set of all *atoms* that appear in the ground structure E, where E can be a rule, a set of rules or a set of logic expressions; $Body(r)$, is the set of literals in the body of a ground rule r; $Facts(E)$, is the set of all facts that appear in the set of rules E; $Heads(E)$, is the set of all atoms that appear in the heads of the set of rules E; if E is unitary, we may use '$Head$' instead of '$Heads$' . We may compound some of these abbreviations, as for instance $Atoms(Body(r))$ whose meaning is straightforward. Each of the abbreviations may also be taken as the conjunction of the elements contained in the respective sets.

Given a 2-valued interpretation I of a logic program P, we represent by I^+ (resp. I^-) the set of its positive literals (resp. atoms whose default negations are true with respect to I). If I is 3-valued, we additionally represent by I^u the set of undefined atoms with respect to I.

The following concepts concern the structure of programs. Let P be a logic program and r, s any two rules of P. **Complete rule graph, CRG(P)**[2]: is the directed graph whose vertices are the rules of P. Two vertices representing rules r and s define an arc from r to s iff $Head(r) \subseteq Atoms(Body(s))$. **Rule depending on a rule**[2]: rule s *depends* on rule r iff there is a directed path in $CRG(P)$ from r to s. **Subprogram relevant to an atom**[3]: a rule $r \in P$ is *relevant* to an atom $a \in H_P$ iff there is a rule s such that $Head(s) = \{a\}$ and s depends on r. The set of all rules of P relevant to a is represented by $Rel_P(a)$, and is named *subprogram (of P) relevant* to a. **Loop**[4]: a set of rules R forms a *loop* (or the rules of set R are *in loop*) iff, for any two rules $r, s \in R$, r depends on s and s depends on r. We say that rule $r \in R$ is in loop through literal $L \in Body(r)$ iff there is a rule $s \in R$ such that $Head(s) = Atoms(L)$. **Rule layering**[3]: the *rule layering* (or just *layering*, for simplicity) of P is the labeling of each rule $r \in P$ with the smallest possible natural number, $layer(r)$, in the following way: for any two rules r and s, (1) if rules r, s are in loop, then $layer(r) = layer(s)$; (2) if rule r depends on rule s but rule s does not depend on rule r, then $layer(r) > layer(s)$. Every integer number T in the image of the layer function defines a *layer* of P, meaning the set of rules of P labeled with number T – we use the expression 'layer' to refer both to a set of all rules with that label, and to the label itself. We represent by $P^{\leqslant T}$ (resp. $P^{>T}$) the set of all rules of P whose layer is less than or equal to (resp. greater than) T. **T-segment of a program**: we say that $P^{\leqslant T}$ is the *T-segment* of P iff $Atoms(P^{\leqslant T}) \cap Heads(P^{>T}) = \emptyset$. We may also say 'segment T' to mean the set of rules corresponding to segment $P^{\leqslant T}$.

Let SEM be a 2-valued semantics and $SEM(P)$ the set of SEM models of a logic program P. Let also the set of atoms $ker_{SEM}(P) = \bigcap_{M \in SEM(P)} M^+$ be dubbed *semantic kernel* of P with respect to SEM (the semantic kernel is not defined if $SEM(P) = \emptyset$). The following properties concern semantics of logic programs. We say that a semantics SEM is: **Existential** iff every normal logic

[1] In this work, if nothing to the contrary is said, by 'logic program', or simply by 'program', we mean a finite set of normal ground rules.
[2] Adapted from (Pinto and Pereira 2011).
[3] Adapted from (Dix 1995b)
[4] Adapted from (Costantini 1995)

program has at least one *SEM* model; **Cautious monotonic**[5] iff for every normal logic program P, and for every set $S \subseteq ker_{SEM}(P)$, we have $ker_{SEM}(P) \subseteq ker_{SEM}(P \cup S)$; **Cut** iff for every normal logic program P, and for every set $S \subseteq ker_{SEM}(P)$, we have $ker_{SEM}(P \cup S) \subseteq ker_{SEM}(P)$; **Cumulative** iff it is cautious monotonic and cut; **Relevant** iff for every normal logic program P we have

$$\forall_{a \in H_P}(a \in ker_{SEM}(P) \Leftrightarrow a \in ker_{SEM}(Rel_P(a))) \quad (2)$$

where $Rel_P(a)$ is the subprogram of P relevant to atom a; **Global to local relevant** iff the logical entailment '$\Rightarrow$' stands in formula (2); **Local to global relevant** iff the logical entailment '$\Leftarrow$' stands in formula (2).

3 Conservative Extensions of the SM Semantics

In this section we define a family of abductive 2-valued semantics[6], the *affix stable model* family, *ASM*, whose members are conservative extensions of the *SM* semantics. For that purpose we need the concepts of *reduction system* and *MH semantics*.

3.1 Reduction System and *MH* Semantics

In (Brass et al. 2001) the authors propose a set of five operations to reduce a program (i.e., eliminate rules or literals) – positive reduction, PR, negative reduction, NR, success, S, failure, F and loop detection, L (see Appendix A for the definitions of these operations). We represent this set of operations as $\mapsto_{WFS} := \{PR, NR, S, F, L\}$. By non-deterministically applying this set of operations on a program P, we obtain the program $\widehat{P}$, the *remainder* of P, which is invariant under a further application of any of the five operations. This transformation is *terminating* and *confluent* (Brass et al. 2001). We denote the transformation of P into $\widehat{P}$ as $P \mapsto_{WFS} \widehat{P}$. We also write $\widehat{P} = remainder_{WFS}(P)$. It is shown in (Brass et al. 2001) that $WFM(P) = WFM(\widehat{P})$, where WFM stands for the well-founded model (Gelder 1993). See Appendix B for an example of the computation of the remainder of a program.

One way to obtain conservative extensions of the *SM* semantics, is to relax some operations of the reduction system $\mapsto_{WFS}$, which yields weaker reduction systems, that is, systems that erase less rules or literals than $\mapsto_{WFS}$. An example of such a semantics is the *minimal hypotheses semantics*, *MH* (Pinto and Pereira 2011), whose reduction system $\mapsto_{MH}$ is obtained from $\mapsto_{WFS}$ by replacing the negative reduction operation, *NR*, by the *layered negative reduction* operation, *LNR*, i.e., $\mapsto_{MH} := \{PR, LNR, S, F, L\}$. *LNR* is a weaker version of *NR* that instead of eliminating any rule r containing say *not* b in the body, in the presence of the fact b, as *NR* does, only eliminates rule r if this rule is not in loop through literal *not* b. We write $P \mapsto_{MH} \mathring{P}$, where $\mathring{P}$ is the *layered remainder* of P. We also write $\mathring{P} = remainder_{MH}(P)$. See Appendix C for an example of the computation of the layered remainder of a program.

3.2 ASM, ASM$^{\mathbf{h}}$ and ASM$^{\mathbf{m}}$ Families

We define *affix stable interpretation* and then use this concept to put forward the definition of *ASM* family.

[5] Adapted from (Dix 1995a; Dix 1995b).
[6] See (Denecker and Kakas 2002) for *abductive semantics*.

Definition 1

Affix Stable Interpretation. Let P be a normal logic program, *SEM* a 2-valued semantics with a corresponding reduction system $\mapsto_{SEM}$, and $X \subseteq Atoms(remainder_{SEM}(P))$. We say that I is an *affix stable interpretation* of P with respect to set X and semantics *SEM* (or simply a *SEM stable interpretation with affix X*) iff $I = WFM(P \cup X)$ and $WFM^u(P \cup X) = \emptyset$,[7] that is, I is the only stable model of the program $P \cup X$. We name X an *affix* (or *hypotheses set*) of interpretation I. We also name *assumable hypotheses* set of program P, $Hyps(P)$, the union of all possible affixes that may be considered to define the stable interpretations (we have $Hyps(P) \subseteq Atoms(remainder_{SEM}(P))$).

Definition 2

Affix Stable Model Semantics Family, ASM. A 2-valued semantics *SEM*, with a corresponding reduction system $\mapsto_{SEM}$, belongs to the *affix stable model* semantics family, *ASM*, iff, given any normal logic program P, $SEM(P)$ contains all the *SM* models of P, in case they exist, plus a subset (possibly empty) of the affix stable interpretations of P chosen by resorting to specifically enounced criteria.

Both semantics *SM* and *MH* belong to the *ASM* family. The two non-disjoint subfamilies of *ASM* next defined, ASM^h and ASM^m, will be the classes whose formal properties we study in the sequel.

Definition 3

$\mathbf{ASM^h}$ and $\mathbf{ASM^m}$ Families. A semantics $SEM \in ASM$ belongs to the ASM^h or ASM^m families iff, for any normal logic program P, the models are computed as follows:

1. For both ASM^h and ASM^m the set of *assumable hypotheses*, $Hyps(P)$, is contained in the set of atoms that appear default negated in $remainder_{SEM}(P)$[8];
2. For semantics in the class ASM^h, the affixes of the models of P are either those non empty minimal with respect to set inclusion, if $Hyps(P) \neq \emptyset$, or else the empty set if $Hyps(P) = \emptyset$. For semantics in the class ASM^m, the models in $SEM(P)$ are always minimal models.

We now refer some examples of ASM^h and ASM^m members, whose definitions can be found in Appendix D. Besides *SM*, *MH* and others, the following are ASM^h family members, referred to subsequently:[9] $\mathbf{MH^{LS}}$, $\mathbf{MH^{Loop}}$, $\mathbf{MH^{Sustainable}}$, $\mathbf{MH^{Sustainable}_{min}}$, $\mathbf{MH^{Regular}}$. Besides *SM* and others, the following are ASM^m family members, referred to subsequently: **Navy**, **Blue**, **Cyan**, **Green**.

4 Characterization of Cumulativity for the $\mathbf{ASM^h \cup ASM^m}$ Class

In this section we lay down a characterization of cumulativity for semantics *SEM* of the $ASM^h \cup ASM^m$ class, via the following theorem.

[7] Notice that $WFM^u(P \cup X)$ is the set of undefined atoms in the model $WFM(P \cup X)$.
[8] The purpose of computing the remainder of a program, is to obtain the assumable hypotheses set of the program.
[9] The first three semantics were suggested by Alexandre Pinto.

Theorem 1

Let SEM be a semantics of the $ASM^h \cup ASM^m$ class. For every program P and for every subset $S \subseteq ker_{SEM}(P)$, the following results stand: (1) SEM is cautious monotonic iff $SEM(P \cup S) \subseteq SEM(P)$; (2) SEM is cut iff $SEM(P) \subseteq SEM(P \cup S)$; (3) SEM is cumulative iff $SEM(P) = SEM(P \cup S)$ – this is a consequence of statements (1) and (2).[10]

The three items of this theorem correspond to refinements of the classical definitions of cautious monotony, cut and cumulativity (see section 2). The new definitions establish the properties by means of relations among sets of models, as opposed to the relations among sets of atoms that characterize the classical definitions.

The results stated in this theorem are advantageous to spot cumulativity failure in semantics of the $ASM^h \cup ASM^m$ class by means of counter-examples (logic programs), when compared with common procedures (e.g., (Dix 1995a; Dix 1995b)). The reason is that common procedures always need the counter-examples to fail cumulativity[11], whilst the results of theorem 1 allow us to spot failure of cumulativity even in some cases where the counter-examples used do not show any failure of this property. To make this point clear see the examples in Appendix E and Appendix F.

It should be stressed that there are 2-valued cumulative semantics to which $SEM(P) \neq SEM(P \cup S)$ for some normal logic program P and some $S \subseteq ker_{SEM}(P)$ (for an example, see the definition of the 2-valued semantics *Picky* in Appendix G). Theorem 1 states this is not the case if $SEM \in ASM^h \cup ASM^m$.

5 Defectivity, Excessiveness and Irregularity

Theorem 1 application for dismissing the cumulativity property by means of counter-examples, demands computing the set of models $SEM(P)$ of a program P, the set $ker_{SEM}(P)$, and after this it needs the computation of the sets of models $SEM(P \cup S)$, $S \in ker_{SEM}(P)$, to look for a case that eventually makes $SEM(P) = SEM(P \cup S)$ false. In this section three structural properties are defined, *defectivity, excessiveness* and *irregularity*, that will turn the dismissal of existence, relevance or cumulativity spottable by means of one model only. It will be shown that for semantics of the $ASM^h \cup ASM^m$ class, defectivity is equivalent to the failure of existence and to the failure of global to local relevance, and also entails the failure of cautious monotony, whilst excessiveness entails the failure of cut, and irregularity is equivalent to the failure of local to global relevance.

5.1 Defectivity

The rationale for the concept of defective semantics is the following: if a segment $P^{\leqslant T}$ has a SEM model M that is not contained in any whole SEM model of P, then we say the semantics SEM is *defective*, in the sense that it 'does not use' all the models of segment T in order to get whole models of P.

[10] Notice that $SEM(P)$ represents the set of all SEM models of P.

[11] The general procedure to spot the failure of cumulativity by resorting to counter-examples is as follows: compute all the SEM models of a program P; add to P subsets $S \subseteq ker_{SEM}(P)$, and compute all the models of the resulting programs $P \cup S$, drawing a conclusion about cumulativity failure only in cases where $ker_{SEM}(P) \neq ker_{SEM}(P \cup S)$.

Definition 4
Defective semantics. A 2-valued semantics *SEM* is called *defective* iff there is a normal logic program P, $SEM(P) \neq \emptyset$, a segment $P^{\leqslant T}$ of P, and a *SEM* model M of the segment $P^{\leqslant T}$, such that $SEM(P^{>T} \cup M^{+}) = \emptyset$. We also say that *SEM* is *defective* with respect to segment T of program P, and that M is a *defective* model of P with respect to segment T and semantics *SEM*.

Example 1
Program $P = \{a \leftarrow not\ b,\ b \leftarrow not\ a,\ c \leftarrow a,\ c \leftarrow not\ c\}$ may be used to show that the *SM* semantics is defective. In fact, the only *SM* model of P is $N = \{a, not\ b, c\}$ with affix $\{a\}$. Meanwhile, $P^{\leqslant 1} = \{a \leftarrow not\ b,\ b \leftarrow not\ a\}$ is a segment that has the stable model $M = \{not\ a, b\}$, and we have $SM(P^{>1} \cup \{b\}) = \emptyset$.

The next theorem shows how conclusions about existence, relevance and cumulativity may be immediately taken in the case of a defective semantics.

Theorem 2
The following relations are valid for any semantics of the $ASM^h \cup ASM^m$ class:

1. Defectivity $\Leftrightarrow$ ¬Existence $\Leftrightarrow$ ¬Global to Local Relevance;
2. Defectivity $\Rightarrow$ ¬Cautious Monotony.

The reader should notice the importance of this theorem: not only defectivity is enough to dismiss existence, relevance and cumulativity, as also these properties appear strongly related for semantics of the class $ASM^h \cup ASM^m$: if existence fails then relevance also fails (through global to local relevance failure); if existence fails then cumulativity also fails (through cautious monotony failure); if relevance fails (through global to local relevance failure), then cumulativity also fails (through cautious monotony failure). Definition 4 above shows the structural nature of defectivity, which allows the verification of the property by wisely constructing a program that satisfies it. This may turn easier the assessment of existence, relevance and cumulativity, when compared to dealing with this issue on the basis of abstract proofs. Even more, the relation between existence and defectivity stated in theorem 2, allows the failure of the existence property to be detected by resorting to counter-examples, even in some cases where the program used as counter-example has models. E.g., program P in Appendix E can be used to detect the failure of existence for *SM* semantics, in spite of the existence of stable models for program P, since it reveals the defectivity of *SM*.[12]

The results stated in theorem 2 also shed some light on the characterization of *SM* semantics with respect to the properties of existence and cumulativity. In (Dix 1995b), section 5.6, the author says that the *SM* is not cumulative and that this fact does not depend on the non existence of stable models (i.e., the author states that lack of cumulativity is not a consequence of lack of existence). Meanwhile theorem 2 above shows that *SM* is non-existential due to being defective, which in turn makes it not cautious monotonic and thus not cumulative. Thus the failure of cumulativity for the *SM* semantics case is indeed a consequence of the failure of existence for this semantics. Moreover, with respect to the *SM* semantics a stronger result relating existence

[12] It should be pointed out that there are 2-valued semantics for which the equivalence $defectivity \Leftrightarrow \neg existence$ fails, e.g., M_P^{Supp} (Apt et al. 1988) which is not defective in spite of failing the existence property – it is the case that M_P^{Supp} is not a *ASM* semantics, since it does not conservatively extend the *SM* semantics.

and cautious monotony may be enounced: these two properties show up equivalence in the sense stated in proposition 3 below. To the best of our knowledge, this connection between these two properties had not yet been stated.

Proposition 3
For the *SM* semantics the following result stands: there is a program P that shows existence failure iff there is a program P^* that shows cautious monotony failure.

5.2 Excessiveness and Irregularity

The rationale of the concept of *excessive* semantics is the following: if a normal logic program P has a model N and a layer $P^{\leqslant T}$ such that for every model $M_* \in SEM(P^{\leqslant T})$ it is the case that $N \notin SEM(P^{>T} \cup M_*^+)$, then we say that model N (and thus the semantics) is *excessive*, in the sense that it 'goes beyond' the semantics of the segment $P^{\leqslant T}$ by not being a 'consequence' of it.

Definition 5
Excessive semantics. A 2-valued semantics *SEM* is called *excessive* iff there is a logic program P, a segment $P^{\leqslant T}$, a model $M \in SEM(P^{\leqslant T})$ and a model $N \in SEM(P)$ such that:

1. $M^+ = N^+_{\leqslant T}$, where $N^+_{\leqslant T} = N^+ \cap Heads(P^{\leqslant T})$;
2. For every model $M_* \in SEM(P^{\leqslant T})$ it is the case that $N \notin SEM(P^{>T} \cup M_*^+)$;
3. There is at last a *SEM* model N^* of P, such that $N^* \in SEM(P^{>T} \cup M^+)$.

We also say that *SEM* is *excessive* with respect to segment T of program P, and that N is an *excessive model* of P with respect to segment T and semantics *SEM*.

In the excessiveness example in Appendix H it is shown that the semantics MH, MH^{LS}, MH^{Loop}, $Navy, Green$ are excessive.

The rationale of the concept of *irregularity* is as follows: given a certain whole model $N \in SEM(P)$, if the set $N^+ \cap Heads(P^{\leqslant T})$ is not a model of a segment $P^{\leqslant T}$, then we say that *SEM* is *irregular*, since N 'is not a consequence' of the semantics of segment T.

Definition 6
Irregular semantics. A 2-valued semantics *SEM* is called *irregular* iff there is a logic program P, a segment $P^{\leqslant T}$ and a *SEM* model N of P, such that for no model M of $P^{\leqslant T}$ do we have $N^+_{\leqslant T} = M^+$, where $N^+_{\leqslant T} = N^+ \cap Heads(P^{\leqslant T})$. We also say that *SEM* is *irregular* with respect to segment T of program P, and that N is an *irregular* model of P with respect to segment T and semantics *SEM*. A model that is not irregular is called *regular*, and a semantics that produces only regular models is called *regular*.[13]

The concepts of excessiveness and irregularity exhibit independence for semantics of the $ASM^h \cup ASM^m$ class, meaning there is a semantics in this class for any of the four possible cases of validity or failure of excessiveness and irregularity. As a matter of fact, it can be shown (Abrantes 2013) that *Blue* is irregular whilst not excessive (i.e., $irregularity \nRightarrow excessiveness$); it is also the case that $MH^{Regular}$ is excessive but not irregular (i.e., $excessiveness \nRightarrow irregularity$).

[13] In comparing excessiveness and irregularity, notice that a whole model can be excessive whilst containing models for all the segments of the program (i.e., be a regular model) - see the excessiveness example in Appendix H.

Also *MH* is excessive and irregular, and *Cyan* is not excessive and is not irregular.

The following result states relations between excessiveness and cut, and between irregularity and relevance.

Theorem 4
The following relations stand for any semantics of the $ASM^h \cup ASM^m$ class:

1. Excessiveness $\Rightarrow \neg$Cut;
2. Irregularity $\Leftrightarrow \neg$Local to Global Relevance.

As excessiveness and irregularity are structural properties, being thus detectable by construction of adequate programs, they facilitate, via this theorem, the dismissal of cut and relevance. For instance, this result together with the excessiveness example in Appendix H, shows that semantics *MH*, MH^{LS}, MH^{Loop}, *Navy* and *Green* are excessive, and thus not cut. Also, this result together with the irregularity example in Appendix H, shows that semantics *MH*, MH^{LS} and MH^{Loop}, *Green*, *Navy* and *Blue* are irregular, and thus not relevant. As was the case for the relation between the properties of existence and cumulativity for the *SM* semantics, our work sheds also some light on the *SM* semantics relevance failure, through the following results.

Proposition 5
Let P be a normal logic program and $M \in SM(P)$. Then M is neither excessive nor irregular.

Corollary 6
SM is (vacuously) local to global relevant.

Notice that this corollary, together with the example in Appendix E and theorem 2, let clear the cause for *SM* semantics relevance failure: *SM* fails relevance because it fails global to local relevance. This is a more precise characterization than just saying that *SM* is not relevant, as usually stated in literature (e.g., (Dix 1995b)).

If we consider the five formal properties of existence ($\exists$), global to local relevance (gl), local to global relevance (lg), cautious monotony (cm) and cut (cut), the validity or failure of each of these properties allow, in the general case, the existence of $2^5 = 32$ types of semantics. Meanwhile, the study we present in this work shows that only 12 such types of semantics may exist in the $ASM^h \cup ASM^m$ class. They are represented in table I 1 in Appendix I.

6 Final Remarks

In this paper we considered the characterization of 2-valued conservative extensions of the *SM* semantics on the properties of existence, relevance and cumulativity. This theoretical endeavor is reasonable under a point of view of prospectively assessing the behavior of such types of semantics with respect to a set of properties that are desirable, both under a computational (relevance and cumulativity) and a semantical (existence) standpoint. For that purpose we focused our study on two subsets of the here defined *ASM* class of 2-valued conservative extensions of the *SM* semantics, the non-disjoint classes ASM^h and ASM^m, whose elements maintain a degree of resemblance with already known 2-valued semantics, such as the *SM* and the *MH* semantics. As a result of this study, refined definitions of cautious monotony, cut and cumulativity were set.

This new definitions turn into an easier job the dismissal of the properties of existence, relevance and cumulativity, as shown in section 4. This study also reveals relations among these properties, unveiled by theorems 2 and 4, that allow to draw conclusions about some of them on basis of held knowledge about others. This last point builds on top of the new structural properties of defectivity, excessiveness and irregularity, which provide an analytical shortcut to assess existence, relevance and cumulativity. The approach taken in this work (characterizing families of semantics, instead of individual semantics), revealed itself advantageous also in clarifying the profile of the well known and studied *SM* semantics, via the results stated in proposition 3 and corollary 6. Our work also states a maximum of 12 types of semantics in the class $ASM^h \cup ASM^m$, with respect to the satisfaction/failure of the properties of existence ($\exists$), global to local relevance (gl), local to global relevance (lg), cautious monotony (cm) and cut (cut).

Finally, the structural approach put forward in this paper has the potential of being used with semantics other than 2-valued ones, and with other strong and weak properties besides existence, relevance or cumulativity.[14]

Acknowledgments

We thank Alexandre Pinto for some important debates on conservative extensions of the *SM* semantics. The work on this paper has been partially supported by Fundação para a Ciência e Tecnologia and Instituto Politécnico de Bragança grant PROTEC : SFHR/49747/2009.

References

ABRANTES, M. 2013. Revision based total semantics for extended normal logic programs. Ph.D. thesis, Universidade Nova de Lisboa.

APT, K., BLAIR, H. A., AND WALKER, A. 1988. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, J. Minker, Ed. Morgan Kaufmann, Los Altos, CA, 89–142.

BRASS, S., DIX, J., FREITAG, B., AND ZUKOWSKI, U. 2001. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 497–538.

COSTANTINI, S. 1995. Contributions to the stable model semantics of logic programs with negation. *Theoretical Computer Science 149,* 2 (2 Oct.), 231–255.

DENECKER, M. AND KAKAS, A. C. 2002. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond'02*. 402–436.

DIX, J. 1995a. A classification theory of semantics of normal logic programs: I. strong properties. *Fundam. Inform 22,* 3, 227–255.

DIX, J. 1995b. A classification theory of semantics of normal logic programs: II. weak properties. *Fundam. Inform 22,* 3, 257–288.

GELDER, A. V. 1993. The alternating fixpoint of logic programs with negation. *J. of Comp. System Sciences 47,* 1, 185–221.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*. MIT Press, 1070–1080.

PINTO, A. M. AND PEREIRA, L. M. 2011. Each normal logic program has a 2-valued minimal hypotheses semantics. *INAP 2011, CoRR abs/1108.5766*.

SWIFT, T. 1999. Tabling for non-monotonic programming. *Ann. Math. Artif. Intell 25,* 3-4, 201–240.

[14] The terms *strong* and *weak* applied to formal properties, are here adopted after (Dix 1995a; Dix 1995b).

Appendix A Reduction Operations

In the definitions below, P_1 and P_2 are two ground logic programs.

1. **Positive reduction, PR.** Program P_2 results from P_1 by *positive reduction* iff there is a rule $r \in P_1$ and a default literal *not* $b \in Body(r)$ such that $b \notin Heads(P_1)$, and $P_2 = (P_1 \setminus \{r\}) \cup \{Head(r) \leftarrow (Body(r) \setminus \{not\ b\})\}$.
2. **Negative reduction, NR.** Program P_2 results from P_1 by *negative reduction* iff there is a rule $r \in P_1$ and a default literal *not* $b \in Body(r)$ such that $b \in Facts(P_1)$, and $P_2 = P_1 \setminus \{r\}$.
3. **Success, S.** Program P_2 results from P_1 by *success* iff there is a rule $r \in P_1$ and a fact $b \in Facts(P_1)$ such that $b \in Body(r)$, and $P_2 = (P_1 \setminus \{r\}) \cup \{Head(r) \leftarrow (Body(r) \setminus \{b\})\}$.
4. **Failure, F.** Program P_2 results from P_1 by *failure* iff there is a rule $r \in P_1$ and a positive literal $b \in Body(r)$ such that $b \notin Heads(P_1)$, and $P_2 = P_1 \setminus \{r\}$.
5. **Loop Detection, L.** Program P_2 results from P_1 by *loop detection* iff there is a set $\mathscr{A}$ of ground atoms such that:

   (a) For each rule $r \in P_1$, if $Head(r) \in \mathscr{A}$, then $Body(r) \cap \mathscr{A} \neq \emptyset$;
   (b) $P_2 := \{r \in P_1 | Body(r) \cap \mathscr{A} = \emptyset\}$.

Appendix B Remainder Computation Example

Let P be the set of all rules below. The remainder $\widehat{P}$ is the non shadowed part of the program. The labels (i)–(v) indicate the operations used in the corresponding reductions: (i) PR, (ii) NR, (iii) S, (iv) F, (v) L.

$\{a \leftarrow \boxed{not\ f}$ (i), $\boxed{e \leftarrow d}$ (v), $a \leftarrow \boxed{not\ b}$ (i), $\boxed{d \leftarrow e}$ (v), $\boxed{b \leftarrow not\ a}$ (ii), $c \leftarrow \boxed{a}$ (iii), $\boxed{d \leftarrow f}$ (iv)$\}$

Appendix C Minimal Hypotheses Models Computation

Let P be the set of rules below, which is equal to the program in Appendix B. The layered remainder $\mathring{P}$ is the non-shadowed part of the program.

$$a \leftarrow \boxed{not\ f} \qquad d \leftarrow f$$
$$a \leftarrow not\ b \qquad e \leftarrow d$$
$$b \leftarrow not\ a \qquad d \leftarrow e$$
$$c \leftarrow \boxed{a}$$

Notice that rule $b \leftarrow not\ a$ is no longer eliminated by the fact a, since this rule and rule $a \leftarrow not\ b$ are in loop, and in the case of rule $b \leftarrow not\ a$ the loop is through the literal *not* a.

The MH models of a program P are computed as follows: (1) Take as assumable hypotheses set, $Hyps(P)$, the set of all atoms that appear default negated in $\mathring{P}$; in the case of the previous program we have $Hyps(P) = \{a, b\}$; (2) Form all programs $P \cup H$, for all possible subsets $H \subseteq Hyps$, $H \neq \emptyset$ (if $Hyps = \emptyset$, then $H = \emptyset$ is the only set to consider); take all the interpretations for which $WFM(P \cup H)$ is a total model (meaning a model that has no undefined literals); H is the *hypotheses set* of the interpretation $WFM(P \cup H)$; (3) Take all the interpretations obtained in the previous point, and chose as MH models the ones that have minimal H sets with respect

to set inclusion. The MH models of program P in the example above, and the corresponding hypotheses sets, are

$$M_1 = \{a, not\ b, c, not\ d, not\ e, not\ f\} \qquad H = \{a\}$$
$$M_2 = \{a, b, c, not\ d, not\ e, not\ f\} \qquad H = \{b\}.$$

Notice that M_1 is the only SM model of P. The MH reduction system keeps some loops intact, which are used as choice devices for generating MH models, allowing us to have $MH(P) \supseteq SM(P)$. The sets H considered may be taken as abductive explanations (Denecker and Kakas 2002) for the corresponding models.

Appendix D Definitions of some Elements of ASM^h and ASM^m Families

Besides SM and others, the following are ASM^h family members.

MH$^{\mathbf{LS}}$: the reduction system is obtained by replacing the success operation in $\mapsto_{MH}$ by the *layered success* operation;[15] MH^{LS} models are computed as in the MH case.

MH$^{\mathbf{Loop}}$: the reduction system is $\mapsto_{MH}$; the assumable hypotheses set of a program P, $Hyps(P)$, is formed by the atoms that appear default negated in literals involved in loops in the layered remainder $\mathring{P}$; MH^{Loop} models are computed as in the MH case.

MH$^{\mathbf{Sustainable}}$: the reduction system is $\mapsto_{MH}$; $MH^{Sustainable}$ models are computed as in the MH case with the following additional condition: if H is a set of hypotheses of a $MH^{Sustainable}$ model M of P, then

$$\forall_{h \in H}\ [(H \setminus \{h\}) \neq \emptyset \Rightarrow h \in WFM^u(P \cup (H \setminus \{h\}))],$$

that is, no single hypothesis may be defined in the well-founded model if we join to P all the other remaining hypotheses.

MH$^{\mathbf{Sustainable}}_{\mathbf{min}}$: the reduction system is $\mapsto_{MH}$; $MH^{Sustainable}_{min}(P)$ retrieves the minimal models contained in $MH^{Sustainable}(P)$ for any normal logic program P. $MH^{Sustainable}_{min}$ also belongs to the ASM^m family, due to the minimality of its models.

MH$^{\mathbf{Regular}}$: the reduction system is $\mapsto_{MH}$; retrieves the same models as MH, except for the irregular ones (cf. Definition 6).

Besides SM, $MH^{Sustainable}_{min}$ (defined above in this appendix) and others, the following are ASM^m family members.

Navy: the reduction system is $\mapsto_{WFS}$. Given a normal logic program P, $Navy(P)$ contains all the minimal models of $\widehat{P}$.[16]

Blue: the reduction system is $\mapsto_{WFS}$. Given a normal logic program P, $Blue(P)$ contains all the models in $Navy(P \cup K)$ where K is obtained after terminating the following algorithm:[17]

(a) Compute $K = kernel_{Navy}(\widehat{P})$;

(b) Compute $K' = kernel_{Navy}(P \cup K)$;

(c) If $K \neq K'$, then let P be the new designation of program $P \cup K'$; go to step (a).

Repeat steps (a) – (c) until $K \neq K'$ comes false in (c).

[15] *Layered success* is an operation proposed by Alexandre Pinto. It weakens the operation of success by allowing it to be performed only in the cases where the rule r, whose body contains the positive literal b to be erased, is not involved in a loop through literal b.

[16] See definition of $\widehat{P}$ in subsection 3.1.

[17] This algorithm is presented in (Dix 1995a).

Cyan: the reduction system is $\mapsto_{WFS}$. Given a normal logic program P, compute $Cyan(P)$ through the steps of *Blue* computation, but taking only the *regular* models (cf. Definition 6) to compute the semantic kernel at steps (a) and (b).
Green: the reduction system is $\mapsto_{WFS}$. Given a normal logic program P, $Green(P)$ contains all the minimal models of $\widehat{P}$ that have the smallest (with respect to set inclusion) subsets of classically unsupported atoms.[18]

Appendix E Example of Cumulativity Failure Detection

The following 1-layer program P is a counter-example for showing, using theorem 1, that *SM* semantics is not cumulative, due to being not cautious monotonic (program P does not allow us to spot the failure of any of these properties by means of the usual definitions of cumulativity and cautious monotony presented in section 2) .

$$\begin{array}{lll} a \leftarrow not\ b, not\ s & d \leftarrow b & d \leftarrow a \\ b \leftarrow not\ a, not\ c & d \leftarrow not\ d & c \leftarrow k \\ c \leftarrow not\ b, not\ k & k \leftarrow a, d & s \leftarrow not\ a, d \end{array}$$

In fact, the *SM* models of P are $\{a,d,c,k\}$ and $\{b,d,s\}$, and thus $ker_{SM}(P) = \{d\}$. Now $P \cup \{d\}$ has the stable models $\{a,d,c,k\}$, $\{b,d,s\}$ and $\{c,d,s\}$, and thus $ker_{SM}(P) = ker_{SM}(P \cup \{d\}) = \{d\}$. Hence no negative conclusion can be afforded about cumulativity, by means of the usual definition of this property. Meanwhile, by using the statement (3) of theorem 1 it is straightforward to conclude that *SM* semantics does not enjoy the property of cumulativity, because $SM(P) \neq SM(P \cup \{d\})$. Moreover, statement (1) of the theorem tells us, via this example, that *SM* semantics is not cautious monotonic because $SM(P \cup \{d\}) \nsubseteq SM(P)$.

Appendix F Proof of Cautious Monotony and Cut Failure

The following 1-layer program $P = \mathring{P}$ is a counter-example for showing, using theorem 1, that none of the semantics MH, MH^{LS}, MH^{Loop}, $MH^{Sustainable}$ and $MH^{Regular}$ is either cautious monotonic or cut (program P does not allow us to spot the failure of any of these properties by means of the usual definitions of cautious monotony and cut presented in section 2) .

$$\begin{array}{ll} u \leftarrow b & a \leftarrow not\ b \\ u \leftarrow c & b \leftarrow not\ c \\ t \leftarrow a & c \leftarrow h, u \\ t \leftarrow h & h \leftarrow not\ h, not\ t \end{array}$$

Let *SEM* represent any of the above semantics. The minimal hypotheses models are the same with respect to any of the four semantics (models are represented considering only positive literals): $\{c,u,a,t\}$ with affix $\{c\}$; $\{b,h,u,c,t\}$ with affix $\{b,h\}$; $\{t,b,u\}$ with affix $\{t\}$. Thus $ker_{SEM}(P) = \{t,u\}$. Now it is the case that the remainder of $P \cup \{u\}$ is the same for any of these

[18] Given a logic program P, a model M of P and an atom $b \in M$, we say that b is classically unsupported by M iff there is no rule $r \in P$ such that $Head(r) = \{b\}$ and all literals in $Body(r)$ are true with respect to M.

semantics:

$$\begin{array}{lll} u \leftarrow b & a \leftarrow not\ b & \\ u \leftarrow c & b \leftarrow not\ c & \\ t \leftarrow a & c \leftarrow h & \\ t \leftarrow h & h \leftarrow not\ h, not\ t & u \leftarrow \end{array}$$

(as a matter of fact, the remainder for the MH^{LS} has the rule $c \leftarrow h,u$ instead of $c \leftarrow h$; but this does not change the sequel of this reasoning). The minimal hypotheses models of $P \cup \{u\}$ are the same with respect to any of the four semantics (models are represented considering only positive literals): $\{c,u,a,t\}$ with affix $\{c\}$; $\{h,u,c,t,a\}$ with affix $\{h\}$; $\{t,b,u\}$ with affix $\{t\}$. Thus $ker_{SEM}(P \cup \{u\}) = \{t,u\} = ker_{SEM}(P)$, and no conclusions about cumulativity can be drawn by means of the usual general procedures. Meanwhile, $M = \{h,u,c,t,a\}$, with affix $\{h\}$, is a minimal affix model of $P \cup \{u\}$ but is not a minimal affix model of P, which by point (1) of theorem 1 renders any of these semantics not cautious monotonic. Also $N = \{b,h,u,c,t\}$, with affix $\{b,h\}$, is a minimal affix model of P, but not a minimal affix model of $P \cup \{u\}$, which by point (2) of theorem 1 renders any of these semantics as not cut.

Appendix G Picky, a Special 2-valued Cumulative Semantics

The semantics *Picky* is defined as follows: for any normal logic program P (1) if $SM(P) = \emptyset$, then $Picky(P) = \emptyset$; (2) if $SM(P) \neq \emptyset$, then (2a) $Picky(P) = SM(P)$ iff $ker_{SM}(P) = ker_{SM}(P \cup S)$, for every $S \subseteq ker_{SM}(P)$; (2b) otherwise $Picky(P) = \emptyset$. This semantics is cumulative, by definition, but it is not always the case that $Picky(P) = Picky(P \cup S)$, $S \subseteq ker_{SM}(P)$: for program P of the example in Appendix E, we have $Picky(P) = \{\{a,d,c,k\},\{b,d,s\}\}$ and $Picky(P \cup \{d\}) = \{\{a,d,c,k\},\{b,d,s\},\{c,d,s\}\}$, which means, by theorem 1, that *Picky* is not cumulative. Notice that *Picky* is not a *ASM* semantics, because it does not conservatively extend the *SM* semantics: for program P in the referred example, we have $SM(P) \neq \emptyset$ and $Picky(P) = \emptyset$.

Appendix H Excessiveness and Irregularity

Excessiveness. The following program P shows that semantics MH, MH^{LS}, MH^{Loop}, *Navy* and *Green* are excessive (the dashed lines divide the program into layers; top layer is layer 1, bottom layer is layer 4),

$$\begin{array}{l} a \leftarrow not\ b \\ b \leftarrow not\ a \\ -----1 \\ u \leftarrow a \\ u \leftarrow b \\ -----2 \\ p \leftarrow not\ p, not\ u \\ -----3 \\ q \leftarrow not\ q, not\ p \\ -----4. \end{array}$$

Let *SEM* represent any of these semantics. It is the case that $N = \{a, u, p, not\ b, not\ q\}$ with affix $\{a, p\}$, is a model of P under any of the referred semantics, and for no *SEM* model $M_* \in SEM(P^{\leqslant 2})$, where $SEM(P^{\leqslant 2}) = \{\{a, not\ b, u\}, \{not\ a, b, u\}\}$, do we have $N \in SEM(P^{>2} \cup M_*^+)$, because atom $u \in M_*^+$ eliminates the rule in layer 3 via layered negative reduction operation (which has here the same effect as negative reduction operation), and thus p belongs to no model in $SEM(P^{>2} \cup M_*^+)$.

Irregularity. Program P below shows that the semantics MH, MH^{LS} and MH^{Loop}, *Green*, *Navy* and *Blue* are all irregular.

$$\begin{aligned} &a \leftarrow not\ b \\ &b \leftarrow not\ a \\ &-----1 \\ &p \leftarrow not\ p, not\ a \\ &q \leftarrow not\ q, not\ b \end{aligned}$$

In fact, all these semantics admit the model $N = \{a, b, not\ p, not\ q\}$. The models of segment $P^{\leqslant 1}$ are $\{a, not\ b\}$ and $\{b, not\ a\}$, none of whose positive sets of atoms equals $N^+_{\leqslant T} = \{a, b\}$. As *Blue* is not excessive, this example shows *irregularity* $\nRightarrow$ *excessiveness*.

Appendix I The 12 possible types of ASM^h and ASM^m semantics

In table I 1 below '0' flags the failure of a property and '1' means the property is verified.

Table I 1. *The* 12 *possible types of* ASM^h *and* ASM^m *semantics*

	$\exists$	*gl*	*lg*	*cm*	*cut*
1	0	0	0	0	0
2	0	0	0	0	1
3	0	0	1	0	0
4	0	0	1	0	1
5	1	1	0	0	0
6	1	1	0	0	1
7	1	1	0	1	0
8	1	1	0	1	1
9	1	1	1	0	0
10	1	1	1	0	1
11	1	1	1	1	0
12	1	1	1	1	1

The 20 missing types of semantics correspond to cases where ($\exists = 0$ and $gl = 1$), or ($\exists = 1$ and $gl = 0$), or ($\exists = 0$ and $cm = 1$), each of these cases going against the statement of theorem 2. The correspondence of the $ASM^h \cup ASM^m$ class semantics presented in this text and

the entries in table I 1 is as follows: 1. $MH^{sustainable}, MH^{Sustainable}_{min}$ 2. $--$ 3. $--$ 4. *SM* 5. MH, MH^{LS}, MH^{Loop}, *Green* 6. $--$ 7. *Navy* 8. *Blue* 9. $MH^{Regular}$ 10. $--$ 11. $--$ 12. *Cyan*. Whether semantics of the $ASM^h \cup ASM^m$ class exist for the types marked with '$--$', may be envisaged as an open issue.

A Simple and Efficient Lock-Free Hash Trie Design for Concurrent Tabling

MIGUEL AREIAS and RICARDO ROCHA
CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
(e-mail: {miguel-areias,ricroc}@dcc.fc.up.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

A critical component in the implementation of a concurrent tabling system is the design of the table space. One of the most successful proposals for representing tables is based on a two-level trie data structure, where one trie level stores the tabled subgoal calls and the other stores the computed answers. In this work, we present a simple and efficient lock-free design where both levels of the tries can be shared among threads in a concurrent environment. To implement lock-freedom we took advantage of the CAS atomic instruction that nowadays can be widely found on many common architectures. CAS reduces the granularity of the synchronization when threads access concurrent areas, but still suffers from low-level problems such as false sharing or cache memory side-effects. In order to be as effective as possible in the concurrent search and insert operations over the table space data structures, we based our design on a hash trie data structure in such a way that it minimizes potential low-level synchronization problems by dispersing as much as possible the concurrent areas. Experimental results in the Yap Prolog system show that our new lock-free hash trie design can effectively reduce the execution time and scale better than previous designs.

KEYWORDS: Tabling, Concurrency, Hash Tries, Lock-Freedom, Performance.

1 Introduction

Tabling (Chen and Warren 1996) is a recognized and powerful implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. Multithreading in Prolog is the ability to perform concurrent computations, in which each thread runs independently but shares the program clauses (Moura 2008). Despite the availability of both multithreading and tabling in some Prolog systems, the efficient implementation of these two features, such that they work together, implies a complex redesign of several components of the underlying engine. XSB was the first Prolog system to combine tabling with multithreading (Marques and Swift 2008). In more recent work (Areias and Rocha 2012b), we have proposed an alternative view to XSB's approach, where each thread views its tables as private but, at the engine level, we use a common table space, i.e., from the thread point of view, the tables are private but, from the implementation point of view, tables are shared among all threads.

A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. To deal with

concurrent table accesses, our initial approach, implemented on top of the Yap Prolog system (Santos Costa et al. 2012), was to use lock-based data structures (Areias and Rocha 2012b). Yap implements the table space using a two-level trie data structure, where one trie level stores the tabled subgoal calls and the other stores the computed answers. More recently (Areias and Rocha 2014), we presented a sophisticated lock-free design to deal with concurrency in both trie levels. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. To implement lock-freedom we took advantage of the *CAS* atomic instruction that nowadays can be widely found on many common architectures. The *CAS* reduces the granularity of the synchronization when threads access concurrent areas, but still suffers from contention points where synchronized operations are done on the same memory locations, leading to low-level problems such as false sharing or cache memory ping pong side-effects.

In this work, we go one step further and we present a simpler and efficient lock-free design based on hash tries that minimizes these problems by dispersing as much as possible the concurrent areas. Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash tables (Bagwell 2001). An essential property of the trie data structure is that common prefixes are stored only once (Fredkin 1962), which in the context of hash tables allows us to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions. The aim of our proposal is to be as effective as possible in the search and insert operations, by exploiting the full potentiality of lock-freedom on those operations, and in such a way that it minimizes the bottlenecks and performance problems mentioned above without introducing significant overheads for sequential execution.

Several approaches do exist in the literature for the implementation of lock-free hash tables, such as Shalev and Shavit split-ordered lists (Shalev and Shavit 2006), Triplett et al. relativistic hash tables (Triplett et al. 2011) or Prokopec et al. CTries (Prokopec et al. 2012). However, to the best of our knowledge, none of them is specifically aimed for an environment with the characteristics of our tabling framework that does not requires concurrent deletion support. In general, a tabled program is deterministic, finite and only executes search and insert operations over the table space data structures. In Yap Prolog, space is recovered when the last running thread abolish a table. Since no delete operations are performed, the size of the tables always grows monotonically during an evaluation. Initial experiments, on top of a 32 core AMD machine, show that our new lock-free hash-trie design can effectively reduce the execution time and scale better than all the previously implemented lock-based and lock-free strategies.

2 Background

A trie is a tree structure where each different path corresponds to a term described by the tokens labeling the nodes traversed. For example, the tokenized form of the term $p(1, f(X))$ is the sequence of 4 tokens $p/2$, 1, $f/1$ and VAR_0, where each variable is represented as a distinct VAR_i constant. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term $p(1, a)$. Since the main functor and the first argument, tokens $p/2$ and 1, are common to

both terms, only one additional node will be required to fully represent this second term in the trie. Figure 1 shows Yap's trie structure that represents both terms.

Whenever the chain of child nodes for a common parent node becomes larger than a predefined threshold value, a hash mechanism is used to provide direct node access and therefore optimize the search. To deal with hash collisions, all previous Yap's approaches implemented a dynamic resizing of the hash tables by doubling the size of the bucket entries in the hash. Our initial approach to support concurrent tabling was lock-based, which required synchronization between threads when performing the hash expansion procedure (Areias and Rocha 2012b). More recently, we proposed a lock-free design for concurrent table accesses that avoids thread synchronization, even when threads are expanding the hash tables (Areias and Rocha 2014). In this work, we present a simpler and efficient lock-free design based on hash tries to implement the hash mechanism inside the subgoal and answer tries.

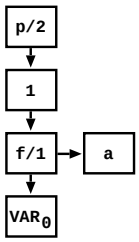


Fig. 1. Trie example

To put our proposal in perspective, Fig. 2 shows a schematic representation of the trie hierarchical levels we are proposing to implement Yap's table space. For each predicate being tabled, Yap implements tables using two levels of tries together with the table entry and subgoal frame auxiliary data structures (Rocha et al. 2005). The first level, the subgoal trie, stores the tabled subgoal calls and the second level, the answer trie, stores the answers for a given call. Then, for each particular subgoal/answer trie, we have as many trie levels as the number of parent/child relationships (for example, the trie in Fig. 1 has 4 trie levels). Finally, to implement hashing inside the subgoal/answer tries, we use another trie-based data structure, the hash trie, which is the focus of the current work. In a nutshell, a hash trie is composed by internal hash arrays and leaf nodes. The leaf nodes store *key* values and the internal hash arrays implement a hierarchy of hash levels of fixed size 2^w. To map a *key* into this hierarchy, we first compute the hash value h for *key* and then use chunks of w bits from h to index the entry in the appropriate hash level. Hash collisions are solved by simply walking down the tree as we consume successive chunks of w bits from the hash value h.

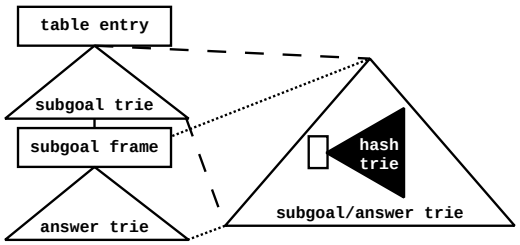


Fig. 2. Trie hierarchical levels overview

3 Our Proposal By Example

We will use three examples to illustrate the different configurations that the hash trie assumes for one, two and three levels (for more levels, the same idea applies). We begin with Fig. 3 showing a small example that illustrates how the concurrent insertion of nodes is done in a hash level.

Figure 3(a) shows the initial configuration for a hash level. Each hash level H_i is formed by a bucket array of 2^w entries and by a backward reference to the previous level (represented as *Prev* in the figures that follow). For the root level, the backward reference is *Nil*. In Fig. 3(a), E_k represents a particular bucket entry of the hash level.

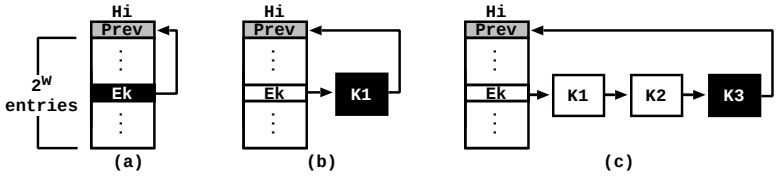


Fig. 3. Insert procedure in a hash level

E_k and the remaining entries are all initialized with a reference to the current level H_i. During execution, each bucket entry stores either a reference to a hash level or a reference to a separate chaining mechanism, using a chain of internal nodes, that deals with the hash collisions for that entry. Each internal node holds a *key* value and a reference to the next-on-chain internal node. Figure 3(b) shows the hash configuration after the insertion of node K_1 on the bucket entry E_k and Fig. 3(c) shows the hash configuration after the insertion of nodes K_2 and K_3 also in E_k. Note that the insertion of new nodes is done at the end of the chain and that any new node being inserted closes the chain by referencing back the current level.

During execution, the different memory locations that form a hash trie are considered to be in one of the following states: black, white or gray. A black state represents a memory location that can be updated by any thread (concurrently). A white state represents a memory location that can be updated only by one (specific) thread (not concurrently). A gray state represents a memory location used only for reading purposes. As the hash trie evolves during time, a memory location can change between black and white states until reaching the gray state, where it is no further updated.

The initial state for E_k is black, because it represents the next synchronization point for the insertion of new nodes. After the insertion of node K_1, E_k moves to the white state and K_1 becomes the next synchronization point for the insertion of new nodes. To guarantee the property of lock-freedom, all updates to black states are done using CAS operations. Since we are using single word *CAS* operations, when inserting a new node in the chain, first we set the node with the reference to the current level and only then the *CAS* operation is executed to insert the new node in the chain.

When the number of nodes in a chain exceeds a *MAX_NODES* threshold value, then the corresponding bucket entry is expanded with a new hash level and the nodes in the chain are remapped in the new level. Thus, instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w. To map our key values into this hierarchy, we use chunks of w bits from the hash values computed by our hash function. For example, consider a *key* value and the corresponding hash value h. For each hash level H_i, we use the $w * i$ least significant bits of h to index the entry in the appropriate bucket array, i.e., we consume h one chunk at a time as we walk down the hash levels. Starting from the configuration in Fig. 3(c), Fig. 4 illustrates the expansion mechanism with a second level hash H_{i+1} for the bucket entry E_k.

The expansion procedure is activated whenever a thread T meets the following two conditions: (i) the key at hand was not found in the chain and (ii) the number of nodes in the chain is equal to the threshold value (in what follows, we consider a threshold value of three nodes). In such case, T starts by pre-allocating a second level hash H_{i+1}, with all entries referring the respective level (Fig. 4(a)). At this stage, the bucket entries in

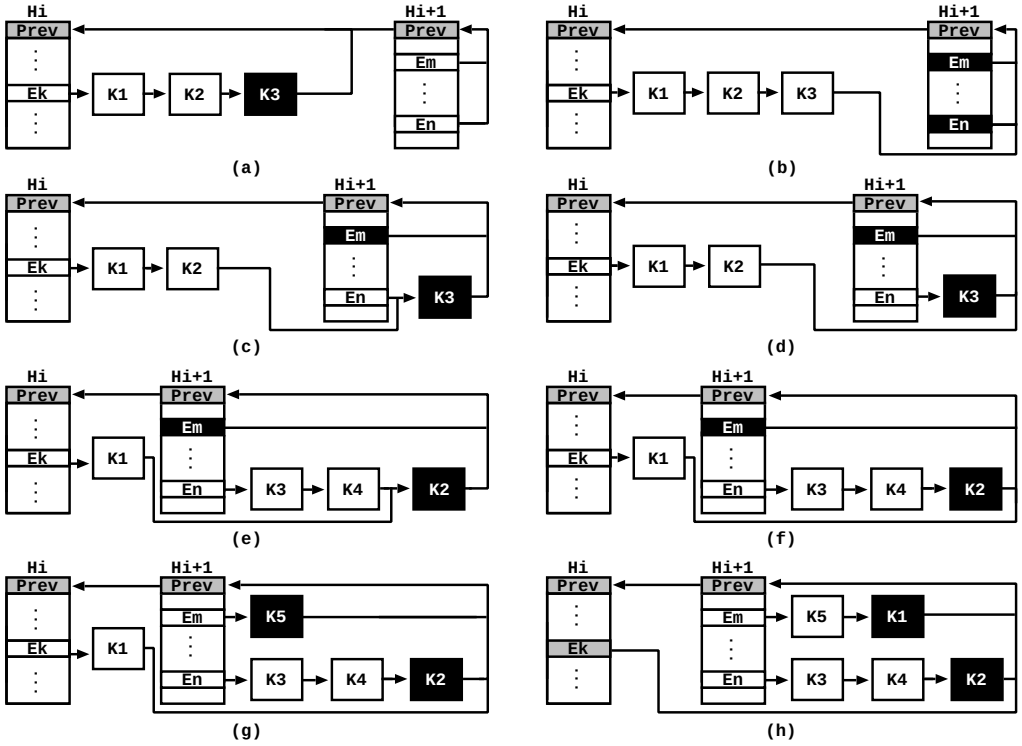


Fig. 4. Expanding a bucket entry with a second level hash

H_{i+1} can be considered white memory locations, because the hash level is still not visible for the other threads. The new hash level is then used to implement a synchronization point with the last node on the chain (node K_3 in the figure) that will correspond to a successful *CAS* operation trying to update H_i to H_{i+1} (Fig. 4(b)). From this point on, the insertion of new nodes on E_k will be done starting from the new hash level H_{i+1}.

If the *CAS* operation fails, that means that another thread has gained access to the expansion procedure and, in such case, T aborts its expansion procedure. Otherwise, T starts the remapping process of placing the internal nodes K_1, K_2 and K_3 in the correct bucket entries in the new level. Figures 4(c) to 4(h) show the remapping sequence in detail. For simplicity of illustration, we will consider only the entries E_m and E_n on level H_{i+1} and assume that K_1, K_2 and K_3 will be remapped to entries E_m, E_n and E_n, respectively. In order to ensure lock-free synchronization, we need to guarantee that, at any time, all threads are able to read all the available nodes and insert new nodes without any delay from the remapping process. To guarantee both properties, the remapping process is thus done in reverse order, starting from the last node on the chain, initially K_3.

Figure 4(c) then shows the hash trie configuration after the successful *CAS* operation that adjusted node K_3 to entry E_n. After this step, E_n moves to the white state and K_3 becomes the next synchronization point for the insertion of new nodes on E_n. Note that the initial chain for E_k has not been affected yet, since K_2 still refers to K_3. Next, on Fig. 4(d), the chain is broken and K_2 is updated to refer to the second level hash H_{i+1}. The process then repeats for K_2 (the new last node on the chain for E_k). First, K_2 is

remapped to entry E_n (Fig. 4(e)) and then it is removed from the original chain, meaning that the previous node K_1 is updated to refer to H_{i+1} (Fig. 4(f)). Finally, the same idea applies to the last node K_1. Here, K_1 is also remapped to a bucket entry on H_{i+1} (E_m in the figure) and then removed from the original chain, meaning in this case that the bucket entry E_k itself becomes a reference to the second level hash H_{i+1} (Fig. 4(h)). From now on, E_K is also a gray memory location since it will be no further updated.

Concurrently with the remapping process, other threads can be inserting nodes in the same bucket entries for the new level. This is shown in Fig. 4(e), where a new node K_4 is inserted before K_2 in E_n and, in Fig. 4(g), where a node K_5 is inserted before K_1 in E_m. As mentioned before, lock-freedom is ensured by the use of CAS operations when updating black state memory locations.

To ensure the correctness of the remapping process, we also need to guarantee that the nodes being remapped are not missed by any other thread traversing the hash trie. Please remember that any chaining of nodes is closed by the last node referencing back the hash level for the node. Thus, if when traversing a chain of nodes, a thread U ends in a second level hash H_{i+1} different from the initial one H_i, this means that U has started from a bucket entry E_k being remapped, which includes the possibility that some nodes initially on E_k were not seem by U. To guarantee that no node is missed, U simply needs to restart its traversal from H_{i+1}.

We conclude the description of our proposal with a last example that shows a expansion procedure involving three hash levels. Starting from the configuration on Fig. 4(b), Fig. 5 assumes a scenario where a set of nodes (K_4, K_5, K_6 and K_7 in the figure) are inserted in the bucket entries E_m and E_n before the beginning of the remapping process of nodes K_1, K_2 and K_3. Again, we will consider only the entries E_m and E_n on level H_{i+1} and assume that K_1, K_2 and K_3 will be remapped to entries E_m, E_n and E_n, respectively.

Figure 5(a) shows the situation where K_3 is scheduled to be remapped to entry E_n on level H_{i+1} but, since the number of nodes on E_n is equal to the threshold value, a preliminary expansion procedure for E_n should be done, which leads to the pre-allocation of a third level hash H_{i+2}. Figure 5(b) then shows the hash trie configuration after the remapping of the nodes on E_n to the level H_{i+2}. Please note that E_n became a gray state memory location since it is now referring the third level hash H_{i+2}, which means that any operation scheduled to E_n should be rescheduled to H_{i+2}. This is the case shown in Fig. 5(c), where K_3 and K_2 were both rescheduled to entry E_z on H_{i+2}. Despite this third level remapping, the chaining reference of the last node on the chain (for example, K_1 in Fig. 5(c)) is still made to refer to the second level hash H_{i+1}. To conclude the example, Fig. 5(d) shows the configuration at the end of the remapping process. Here, K_1 is remapped to the bucket entry E_m on H_{i+1} and removed from the initial chain, meaning that E_k itself becomes a reference to H_{i+1} and moves to a gray state.

For each configuration shown, the reader is encourage to verify that, at any moment, all threads are able to access all available nodes. Consider, for example, the configuration shown in Fig. 5(c) and a thread entering on level H_i searching for a node with the key K_7. The thread would begin by hashing the key K_7 on level H_i and obtain the bucket entry E_k. Then, it would follow the chain of nodes (K_1 in this case) and reach level H_{i+1}. At level H_{i+1}, it would hash again the key K_7, obtain the bucket entry E_n and follow the reference to level H_{i+2}. Finally, it would hash one more time the key K_7, now for level H_{i+2}, obtain the entry E_x and follow the chain until reaching node K_7.

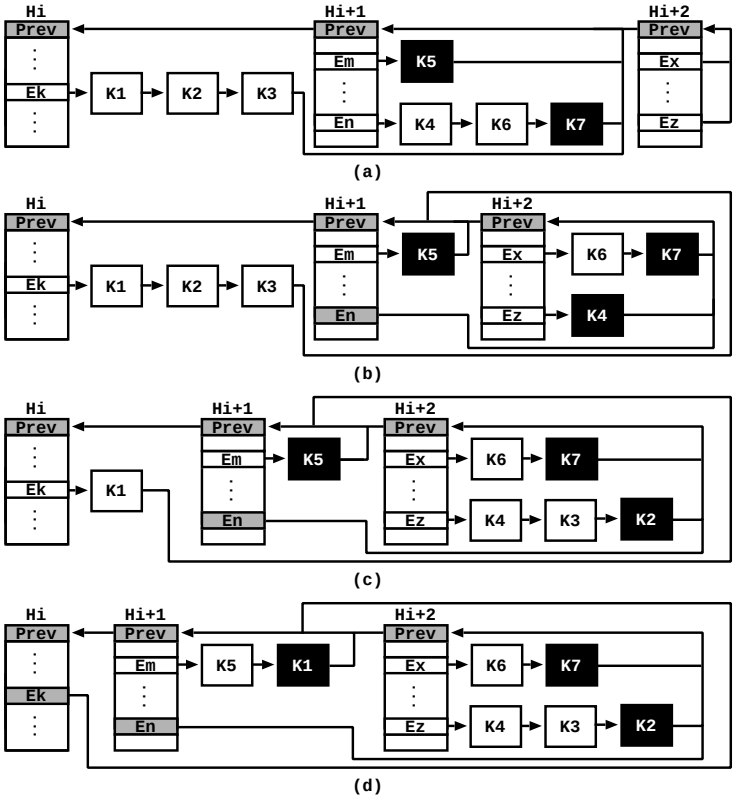


Fig. 5. Remapping nodes on a third level hash

We argue that a key design decision in our approach is thus the combination of hash tries with the use of a separate chaining (with a threshold value) to resolve hash collisions (the original hash trie design expands a bucket entry when a second key is mapped to it). Also, to ensure that nodes being remapped are not missed by any other thread traversing the hash trie, any chaining of nodes is closed by the last node referencing back the hash level for the node, which allows to detect the situations where a node changes level. This is very important because it allows to implement a clean design to resolve hash collisions by simply moving nodes between the levels. In our design, updates and expansions of the hash levels are never done by using replacement of data structures (i.e., create a new one to replace the old one), which also avoids the complex mechanisms necessary to support the recovering of the unused data structures. Another important design decision which minimizes the low-level synchronization problems leading to false sharing or cache memory side-effects, is the insertion of nodes done at the end of the separate chain. Inserting nodes at the end of the chain allows for dispersing as much as possible the memory locations being updated concurrently (the last node is always different) and, more importantly, reduces the updates for the memory locations accessed more frequently, like the bucket entries for the hash levels (each bucket entry is at most only updated twice).

4 Performance Evaluation

To put our results in perspective, we compared our new lock-free hash trie design (LFHT) against all the previously implemented Yap's lock-based and lock-free strategies for concurrent tabling. For the sake of simplicity, here we will only consider Yap's best lock-based strategy (LB) and the lock-free design (LF) presented in (Areias and Rocha 2014). For benchmarking, we used the set of tabling benchmarks from (Areias and Rocha 2012a) which includes 19 different programs in total. We choose these benchmarks because they have characteristics that cover a wide number of scenarios in terms of trie usage. The benchmarks create different trie configurations with lower and higher number of nodes and depths, and also have different demands in terms of trie traversing.

Since the system's performance is highly dependent on the available concurrency that a particular program might have, our initial goal was to evaluate the robustness of our implementation when exposed to worst case scenarios and, for that, we ran the benchmarks with all threads executing the same query goal. By doing that, we avoid the peculiarities of the program at hand and we try to focus on measuring the real value of our new design. Since, all threads are executing the same query goal, it is expected that all threads will access the table space, to check/insert for subgoals and answers, at similar times, thus stressing the synchronization on common memory locations, which can increase the aforementioned problems of false sharing and cache memory side-effects and thus penalize the less robust designs.

The environment for our experiments was a machine with 2x16 (32) Core AMD Opteron (tm) Processor 6274 @ 2.2 GHz with 32 GBytes of memory and running the Linux kernel 3.8.3-1.fc17.x86_64 with Yap Prolog 6.3. We experimented with intervals of 8 threads up to 32 threads and all results are the average of 5 runs for each benchmark. Figure 6 shows the average execution time, in seconds, and the average overhead, compared against the respective execution time with one thread, for the LFHT, LF and LB designs when running the set of tabling benchmarks with all threads executing the same query goal.

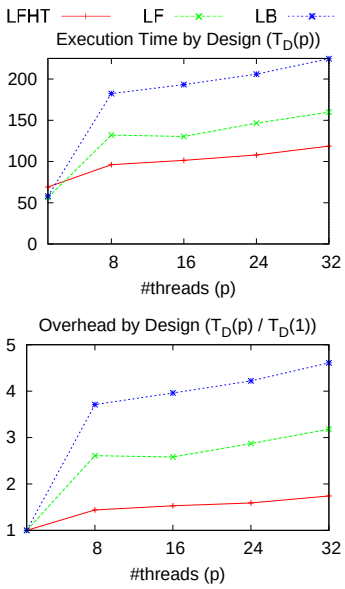


Fig. 6. Average execution time, in seconds, and average overhead, against the execution time with one thread, for the set of tabling benchmarks with all threads executing the same query goal

The results clearly show that the new LFHT design achieves the best performance for both the execution time and the overhead. As expected, LF is the second best and LB is the worst. In general, our design clearly outperforms the other designs with a overhead of at most 1.74 for 32 threads (the number of cores in the machine). Another important observation is that both LF and LB show an initial high overhead in the execution time in most experiments, mainly when going from 1 to 8 threads, in contrast to LFHT that shows more smooth curves. The difference between LFHT and LF/LB for the overhead ratio in these benchmarks clearly shows the distinct potential of the LFHT design.

Besides measuring the value of our new design through the use of worst case scenarios, we conclude the paper by showing the potential of our work to speedup the execution of tabled programs. Other works have already showed the capabilities of the use of multithreaded tabling to speedup tabled execution (Marques and Swift 2008; Marques et al. 2010). Here, for each program, we considered a set of different queries and then we ran this set with different number of threads. To do that, we implemented a naive scheduler in Prolog code that initially launches the number of threads required and then uses a mutex to synchronize access to the pool of queries. We experimented with a Path program using a grid-like configuration and with two well-known ILP data-sets, the Carcinogenesis and Mutagenesis data-sets. We used the same 32 Core AMD machine, experimented with intervals of 8 threads up to 32 threads and the results that follow are the average of 5 runs. Figure 7 shows the average execution time, in seconds, and the average speedup, compared against the respective execution time with one thread, for running the naive scheduler on top of these three programs with the LFHT design.

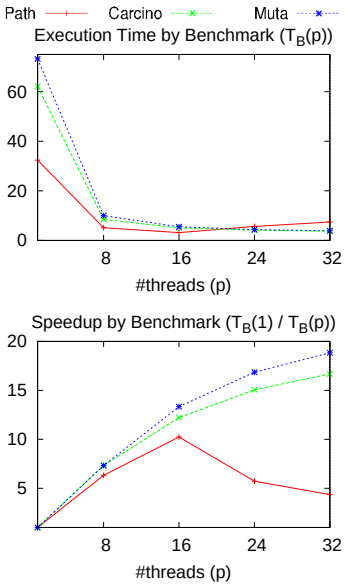


Fig. 7. Execution time, in seconds, and speedup, against the execution time with one thread, for running the naive scheduler program with the LFHT design

The results show that our design has potential to speedup the execution of tabled programs. For the Path benchmark, the speedup increases up to 10.24 with 16 threads, but then it starts to slow down. We believe that this behavior is related with the large number of tabled dependencies in the program. For the Carcino and Muta benchmarks, the speedup increases up to a value of 16.68 and 18.84 for 32 threads, respectively. Note that our goal with these experiments was not to achieve maximum speedup because this would require to take into account the peculiarities of each program and eventually develop specialized schedulers for each one, which is orthogonal to the focus of this work.

5 Conclusions

We have presented a novel, simple and efficient lock-free design for concurrent tabling. A key design decision in our approach is the combination of hash tries with the use of a separate chaining closed by the last node referencing back the hash level for the node. This allows us to implement a clean design to solve hash collisions by simply moving nodes between the levels. In our design, updates and expansions of the hash levels are never done by using replacement of data structures (i.e., create a new one to replace the old one), which also avoids the need for memory recovery mechanisms. Another important design decision which minimizes the bottlenecks and performance problems leading to false sharing or cache memory side-effects, is the insertion of nodes done at the end of the separate chain. This allows for dispersing as much as possible the memory locations being updated concurrently and, more importantly, reduces the updates for the memory locations accessed more frequently, like the bucket entries for the hash levels.

Experimental results in the context of Yap's concurrent tabling environment, showed that our design clearly achieved the best results for the execution time, speedups and overhead ratios. In particular, for worst case scenarios, our design clearly outperformed the previous designs with a superb overhead always below 1.74 for 32 threads or less. We thus argue that our design is the best proposal to support concurrency in general purpose multithreaded tabling applications.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within project SIBILA (NORTE-07-0124-FEDER-000059). Miguel Areias is funded by the FCT grant SFRH/BD/69673/2010.

References

Areias, M. and Rocha, R. 2012a. An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs. In International Conference on Parallel and Distributed Systems. IEEE Computer Society, 636–643.

Areias, M. and Rocha, R. 2012b. Towards Multi-Threaded Local Tabling Using a Common Table Space. Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue 12, 4 & 5, 427–443.

Areias, M. and Rocha, R. 2014. On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs. In International Symposium on Practical Aspects of Declarative Languages. Number 8324 in LNCS. Springer-Verlag, 168–183.

Bagwell, P. 2001. Ideal Hash Trees. Es Grands Champs 1195.

Chen, W. and Warren, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43, 1, 20–74.

Fredkin, E. 1962. Trie Memory. Communications of the ACM 3, 490–499.

Marques, R. and Swift, T. 2008. Concurrent and Local Evaluation of Normal Programs. In International Conference on Logic Programming. Number 5366 in LNCS. Springer-Verlag, 206–222.

Marques, R., Swift, T., and Cunha, J. C. 2010. A Simple and Efficient Implementation of Concurrent Local Tabling. In International Symposium on Practical Aspects of Declarative Languages. Number 5937 in LNCS. Springer-Verlag, 264–278.

Moura, P. 2008. ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates.

Prokopec, A., Bronson, N. G., Bagwell, P., and Odersky, M. 2012. Concurrent Tries with Efficient Non-Blocking Snapshots. In ACM Symposium on Principles and Practice of Parallel Programming. ACM, 151–160.

Rocha, R., Silva, F., and Santos Costa, V. 2005. On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming 5, 1 & 2, 161–205.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog System. Journal of Theory and Practice of Logic Programming 12, 1 & 2, 5–34.

Shalev, O. and Shavit, N. 2006. Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM 53, 3, 379–405.

Triplett, J., McKenney, P. E., and Walpole, J. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In USENIX Annual Technical Conference. USENIX Association, 11–11.

Grounding Bound Founded Answer Set Programs

Rehan Abdul Aziz, Geoffrey Chu and Peter J. Stuckey

National ICT Australia, Victoria Laboratory,†
Department of Computing and Information Systems,
University of Melbourne, Australia
Email: raziz@student.unimelb.edu.au, gchu@csse.unimelb.edu.au, pjs@csse.unimelb.edu.au

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Bound Founded Answer Set Programming (BFASP) is an extension of Answer Set Programming (ASP) that extends stable model semantics to numeric variables. While the theory of BFASP is defined on ground rules, in practice BFASP programs are written as complex non-ground expressions. Flattening of BFASP is a technique used to simplify arbitrary expressions of the language to a small and well defined set of primitive expressions. In this paper, we first show how we can flatten arbitrary BFASP rule expressions, to give equivalent BFASP programs. Next, we extend the bottom-up grounding technique and magic set transformation used by ASP to BFASP programs. Our implementation shows that for BFASP problems, these techniques can significantly reduce the ground program size, and improve subsequent solving.

KEYWORDS: Answer Set Programming, Grounding, Flattening, Constraint ASP, Magic Sets

1 Introduction

Many problems in the areas of planning or reasoning can be efficiently expressed using Answer Set Programming (ASP) (Baral 2003). ASP enforces stable model semantics (Gelfond and Lifschitz 1988) on the program, which disallows solutions representing circular reasoning. For example, given only rules $b \leftarrow a$ and $a \leftarrow b$, the assignment $a = true, b = true$ would be a solution under the logical semantics normally used by Boolean Satisfiability (SAT) (Mitchell 2005) solvers or Constraint Programming (CP) (Marriott and Stuckey 1998) solvers, but would not be a solution under the stable model semantics used by ASP solvers.

Bound Founded Answer Set Programming (BFASP) (Aziz et al. 2013) is an extension of ASP to allow *founded* integer and real variables. This makes it possible to concisely express and efficiently solve problems involving inductive definitions of numeric variables where we want to disallow circular reasoning. As an example consider the Road Construction problem (*RoadCon*). We wish to decide which roads to build such that the shortest paths between various cities are acceptable, with the minimal total cost. This can be modeled as:

A complete version of this paper that includes examples can be found at http://arxiv.org/abs/1405.3362

† NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

$$\text{minimize} \sum_{e \in Edge} built[e] \times cost[e]$$
$$\forall y \in Node : sp[y, y] \leqslant 0$$
$$\forall y \in Node, e \in Edge : sp[from[e], y] \leqslant len[e] + sp[to[e], y] \leftarrow built[e]$$
$$\forall y \in Node, e \in Edge : sp[to[e], y] \leqslant len[e] + sp[from[e], y] \leftarrow built[e]$$
$$\forall p \in Demand : sp[d\_from[p], d\_to[p]] \leqslant demand[p]$$

The decisions are which edges e are built ($built[e]$). The aim is to minimize the total cost of the edges $cost[e]$ built. The first rule is a base case that says that shortest path from a node to itself is 0. The second constraint defines the shortest path $sp[x, y]$ from x to y: the path from x to y is no longer than from x to z along edge e if it is built plus the shortest path from z to y; and the third constraint is similar for the other direction of the edge. The last constraint ensures that the shortest path for each of a given set of paths $p \in Demand$ is no longer than its maximal allowed distance $demand[p]$. The above model has a trivial solution with cost 0 by setting $sp[x, y] = 0$ for all x, y. In order to avoid this, we require that the sp variables are (upper-bound) *founded* variables, that is they take the largest possible justified value. The first three constraints are actually *rules* which justify upper bounds on sp, the last constraint is a restriction that needs to be met and cannot be used to justify upper bounds. Solving such a BFASP is challenging, mapping to CP models leads to inefficient solving, and hence we need a BFASP solver which can reason directly about *unfounded sets* (Van Gelder et al. 1988) of numeric assumptions. Note that *Constraint ASP* (CASP) and hybrid systems such as those given by (Mellarkod et al. 2008; Gebser et al. 2009; Drescher and Walsh 2012; Liu et al. 2012; Balduccini 2009; Aziz et al. 2013a) cannot solve the above problem without grounding the numeric domain to propositional variables and running into the grounding bottleneck. BFASP has been shown to subsume CP, ASP, CASP and Fuzzy ASP (Nieuwenborgh et al. 2006; Blondeel et al. 2013), see (Aziz et al. 2013) for details.

The above encoding for Road Construction problem is a *non-ground* BFASP since it is parametric in the data: $Node$, $Edge$, $Demand$, $cost$, $from$, to, len, d_from, d_to and $demand$. In this paper we consider how to efficiently create a ground BFASP from a non-ground BFASP given the data. This is analogous to *flattening* (Stuckey and Tack 2013) of constraint models and *grounding* (Syrjanen 2009; Gebser et al. 2007; Perri et al. 2007) of ASP programs. The contributions of this paper are: a flattening algorithm that transforms complex expressions to primitive forms while preserving the stable model semantics, a generalization of bottom-up grounding for normal logic programs to BFASPs and a generalization of the magic set transformation (Bancilhon et al. 1985; Beeri and Ramakrishnan 1991) for normal logic programs to BFASPs.

2 Preliminaries

2.1 Constraints and Answer Set Programming

We consider three types of variables: integer, real, and Boolean. Let $\mathcal{V}$ be a set of variables. A *domain* D maps each variable $x \in \mathcal{V}$ to a set of constant values $D(x)$. A *valuation* (or assignment) θ over variables $vars(\theta) \subseteq \mathcal{V}$ maps each variable $x \in vars(\theta)$ to a value $\theta(x)$. A restriction of assignment θ to variables V, $\theta|_V$, is the the assignment θ' over $V \cap vars(\theta)$ where $\theta'(v) = \theta(v)$. A *constraint* c is a set of assignments over the variables $vars(c)$, representing the solutions of the constraint. A constraint c is *monotonically increasing (resp. decreasing)* w.r.t. a variable $y \in vars(c)$ if for all solutions θ that satisfy c, increasing (resp. decreasing) the value of y also creates a solution, that is θ' where $\theta'(y) > \theta(y)$ (resp. $\theta'(y) < \theta(y)$), and

$\theta'(x) = \theta(x), x \in vars(c) - \{y\}$, is also a solution of c. A *constraint program (CP)* is a collection of variables $\mathscr{V}$ and constraints C on those variables ($vars(c) \subseteq \mathscr{V}, c \in C$). A *positive-CP* P is a CP where each constraint is increasing in exactly one variable and decreasing in the rest. The *minimal* solution of a positive-CP is an assignment θ that satisfies P s.t. there is no other assignment θ' that also satisfies P and there exists a variable v for which $\theta'(v) < \theta(v)$. Note that for Booleans, $true > false$. A positive-CP P always has a unique minimal solution. If we have bounds consistent propagators for all the constraints in the program, then this unique minimal solution can be found simply by performing bounds propagation on all constraints until a fixed point is reached, and then setting all variables to their lowest values.

A *normal logic program* P is a collection of *rules* of the form: $b_0 \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg b'_1 \wedge \ldots \wedge \neg b'_m$ where $\{b_0, b_1, \ldots, b_n, b'_1, \ldots, b'_m\}$ are Boolean variables. b_0 is the *head* of the rule while the RHS of the reverse implication is the *body* of the rule. A rule without any negative literals is a *positive rule*. A *positive program* is a collection of positive rules. The *least model* of a positive program is an assignment θ that assigns *true* to the minimum number of variables. The *reduct* of P w.r.t. an assignment θ is written P^θ and is a positive program obtained by transforming each rule r of P as follows: if there exists an i for which $\theta(b'_i) = true$, discard the rule, otherwise, discard all negative literals $\{b'_1, \ldots, b'_m\}$ from the rule. The stable models of P are all assignments θ for which the least model of P^θ is equal to θ. Note that if we consider a logic program as a constraint program, then a positive program is a positive-CP and the least model of that program is equivalent to the minimal solution defined above.

2.2 Bound Founded Answer Set Programs (BFASP)

BFASP is an extension of ASP that extends its semantics over integer and real variables. In BFASP, the set of variables is a union of two disjoint sets: standard $\mathscr{S}$ and *founded* variables $\mathscr{F}$.[1] A rule r is a pair (c, y) where c is a constraint, $y \in \mathscr{F}$ is the head of the rule and it is increasing in c. A bound founded answer set program (BFASP) P is a tuple $(\mathscr{S}, \mathscr{F}, C, R)$ where C and R are sets of constraints and rules respectively (also accessed as $constraints(P)$ and $rules(P)$ resp.). Given a variable $y \in \mathscr{F}$, $rules(y)$ is the set of rules with y as their heads. Each standard variable s is associated with a lower and an upper bound, written $lb(s)$ and $ub(s)$ respectively.

The reduct of a BFASP P w.r.t. an assignment θ is a positive-CP made from each rule $r = (c, y)$ by replacing in c every variable $x \in vars(c) - \{y\}$ s.t. x is a standard variable or c is not decreasing in x, by its value $\theta(x)$ to create a positive-CP constraint c'. Let r^θ denote this constraint. If r^θ is not a tautology, it is included in the reduct P^θ. An assignment θ is a stable solution of P iff i) it satisfies all the constraints in P and ii) it is the minimal solution that satisfies P^θ. For a variable $y \in \mathscr{F}$, the *unconditionally justified bound* of y, written $ujb(y)$, is a value that is unconditionally justified by the rules of the program regardless of what the standard variables are fixed to. E.g. if we have a rule: $(y \geqslant 3 + x, y)$ where x is a standard variable with domain $[0, 10]$, then we can set $ujb(y) = 3$. For any Boolean, we assume that ujb is fixed to $false$.

The focus of this paper is BFASPs where every rule is written in the form $(y \geqslant f(x_1, \ldots, x_n), y)$. Recall that we consider the domains of Boolean variables to be ordered such that $true > false$. So for example, an ASP rule such as $a \leftarrow b \wedge c$ can equivalently be written as: $a \geqslant f(b, c)$ where

[1] For the rest of this paper we only consider *lower bound* founded variables, analogous to founded Booleans. Upper bound founded variables can be implemented as negated lower bound founded variables, e.g. replace $sp[x, y]$ in the Road Construction example by $-nsp[x, y]$ where $nsp[x, y]$ is lower bound founded.

f is a Boolean that returns the value of $b \wedge c$. $f(x_1, \ldots, x_n)$ is essentially an expression tree where the leaf nodes are the variables $x_1, \ldots, x_n$.

The *local dependency graph* for a BFASP P is defined over founded variables. For each rule $r = (y \geqslant f(x_1, \ldots, x_n), y)$, there is an edge from y to all founded x_i. Each edge is marked increasing, decreasing, or non-monotonic, depending on whether f is increasing, decreasing, or non-monotonic in x_i. A BFASP is *locally valid* iff no edge within an SCC is marked non-monotonic. A program is *locally stratified* if all the edges between any two nodes in the same component are marked increasing.

2.3 Non-ground BFASPs

A *non-ground BFASP* is a BFASP where sets of variables are grouped together in variable arrays, and sets of ground rules are represented by non-ground rules via universal quantification over index variables. For example, if we have arrays of variables a, b, c, then we can represent the ground rules: $(a[1] \geqslant b[1] + c[1], a[1]), (a[2] \geqslant b[2] + c[2], a[2]), (a[3] \geqslant b[3] + c[3], a[3])$ by $\forall i \in [1, 3] : (a[i] \geqslant b[i] + c[i], a[i])$. Variables can be grouped together in arrays of any dimension and non-ground BFASP rules have the following form: $\forall \bar{i} \in \bar{D}$ where $con(\bar{i}) : (y[l_0(\bar{i})] \geqslant f(x_1[l_1(\bar{i})], \ldots, x_n[l_n(\bar{i})]), y[l_0(\bar{i})])$, where $\bar{i}$ is a set of index variables $i_1, \ldots, i_m$, $\bar{D}$ is a set of domains $D_1, \ldots, D_m$, *con* is a *constraint* over the index variables which constrains these variables, $l_0, \ldots, l_n$ are functions over the index variables which return a tuple of array indices, $y, x_1, \ldots, x_n$ are arrays of variables and f is a function over the x_i variables. Let $gen(r) \equiv \bar{i} \in \bar{D} \wedge con(\bar{i})$ denote the *generator constraint* for a non-ground rule r. Note that we require the generator constraint in each rule to constrain the index variables so that f is always defined.

Variable arrays can contain either founded variables, standard variables, or parameters (which can simply be considered fixed standard variables), although all variables in a variable array must be of the same type. Note that the array names in our notation correspond to predicate names in standard ASP syntax, and our index variables correspond to ASP "local variables." Given a non-ground rule r, let $grnd(r)$ be the set of ground rules obtained by substituting all possible values of the index variables that satisfy $gen(r)$ into the quantified expression. Similarly given a non-ground BFASP P, let $grnd(P)$ be the grounded BFASP that contains the grounding of all its rules and constraints. The *predicate dependency graph*, validity and stratification are defined similarly for array variables and non-ground rules as the local dependency graph, local validity and local stratification respectively are defined for ground variables and ground rules. All our subsequent discussion is restricted to valid BFASPs.

3 Flattening

A ground BFASP may contain constraints and rules whose expressions are not *flat*, i.e., they are expression trees with height greater than one. Such expressions are not supported by constraint solvers and we need to flatten these expressions to primitive forms. We omit consideration of flattening constraints since this is the same as in standard CP (Stuckey and Tack 2013). It can be shown that the standard CP flattening approach in which a subexpression is replaced with a standard variable and a constraint is added that equates the introduced variable with the subexpression, does *not* preserve stable model semantics. To preserve the stable model semantics, it is necessary to use introduced *founded* variables to represent subexpressions containing founded variables. We now describe the central result used in our flattening algorithm.

```
flat(P)
    P_flat := ∅
    R := rules(P)
    T := constraints(P)
    for(r ∈ R)
        R := R \ {r}
        flatRule(r, R, T)
        r := simplify(r)
        P_flat ∪= {r}
    for(c ∈ T) P_flat ∪= cp_flat(c)
    return P_flat
```

```
flatRule(r = (y ⩾ f(e_1, ..., e_n)), R, T)
    for(each non-terminal e_i)
        if(e_i does not contain founded vars)
            replace e_i with standard var y' in r
            T ∪={y' = e_i}
        elif(f is increasing in e_i)
            replace e_i with founded var y' in r
            R ∪={(y' ⩾ e_i, y')}
        elif(f is decreasing in e_i)
            replace e_i with founded var −y' in r
            R ∪={(y' ⩾ −e_i, y')}
```

Theorem 1
Let P be a BFASP containing a rule $r = (y \geqslant f_1(x_1, \ldots, x_k, f_2(x_{k+1}, \ldots, x_n)), y)$ where f_1 is increasing in the argument where f_2 appears, and where if a variable occurs among both $x_1, \ldots, x_k$ and $x_{k+1}, \ldots, x_n$, then f_1 and f_2 have the same monotonicity w.r.t. it. Let P' be P with r replaced by the two rules: $r_1 = (y \geqslant f_1(x_1, \ldots, x_k, y'), y)$ and $r_2 = (y' \geqslant f_2(x_{k+1}, \ldots, x_n), y')$ where y' is an introduced founded variable. Then the stable solutions of P' restricted to the variables of P are equivalent to the stable solutions of P.

As a corollary, if f_1 is decreasing in the argument where f_2 appears, we can replace f_2 by a founded variable $-y'$ and add the rule $(y' \geqslant -f_2(x_k, \ldots, x_n), y')$ instead. Not all valid rule forms are supported by Theorem 1, because we require that multiple occurrences of the same variable in the expression must have the same monotonicity w.r.t. the root expression. Note that if a subexpression does not contain any founded variables at all, i.e., only contains standard variables, parameters or constants, then a standard CP flattening step is sufficient. Let us now describe our flattening algorithm flat for ground BFASPs and later extend it to non-ground BFASPs. We put all the rules and constraints of the program in sets R and T respectively. For every rule $r = (y \geqslant f(e_1, \ldots, e_n), y) \in R$, where f is the top level function in that rule, and $e_1, \ldots, e_n$ are the expressions which form f's arguments, we call flatRule which works as follows. If there is some e_i which is not a terminal, i.e., not a constant, parameter or variable, then we have two cases. If e_i does not contain any founded variables, we simply replace it with standard variable y' and add the constraint $y' = e_i$ to T. Otherwise, we apply the transformation described in Theorem 1. After flatRule, we simplify r as much as possible through the subroutine simplify, e.g., by getting rid of double negations, pushing negations inside the expressions as much as possible etc. Finally, we flatten all the constraints in T using the standard CP flattening algorithm cp_flat as described in (Stuckey and Tack 2013). Since we replace all decreasing subexpressions by negated introduced variables and simplify expressions by pushing negations towards the variables, we handle negation through simple rule forms like $(y \geqslant -x, y)$, $(y \geqslant \neg x, y)$ etc.

The algorithm can be extended to non-ground rules by defining the index set of the introduced variables to be equal to the domain of index variables as given in the generator of the rule in which they replace an expression. Moreover, the generator expression of an intermediate rule stays the same as that of the original rule from which it is derived.

4 Grounding

ASP grounders keep track of variables that have been created and instantiate further rules based on that. For example, if the variables b and c have been created, then the rule $a \leftarrow b \wedge c$ justifies a bound on a and therefore, must be included in the final program. The justification of all positive literals in a rule potentially justify its head. However, for a rule, if any one positive variable in its body does not have any rule supporting it, then that rule can safely be ignored until a justification

c	ϕ_r
$y \geqslant sum(x_1, \ldots, x_n)$	$(\sum_i ujb(x_i) > ujb(y)) \vee$ $((\wedge_i ujb(x_i) > -\infty \vee cr(x_i)) \wedge (\vee_i cr(x_i)))$
$y \geqslant max(x_1, \ldots, x_n)$	$\vee_i (ujb(x_i) > ujb(y) \vee cr(x_i))$
$y \geqslant min(x_1, \ldots, x_n)$	$\wedge_i (ujb(x_i) > ujb(y) \vee cr(x_i))$
$y \geqslant product(x_1, \ldots, x_n)$ where $\wedge_i x_i > 0$	$\prod_i ujb(x_i) > ujb(y) \vee (\vee_i cr(x_i)))$
$y \geqslant x \leftarrow r$	$cr(r) \wedge (ujb(x) > ujb(y) \vee cr(x))$
$y \leftarrow x \geqslant 0$	$ujb(x) \geqslant 0 \vee cr(x)$
$y \leftarrow \wedge_i x_i$	$\wedge_i cr(x_i)$
$y \leftarrow \vee_i x_i$	$\vee_i cr(x_i)$
$y \geqslant -x$	$-ub(x) > ujb(y)$
$y \leftarrow \neg x$	$true$
$y \geqslant 1/x$ where $x > 0$	$1/-ub(x) > ujb(y)$

Table 1. *Grounding conditions for rule* $r = (c, y)$

for that variable has been found. In case a justification is never found for that variable, then the rule is *useless*, i.e., excluding the rule from the program does not change its stable solutions.

We propose a simple grounding algorithm for non-ground BFASPs which can be implemented by simply maintaining a set of ground rules and variables as done in ASP grounders, but which may generate useless rules in addition to all the useful ones. The idea is that for each variable v, we only keep track of whether v can potentially be justified above its ujb value, rather than keeping track of whether it can be justified above each value in its domain. If it can be justified above its ujb, then when v appears in the body of a rule, we assume that v can be justified to any possible bound for the purpose of calculating what bound can be justified on the head.

We refer to a variable x as being *created*, written $cr(x)$, if it can go above its ujb value. More formally, $cr(x)$ is a founded Boolean with a rule: $cr(x) \leftarrow x > ujb(x)$. While that is how we define $cr(x)$, we do not explicitly have a variable $cr(x)$ or the above rule in our implementation. Instead, we implement it by maintaining a set Q of variables that have been created. Initially, Q is empty. We recursively look at each non-ground rule to see if the newly created variables make it possible for more head variables to be justified above their ujb values. If so, we create those variables and add them to Q. In order to do this, we need to find necessary conditions under which the head variable can be justified above its ujb. In order to simplify the presentation, we are going to define ujb for constants, standard variables and parameters as well. For a constant x, we define $ujb(x)$ to be the value of x. For parameters and standard variables x, we define $ujb(x) = ub(x)$.[4] Note that for soundness, the ujb values of founded variables only have to be correct (e.g. $-\infty$ for all variables) although tighter ujb values can improve the efficiency of our algorithm. Table 1 gives a non-exhaustive list of necessary conditions for the head variable to be justified above its ujb value for different rule forms.

Let us now make a few observations about the conditions given in Table 1. A key point is that for many rule forms ϕ_r can evaluate to $true$, even without any variable in the body getting created. All such rules that evaluate to true give us a starting point for initializing Q in our implementation. The linear case (*sum*) deserves some explanation. It is made up of two disjuncts, the first of which is an evaluation of the initial condition, i.e., whether the sum of ujb values of all variables is greater than the ujb of the head. If this condition is true, then the rule needs to be grounded unconditionally. If this is false, then the second disjunct becomes important. The second disjunct itself is a conjunction of two more conditions. The first one says that all variables must be greater than $-\infty$ in order for the rule to justify a finite value on the head. In the case

[4] Upper and lower bounds for a parametric array can be established by simply parsing the array.

createCPs(P)
  for($r \in rules(P) : \phi_r = \bigwedge_{i=1}^{n} cr(x_i[\bar{l}_i])$)
    $cp[r] := true$ % new constraint program
    $cp[r] := cp[r] \wedge gen(r)$
    for($i \in 1 \ldots n$)
      $set[r,i] := \emptyset$
      $cp[r] := cp[r] \wedge \bar{l}_i \in \ll set[r,i] \gg$
  for($r \in rules(P) : \phi_r = \bigvee_{i=1}^{n} cr(x_i[\bar{l}_i])$)
    for($i \in 1 \ldots n$)
      $cp[r,i] := true$ % new constraint program
      $cp[r,i] := cp[r,i] \wedge gen(r)$
      $set[r,i] := \emptyset$
      $cp[r,i] := cp[r,i] \wedge \bar{l}_i \in \ll set[r,i] \gg$

ground(P)
  $C := \{\mathsf{groundAll}(c) : c \in constraints(P)\}$
  $R' := \{\mathsf{groundAll}(r) : r \in rules(P) : \phi_r = true\}$
  while($R' \neq \emptyset$)
    $H := \mathsf{heads}(R')$
    $Q \cup\!= H$
    $R' := \emptyset$
    for($r \in rules(P) : H \cap vars(\phi_r) \neq \emptyset$)
      if($\phi_r = \bigwedge_{i=1}^{n} cr(x_i[\bar{l}_i]) \vee \phi_r = \bigvee_{i=1}^{n} cr(x_i[\bar{l}_i])$)
        for($i \in 1 \ldots n$)
          $dom := \{\bar{m} \mid x[\bar{m}] \in Q\}$
          $set[r,i] := dom \setminus set[r,i]$
          if(ϕ_r is conj) $R' \cup\!= \mathsf{search}(\mathrm{cp[r]}) \setminus R$
          if(ϕ_r is disj) $R' \cup\!= \mathsf{search}(\mathrm{cp[r,i]}) \setminus R$
          $R \cup\!= R'$
          $set[r,i] := dom$
  for($y \in vars(R) \cap \mathscr{F}$) $R \cup\!= (y \geqslant ujb(y), y)$

where all variables already have a finite ujb, the second conjunct says that at least one of them must be created for the rule to be grounded . Finally, observe that after plugging all values of ujb, all conditions given in the table simplify to one of the following four forms: $true$, $false$, $\vee_i cr(x_i)$ or $\wedge_i cr(x_i)$. Note that the grounding conditions are significantly more sophisticated than the simple conjunctive condition for normal rules. More specifically, after simplification, we can get a disjunctive condition which has no analog in ASP.

We are now ready to present the main bottom-up grounding algorithm. Logically, our grounding algorithm starts with $ujb(x)$ for all x, adds $(x \geqslant ujb(x), x)$ to the program and then finds all the ground rules that are not made redundant by these rules. createCPs is a preprocessing step that creates constraint programs for rules in a BFASP P whose conditions are either conjunctions or disjunctions. For a rule with a conjunctive condition, it only creates one program, while for one with a disjunctive condition, it creates one constraint program for each variable in the condition. Each program is initialized with the $gen(r)$ which defines the variables and some initial constraints given in the where clause in the generator of non-ground rule. Furthermore, for each array literal in ϕ_r, a constraint is posted on its literal (which is a function of index variables in the rule), to be in the domain given by the *current* value of the *set* variable (the reason for the Quine quotes) which is initially set to empty. ground is called after preprocessing. Q and R are sets of ground variables and rules respectively. groundAll is a function that grounds a non-ground rule or constraint completely, and returns the set of all rules and constraints respectively. Initially, we ground all constraints in P and rules for which ϕ_r evaluates to true. R' is a temporary variable that represents the set of new ground rules from the last iteration. In each iteration, we only look for non-ground rules that have some variable in their conditions that is created in the previous iteration. heads takes a set of ground rules as its input and returns their heads. In each iteration, through Q, we manipulate the *set* constraint to get new rule instantiations. For each variable in the clause, we make *set* equal to the new index values created for that variable. For both the conjunctive and the disjunctive case, this optimization only tries out new values of recently created variables to instantiate new rules. search takes a constraint program as its input, finds all its solutions, instantiates the non-ground rule for each solution, and returns the set of these ground rules. After creating new rules due to the new values in *set*, we make it equal to all values of the variable in Q. The fixed point calculation stops when no new rules are created. Finally, for every founded variable y, we add $(y \geqslant ujb(y), y)$ as a rule so that if the ujb relied on some rules that were ignored during grounding, then this ensures that $ujb(y)$ is always justified.

5 Magic set transformation

Let us first define the *query* of a BFASP. To build the query Q for a BFASP P, we ground all its constraints and its objective function, and put all the variables that appear in them in Q.[5] Note that our query does not have any free variables and only contains ground variables. Therefore, we do not need adornment strings to propagate binding information as in the original magic set technique. The original magic set technique has three stages: adorn, generate and modify. For the reason described above, we only describe the latter two.

The purpose of the magic set technique is to simulate a top-down computation through bottom-up grounding. For every variable a in the original program, we create a *magic variable* m_a that represents whether we care about a. Additionally, there are *magic rules* that specify when a magic variable should be created. Consider a simple rule $(a \geqslant b + c, a)$ where ujb of all variables is equal to $-\infty$. Suppose we are interested in computing a, we model this by setting m_a to true. Since b is required to compute the value of a, we add a magic rule $m\_b \leftarrow m\_a$. We do not care about c until a finite bound on b is justified (until b is created), so we generate a tighter magic rule for c: $m\_c \leftarrow m\_a \wedge cr(b)$.

We can utilize the necessary conditions for a useful grounding of a rule r as given by ϕ_r. Recall that after evaluating the initial conditions, ϕ_r reduces to true, false, a conjunction or a disjunction. The above generation of magic rules for the rule $(a \geqslant b + c, a)$ is an example of the conjunctive case. For a disjunction, the magic rules are even simpler. For every $cr(x)$ in the disjunction, we create the magic rule $m\_x \leftarrow m\_a$. Note that not all variables in the original rule appear in the condition; some might get removed in the simplification or not be included in the original condition at all. We can ignore them for grounding, but we are interested in their values as soon as we know that the rule can be useful. Therefore, as soon as the magic variable for the head is created, and ϕ_r is satisfied, we are interested in all the variables in the rule that do not appear in ϕ_r. Finally, we define the modification step for a rule $r = (y \geqslant f(\bar{x}), y)$, written $\mathsf{modify}(r)$, as changing it to $r = (y \geqslant f(\bar{x}) \leftarrow m_y, y)$. The pseudo-code for generation of magic rules and modification of the original rule is given as the function magic that takes a rule as its input. It adds magic rules for a rule to a set P. The first two if conditions handle the disjunctive and conjunctive case respectively. The **for** loop that follows generates magic rules for variables that are not in ϕ_r.

The entire bottom-up calculation with magic sets is as follows. First, create magic variables for all the variables in the program and call magic for every rule in the program. If the magic rules generated and/or the original rule after modification are not primitive expressions, flatten them. Then, call ground on the resulting program. While grounding the constraints, build the query by including m_v in Q for every ground variable v that is in some ground constraint. After grounding, filter all the magic variables from Q, and magic rules from R.

If a given BFASP program is unstratified, then the algorithm described above is not sound. There might be parts of the program that are unreachable from the founded atoms appearing in the query but are inconsistent. We refer the reader to (Faber et al. 2007) for further details. We overcome this by including in the query all ground magic variables of all array variables that are part of a component in the dependency graph in which there is some decreasing (negative) edge between any two of its nodes. The following result establishes correctness of our approach.

[5] Technically if the problem has output variables, whose value will be printed, they too need to be added to Q.

```
magic(r)
  a := head(r)
  if(φ_r = ⋁_{i=1}^{n} cr(x_i))
    for(i ∈ 1...n) P ∪= gen(r) : (m_x_i ← m_a, m_x_i)
  if(φ_r = ⋀_{i=1}^{n} cr(x_i))
    for(i ∈ 1...n)
      b := m_a
      P ∪= gen(r) : (m_x_i ← b, m_x_i)
      b := b ∧ cr(x_i)
  for(v ∈ vars(r) \ (vars(φ_r) ∪ {a}))
    P ∪= gen(r) : (m_v ← m_a ∧ φ_r, m_v)
  modify(r)
```

Theorem 2
Given a BFASP P, let G be equal to $grnd(P)$ and M be a ground BFASP produced by running the magic set transformation after including the unstratified parts of the program in the initial query for a given non-ground BFASP P. The stable solutions of G restricted to the variables $vars(M)$ are equivalent to the stable solutions of M. That is, if θ' is a stable solution of G, then $\theta'|_{vars(M)}$ is a stable solution of M and if θ is a stable solution of M, then there exists θ' s.t. θ' is a stable solution of G and $\theta'|_{vars(M)} = \theta$.

6 Experiments

We show the benefits of bottom-up grounding and magic sets for computing with BFASPs on a number of benchmarks: *RoadCon*, *UtilPol* and *CompanyCon*.[7] In utilitarian policies (*UtilPol*), a government decides a set of policies to enact while minimizing the cost. Additionally, there are different citizens and each citizen's happiness depends on the enacted policies and happiness of other citizens. There is a citizen t whose happiness should be above a given value. Company controls (*CompanyCon*) is a problem related to stock markets. The parameters of the problem are the number of companies, each company's ownership of stocks in other companies, and a source company that wants to *control* a destination company. The decision variables are the number of stocks that the source company buys in every other company. A company c controls a company d if the number of stocks that c owns in d plus the number of stocks that other companies that c controls own in d is greater than 50 percent of total number of stocks of company d. The objective is to minimize the total cost of stocks bought. All experiments were performed on a machine running Ubuntu 12.04.1 LTS with 8 GB of physical memory and Intel(R) Core(TM) i7-2600 3.4 GHz processor. Our implementation extends MiniZinc 2.0 (LIBMZN) and uses the solver CHUFFED extended with founded variables and rules as described in our previous work (Aziz et al. 2013). Each time in the tables is the median time in seconds of 10 different instances.

Table 2 shows the results for *RoadCon*. N is the number of nodes, and SCCs is the minimum number of strongly connected components in the graph. We compare exhaustive grounding (simply creating $grnd(P)$) against bottom-up grounding, and bottom-up grounding with magic set transformation. A — represents either the flattener/solver did not finish in 10 minutes or that it ran out of memory. Using bottom-up grounding, the founded variables representing shortest paths between two nodes that are not in the same SCC and the corresponding useless rules are not created. Clearly bottom-up grounding is far superior to naively grounding everything, and magic sets substantially improves on this. Tables 3 and 4 show the results for utilitarian policies

[7] All problem encodings and instances can be found at: `www.cs.mu.oz.au/~pjs/bound_founded/`

		Exhaustive		Bottom-up		Magic	
N	SCCs	Flat	Solve	Flat	Solve	Flat	Solve
100	5	4.25	3.34	1.37	.64	.27	.04
300	15	39.02	—	4.19	1.25	.41	.07
600	20	237.97	—	19.70	22.56	.83	.96
900	30	—	—	30.44	127.90	1.17	4.74
1400	45	—	—	56.99	398.29	1.79	25.66

Table 2. *Road Construction RoadCon*

Instance				Bottom-up		Magic	
C	P	C_r	P_r	Flat	Solve	Flat	Solve
50	100	5	5	2.02	1.90	.48	.01
100	300	10	30	16.62	91.66	2.97	.07
100	500	10	30	24.78	—	4.39	.09
250	350	105	105	83.45	—	35.16	18.40
250	400	110	110	88.61	—	39.32	452.17
300	400	125	150	140.36	—	57.09	—

Table 3. *UtilPol*

Instance		Bottom-up		Magic	
C	C_r	Flat	Solve	Flat	Solve
1000	15	24.27	5.20	.79	.70
1500	25	53.66	17.52	1.39	2.07
2000	35	94.38	66.81	2.31	8.75
3000	50	209.70	86.35	1.71	17.87
3500	60	—	—	5.58	19.18
5000	80	—	—	9.63	51.45

Table 4. *CompanyCon*

and company controls respectively. The running time for exhaustive and bottom-up for these benchmark are similar, therefore, the comparison is only given for bottom-up vs. magic sets. For *UtilPol*, C and P represent the number of citizens and policies respectively, C_r represents the maximum number of relevant citizens on which the happiness of t directly or indirectly depends and P_r is the maximum number of policies on which the happiness of t and other citizens in C_r depends. This is the part of the instance that is relevant to the query and the rest is ignored when magic sets are enabled. It can be seen that magic sets outperform regular bottom-up grounding, especially when the relevant part of the instance is small compared to the entire instance. Note that when P_r is small, the flattening time for magic sets is greater that the solving time since the resulting set of rules is actually simple. This changes, however, as P_r is increased. For *CompanyCon*, C is the number of total companies while C_r is the maximum number of companies reachable from the destination in the given ownership graph. The table shows that if C_r is small compared to C, magic sets can give significant advantages.

7 Conclusion

Bound Founded Answer Set Programming extends ASP to disallow circular reasoning over numeric entities. In this paper, we show how we can flatten and ground a non-ground BFASP while preserving its semantics, thus creating an executable specification of the BFASP problem. We show that using bottom-up grounding and magic sets transformation we can significantly improve the efficiency of computing BFASPs. The existing magic set techniques are only defined for the normal rule form, involving only founded Boolean variables. We have extended magic sets to BFASP, a formalism that has significantly more sophisticated rule forms and has both standard and founded variables, that can moreover be Boolean or numeric.

References

AZIZ, R. A., CHU, G., AND STUCKEY, P. J. 2013. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming 13,* 4–5, 517–532. Proceedings of the 29th International Conference on Logic Programming.

AZIZ, R. A., STUCKEY, P. J., AND SOMOGYI, Z. 2013a. Inductive definitions in constraint programming. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference*, B. Thomas, Ed. CRPIT, vol. 135. ACS, 41–50.

BALDUCCINI, M. 2009. Representing constraint satisfaction problems in answer set programming. In *In ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)*.

BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J. D. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 1–15.

BARAL, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

BEERI, C. AND RAMAKRISHNAN, R. 1991. On the power of magic. *The Journal of Logic Programming 10*, 255 – 299.

BLONDEEL, M., SCHOCKAERT, S., VERMEIR, D., AND DE COCK, M. 2013. Fuzzy answer set programming: An introduction. In *Soft Computing: State of the Art Theory and Novel Applications*. Springer, 209–222.

DRESCHER, C. AND WALSH, T. 2012. Answer set solving with lazy nogood generation. In *Technical Communications of the 28th International Conference on Logic Programming*. 188–200.

FABER, W., GRECO, G., AND LEONE, N. 2007. Magic sets and their application to data integration. *Journal of Computer and System Sciences 73,* 4, 584–609.

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming*. Springer, 235–249.

GEBSER, M., SCHAUB, T., AND THIELE, S. 2007. Gringo : A new grounder for answer set programming. In *LPNMR*. 266–271.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*. MIT Press, 1070–1080.

LIU, G., JANHUNEN, T., AND NIEMELA, I. 2012. Answer set programming via mixed integer programming. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, 32–42.

MARRIOTT, K. AND STUCKEY, P. 1998. *Programming with Constraints: an Introduction*. MIT Press.

MELLARKOD, V. S., GELFOND, M., AND ZHANG, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

MITCHELL, D. G. 2005. A SAT solver primer. *Bulletin of the EATCS 85*, 112–132.

NIEUWENBORGH, D. V., COCK, M. D., AND VERMEIR, D. 2006. Fuzzy answer set programming. In *Proceedings of Logics in Artificial Intelligence, 10th European Conference, JELIA 2006*. Springer Berlin Heidelberg, 359–372.

PERRI, S., SCARCELLO, F., CATALANO, G., AND LEONE, N. 2007. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence 51,* 2-4, 195–228.

STUCKEY, P. J. AND TACK, G. 2013. Minizinc with functions. In *Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Number 7874 in LNCS. Springer, 268–283.

SYRJANEN, T. 2009. Logic programs and cardinality constraints – theory and practice. Ph.D. thesis, Faculty of Information and Natural Sciences, Aalto University.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1988. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*. ACM, 221–230.

Appendix A Proofs of theorems

Theorem 1

Let P be a BFASP containing a rule $r = (y \geqslant f_1(x_1, \ldots, x_k, f_2(x_{k+1}, \ldots, x_n)), y)$ where f_1 is increasing in the argument where f_2 appears, and where if a variable occurs among both $x_1, \ldots, x_k$ and $x_{k+1}, \ldots, x_n$, then f_1 and f_2 have the same monotonicity w.r.t. it. Let P' be P with r replaced by the two rules: $r_1 = (y \geqslant f_1(x_1, \ldots, x_k, y'), y)$ and $r_2 = (y' \geqslant f_2(x_{k+1}, \ldots, x_n), y')$ where y' is an introduced founded variable. Then the stable solutions of P' restricted to the variables of P are equivalent to the stable solutions of P.

Proof

For a rule $s = (c, head)$, let $con(s) = c$. By construction, $con(r) \Leftrightarrow \exists_{y'}(con(r_1) \wedge con(r_2))$ and all the other constraints in P and P' are identical. Also, given any assignment θ' of P', since f_1 is increasing in the argument where f_2 appears, y' will be left as a variable in ${r_1}^{\theta}$. Consider an assignment θ' over $vars(P')$, and let $\theta = \theta'|_{vars(P)}$. Recall that the reduct of a program with respect to an assignment replaces all the standard variables and founded variables that are not decreasing in any rule's constraint with its value in that assignment. Since f_1 and f_2 have the same monotonicity w.r.t. any variable common in $\{x_1, \ldots, x_k\}$ and $\{x_{k+1}, \ldots, x_n\}$, it will either be replaced by its assignment value in both f_1 and f_2 or not be replaced at all. Therefore, the relation $r^{\theta} \Leftrightarrow \exists_{y'}({r_1}^{\theta'} \wedge {r_2}^{\theta'})$ is also valid. Furthermore, all other constraints in P^{θ} and $P'^{\theta'}$ are identical.

Suppose θ is a stable solution of P. Let θ' be the extension of θ to variable y' s.t. $\theta'(y') = f_2(\theta(x_k), \ldots, \theta(x_n))$. Clearly, this choice of $\theta'(y')$ allows θ' to satisfy all the constraints of P' and allows $\theta'|_{vars(P'^{\theta'})}$ to satisfy all the constraints of $P'^{\theta'}$. To prove that θ' is a stable solution of P', we just need to show that there is no smaller solution of $P'^{\theta'}$ than $\theta'|_{vars(P'^{\theta'})}$. Since $r^{\theta} \Leftrightarrow \exists_{y'}({r_1}^{\theta'} \wedge {r_2}^{\theta'})$ and all other constraints in P^{θ} and $P'^{\theta'}$ are identical, $P'^{\theta'}$ must force the same lower bounds on the variables in $vars(P^{\theta})$ as P^{θ} does. Hence, none of those values can go any lower. Also, ${r_2}^{\theta'}$ forces $y' \geqslant f_2(\theta(x_k), \ldots, \theta(x_n))$, and so $f_2(\theta(x_k), \ldots, \theta(x_n))$ is the lowest possible value for y'. Hence $\theta'|_{vars(P'^{\theta'})}$ is the minimal solution of $P'^{\theta'}$ and θ' is a stable solution of P'.

Suppose θ' is a stable solution of P'. Let $\theta = \theta'|_{vars(P)}$. Since $con(r) \Leftrightarrow \exists_{y'}(con(r_1) \wedge con(r_2))$ and all the other constraints in P and P' are identical, θ satisfies all the constraints in P. Since $r^{\theta} \Leftrightarrow \exists_{y'}({r_1}^{\theta'} \wedge {r_2}^{\theta'})$ and all other constraints in P^{θ} and $P'^{\theta'}$ are identical, $\theta|_{vars(P^{\theta})}$ satisfies all the constraints in P^{θ}. To prove that θ is a stable solution of P, we just need to show that there is no smaller solution of P^{θ} than $\theta|_{vars(P^{\theta})}$. Since $r^{\theta} \Leftrightarrow \exists_{y'}({r_1}^{\theta'} \wedge {r_2}^{\theta'})$ and all other constraints in P^{θ} and $P'^{\theta'}$ are identical, P^{θ} must force the same lower bounds on the variables in $vars(P^{\theta})$ as $P'^{\theta'}$ does. Hence, none of those values can go any lower, $\theta|_{vars(P^{\theta})}$ is the minimal solution of P^{θ} and θ is a stable solution of P. □

Theorem 2

Given a BFASP P, let G be equal to $grnd(P)$ and M be a ground BFASP produced by running the magic set transformation after including the unstratified parts of the program in the initial query for a given non-ground BFASP P. The stable solutions of G restricted to the variables $vars(M)$ are equivalent to the stable solutions of M. That is, if θ' is a stable solution of G, then $\theta'|_{vars(M)}$ is a stable solution of M and if θ is a stable solution of M, then there exists θ' s.t. θ' is a stable solution of G and $\theta'|_{vars(M)} = \theta$.

Proof

Let us first argue about the correctness of our grounding approach presented in Section 4. We can analyze each row in Table 1 and reason that until the condition is satisfied, the rule can be ignored without changing the stable solutions of the program. We only provide a brief sketch and do not analyze each case in the table. Say, e.g. for $y \geqslant max(x_1, \ldots, x_n)$, if the condition is not satisfied, this means that no x_i has a rule in the program that justifies a value higher than its ujb, and no x_i initially justifies a bound on y that is greater than $ujb(y)$. If we include a ground version of this rule in the program, then after taking the reduct w.r.t. any assignment, the rule can never justify any bound on the head, and hence can safely be eliminated.

Let P_i be part of $grnd(P)$ that is not included in M. It can be seen from the description of magic set transformation that any variable in P_i either cannot be reached from any variable in M in the dependency graph of P, or can only be reached through useless rules. Since useless rules can be eliminated as argued above, we conclude that no variable in M can reach any variable in P_i in the dependency graph. This obviously also holds for dependency graph of respective reduced program w.r.t. some assignment. This means that for a given assignment θ', the minimal order computation can first be performed on $M^{\theta'}$ which fixes all the variables in $vars(M)$, and then on $P_i^{\theta'}$ which fixes all the remaining variables, i.e., variables in $vars(P) - vars(M)$. Combining both the minimal solutions would be the same as computing the minimal solution for $G^{\theta'}$. This proves the first result.

For the second result, since all unstratified parts in P are included in M, all the intra-component edges in the dependency graph of P_i are marked increasing (positive). It can be shown that for such a program, once we fix all the standard variables appearing in any rule in P_i, there is a unique stable solution that can be computed as the *iterated least fixpoint* of P_i. This is similar to the well known result for logic programs that states that for a stratified program, the unique stable solution can be computed as the iterated least fixpoint of the program (Corollary 2 in (Gelfond and Lifschitz 1988)). Therefore, if we are given a stable solution θ for M, we can extend it to θ' by fixing all the unfixed standard variables to any value, and then computing the iterated least fixpoint, which will extend θ' over founded variables of P_i, and will be a unique stable solution given the values of all standard variables. □

Towards an ASP-Based Architecture for Autonomous UAVs in Dynamic Environments (Extended Abstract)

Marcello Balduccini, William C. Regli, Duc N. Nguyen

Applied Informatics Group
Drexel University
Philadelphia, PA, USA

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Traditional AI reasoning techniques have been used successfully in many domains, including logistics, scheduling and game playing. This paper is part of a project aimed at investigating how such techniques can be extended to coordinate teams of unmanned aerial vehicles (UAVs) in dynamic environments. Specifically challenging are real-world environments where UAVs and other network-enabled devices must communicate to coordinate – and communication actions are neither reliable nor free. Such network-centric environments are common in military, public safety and commercial applications, yet most research (even multi-agent planning) usually takes communications among distributed agents as a given. We address this challenge by developing an agent architecture and reasoning algorithms based on Answer Set Programming (ASP). Although ASP has been used successfully in a number of applications, to the best of our knowledge this is the first practical application of a complete ASP-based agent architecture. It is also the first practical application of ASP involving a combination of centralized reasoning, decentralized reasoning, execution monitoring, and reasoning about network communications.

1 Introduction

Unmanned Aerial Vehicles (UAVs) promise to revolutionize the way in which we use our airspace. From talk of automating the navigation for major shipping companies to the use of small helicopters as ”deliverymen” that drop your packages at the door, it is clear that our airspaces will become increasingly crowded in the near future. This increased utilization and congestion has created the need for new and different methods of coordinating assets using the airspace. Currently, airspace management is the job for mostly human controllers. As the number of entities using the airspace vastly increases—many of which are autonomous—the need for improved autonomy techniques becomes evident.

The challenge in an environment full of UAVs is that the world is highly dynamic and the communications environment is uncertain, making coordination difficult. Communicative actions in such setting are neither reliable nor free.

The work discussed here is in the context of the development of a novel application of network-aware reasoning and of an intelligent mission-aware network layer to the problem of UAV coordination. Typically, AI reasoning techniques do not consider realistic network models, nor does the network layer reason dynamically about the needs of the mission plan. With network-aware reasoning, a reasoner (either centralized or decentralized) factors in the communications network and its conditions.

In this paper we provide a general overview of the approach, and then focus on the aspect

of network-aware reasoning. We address this challenge by developing an agent architecture and reasoning algorithms based on Answer Set Programming (ASP, (Gelfond and Lifschitz 1991; Marek and Truszczynski 1999; Baral 2003)). ASP has been chosen for this task because it enables high flexibility of representation, both of knowledge and of reasoning tasks. Although ASP has been used successfully in a number of applications, to the best of our knowledge this is the first practical application of a complete ASP-based agent architecture. It is also the first practical application of ASP involving a combination of centralized reasoning, decentralized reasoning, execution monitoring, and reasoning about network communications.

The next section describes relevant systems and reasoning techniques, and is followed by a motivating scenario that applies to UAV coordination. The Technical Approach section describes network-aware reasoning and demonstrates the level of sophistication of the behavior exhibited by the UAVs using an example problem instance. Finally, we draw conclusions and discuss future work.

2 Related Work

Incorporating network properties into planning and decision-making has been investigated in (Usbeck et al. 2012). The authors' results indicate that plan execution effectiveness and performance is increased with the increased network-awareness during the planning phase. The UAV coordination approach in this current work combines network-awareness during the reasoning processes with a plan-aware network layer.

The problem of mission planning for UAVs under communication constraints has been addressed in (Kopeikin et al. 2013), where an ad-hoc task allocation process is employed to engage under-utilized UAVs as communication relays. In our work, we do not separate planning from the engagement of under-utilized UAVs, and do not rely on ad-hoc, hard-wired behaviors. Our approach gives the planner more flexibility and finer-grained control of the actions that occur in the plans, and allows for the emergence of sophisticated behaviors without the need to pre-specify them.

The architecture adopted in this work is an evolution of (Balduccini and Gelfond 2008), which can be viewed as an instantiation of the BDI agent model (Rao and Georgeff 1991; Wooldridge 2000). Here, the architecture has been extended to include a centralized mission planning phase, and to reason about other agents' behavior. Recent related work on logical theories of intentions (Blount et al. 2014) can be further integrated into our approach to allow for a more systematic hierarchical characterization of actions, which is likely to increase performance.

Traditionally, AI planning techniques have been used (to great success) to perform multi-agent teaming, and UAV coordination. Multi-agent teamwork decision frameworks such as the ones described in (Pynadath and Tambe 2002) may factor communication costs into the decision-making. However, the agents do not actively reason about other agent's observed behavior, nor about the communication process. Moreover, policies are used as opposed to online reasoning about models of domains and of agent behavior.

The reasoning techniques used in the present work have already been successfully applied to domains ranging from complex cyber-physical systems to workforce scheduling. To the best of our knowledge, however, they have never been applied to domains combining realistic communications and multiple agents.

Finally, high-fidelity multi-agent simulators (e.g., AgentFly (David Sislak and Pechoucek 2012)) do not account for network dynamism nor provide a realistic network model. For this

reason, we base our simulator on the Common Open Research Emulator (CORE) (Ahrenholz 2010). CORE provides network models in which communications are neither reliable nor free.

3 Motivating Scenario

To motivate the need for network-aware reasoning and mission-aware networking, consider a simple UAV coordination problem, depicted in Figure 2, in which two UAVs are tasked with taking pictures of a set of three targets, and with relaying the information to a home base.

Fixed relay access points extend the communications range of the home base. The UAVs can share images of the targets with each other and with the relays when they are within radio range. The simplest solution to this problem consists in entirely disregarding the networking component of the scenario, and generating a mission plan in which each UAV flies to a different set of targets, takes pictures of them, and flies back to the home base, where the pictures are transferred. This solution, however, is not satisfactory. First of all, it is inefficient, because it requires that the UAVs fly all the way back to the home base before the images can be used. The time it takes for the UAVs to fly back may easily render the images too outdated to be useful. Secondly, disregarding the network during the reasoning process may lead to mission failure — especially in the case of unexpected events, such as enemy forces blocking transit to and from the home base after a UAV has reached a target. Even if the UAVs are capable of autonomous behavior, they will not be able to complete the mission unless they take advantage of the network.

Another common solution consists of acknowledging the availability of the network, and assuming that the network is constantly available throughout plan execution. A corresponding mission plan would instruct each UAV to fly to a different set of targets, and take pictures of them, while the network relays the data back to the home base. This solution is *optimistic* in that it assumes that the radio range is sufficient to reach the area where the targets are located, and that the relays will work correctly throughout the execution of the mission plan.

This optimistic solution is more efficient than the previous one, since the pictures are received by the home base soon after they are taken. Under realistic conditions, however, the strong assumptions it relies upon may easily lead to mission failure—for example, if the radio range does not reach the area where the targets are located.

In this work, the reasoning processes take into account not only the presence of the network, but also its configuration and characteristics, taking advantage of available resources whenever possible. The mission planner is given information about the radio range of the relays and determines, for example, that the targets are out of range. A possible mission plan constructed by this information into account consists in having one UAV fly to the targets and take pictures, while the other UAV remains in a position to act as a network bridge between the relays and the UAV that is taking pictures. This solution is as efficient as the optimistic solution presented earlier, but is more robust, because it does not rely on the same strong assumptions.

Conversely, when given a mission plan, an intelligent network middleware service capable of sensing conditions and modifying network parameters (e.g., modify network routes, limit bandwidth to certain applications, and prioritize network traffic) is able to adapt the network to provide optimal communications needed during plan execution. A relay or UAV running such a middleware is able to interrupt or limit bandwidth given to other applications to allow the other UAV to transfer images and information toward home base. Without this traffic prioritization, network capacity could be reached prohibiting image transfer.

4 Technical Approach

In this section, we formulate the problem in more details, discuss the design of the agent architecture and of the reasoning modules, and demonstrate the sophistication of the resulting behavior of the agents in two scenarios. We assume familiarity with ASP, and refer the reader to (Gelfond and Lifschitz 1991; Niemelä and Simons 2000; Baral 2003) for an introduction on the topic.

4.1 Problem Formulation

A problem instance for coordinating UAVs to observe targets and deliver information (e.g., images) to a home base is defined by a set of UAVs, $u_1, u_2, \ldots$, a set of targets, $t_1, t_2, \ldots$, a (possibly empty) set of fixed radio relays, $r_1, r_2, \ldots$, and a home base. The UAVs, the relays, and the home base are called radio nodes (or network nodes). Two nodes are in radio contact if they are within a distance ρ from each other, called radio range[1], or if they can relay information to each other through intermediary radio nodes that are themselves within radio range. The UAVs are expected to travel from the home base to the targets to take pictures of the targets and deliver them to the home base. A UAV will automatically take a picture when it reaches a target. If a UAV is within radio range of a radio node, the pictures are automatically shared. From the UAVs' perspective, the environment is only partially observable. Features of the domain that are observable to a UAV u are (1) which radio nodes u can and cannot communicate with by means of the network, and (2) the position of any UAV that is near u.

The goal is to have the UAVs take a picture of each of the targets so that (1) the task is accomplished as quickly as possible, and (2) the total "staleness" of the pictures is as small as possible. Staleness is defined as the time elapsed from the moment a picture is taken, to the moment it is received by the home base. While the UAVs carry on their tasks, the relays are expected to actively prioritize traffic over the network in order to ensure mission success and further reduce staleness.

4.2 Agent Architecture

The architecture used in this project follows the BDI agent model (Rao and Georgeff 1991; Wooldridge 2000), which provides a good foundation because of its logical underpinning, clear structure and flexibility. In particular, we build upon ASP-based instances of this model (Baral and Gelfond 2000; Balduccini and Gelfond 2008) because they employ directly-executable logical languages featuring good computational properties while at the same time ensuring elaboration tolerance (McCarthy 1998) and elegant handling of incomplete information, non-monotonicity, and dynamic domains.

A sketch of the information flow throughout the system is shown in Figure 1a.[2] Initially, a centralized *mission planner* is given a description of the domain and of the problem instance, and finds a plan that uses the available UAVs to achieve the goal.

Next, each UAV receives the plan and begins executing it individually. As plan execution unfolds, the communication state changes, potentially affecting network connectivity. For example,

[1] For simplicity, we assume that all the radio nodes use comparable network devices, and that thus ρ is unique throughout the environment.

[2] The tasks in the various boxes are executed only when necessary.

the UAVs may move in and out of range of each other and of the other network nodes. Unexpected events, such as relays failing or temporarily becoming disconnected, may also affect network connectivity. When that happens, each UAV reasons in a decentralized, autonomous fashion to overcome the issues. As mentioned earlier, the key to taking into account, and hopefully compensating for, any unexpected circumstances is to actively employ, in the reasoning processes, realistic and up-to-date information about the communications state.

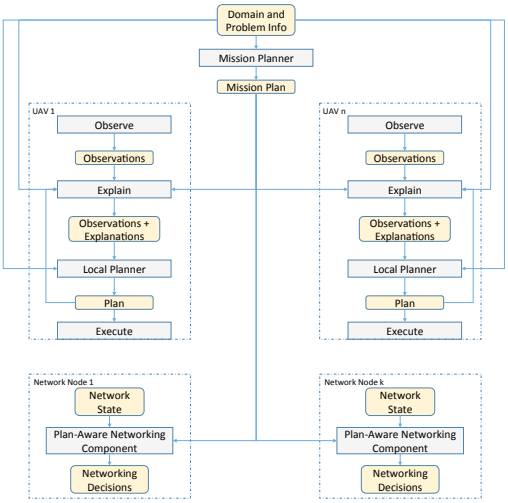
The control loop used by each UAV is shown in Figure 1b. In line with (Gelfond and Lifschitz 1991; Marek and Truszczynski 1999; Baral 2003), the loop and the I/O functions are implemented procedurally, while the reasoning functions (*Goal_Achieved*, *Unexpected_Observations*, *Explain_Observations*, *Compute_Plan*) are implemented in ASP. The loop takes in input the mission goal and the mission plan, which potentially includes courses of actions for multiple UAVs. Functions New_Observations, Next_Action, Tail, Execute, Record_Execution perform basic manipulations of data structures, and interface the agent with the execution and perception layers. Functions *Next_Action* and Tail are assumed to be capable of identifying the portions of the mission plan that are relevant to the UAV executing the loop. The remaining functions occurring in the control loop implement the reasoning tasks. Central to the architecture is the maintenance of a history of past observations and actions executed by the agent. Such history is stored in variable H and updated by the agent when it gathers observations about its environment and when it performs actions. It is important to note that variable H is local to the specific agent executing the loop, rather than shared among the UAVs (which would be highly unrealistic in a communication-constrained environment). Thus, different agents will develop differing views of the history of the environment as execution unfolds. At a minimum, the difference will be due to the fact that agents cannot observe each other's actions directly, but only their consequences, and even those are affected by the partial observability of the environment.

Details on the control loop can be found in (Balduccini and Gelfond 2008). With respect to that version of the loop, the control loop used in the present work does not allow for the selection of a new goal at run-time, but it extends the earlier control loop with the ability to deal with, and reason about, an externally-provided, multi-agent plan, and to reason about other agents' behavior. We do not expect run-time selection of goals to be difficult to embed in the control loop presented here, but doing so is out of the scope of the current phase of the project.

4.3 Network-Aware Reasoning

The major reasoning tasks (centralized mission planning, as well as anomaly detection, explanation and planning within each agent) are reduced to finding models of answer-set based formalizations of the corresponding problems. Central to all the reasoning tasks is the ability to represent the evolution of the environment over time. Such evolution is conceptualized into a *transition diagram* (Gelfond and Lifschitz 1993), a graph whose nodes correspond to states of the environment, and whose arcs describe state transitions due to the execution of actions. Let $\mathscr{F}$ be a collection of *fluents*, expressions representing relevant properties of the domain that may change over time, and let $\mathscr{A}$ be a collection of *actions*. A *fluent literal* l is a fluent $f \in \mathscr{F}$ or its negation $\neg f$. A *state* σ is a complete and consistent set of fluent literals.

The transition diagram is formalized in ASP by rules describing the direct effects of actions, their executability conditions, and their indirect effects (also called state constraints). The succession of moments in the evolution of the environment is characterized by discrete *steps*, associated with non-negative integers. The fact that a certain fluent f is true at a step s is encoded by an



Input: M: mission plan;
G: mission goal;
Vars: H : history;
P: current plan;

$P := M$;
$H :=$ New_Observations();
while $\neg$Goal_Achieved(H, G) **do**
  if Unexpected_Observations(H) **then**
    $H :=$ Explain_Observations(H);
    $P :=$ Compute_Plan(G, H, P);
  end if
  $A :=$ Next_Action(P);
  $P :=$ Tail(P);
  Execute(A);
  $H :=$ Record_Execution(H, A);
  $H := H \cup$ New_Observations();
loop

Fig. 1: (a) Information flow *(left)*; (b) Agent Control Loop *(right)*.

atom $h(f,s)$. If f is false, this is expressed by $\neg h(f,s)$. The occurrence of an action $a \in \mathscr{A}$ at step s is represented as $o(a,s)$.

The history of the environment is formalized in ASP by two types of statements: $obs(f,true,s)$ states that f was observed to be true at step s (respectively, $obs(f,false,s)$ states that f was false); $hpd(a,s)$ states that a was observed to occur at s. Because in the this paper other agents' actions are not observable, the latter expression is used only to record an agent's own actions.

Objects in the UAV domain discussed in this paper are the home base, a set of fixed relays, a set of UAVs, a set of targets, and a set of waypoints. The waypoints are used to simplify the path-planning task, which we do not consider in the present work. The locations that the UAVs can occupy and travel to are the home base, the waypoints, and the locations of targets and fixed relays. The current location, l, of UAV u is represented by a fluent $at(u,l)$. For each location, the collection of its neighbors is defined by relation $next(l,l')$. UAV motion is restricted to occur only from a location to a neighboring one. The direct effect of action $move(u,l)$, intuitively stating that UAV u moves to location l, and its executability condition are described by the following rules:

$$h(at(U,L2),S+1) \leftarrow o(move(U,L2),S), h(at(U,L1),S), next(L1,L2).$$

$$\leftarrow o(move(U,L2),S), h(at(U,L1),S), \text{not } next(L1,L2).$$

The fact that two radio nodes are in radio contact is encoded by fluent $in\_contact(r_1, r_2)$. The next two rules provide a recursive definition of the fluent, represented by means of state constraints:

$$h(in\_contact(R1,R2),S) \leftarrow$$
$$R1 \neq R2,\ \neg h(down(R1),S),\ \neg h(down(R2),S),$$
$$h(at(R1,L1),S),\ h(at(R2,L2),S),\ range(Rg),\ dist2(L1,L2,D),\ D \leqslant Rg^2.$$

$$h(in\_contact(R1,R3),S) \leftarrow$$
$$R1 \neq R2,\ R2 \neq R3,\ R1 \neq R3,\ \neg h(down(R1),S),\ \neg h(down(R2),S),$$
$$h(at(R1,L1),S),\ h(at(R2,L2),S),$$
$$range(Rg),\ dist2(L1,L2,D),\ D \leqslant Rg^2,$$
$$h(in\_contact(R2,R3),S).$$

The first rule defines the base case of two radio nodes that are directly in range of each other. Relation $dist2(l_1, l_2, d)$ calculates the square of the distance between two locations. Fluent $down(r)$ holds if radio r is known to be out-of-order, and a suitable axiom (not shown) defines the closed-world assumption on it. In the formalization, $in\_contact(R1,R2)$ is a *defined positive fluent*, i.e., a fluent whose truth value, in each state, is completely defined by the current value of other fluents, and is not subject to inertia. The formalization of $in\_contact(R1,R2)$ is thus completed by a rule capturing the closed-world assumption on it:

$$\neg h(in\_contact(R1,R2),S) \leftarrow$$
$$R1 \neq R2,$$
$$\text{not } h(in\_contact(R1,R2),S).$$

Functions Goal_Achieved and Unexpected_Observations, in Figure 1a, respectively check if the goal has been achieved, and whether the history observed by the agent contains any unexpected observations. Following the definitions from (Balduccini and Gelfond 2003), observations are unexpected if they contradict the agent's expectations about the corresponding state of the environment. This definition is captured by the *reality-check axiom*, consisting of the constraints:

$$\leftarrow obs(F,true,S),\ \neg h(F,S).$$
$$\leftarrow obs(F,false,S),\ h(F,S).$$

Function Explain_Observations uses a diagnostic process along the lines of (Balduccini and Gelfond 2003) to identify a set of exogenous actions (actions beyond the control of the agent that may occur unobserved), whose occurrence explains the observations. To deal with the complexities of reasoning in a dynamic, multi-agent domain, the present work extends the previous results on diagnosis by considering multiple types of exogenous actions, and preferences on the resulting explanations. The simplest type of exogenous action is $break(r)$, which occurs when radio node r breaks. This action causes fluent $down(r)$ to become true. Actions of this kind may be used to explain unexpected observations about the lack of radio contact. However, the agent must also be able to cope with the limited observability of the position and motion of the other agents. This is accomplished by encoding commonsensical statements (encoding omitted) about the behavior of other agents, and about the factors that may affect it. The first such statement says that *a UAV will normally perform the mission plan, and will stop performing actions when its portion of the mission plan is complete.* Notice that a mission plan is simply a sequence of actions. There is no need to include pre-conditions for the execution of the actions it contains, because those can be easily identified by each agent, at execution time, from the formalization of the domain.

The agent is allowed to hypothesize that *a UAV may have stopped executing the mission plan* (for example, if the UAV malfunctions or is destroyed). *Normally, the reasoning agent will expect a UAV that aborts execution to remain in its latest location.* In certain circumstances, however, a UAV may need to deviate completely from the mission plan. To accommodate for this situation, the agent may hypothesize that *a UAV began behaving in an unpredictable way (from the agent's point of view) after aborting plan execution.* The following choice rule allows an agent to consider all of the possible explanations:

$$\{\ hpd(break(R),S), hpd(aborted(U,S)), hpd(unpredictable(U,S))\ \}.$$

A constraint ensures that unpredictable behavior can be considered only if a UAV is believed to have aborted the plan. If that happens, the following choice rule is used to consider all possible courses of actions from the moment the UAV became unpredictable to the current time step.

$$\{hpd(move(U,L),S') : S' \geqslant S : S' < currstep\} \leftarrow hpd(unpredictable(U,S)).$$

In practice, such a thought process is important to enable coordination with other UAVs when communications between them are impossible, and to determine the side-effects of the inferred courses of actions and potentially take advantage of them (e.g., "the UAV must have flown by target t_3. Hence, it is no longer necessary to take a picture of t_3"). A *minimize* statement ensures that only cardinality-minimal diagnoses are found:

$$\#minimize[hpd(break(R),S), hpd(aborted(U,S)), hpd(unpredictable(U,S))].$$

An additional effect of this statement is that the reasoning agent will prefer simpler explanations, which assume that a UAV aborted the execution of the mission plan and stopped, over those hypothesizing that the UAV engaged in an unpredictable course of actions.

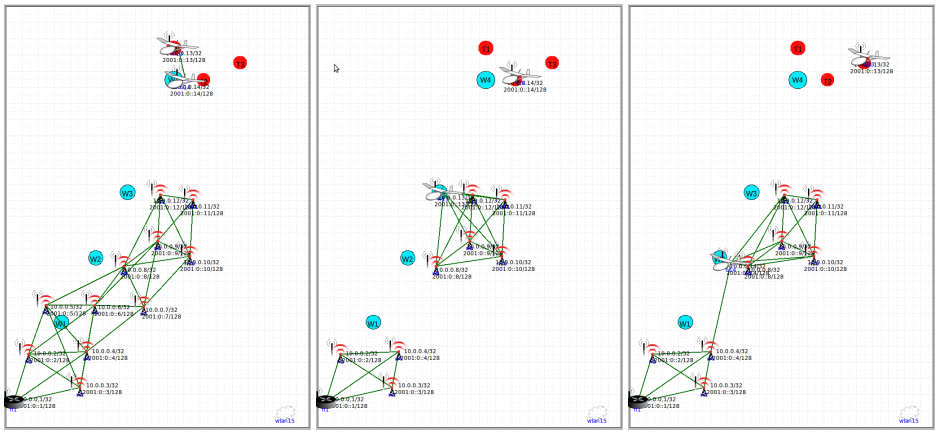
Function Compute_Plan, as well as the mission planner, compute a new plan using a rather traditional approach, which relies on a choice rule for generation of candidate sequences of actions, constraints to ensure the goal is achieved, and *minimize* statements to ensure optimality of the plan with respect to the given metrics.

Next, we outline a scenario demonstrating the features of our approach, including the ability to work around unexpected problems autonomously.

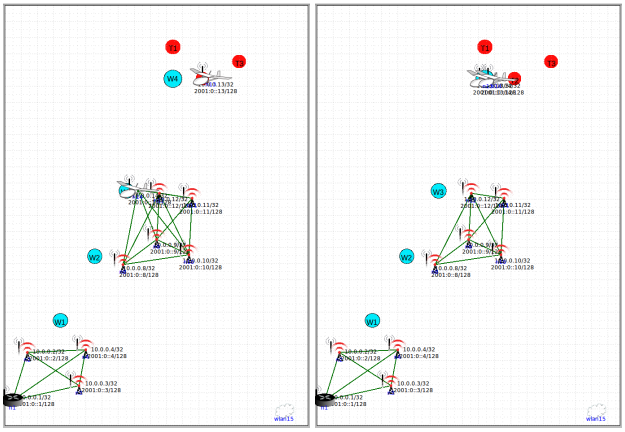
Example Instance. Consider the environment shown in in Figure 2. Two UAVs, u_1 and u_2 are initially located at the home base in the lower left corner. The home base, relays and targets are positioned as shown in the figure, and the radio range is set to 7 grid units.

The mission planner finds a plan in which the UAVs begin by traveling toward the targets. While u_1 visits the first two targets, u_2 positions itself so as to be in radio contact with u_1 (Figure 2a). Upon receipt of the pictures, u_2 moves within range of the relays to transmit the pictures to the home base. At the same time, u_1 flies toward the final target, where it will be reached by u_2 to exchange the final picture.

Now let us consider the impact of unexpected events during mission execution: while u_2 is flying back to re-connect with the relays, it observes ("Observe" step of the architecture from Figure 1) that the home base is unexpectedly not in radio contact (Figure 2b). Hence, u_2 uses the available observations to determine plausible causes ("Explain" step of the architecture). In this instance, u_2 observes that relays r_5, r_6, r_7 and all the network nodes South of them are not reachable via the network. Based on knowledge of the layout of the network, u_2 determines that the simplest plausible explanation is that those three relays must have stopped working while



Step 5: u_1 transmitting to u_2. Step 6: Nodes 5-7 have failed. Step 7: u_2 re-plans, moves closer to home base.



Step 8: u_2 moves toward u_1. Step 9: u_2 and u_1 reconnect and move back toward home base.

Fig. 2: Re-planning after relay node failure between steps 5 and 6 forcing the UAVs to re-plan.

u_2 was out of radio contact (e.g., started malfunctioning or have been destroyed). Next, u_2 re-plans ("Local Planner" step of the architecture). *The plan is created based on the assumption that u_1 will continue executing the mission plan. This assumption can be later withdrawn if observations prove it false.* Following the new plan, u_2 moves further South towards the home base (Figure 2c). Simultaneously, u_1 continues with the execution of the mission plan, unaware that the connectivity has changed and that u_2 has deviated from the mission plan. After successfully relaying the pictures to the home base, u_2 moves back towards u_1. UAV u_1, on the other hand, reaches the expected rendezvous point, and observes that u_2 is not where expected (Figure 2d). UAV u_1 does not know the actual position of u_2, but its absence is evidence that u_2 must have deviated from the plan at some point in time. Thus, u_1's must now replan. Not knowing u_2's state, u_1's plan is to fly South to relay the missing picture to the home base on its own. This plan still does not deal with the unavailability of r_5, r_6, r_7, since u_1 has not yet had a chance to get in radio contact with the relays and observe the current network connectivity state. The two UAVs continue with the execution of their new plans and eventually meet, unexpectedly for both

(Figure 2e). At that point, they automatically share the final picture. Both now determine that the mission can be completed by flying South past the failed relays, and execute the corresponding actions.

5 Conclusion and Future Work

This paper discussed a novel application of an ASP-based intelligent agent architecture to the problem of UAV coordination while taking into account network communications. Our work demonstrates the advantages deriving from such network-aware reasoning. In on-going experimental evaluations, our approach yielded a reduction in mission length of up to 30% and in total staleness between 50% and 100%. We expect that, in more complex scenarios, the advantage of a realistic networking model will be even more evident.

References

AHRENHOLZ, J. 2010. Comparison of CORE network emulation platforms. In *IEEE Military Communications Conf.*

BALDUCCINI, M. AND GELFOND, M. 2003. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP) 3*, 4–5 (Jul), 425–461.

BALDUCCINI, M. AND GELFOND, M. 2008. The AAA Architecture: An Overview. In *AAAI Spring Symp.: Architectures for Intelligent Theory-Based Agents.*

BARAL, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press.

BARAL, C. AND GELFOND, M. 2000. Reasoning Agents In Dynamic Domains. In *Workshop on Logic-Based Artificial Intelligence.* Kluwer Academic Publishers, 257–279.

BLOUNT, J., GELFOND, M., AND BALDUCCINI, M. 2014. Towards a Theory of Intentional Agents. In *Knowledge Representation and Reasoning in Robotics.* AAAI Spring Symp. Series.

DAVID SISLAK, PREMYSL VOLF, S. K. AND PECHOUCEK, M. 2012. *AgentFly: Scalable, High-Fidelity Framework for Simulation, Planning and Collision Avoidance of Multiple UAVs.* Wiley Inc., Chapter 9, 235–264.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9*, 365–385.

GELFOND, M. AND LIFSCHITZ, V. 1993. Representing Action and Change by Logic Programs. *Journal of Logic Programming 17*, 2–4, 301–321.

KOPEIKIN, A. N., PONDA, S. S., JOHNSON, L. B., AND HOW, J. P. 2013. Dynamic Mission Planning for Communication Control in Multiple Unmanned Aircraft Teams. *Unmanned Systems 1*, 1, 41–58.

MAREK, V. W. AND TRUSZCZYNSKI, M. 1999. *The Logic Programming Paradigm: a 25-Year Perspective.* Springer Verlag, Berlin, Chapter Stable Models and an Alternative Logic Programming Paradigm, 375–398.

MCCARTHY, J. 1998. Elaboration Tolerance.

NIEMELÄ, I. AND SIMONS, P. 2000. *Logic-Based Artificial Intelligence.* Kluwer Academic Publishers, Chapter Extending the Smodels System with Cardinality and Weight Constraints.

PYNADATH, D. V. AND TAMBE, M. 2002. The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models. *JAIR 16*, 389–423.

RAO, A. S. AND GEORGEFF, M. P. 1991. Modeling Rational Agents within a BDI-Architecture. In *Proc. of the Int'l Conf. on Principles of Knowledge Representation and Reasoning.*

USBECK, K., CLEVELAND, J., AND REGLI, W. C. 2012. Network-centric ied detection planning. *IJIDSS 5*, 1, 44–74.

WOOLDRIDGE, M. 2000. *Reasoning about Rational Agents.* MIT Press.

A Well-Founded Semantics for FOL-Programs

Yi Bi[1], Jia-Huai You[2], Zhiyong Feng[1]
[1]*Tianjin University, Tianjin China*
[2]*University of Alberta, Edmonton T6G 2E8, Canada*

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

An FOL-program consists of a background theory in a decidable fragment of first-order logic and a collection of rules possibly containing first-order formulas. The formalism stems from recent approaches to tight integrations of ASP with description logics. In this paper, we define a well-founded semantics for FOL-programs based on a new notion of unfounded sets on consistent as well as inconsistent sets of literals, and study some of its properties. The semantics is defined for all FOL-programs, including those where it is necessary to represent inconsistencies explicitly. The semantics supports a form of combined reasoning by rules under closed world as well as open world assumptions, and it is a generalization of the standard well-founded semantics for normal logic programs. We also show that the well-founded semantics defined here approximates the well-supported answer set semantics for normal DL programs.

KEYWORDS: Logic Programs, Well-Founded Semantics, First-Order Logic.

1 Introduction

Recent literature has shown extensive interests in combining ASP with fragments of classical logic, such as description logics (DLs) (see, e.g., (de Bruijn et al. 2008; de Bruijn et al. 2007; Eiter et al. 2008; Lukasiewicz 2010; Motik and Rosati 2010; Rosati 2005; Rosati 2006; Shen and Wang 2011; Yang et al. 2011)). A program in this context is a combined knowledge base $KB = (L, \Pi)$, where L is a knowledge base of a decidable fragment of first-order logic and Π a set of rules possibly containing first-order formulas or interface facilities. In this paper, we use *FOL-program* as an umbrella term for approaches that allow first-order formulas to appear in rules (the so-called *tight* integration), for generality. The goal of this paper is to formulate a well-founded semantics for these programs with the following features.

- The class of all FOL-programs are supported.
- Combined reasoning with closed world as well as open world assumptions is supported.

Under the first feature, we shall allow an atom with its predicate shared in the first-order theory L to appear in a rule head. This can result in two-way flow of information and enable inferences within each component automatically. For example, assume L contains a formula that says students are entitled to educational discount, $\forall x\, St(x) \supset EdDiscount(x)$. Using the notation of DL, we would write $St \sqsubseteq EdDiscount$. Suppose in an application anyone who is not employed full time but registered for an evening class is given the benefit of a student. We can write a rule

$$St(X) \leftarrow EveningClass(X), not\ HasJob(X).$$

Thus, that such a person enjoys educational discount can be inferred directly from the underlying knowledge base L.

To support all FOL-programs, we need to consider the possibility of inconsistencies arising in the construction of the intended well-founded semantics. For example, consider an FOL-program, $KB = (L, \Pi)$, where $L = \{\forall x A(x) \supset C(x), \neg C(a)\}$ and $\Pi = \{A(a) \leftarrow not\ B(a);$ $B(a) \leftarrow B(a)\}$. Suppose the Herbrand base is $\{A(a), B(a)\}$. In an attempt to compute the well-founded semantics of KB by an iterative process, we begin with the empty set; while L entails $\neg A(a)$, since $B(a)$ is false by closed world reasoning, we derive $A(a)$ resulting in an inconsistency. This reasoning process suggests that during an iterative process a consistent set of literals may be mapped to an inconsistent one and, in general, whether inconsistencies arise or not is not known *a priori* without actually performing the computation.

That the well-founded semantics of an FOL-program is defined by an inconsistent set can be useful on its own, or in the computation of (suitably defined) answer sets of the program. If we have computed the well-founded semantics which is inconsistent, we need not pursue the task of computing answer sets of the same program, because they do not exist.

In complex real world reasoning by rules, it is sometimes desirable that not all predicates are reasoned with under the closed world assumption. Some conditions may need to be established under the open world assumption. We call this *combined reasoning*. For example, we may write a rule

$$PrescribeTo(X, Q) \leftarrow Effective(X, Z), Contract(Q, Z), \neg AllergicTo(Q, X)$$

to describe that an antibiotic is prescribed to a patient who contracted a bacterium, if the antibiotic against that bacterium is effective and patient is not allergic to it. Though *Effective* can be reasoned with under the closed world assumption, it may be preferred to judge whether a patient is not allergic to an antibiotic under the open world assumption, e.g., it holds if it can be proved classically. This is in contrast with closed world reasoning whereas one may infer nonallergic due to lack of evidence for allergy.

To our knowledge, there has been no well-founded semantics defined for all FOL-programs. The closest that one can find is the definition for a subset of FOL-programs (Lukasiewicz 2010), which relies on syntactic restrictions so that the least fixpoint is computed over consistent sets of literals. To ensure that the construction is well-defined, it is assumed that DL axioms must be, or can be converted to, *tuple generating dependencies* (which are essentially Horn rules) plus constraints. Thus, the approach cannot be lifted to handle first-order formulas in general. In addition, to the best of our knowledge, no combined reasoning is ever supported under any well-founded semantics.

As motivated above, in this paper we first define a well-founded semantics for FOL-programs based on a new notion of unfounded sets. We show that the semantics generalizes the well-founded semantics for normal logic programs. Also, we prove that the well-founded semantics defined here approximates the well-supported answer set semantics for the language of (Shen and Wang 2011); namely, all well-founded atoms (resp. unfounded atoms) of a program remain to be true (resp. false) in any well-supported answer set. This makes it possible to use the mechanism of constructing the well-founded semantics as constraint propagation in an implementation of computing well-supported answer sets.

The paper is organized as follows. The next section introduces the language and notations. In Section 3 we define the well-founded semantics. Section 4 studies some properties and relates

to the well-supported answer set semantics, followed by related work in Section 5. Section 6 concludes the paper and points to future directions.

2 Language and Notation

We assume a language of a decidable fragment of first-order logic, denoted $\mathscr{L}_\Sigma$, where $\Sigma = \langle F^n; P^n \rangle$, called a signature, and F^n and P^n are disjoint countable sets of n-ary function and n-ary predicate symbols, respectively. Constants are 0-ary functions. *Terms* are variables, constants, or functions in the form $f(t_1, \ldots, t_n)$, where each t_i is a term and $f \in F^n$. *First-order formulas*, or just *formulas*, are defined as usual, so are the notions of *satisfaction*, *model*, and *entailment*.

Let Φ_P be a finite set of predicate symbols and Φ_C a nonempty finite set of constants such that $\Phi_C \subseteq F^n$. An *atom* is of the form $P(t_1, \ldots, t_n)$ where $P \in \Phi_P$ and each t_i is either a constant from Φ_C or a variable. A *negated atom* is of the form $\neg A$ where A is an atom. We do not assume any other restriction on the vocabularies, that is, Φ_P and P^n may have predicate symbols in common.

An *FOL-program* is a combined knowledge base $KB = (L, \Pi)$, where L is a first-order theory of $\mathscr{L}_\Sigma$ and Π a *rule base*, which is a finite collection of rules of the form

$$H \leftarrow A_1, \ldots, A_m, not\ B_1, \ldots, not\ B_n \tag{1}$$

where H is an atom, and A_i and B_i are atoms or formulas. By abuse of terminology, each A_i is called a *positive literal* and each *not* B_i is called a *negative literal*.

For any rule r, we denote by $head(r)$ the head of the rule and $body(r)$ its body, and we define $pos(r) = \{A_1, \ldots, A_m\}$ and $neg(r) = \{B_1, \ldots, B_n\}$.

A *ground instance* of a rule r in Π is obtained by replacing every free variable with a constant from Φ_C. In this paper, we assume that a rule base Π is already grounded if not said otherwise. When we refer to an atom/literal/formula, by default we mean it is a ground one.

Given an FOL-program $KB = (L, \Pi)$, the *Herbrand base* of Π, denoted HB_Π, is the set of all ground atoms $P(t_1, \ldots, t_n)$, where $P \in \Phi_P$ occurs in KB and $t_i \in \Phi_C$.

We denote by Ω the set of all predicate symbols appearing in HB_Π such that $\Omega \subseteq P^n$. For distinction, we call atoms whose predicate symbols are not in Ω *ordinary*, and all the other formulas *FOL-formulas*. If $L = \emptyset$ and Π only contains rules of the form (1) where all H, A_i and B_j are ordinary atoms, then KB is called a *normal logic program*.

Any subset $I \subseteq HB_\Pi$ is called an *interpretation* of Π. It is also called a *total* interpretation or a *2-valued* interpretation. If I is an interpretation, we define $\bar{I} = HB_\Pi \backslash I$.

Let Q be a set of atoms. We define $\neg.Q = \{\neg A \mid A \in Q\}$. For a set of atoms and negated atoms S, we define $S^+ = \{A \mid A \in S\}$, $S^- = \{A \mid \neg A \in S\}$, and $S|_\Omega = \{A \in S \mid pred(A) \in \Omega\}$, where $pred(A)$ is the predicate symbol of A. Let $Lit_\Pi = HB_\Pi \cup \neg.HB_\Pi$. A subset $S \subseteq Lit_\Pi$ is consistent if $S^+ \cap S^- = \emptyset$. For a first-order theory L, we say that $S \subseteq Lit_\Pi$ is *consistent with* L if the first-order theory $L \cup S|_\Omega$ is consistent (i.e., the theory is *satisfiable*). Note that when we say S is consistent with L, both S and L must be consistent. Similarly, a (2-valued) interpretation I is consistent with L if $L \cup I|_\Omega \cup \neg.\bar{I}|_\Omega$ is consistent. We denote by Lit^c_Π the set of all consistent subsets of Lit_Π. For any $S \in Lit^c_\Pi$, S' is called a *consistent extension* of S if $S \subseteq S' \in Lit^c_\Pi$.

Definition 1

Let $KB = (L, \Pi)$ be an FOL-program and $I \subseteq HB_\Pi$ an interpretation. Define the satisfaction relation under L, denoted $\models_L$, as follows (the definition extends to conjunctions of literals):

1. For any ordinary atom $A \in HB_\Pi$, $I \models_L A$ if $A \in I$ and $I \models not\ A$ if $A \notin I$.
2. For any FOL-formula A, $I \models_L A$ if $L \cup I|_\Omega \cup \neg.\bar{I}|_\Omega \models A$, and $I \models_L not\ A$ if $I \not\models_L A$.

Let $KB = (L, \Pi)$ be an FOL-program. For any $r \in \Pi$ and $I \subseteq HB_\Pi$, $I \models_L r$ if $I \not\models_L body(r)$ or $I \models_L head(r)$. I is a *model* of KB if I is consistent with L and I satisfies all rules in Π.

Example 1
To illustrate the flexibility provided by the parameter Ω, suppose we have a program $KB = (L, \Pi)$ where Π contains a rule that says any unemployed with disability receives financial assistance, with an FOL-formula in the body

$$Assist(X) \leftarrow Disabled(X), not\ Employed(X)$$

Assume $\Omega = \Phi_P = \{Assist, Employed\}$. Then, $Employed$ is interpreted under the closed world assumption and $Disabled$ under the open world assumption. Indeed, unemployment can be established by closed world reasoning for lack of evidence of employment, but disability requires a direct proof.

3 Well-Founded Semantics

We first define the notion of unfounded set. Intuitively, the atoms in an unfounded set can be safely assigned to false, due to persistent inability to derive their positive counterparts.

Definition 2
(Unfounded set) Let $KB = (L, \Pi)$ be an FOL-program and $I \subseteq Lit_\Pi$. If $I \cup L$ is consistent, then a set $U \subseteq HB_\Pi$ is an *unfounded set* of KB relative to I iff for every $H \in U$ and $r \in \Pi$, both of the following conditions are satisfied

(a) If $head(r) = H$, then
  (i) $\neg A \in I \cup \neg.U$ for some ordinary atom $A \in pos(r)$, or
  (ii) $B \in I$ for some ordinary atom $B \in neg(r)$, or
  (iii) for some FOL-formula $A \in pos(r)$, it holds that $L \cup S|_\Omega \not\models A$, for all $S \in Lit_\Pi^c$ with $I \cup \neg.U \subseteq S$, or
  (iv) for some FOL-formula $A \in neg(r)$, $L \cup I|_\Omega \models A$.

(b) $L \cup S|_\Omega \not\models H$ for all $S \in Lit_\Pi^c$ with $I \cup \neg.U \subseteq S$.

If $I \cup L$ is inconsistent, the unfounded set of KB relative to I is HB_Π.

That H is unfounded relative to I if both conditions (a) and (b) are satisfied when $I \cup L$ is consistent; in particular, condition (a.iii) ensures that a positive occurrence of an FOL-formula in the rule body is not entailed, for all consistent extensions of $I \cup \neg.U$; and condition (b) ensures the inability to infer its positive counterpart, independent of any rules.

An FOL-formula may contain shared predicates in Ω, and those not in Ω hence not shared. The latter are supposed to be interpreted under open world assumption. Continuing with Example 1 above, let $KB = (L, \Pi)$, where

$$L = \{\forall x\ Certified(x) \supset Disabled(x)\}$$
$$\Pi = \{Assist(a) \leftarrow Disabled(a), not\ Employed(a)\}$$

Assume $Assist, Employed \in \Omega$ while $Certified$ and $Disabled$ are not. Let $\Phi_C = \{a\}$, and thus $HB_\Pi = \{Assist(a), Employed(a)\}$. Clearly, $\{Assist(a), Employed(a)\}$ is an unfounded set relat-

ive to $I = \emptyset$, in particular because $Disabled(a)$ is not derivable under all consistent extensions of I. Note that, since $Disabled(a)$ is not in HB_Π, it is not part of an unfounded set.[1]

Lemma 1
Let $KB = (L, \Pi)$ be an FOL-program and $I \subseteq Lit_\Pi$. A set of unfounded sets of KB relative to I is closed under union, and the greatest unfounded set of KB relative to I exists, which is the union of all unfounded sets of KB relative to I.

Proof
If I is inconsistent, the claims hold trivially. For a consistent I, suppose U_1, $U_2 \subseteq HB_\Pi$ are unfounded sets (of KB relative to I), we show that $U_1 \cup U_2$ is also an unfounded set. Let $A \in U_1$. Since both (a) and (b) in Definition 2 hold for U_1 and U_2 separately, in particular, each consistent extension of $I \cup \neg.(U_1 \cup U_2)$ is a consistent extension of $I \cup \neg.U_1$, (a) and (b) also hold for $U_1 \cup U_2$. Thus $A \in U_1 \cup U_2$. By symmetry, the same argument applies to U_2. Therefore, the union of all unfounded sets is an unfounded set, which is the greatest among all unfounded sets. □

We define the operators which will be used to define the well-founded semantics.

Definition 3
Let $KB = (L, \Pi)$ be an FOL-program. Define T_{KB}, U_{KB}, Z_{KB} as mappings of $2^{Lit_\Pi} \rightarrow HB_\Pi$, and W_{KB} as a mapping of $2^{Lit_\Pi} \rightarrow 2^{Lit_\Pi}$, as follows:

(i) If $I \cup L$ is inconsistent, then $T_{KB}(I) = HB_\Pi$; otherwise, $H \in T_{KB}(I)$ iff $H \in HB_\Pi$ and either (a) or (b) below holds

  (a) some $r \in \Pi$ with $head(r) = H$ exists such that
    (1) for any ordinary atom A, $A \in I$ if $A \in pos(r)$ and $\neg A \in I$ if $A \in neg(r)$,
    (2) for any FOL-formula $A \in pos(r)$, $L \cup I|_\Omega \models A$, and
    (3) for any FOL-formula $B \in neg(r)$, $L \cup S|_\Omega \not\models B$, for all $S \in Lit^c_\Pi$ with $I \subseteq S$.

  (b) $L \cup I|_\Omega \models H$.

(ii) $U_{KB}(I)$ is the greatest unfounded set of KB relative to I.
(iii) $Z_{KB}(I) = \{A \in HB_\Pi \mid L \cup I|_\Omega \models \neg A\}$.
(iv) $W_{KB}(I) = T_{KB}(I) \cup \neg.U_{KB}(I) \cup \neg.Z_{KB}(I)$.

The operator T_{KB} is a consequence operator. An atom is a consequence, either due to a derivation via a rule (case (i.a)), or because it is entailed by L, given I (case (i.b)). In the first case, the body of such a rule should be satisfied not only by I, but by all consistent extensions of I. In the case (i.a.1) or (i.a.2), it is sufficient that the body is satisfied by I only because the classical entailment relation is monotonic. For the case (i.a.3) the condition needs to be stated explicitly.

There are two features in this definition that are non-conventional. The first is the operator Z_{KB} - interacting an FOL knowledge base with rules may result in *direct negative consequences*. In the second, all operators here are defined on all subsets of Lit_Π, including inconsistent ones.[2]

Lemma 2
The operators T_{KB}, U_{KB}, Z_{KB}, and W_{KB} are all monotonic.

[1] Placed under the context of 2-valued logic, the reasoning here is analogue to parallel circumscription (McCarthy 1980), where the predicates *Employed* and *Assist* are minimized with predicates *Certified* and *Disabled* varying.
[2] When inconsistency arises, the fixpoint operator here leads to triviality. This is the most common treatment when the underlying entailment relation is the classical one. However, we remark that this is only one possible choice.

Proof
Let $I_1 \subseteq I_2 \subseteq Lit_\Pi$ and $H \in T_{KB}(I_1)$. Since any condition in part (i) of Definition 3 that applies under I_1 applies under I_2 (including the case where one of them, or both, are inconsistent with L), thus the set of all consistent extensions of I_2 is a subset of all consistent extensions of I_1, and therefore we have $T_{KB}(I_1) \subseteq T_{KB}(I_2)$. The same argument applies to U_{KB} and Z_{KB}. Since T_{KB}, U_{KB}, and Z_{KB} are monotonic, it follows from the definition that the operator W_{KB} is also monotonic. □

As W_{KB} is monotone on the complete lattice $\langle 2^{Lit_\Pi}, \subseteq \rangle$, according to the Knaster-Tarski fixpoint theorem (Tarski 1955), its least fixpoint, $lfp(W_{KB})$, exists.

Definition 4
Let $KB = (L, \Pi)$ be an FOL-program. The *well-founded semantics* of KB (relative to Ω) is defined by $lfp(W_{KB})$.

We allow the well-founded semantics of an FOL-program to be defined by an inconsistent set, independent of how the semantics may be used. This may be utilized in the computation of answer sets. Suppose under a suitable definition of answer sets for an FOL-program KB, $lfp(W_{KB})$ approximates all answer sets of KB.[3] If the computed $lfp(W_{KB})$ is inconsistent then we know KB has no answer sets.

Example 2
Let $KB = (\{\neg A(a)\}, \Pi)$ where $\Pi = \{A(a) \leftarrow not\ B(a), B(a) \leftarrow B(a)\}$. Let $\Omega = \Phi_P = \{A, B\}$ and $\Phi_C = \{a\}$. $lfp(W_{KB})$ is constructed as follows (where $W^0_{KB} = \emptyset$ and $W^{i+1}_{KB} = W_{KB}(W^i_{KB})$, for all $i \geqslant 0$):

$$\begin{aligned} W^0_{KB} &= \emptyset, \\ W^1_{KB} &= \{\neg A(a), \neg B(a)\}, \\ W^2_{KB} &= \{\neg A(a), \neg B(a), A(a)\}, \\ W^3_{KB} &= Lit_\Pi, \\ W^4_{KB} &= W^3_{KB}. \end{aligned}$$

As a result, the well-founded semantics of KB is inconsistent. It is interesting to note that KB has a model, $\{B(a)\}$. This means that we cannot determine whether the well-founded semantics for an FOL-program is consistent or not, based on the existence of a model; when an iterative process is carried out, we have to deal with the possibility that inconsistencies may arise.

Example 3
Consider $KB = (L, \Pi)$ where $L = \{\forall x B(x) \supset A(x), \neg A(a) \vee C(a)\}$ and Π consists of

$$\begin{aligned} &B(a) \leftarrow B(a). \\ &A(a) \leftarrow (\neg C(a) \wedge B(a)). \\ &R(a) \leftarrow not\ C(a), not\ A(a). \end{aligned}$$

Let $\Phi_P = \{A, B, R\}$, $\Omega = \{A, B\}$, and $\Phi_C = \{a\}$. Hence $HB_\Pi = \{A(a), B(a), R(a)\}$. For $I_0 = \emptyset$, we have $T_{KB}(I_0) = \emptyset$, $U_{KB}(I_0) = \{B(a), A(a)\}$, and $Z_{KB} = \emptyset$. $B(a)$ is in $U_{KB}(I_0)$ because $B(a)$ is not derivable by any rule based on I_0, and $L \cup S|_\Omega \not\models B(a)$ for all $S \in Lit^c_\Pi$ with $I_0 \cup \neg.U_{KB}(I_0) \subseteq S$ (condition (b) in Definition 2). Similarly, $A(a)$ is in $U_{KB}(I_0)$. Since $C(a)$ is not derivable under

[3] We will show later in this paper that the well-supported answer sets of (Shen and Wang 2011) fall into this category.

all consistent extensions, we derive $R(a)$. Therefore, $lfp(W_{KB}) = \{\neg B(a), \neg A(a), R(a)\}$. Note that since $C(a) \notin HB_\Pi$ its truth value is not part of the well-founded semantics.

4 Properties and Relations

We first show that the well-founded semantics is a generalization of the well-founded semantics for normal logic programs.

Theorem 1
Let $KB = (\emptyset, \Pi)$ be a normal logic program. Then, the *WFS* of *KB* coincides with the *WFS* of Π.

Proof
The *WFS* of a normal program Π is defined by the least fixpoint of a monotone operator W_Π on the set of consistent subsets of $HB_\Pi \cup \neg.HB_\Pi$:

- $T_\Pi(S) = \{head(r) \mid r \in \Pi, pos(r) \cup \neg.neg(r) \subseteq S\}$
- $W_\Pi(S) = T_\Pi(S) \cup \neg.U_\Pi(S)$

where $U_\Pi(S)$ is the greatest unfounded set of Π w.r.t. S. A set $U \subseteq HB_\Pi$ is an unfounded set of Π w.r.t. S, if for every $a \in U$ and every rule $r \in \Pi$ with $head(r) = a$, either (i) $\neg b \in S \cup \neg.U$ for some $b \in pos(r)$, or (ii) $b \in S$ for some atom $b \in neg(r)$.

Then, it is immediate that the notion of unfounded set and the greatest unfounded set for normal logic program $KB = (\emptyset, \Pi)$ coincide with those for Π, respectively. Note that $Z_{KB}(I) \subseteq U_{KB}(I)$ when $L = \emptyset$. It is easy to see that the operator T_{KB} for normal program $KB = (\emptyset, \Pi)$ reduces to T_Π for normal program Π. □

The well-supported answer set semantics is defined for what are called *normal DL logic programs* (Shen and Wang 2011), which applies to FOL-programs. There is however a subtle difference: in the definition of the entailment relation, the W_{KB} operator uses 3-valued evaluation while the well-supported semantics is based on the notion of 2-valued *up to satisfaction*.

Definition 5
(Up to satisfaction) Let $KB = (L, \Pi)$, l a literal, and E and I two interpretations with $E \subseteq I \subseteq HB_\Pi$. The relation E *up to* I *satisfies* l *under* L, denoted $(E, I) \models_L l$, is defined as: $(E, I) \models_L l$ if $\forall F, E \subseteq F \subseteq I$, $F \models_L l$. The definition extends to conjunctions of literals.

The entailment relation, $F \models_L l$, is based 2-valued satisfiability (cf. Def. 1), i.e., $F \models_L l$ is $L \cup F|_\Omega \cup \neg.\bar{F}|_\Omega \models l$, while in 3-valued satisfiability, $S \models_L l$ is $L \cup S|_\Omega \models l$.

Given an FOL-program $KB = (L, \Pi)$, an immediate consequence operator is defined as:

$$\mathscr{T}_{KB}(E, I) = \{head(r) \mid r \in \Pi, (E, I) \models_L body(r)\}. \quad (2)$$

The operator $\mathscr{T}_{KB}$ is monotonic on its first argument E with I fixed (Shen and Wang 2011). Thus, for any model I of *KB*, we can compute a fixpoint, denoted $\mathscr{T}^{\alpha}_{KB}(\emptyset, I)$.

Definition 6
Let $KB = (L, \Pi)$ be an FOL-program and I a model of *KB*. I is an *answer set of KB* if for every $A \in I$, either $A \in \mathscr{T}^{\alpha}_{KB}(\emptyset, I)$ or $L \cup \mathscr{T}^{\alpha}_{KB}(\emptyset, I)|_\Omega \cup \neg.\bar{I}|_\Omega \models A$.

The next theorem shows that the well-founded semantics of an FOL-program approximates its well-supported answer set semantics.

Theorem 2
Let $KB = (L, \Pi)$ be an FOL-program. Then every well-supported answer set of KB includes all atoms $H \in HB_\Pi$ that are well-founded and no atoms $H \in HB_\Pi$ that are unfounded or are in $Z_{KB}(lfp(W_{KB}))$.

Proof
To prove the assertion, it is sufficient to show that if $lfp(W_{KB})$ is consistent, then for every well-supported answer set I, all atoms in $lfp(W_{KB})$ are in I and all negated atoms in $lfp(W_{KB})$ are in $\neg.\bar{I}$.

We consider the fixpoint construction by the operators $\mathscr{T}_{KB}(\cdot, I)$ and W_{KB}. Let us use a short notation for the respective sequences by

$$\mathscr{T}^0_{KB} = \emptyset, \ldots, \mathscr{T}^{k+1}_{KB} = \mathscr{T}_{KB}(\mathscr{T}^k_{KB}, I), \ldots \tag{3}$$

$$W^0_{KB} = \emptyset, \ldots, W^{k+1}_{KB} = W(W^k_{KB}), \ldots \tag{4}$$

for all $k \geqslant 0$. Define $E_0 = \emptyset$ and $E_i = \{H \mid (\mathscr{T}^{i-1}_{KB}, I) \models_L H\}$ for all $i \geqslant 1$. We show that $W^i_{KB} \subseteq E_i \cup \mathscr{T}^i_{KB} \cup \neg.\bar{I}$, for all $i \geqslant 0$. The base case is trivial. For the inductive step, assume for any $k \geqslant 0$ the subset relation holds and we show it holds for $k+1$. The proof is conducted on two cases: **(I)** Assume an atom $H \in W^{k+1}_{KB}$ and show $H \in E_{k+1} \cup \mathscr{T}^{k+1}_{KB}$, and **(II)** Assume a negated atom $\neg H \in W^{k+1}_{KB}$ and show $\neg H \in \neg.\bar{I}$.

By definition and monotonicity of the operator $\mathscr{T}_{KB}$, $(\mathscr{T}^i_{KB}, I) \models_L E_i$, and it follows from the first-order entailment that for any atom $H \in HB_\Pi$,

$$(\mathscr{T}^i_{KB}, I) \models_L H \text{ iff } (E_i \cup \mathscr{T}^i_{KB}, I) \models_L H. \tag{5}$$

By definition,

$$W^{k+1}_{KB} = T_{KB}(W^k_{KB}) \cup \neg.U_{KB}(W^k_{KB}) \cup \neg.Z_{KB}(W^k_{KB})$$

By Proposition 1 of (Shen 2011), for any FOL-formula H,

$$(E, I) \models_L H \text{ iff } L \cup E|_\Omega \cup \neg.\bar{I}|_\Omega \models H. \tag{6}$$

By Proposition 2 (Shen 2011), for any ordinary atom H,

$$(E, I) \models_L H \text{ iff } H \in E; (E, I) \models_L not\ H \text{ iff } H \notin I. \tag{7}$$

(I) For any atom $H \in W^{k+1}_{KB}$, we have $H \in T_{KB}(W^k_{KB})$. If condition (i.b) in Definition 3 holds, we have $L \cup W^k_{KB}|_\Omega \models H$. By induction hypothesis, $L \cup (E_k \cup \mathscr{T}^k_{KB})|_\Omega \cup \neg.\bar{I}|_\Omega \models H$. Then by (6) and (5), $(\mathscr{T}^k_{KB}, I) \models_L H$. Thus $H \in E_{k+1}$. If condition (i.a) in Definition 3 holds, we consider the following four cases:

1. For any ordinary atom $A \in pos(r)$, $A \in W^k_{KB}$, thus $(\mathscr{T}^k_{KB}, I) \models_L A$ by (7) and (5).
2. For any ordinary atom $A \in neg(r)$, $\neg A \in W^k_{KB}$, thus $(\mathscr{T}^k_{KB}, I) \models_L not\ A$.
3. For any FOL-formula $A \in pos(r)$, $L \cup W^k_{KB}|_\Omega \models A$, then $(\mathscr{T}^k_{KB}, I) \models_L A$.
4. For any FOL-formula $A \in neg(r)$, $L \cup (W^k_{KB})'|_\Omega \not\models A$ for every $(W^k_{KB})'$ such that $W^k_{KB} \subseteq (W^k_{KB})' \in Lit^c_\Pi$, we have $(\mathscr{T}^k_{KB}, I) \not\models_L A$, since every total interpretation is a partial one.

Hence $H \in \mathscr{T}^{k+1}_{KB}$.

(II) For any negated atom $\neg H \in W^{k+1}_{KB}$, either $H \in U_{KB}(W^k_{KB})$ or $H \in Z_{KB}(W^k_{KB})$. For the case $H \in U_{KB}(W^k_{KB})$, if condition (b) in Definition 2 holds, we have $(\mathscr{T}^k_{KB}, I) \not\models_L H$ by (5), (6) and induction hypothesis, in addition to the fact that every total interpretation is a partial one. Then $H \notin E_{k+1}$. For condition (a) in Definition 2, we also consider the following four situations:

1. For any ordinary atom $A \in pos(r)$, $\neg A \in W^k_{KB} \cup \neg.U$, thus $(\mathscr{T}^k_{KB}, I) \not\models_L A$, by a similar argument above.
2. For any ordinary atom $A \in neg(r)$, $A \in W^k_{KB}$, thus $(\mathscr{T}^k_{KB}, I) \models_L A$.
3. For any FOL-formula $A \in pos(r)$, $L \cup (W^k_{KB})'|_\Omega \not\models A$, for every $W^k_{KB} \subseteq (W^k_{KB})' \in Lit^c_\Pi$, then $(\mathscr{T}^k_{KB}, I) \not\models_L A$, since every total interpretation is a partial one.
4. For any FOL-formula $A \in neg(r)$, $L \cup W^k_{KB}|_\Omega \models A$, we have $(\mathscr{T}^k_{KB}, I) \models_L A$.

We have $H \notin \mathscr{T}^{k+1}_{KB}$. Hence $H \notin E_{k+1} \cup \mathscr{T}^{k+1}_{KB}$

For any $\neg H \in W^k_{KB}$, $\neg H \in \neg.\bar{I}$, since the operator E and $\mathscr{T}_{KB}$ only generate positive atoms. We then have $H \notin E_k \cup \mathscr{T}^k_{KB}$. As k is arbitrary, we have $H \in U_{KB}(lfp(W_{KB}))$ and $H \notin E_\alpha \cup \mathscr{T}^\alpha_{KB}$, where E_α and $\mathscr{T}^\alpha_{KB}$ are the respective fixpoints. Since $E_\alpha \cup \mathscr{T}^\alpha_{KB} = I$ (cf. definition 6), we get $H \in \bar{I}$. Similarly, if $H \in Z_{KB}(W^k_{KB})$, then $H \in \bar{I}$. We thus have proved that $W^{k+1}_{KB} \subseteq E_{k+1} \cup \mathscr{T}^{k+1}_{KB} \cup \neg.\bar{I}$. □

5 Related Work

The most relevant work in defining well-founded semantics for combing rules with DLs are (Eiter et al. 2011; Lukasiewicz 2010). The former embeds *dl-atoms* in rule bodies to serve as queries to the underlying ontology, and it does not allow the predicate in a rule head to be shared in the ontology. In both approaches, syntactic restrictions are posted so that the least fixpoint is always constructed over sets of consistent literals. It is also a unique feature in our approach that combined reasoning with closed world and open world is supported.

A program in FO(ID) has a clear knowledge representation "task" - the rule component is used to define concepts, whereas the FO component may assert additional properties of the defined concepts. All formulas in FO(ID) are interpreted under closed world assumption. Thus, FOL-programs and FO(ID) have fundamental differences in basic ideas. On semantics, FOL-formulas can be interpreted under open world and closed world flexibly. On modeling, the rule set in FO(ID) is built on ontologies, thus information can only flow from a first order theory to rules. But in FOL-programs, the first order theory and rules are tightly integrated, and thus information can flow from each other bilaterally.

6 Conclusion and Future Directions

In this paper we have defined a new well-founded semantics for FOL-programs, where arbitrary FOL-formulas are allowed to appear in rule bodies and an atom with its predicate shared with first-order theory to appear in a rule head. Combined reasoning with closed world as well as open world is supported. Moreover, inconsistencies are dealt with explicitly, and thus the task of computing answer sets can be prejudged in case that the well-founded semantics is an inconsistent set. We have shown that the well-founded semantics is an appropriate approximation of the well-supported answer set semantics defined in (Shen and Wang 2011).

As future work, we will study the approximation fixpoint theory (AFT) (Denecker et al. 2000; Denecker et al. 2004), and investigate whether and how well-founded and stable semantics of FOL-programs can be defined uniformly under an extended approximation fixpoint theory. We are also interested in possible different approximating operators for alternative semantics of FOL-programs. In (Denecker et al. 2004) the authors show that the theory of consistent approximations can be applied to the entire bilattice $\mathscr{L}^2$ (including inconsistent elements), under the assumption

that an approximating operator $\mathscr{A}$ is *symmetric*. This symmetry assumption guarantees that no transition from a consistent state to an inconsistent one may take place. As argued at the outset of this paper, this is precisely what we cannot assume for a definition of well-founded semantics for all FOL-programs.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (NSFC) grants 61373035 and 61373165, and by National High-tech R&D Program of China (863 Program) grant 2013AA013204.

References

DE BRUIJN, J., EITER, T., AND TOMPITS, H. 2008. Embedding approaches to combining rules and ontologies into autoepistemic logic. In *Proc. KR 2008*. 485–495.

DE BRUIJN, J., PEARCE, D., POLLERES, A., AND VALVERDE, A. 2007. Quantified equilibrium logic and hybrid rules. In *Proc. RR 2007*. 58–72.

DENECKER, M., MAREK, V., AND TRUSZCZYŃSKI, M. 2000. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *Logic-based Artificial Intelligence*. 127–144.

DENECKER, M., MAREK, V., AND TRUSZCZYNSKI, M. 2004. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Information and Computation 192,* 1, 84–121.

EITER, T., IANNI, G., LUKASIEWICZ, T., SCHINDLAUER, R., AND TOMPITS, H. 2008. Combining answer set programming with description logics for the semantic web. *Artifical Intelligence 172,* 12-13, 1495–1539.

EITER, T., LUKASIEWICZ, T., IANNI, G., AND SCHINDLAUER, R. 2011. Well-founded semantics for description logic programs in the semantic web. *ACM Transactions on Computational Logic 12,* 2. Article 3.

LUKASIEWICZ, T. 2010. A novel combination of answer set programming with description logics for the semantic web. *IEEE TKDE 22,* 11, 1577–1592.

MCCARTHY, J. 1980. Circumscription - a form of non-monotonic reasoning. *Artifical Intelligence 13,* 27-39, 171–172.

MOTIK, B. AND ROSATI, R. 2010. Reconciling description logics and rules. *Journal of the ACM 57,* 5, 1–62.

ROSATI, R. 2005. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics 3,* 1, 61–73.

ROSATI, R. 2006. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. KR'06*. 68–78.

SHEN, Y.-D. 2011. Well-supported semantics for description logic programs. In *Proc. IJCAI-11*. 1081–1086.

SHEN, Y.-D. AND WANG, K. 2011. Extending logic programs with description logic expressions for the semantic web. In *Proc. International Semantic Web Conference*. 633–648.

TARSKI, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics 5:2*, 285–309.

YANG, Q., YOU, J.-H., AND FENG, Z. 2011. Integrating rules and description logics by circumscription. In *Proc. AAAI-11*.

FO(C): *A Knowledge Representation Language of Causality*

Bart Bogaerts, Joost Vennekens, Marc Denecker
Department of Computer Science, KU Leuven
E-Mail: firstname.lastname@cs.kuleuven.be

Jan Van den Bussche
Hasselt University & transnational University of Limburg
E-Mail: jan.vandenbussche@uhasselt.be

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Cause-effect relations are an important part of human knowledge. In real life, humans often reason about complex causes linked to complex effects. By comparison, existing formalisms for representing knowledge about causal relations are quite limited in the kind of specifications of causes and effects they allow. In this paper, we present the new language C-Log, which offers a significantly more expressive representation of effects, including such features as the creation of new objects. We show how C-Log integrates with first-order logic, resulting in the language FO(C). We also compare FO(C) with several related languages and paradigms, including inductive definitions, disjunctive logic programming, business rules and extensions of Datalog.

1 Introduction

Cause-effect relations are an important part of human knowledge. There exist a number of knowledge representation languages (McCain and Turner 1996; Vennekens et al. 2009; Cabalar 2012) in which logic programming style rules are used to represent such relations. The basic idea in all these approaches is that the head of such a rule represents an effect that is caused by its body. In this paper, we are particularly concerned with CP-logic (Vennekens et al. 2009). More specifically, we consider the variant of CP-logic without probabilities, and we will extend this language with three features: dynamic non-determinsitic choice; object creation; and recursive nesting of cause-effect relations. We call the resulting language C-Log. In this paper, we present C-Log and its informal semantics. For the formal semantics, we refer to an accompanying technical report (Bogaerts et al. 2014a). We also present the integration of C-Log with first-order logic, and thus show that C-Log fits in the FO(·) Knowledge Base System project (Denecker 2012).

Let us begin by recalling the guiding principles behind CP-logic. When compared to predecessors, such as the causal logic of McCain and Turner (1996), one of the important contributions of this languages is to add two modelling principles that are common in causal modelling. The first is the distinction between *endogenous* and *exogenous* properties, i.e., those whose value is determined by the causal laws in the model and those whose value is not, respectively (Pearl 2000). The second is the *default-deviant* assumption, used also by, e.g., Hall (2004) and Hitchcock (2007). The idea here is to assume that each

endogenous property of the domain has some "natural" state, that it will be in whenever nothing is acting upon it. For ease of notation, CP-logic identifies the default state with falsity, and the deviant state with truth. For example, consider the following simplified model of a bicycle, in which a pair of gear wheels can be put in motion by pedalling:

$$\text{Turn(BigGear)} \leftarrow \text{Pedal}. \tag{1}$$

$$\text{Turn(BigGear)} \leftarrow \text{Turn(SmallGear)}. \tag{2}$$

$$\text{Turn(SmallGear)} \leftarrow \text{Turn(BigGear)}. \tag{3}$$

Here, Pedal is exogenous, while Turn(BigGear) and Turn(SmallGear) are endogenous. The semantics of this causal model is given by a straightforward "execution" of the rules. The domain starts out in an initial state, in which all endogenous atoms have their default value *false* and the exogenous atom Pedal has some fixed value. If Pedal is true, then the first rule is applicable and may be fired ("Pedal causes Turn(BigGear)") to produce a new state of the domain in which Turn(BigGear) now has its deviant value *true*. In this way, we construct the following sequence of states (we abbreviate symbols by their first letter):

$$\{\text{P}\} \overset{(1)}{\rightarrow} \{\text{P}, \text{T(B)}\} \overset{(3)}{\rightarrow} \{\text{P}, \text{T(B)}, \text{T(S)}\} \overset{(2)}{\rightarrow} \{\text{P}, \text{T(B)}, \text{T(S)}\} \tag{4}$$

Note that firing rule (2) does not change the state of the world, because its effect is already true. Moreover, it is obvious that this will always be the case, so this rule may seem redundant. However, many interesting applications of causal models require the use of interventions (Pearl 2000), e.g., to evaluate counterfactuals or to predict the effects of actions. As shown by Vennekens et al. (2010), rule (2) allows CP-logic to represent this example in a way that produces the correct results for all conceivable interventions in a manner that is more modular and more concise than, among others, Pearl's structural models (Pearl 2000).

After rules (1), (3) and (2) have all fired, there are no more rules left whose body is satisfied and that have not yet fired. At this point, the process is at an end and the domain has reached a final state. It is this final state, rather than the details of the intermediate process, that we are really interested in. One of the most important properties of CP-logic is that, while there may be any number of different processes derived from a causal theory, the final state that is eventually reached is unique for any given interpretation for the exogenous predicates—at least, for examples such as this one. In general, CP-logic also allows rules with a *non-deterministic* effect, such as:

$$(\text{Turn(SmallGear)} : 0.99)\ \mathbf{Or}\ (\text{ChainBreaks} : 0.01) \leftarrow \text{Turn(BigGear)}.$$

Now, the cause Turn(BigGear) produces one of two possible effects, and there is an associated probability distribution over these two possibilities. The effect on the semantics is that, instead of a linear progression of states as in (4), we get a tree structure in which each firing of a non-deterministic rule introduces a branching of possibilities. When considering also the probabilities associated to non-deterministic choices, the tree defines a probability distribution over its leaves, i.e., over the final states that may be reached. It was shown in (Vennekens et al. 2009) that, given a specific interpretation for the exogenous atoms, this distribution is unique, even though there may exists many probability trees that produce it.

In many circumstances, the precise values of the probabilities are not of interest. In such cases, a non-probabilistic variant of CP-logic may be used, in which these are omitted. The head of a rule is then simply a disjunction:

$$\text{Turn(SmallGear)}\ \mathbf{Or}\ \text{ChainBreaks} \leftarrow \text{Turn(BigGear)}.$$

The trees then no longer produce a probability distribution over final states, but they describe the set of all final states that may be reached. In other words, this formalism has a possible world semantics. It is this non-probabilistic variant that concerns us in this paper.

Like other rule-based approaches to causality, CP-logic uses a very simple way of specifying the possible effects of some cause, namely, as a disjunction of ground atoms. Clearly, this does not—or, at least, not directly—cover many interesting phenomena that may occur in practice:

- A robot enters a room, opens some of the doors in this room, and then leaves by one of the doors that are open. The robot's leaving corresponds to a non-deterministic choice between a *dynamic* set of alternatives, which is determined by the robot's own actions, and therefore cannot be hard-coded into the head of a rule. A language construct for representing such choices is present in P-log (Baral et al. 2004).
- A stallion and a mare that are put in the same field may cause the birth of a foal. Therefore, not only the properties of these horses are governed by causal laws, but also their very existence.
- A horse being the parent of a foal is itself a cause for its own height to have a causal link to the height of the foal. Therefore, causal laws may be nested, in the sense that an effect can itself again consist of an entire causal law.

The goal of this paper is to develop an expressive knowledge representation language that is able to represent these more complex effects, and others like them, in a direct way. Moreover, we want to do this in a way that extends the approach of CP-logic. To summarise, the formal semantics of the language should consist of a set of possible worlds, each of which can be constructed by a non-deterministic causal process. This process will take place in the context of a fixed interpretation for the exogenous atoms. It will start from an initial state in which each of the endogenous atoms is at its default value false. The causal laws of our language will then "fire" and flip atoms to their deviant value, until no more such flips are possible. Whereas in CP-logic these flips happen one atom at a time, our extended language will flip *sets* of atoms at the same time. Moreover, our logic will present syntax and semantics for object-creation, as is needed in the second of the above examples.

The rest of this paper is structured as follows: we start by introducing causal effect expressions (CEEs) and their informal semantics in Section 2. In Section 3, we explain how C-Log is integrated with first-order logic, resulting in FO(C), a member of the FO(·) family of extensions of first-order logic. We conclude in Section 4 by comparing C-Log with various other paradigms, including inductive definitions (Denecker and Ternovska 2008), disjunctive logic programming with existential quantifications (You et al. 2013), Business Rules systems (Business Rules Group 2000) and Datalog extensions (Green et al. 2012).

2 Syntax and Informal Semantics

We assume familiarity with basic concepts of first-order logic (FO). Vocabularies, formulas, and terms are defined as usual. A Σ-structure $\mathcal{I}$ interprets all symbols (including variable symbols) in a vocabulary Σ; $D^{\mathcal{I}}$ denotes the domain of $\mathcal{I}$ and $\sigma^{\mathcal{I}}$, with σ a symbol in Σ, denotes the interpretation of σ in $\mathcal{I}$. We use $\mathcal{I}[\sigma : v]$ for the structure $\mathcal{J}$ that equals $\mathcal{I}$, except on σ: $\sigma^{\mathcal{J}} = v$. *Domain atoms* are atoms of the form $P(\overline{d})$ where the d_i are domain elements. We use restricted quantifications (Preyer and Peter 2002), e.g., in FO, these are formulas of the form $\forall x[\psi] : \varphi$ or $\exists x[\psi] : \varphi$, meaning that φ holds for all (resp. for a) x such that ψ holds. The above expressions are syntactic sugar for $\forall x : \psi \Rightarrow \varphi$ and $\exists x : \psi \wedge \varphi$, but such a reduction is not possible for other restricted quantifiers that we will define below. We call ψ the *qualification* and φ the *assertion* of the restricted quantifications. From now on, let Σ be a relational vocabulary, i.e., Σ consists only of predicate, constant and variable symbols.

2.1 Syntax

Definition 2.1
Causal effect expressions (CEE) are defined inductively as follows:

- if $P(\overline{t})$ is an atom, then $P(\overline{t})$ is a CEE,
- if φ is a first-order formula and C' is a CEE, then $C' \leftarrow \varphi$ is a CEE,
- if C_1 and C_2 are CEEs, then C_1 **And** C_2 is a CEE,
- if C_1 and C_2 are CEEs, then C_1 **Or** C_2 is a CEE,
- if x is a variable, φ is an FO formula and C' is a CEE, then **All** $x[\varphi] : C'$ is a CEE,
- if x is a variable, φ an FO formula and C' a CEE, then **Select** $x[\varphi] : C'$ is a CEE,
- if x is a variable and C' is a CEE, then **New** $x : C'$ is a CEE.

We call a CEE an *atom-expression* (respectively *rule*-, **And**-, **Or**-, **All**-, **Select**- or **New**-*expression*) if it is of the corresponding form. We call a predicate symbol P *endogenous* in C if P occurs as the symbol of a (possibly nested) atom-expression in C, i.e., if P occurs in C but not only in first-order formulas, i.e., not only in qualifications of restricted C-Log quantifications (**All** and **Select**) or conditions of rule-expressions. All other symbols are called *exogenous* in C. This is a straightforward generalisation of the same notions in CP-logic. An occurrence of a variable x is *bound* in a CEE if it occurs in the scope of a quantification over that variable ($\forall x$, $\exists x$, **All** x, **Select** x, or **New** x) and *free* otherwise. A variable is *free* in a CEE if it has free occurrences. A *causal theory*, or C-Log *theory* is a CEE without free variables. By abuse of notation, we often represent a causal theory as a set of CEEs; the intended causal theory is the **And**-conjunction of these CEEs. We often use Δ for a causal theory and C, C', C_1 and C_2 for its subexpressions.

2.2 Informal Semantics of CEEs

We now present the informal semantics of CEEs, due to space restrictions, the formalisation of this semantics is lacking in this paper. For a complete description of the formal semantics, we refer to an accompanying technical report (Bogaerts et al. 2014a). A CEE is a description of a set of causal laws. In the context of a state of affairs—which we represent, as usual, by a structure—a CEE non-deterministically describes a set of effects,

i.e., a set of events that take place and change the state of affairs. We call such a set the *effect set* of the CEE. From a CEE C, we can derive causal processes similar to (4); a causal process is a sequence of intermediate states, starting from the default state, such that, at each state, the effects described by C take place. The process ends if the effects no longer cause changes to the state. A structure is a model of a CEE if it is the final result of such a process. We now explain in a compositional way what the effect set of a CEE is in a given state of affairs.

The effect of an atom-expression A is that A is flipped to its deviant state. A conditional effect, i.e., a rule expression, causes the effect set of its head if its body is satisfied in the current state, and nothing otherwise. The effect set described by an **And**-expression is the union of the effect sets of its two subexpressions; an **All**-expression $\mathbf{All}\, x[\varphi] : C'$ causes the union of all effect sets of $C'(x)$ for those x's that satisfy φ. An expression $C_1\, \mathbf{Or}\, C_2$ non-deterministically causes either the effect set of C_1 or the effect set of C_2; a **Select**-expression $\mathbf{Select}\, x[\varphi] : C'$ causes the effect set of C' for a non-deterministically chosen x that satisfies φ. An object-creating CEE $\mathbf{New}\, x : C'$ causes the creation of a new domain element n and the effect set of $C'(n)$.

Example 2.2

Permanent residence in the United States can be obtained in several ways. One way is passing the naturalisation test. Another way is by playing the "Green Card Lottery", where each year a number of lucky winners are randomly selected and granted permanent residence. We model this as follows:

$$\mathbf{All}\, p[\mathrm{Applied}(p) \wedge \mathrm{PassedTest}(p)] : \mathrm{PermRes}(p)$$
$$(\mathbf{Select}\, p[\mathrm{Applied}(p)] : \mathrm{PermRes}(p)) \leftarrow \mathrm{Lottery}.$$

The first CEE describes the "normal" way to obtain permanent residence; the second rule expresses that one winner is selected among everyone who applies. If $\mathscr{I}$ is a structure in which Lottery holds, due to the non-determinism, there are many possible effect sets of the above CEE, namely all sets $\{\mathrm{PermRes}(p) \mid p \in D^{\mathscr{I}} \wedge p \in \mathrm{Applied}^{\mathscr{I}} \wedge \mathrm{PassedTest}(p)^{\mathscr{I}}\}$ $\cup\, \{\mathrm{PermRes}(d)\}$ for some $d \in \mathrm{Applied}^{\mathscr{I}}$. The two CEEs are considered independent: the winner could be one of the people that obtained it through standard application, as well as someone else. Note that in the above, there is a great asymmetry between Applied(p), which occurs as a qualification of **Select**-expression, and PermRes(p), which occurs as a caused atom, in the sense that the effect will never cause atoms of the form Applied(p), but only atoms of the form PermRes(p). This is one of the cases where the qualification of an expression cannot simply be eliminated.

Example 2.3

Hitting the "send" button in your mail application causes the creation of a new package containing a specific mail. That package is put on a channel and will be received some (unknown) time later. As long as the package is not received, it stays on the channel. In C-Log, we model this as follows:

$$\mathbf{All}\, m, t[\mathrm{Mail}(m) \wedge \mathrm{HitSend}(m,t)] : \mathbf{New}\, p : \mathrm{Pack}(p)\, \mathbf{And}\, \mathrm{Cont}(p,m)\, \mathbf{And}\, \mathrm{OnCh}(p, t+1)$$
$$\mathbf{And}\, \mathbf{Select}\, d[d > 0] : \mathrm{Received}(p, t+d)$$
$$\mathbf{All}\, p, t[\mathrm{Pack}(p) \wedge \mathrm{OnCh}(p,t) \wedge \neg \mathrm{Received}(p,t)] : \mathrm{OnCh}(p, t+1)$$

Suppose an interpretation $\text{HitSend}^{\mathcal{I}} = \{(\text{MyMail}, 0)\}$ is given. A causal process then unfolds as follows: it starts in the initial state, where all endogenous predicates are false. The effect set of the above causal effect in that state consists of 1) the creation of one new domain element, say $\_p$, and 2) the caused atoms $\text{Pack}(\_p)$, $\text{Cont}(\_p, \text{MyMail})$, $\text{OnCh}(\_p, 1)$ and $\text{Received}(\_p, 7)$, where instead of 7, we could have chosen any number greater than zero. Next, it continues, and in every step t, before receiving the package, an extra atom $\text{OnCh}(p, t+1)$ is caused. Finally, in the seventh step, no more atoms are caused; the causal process ends. The final state is a model of the causal theory.

3 FO(C): Integrating FO and C-Log

First-order logic and C-Log have a straightforward integration, FO(C). Theories in this logic are sets of FO sentences and CEEs. A model of such a theory is a structure that satisfies each of its expressions (each of its CEEs and formulas). An illustration is the mail protocol from Example 2.3, which we can extend with the "observation" that at at some time point, two packages are on the channel: $\exists t, p_1, p_2 : \text{OnCh}(p_1, t) \wedge \text{OnCh}(p_2, t) \wedge p_1 \neq p_2$. Models of this theory represent states of affairs where at least once two packages are on the channel simultaneously. This entirely differs from **And**-conjoining our CEE with

$$\textbf{Select}\, t, p_1, p_2[p_1 \neq p_2] : \text{OnCh}(p_1, t)\, \textbf{And}\, \text{OnCh}(p_2, t).$$

The resulting CEE would have unintended models in which two packages suddenly appear on the channel for no reason.

In FO(C), **New**-expressions can be simulated with **Select**-expressions together with FO axioms expressing the unicity of the newly "created" objects. E.g.,

$$\textbf{New}\, x : P(x, a)\, \textbf{And}\, \textbf{New}\, x : Q(x)$$

is simulated by introducing auxiliary unary predicates N_1 and N_2 that identify the objects created by the expressions and writing:

$$\{(\textbf{Select}\, x[\mathbf{t}] : (N_1(x)\, \textbf{And}\, P(x, a)))\, \textbf{And}\, \textbf{Select}\, x[\mathbf{t}] : (N_2(x)\, \textbf{And}\, Q(x))\}$$
$$\forall x : \neg(N_1(x) \wedge N_2(x))$$

It is clear that **New**-expressions are more natural and more modular than this simulation.

Despite the syntactical correspondence between CEEs and FO formulas (**And** corresponds to $\wedge$, **All** to $\forall$, ...), it is obvious that they have an entirely different meaning, and that both are useful. This is why we chose to introduce new connectives rather than overloading the ones of FO. The logic FO(C) has further interesting extensions, e.g., by adding aggregates in FO formulas, including in qualifications and conditions of CEEs.

4 Comparison and Future Work

In this section, we compare FO(C) to other existing paradigms. This comparison is only an initial study. By the time of publishing, a more extended version of a comparison between FO(C) and other paradigms has appeared (Bogaerts et al. 2014b).

Due to its simple recursive syntax, FO(C) is a very general logic that generalises several existing logics and shows overlaps with many others in different areas of computational logic. C-Log is an extension of (the non-probabilistic version of) CP-logic. FO(C) is an

extension of the logic FO(ID) (Denecker and Ternovska 2008). An FO(ID) theory is a set of FO sentences and inductive definitions (ID), which are sets of rules of the form

$$\forall \overline{x} : P(\overline{t}) \leftarrow \varphi,$$

where φ is an FO formula. Such a rule corresponds to a CEE

$$\mathbf{All}\, \overline{x}[\varphi] : P(\overline{t})$$

or equivalently,

$$\mathbf{All}\, \overline{x}[\mathbf{t}] : (P(\overline{t}) \leftarrow \varphi)$$

and a definition corresponds to the **And**-conjunction of its rules. The semantics of FO(ID) corresponds exactly to the semantics of the corresponding FO(C) theory (Bogaerts et al. 2014a). Denecker et al. (1998) already pointed to the correspondence between causality and inductive definitions and exploited it for solving the causal *ramification problem* of temporal reasoning (McCarthy and Hayes 1969). The CEEs presented here can be seen as a non-deterministic extension of inductive definitions with an informal semantics based on causal processes.

FO(C) shows similarity to extensions of disjunctive logic programming (DLP) such as DLP with existential quantification in rule heads (You et al. 2013) and the stable semantics for FO as defined by Ferraris et al. (2011). Here constraints correspond to FO sentences in FO(C) and other rules correspond to C-Log expressions. However, there is an important semantical difference. Suppose we want to express Example 2.2, where all people passing a test and one random person are given permanent residence in the United States. The E-disjunctive program

$$\exists X : permres(X) \text{ :- } lottery$$
$$\forall X : permres(X) \text{ :- } passtest(X)$$

is similar to

$$(\mathbf{Select}\, x[\mathbf{t}] : permres(x)) \leftarrow lottery$$
$$\mathbf{And\ All}\, x[passtest(x)] : permres(x)$$

Semantically, the E-disjunctive program imposes a minimality condition: the lottery is always won by a person succeeding the test, if there exists one. On the other hand, in FO(C) the two rules execute independently, and models might not be minimal. In this example, it is the latter that is intended. We believe that one advantage of C-Log is its clear causal informal semantics. On the other hand, there are ways to simulate the causal semantics and the **New** operator of C-Log in E-disjunctive programs while it follows from complexity arguments that not all E-disjunctive programs can be expressed in FO(C) (Bogaerts et al. 2014c).

Other semantics than the stable semantics for DLP have been developed. For example, Brass and Dix (1996) defined D-WFS, a well-founded semantics for DLP. This semantics has the property that if a program contains two identical lines, one of them can be removed. However, in our context, a duplicate effect means that a same causal effect happens twice (maybe for different reasons), independently, and hence different choices might be made in each of these rules.

The logic of cause and change (McCain and Turner 1996) differs from C-Log in several important aspects; in McCain & Turner's logic both true and false atoms need a cause. In C-Log on the other hand, endogenous predicates can be false (the default value) without reason but can only be true (the deviant value) if caused. Moreover, we rule out unfounded "cyclic" causation. For instance, if Pedal is false, in C-Log, Turn(BigGear) and Turn(SmallGear) are false but in McCain and Turner's logic they may be true and caused by each other. We call this "spontaneous generation" and do not admit it in C-Log.

We find operators similar to those of C-Log in several other formalisms. For example, **Select-**, **All-**, **Or-** and rule-expressions are present in the subformalism of the language Event-B that serves to specify effects of actions (Abrial 2010). The **New** operator is found in various other rule based paradigms, for example in Business Rules systems (Business Rules Group 2000). The JBoss manual (Browne 2009) contains the following rule:

```
when Order( customer == null ) then  insertLogical(new
   ValidationResult( validation.customer.missing ));
```

meaning that if an order is created without customer, a new *ValidationResult* is created with the message that the customer is missing. This can be translated to C-Log as follows:

$$\textbf{All}\ y[\text{Order}(y) \wedge \text{NoCustumer}(y)] : \textbf{New}\ x : \text{ValidationR}(x)\ \textbf{And}\ \text{Message}(x, \text{"}\ldots\text{"}).$$

Another field in which related language constructions have been developed is the field of deductive databases. Abiteboul and Vianu (1991) considered various extensions of Datalog, resulting in non-deterministic semantics for queries and updates. One of the studied extensions is object creation. Such an extension is present in the LogicBlox system (Green et al. 2012). An example from the latter paper is the rule: $President(p), presidentOf[c] = p \leftarrow Country(c)$ which means that for every country c, a new (anonymous) "derived entity" of type $President$ is created. Of course, this president is not a new person, but it is new with respect to a given database. Such rules with implicit existentially quantified head variables correspond with **New**-expressions in C-Log.

Other Datalog extensions with other forms of object creation exist. For example Van den Bussche and Paredaens (1995) discuss a version with creation of sets and compare its expressivity with simple object creation.

Non-deterministic choices have been studied intensively in the context of deductive databases. Krishnamurthy and Naqvi (1988) introduced a non-deterministic choice in Datalog. This choice was *static*: choice models are constructed in three steps. First, models are calculated while ignoring choices (choosing everything); second, this model is used to select a number of choices for all occurrences of *choice goals* and third, models are recalculated with respect to these choice goals. In other work (Saccà and Zaniolo 1990; Giannotti et al. 1991), it is argued that static choices do not behave well in the presence of recursion; hence *dynamic* choices were introduced. Saccà and Zaniolo (1990) use stable models to provide a model-theoretic description of these dynamic choices. Weidong and Jinghong (1996) introduced an alternative choice principle on predicates P. There, the values in certain argument positions in the tuples of P are chosen non-deterministically in function of the values at the other argument positions. The semantics of that logic is based on the well-founded semantics; this choice principle is very different from the principle in C-Log. Compared to these, C-Log resembles most the language of Saccà and

Zaniolo (1990); the difference is that C-Log supports a recursive syntax and is based on the well-founded semantics, whereas Saccà and Zaniolo (1990) use stable semantics.

The above similarities suggest that FO(C) is a promising language to study and unify many existing logical paradigms and to provide a clear informal semantics for them. An in-depth semantical analysis of the exact relationship between FO(C) and the languages described above is an interesting topic for future work. Another research challenge is extending FO(C) with types, function symbols, arithmetic, etc. in order to make it useful as a KR-language. We need to study the complexity of various inference tasks in FO(C), and develop and implement algorithms for these various tasks. By the time of publication, a first study of complexity and inference in FO(C) has appeared (Bogaerts et al. 2014c). Another research question is to add probabilities to C-Log to obtain an extension of the probabilistic CP-logic, and possibly also of other related logics such as BLOG (Milch et al. 2005) and P-Log (Baral et al. 2004).

References

Abiteboul, S. and Vianu, V. 1991. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci. 43,* 1, 62–124.

Abrial, J.-R. 2010. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press.

Baral, C., Gelfond, M., and Rushton, N. 2004. Probabilistic reasoning with answer sets. In *Proc. Logic Programming and Non Monotonic Reasoning, LPNMR'04.* Springer-Verlag, 21–33.

Bogaerts, B., Vennekens, J., Denecker, M., and Van den Bussche, J. 2014a. C-Log: A knowledge representation language of causality. Tech. Rep. CW 656, Departement of Computer Science, Katholieke Universiteit Leuven.

Bogaerts, B., Vennekens, J., Denecker, M., and Van den Bussche, J. (in press) 2014b. FO(C) and related modelling paradigms. In *Proceedings of the Fifteenth International Workshop on Non-Monotonic Reasoning, NMR 2014, Vienna, Austria, September 17-19.*

Bogaerts, B., Vennekens, J., Denecker, M., and Van den Bussche, J. (in press) 2014c. Inference in the FO(C) modelling language. In *ECAI 2014 - 21th European Conference on Artificial Intelligence, Prague, Czech Republic, August 18-22, 2014, Proceedings.*

Brass, S. and Dix, J. 1996. Characterizing D-WFS: Confluence and iterated GCWA. In *Logics in Artificial Intelligence*, J. J. Alferes, L. M. Pereira, and E. Orlowska, Eds. Lecture Notes in Computer Science, vol. 1126. Springer Berlin Heidelberg, 268–283.

Browne, P. 2009. *JBoss Drools Business Rules.* From technologies to solutions. Packt Publishing, Limited.

Business Rules Group. 2000. Defining Business Rules ~ What Are They Really? Tech. rep.

Cabalar, P. 2012. Causal logic programming. In *Correct Reasoning*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds. Lecture Notes in Computer Science, vol. 7265. Springer, 102–116.

Denecker, M. 2012. The FO(·) knowledge base system project: An integration project (invited talk). In *ASPOCP.*

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL) 9,* 2 (Apr.), 14:1–14:52.

Denecker, M., Theseider-Dupré, D., and Van Belleghem, K. 1998. An inductive definition approach to ramifications. *Linkoping Electronic Articles in Computer and Information Science 3,* 7 (Jan.), 1–43.

Ferraris, P., Lee, J., and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence 175*, 236–263.

Giannotti, F., Pedreschi, D., Saccà, D., and Zaniolo, C. 1991. Non-determinism in deductive

databases. In *Deductive and Object-Oriented Databases*, C. Delobel, M. Kifer, and Y. Masunaga, Eds. Lecture Notes in Computer Science, vol. 566. Springer Berlin Heidelberg, 129–146.

Green, T. J., Aref, M., and Karvounarakis, G. 2012. Logicblox, platform and language: A tutorial. In *Datalog*, P. Barceló and R. Pichler, Eds. LNCS, vol. 7494. Springer, 1–8.

Hall, N. 2004. Two concepts of causation. In *Causation and Counterfactuals*.

Hitchcock, C. 2007. Prevention, preemption, and the principle of sufficient reason. *Philosophical review 116,* 4.

Krishnamurthy, R. and Naqvi, S. A. 1988. Non-deterministic choice in datalog. In *JCDKB* (2002-01-03). 416–424.

McCain, N. and Turner, H. 1996. Causal theories of action and change. In *AAAI/IAAI*. AAAI Press, 460–465.

McCarthy, J. and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, 463–502.

Milch, B., Marthi, B., Russell, S. J., Sontag, D., Ong, D. L., and Kolobov, A. 2005. Blog: Probabilistic models with unknown objects. In *IJCAI*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 1352–1359.

Pearl, J. 2000. *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

Preyer, G. and Peter, G. 2002. *Logical Form and Language*. Clarendon Press.

Saccà, D. and Zaniolo, C. 1990. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*. ACM Press, 205–217.

Van den Bussche, J. and Paredaens, J. 1995. The expressive power of complex values in object-based data models. *Information and Computation 120*, 220–236.

Vennekens, J., Bruynooghe, M., and Denecker, M. 2010. Embracing events in causal modelling: Interventions and counterfactuals in CP-logic. In *Logics in Artificial Intelligence*, T. Janhunen and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 6341. Springer Berlin Heidelberg, 313–325.

Vennekens, J., Denecker, M., and Bruynooghe, M. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming 9,* 3, 245–308.

Weidong, C. and Jinghong, Z. 1996. Nondeterminism through well-founded choice. *The Journal of Logic Programming 26,* 3, 285–309.

You, J.-H., Zhang, H., and Zhang, Y. 2013. Disjunctive logic programs with existential quantification in rule heads. *Theory and Practice of Logic Programming 13*, 563–578.

A Framework for Bottom-Up Simulation of SLD-Resolution

STEFAN BRASS

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
(*e-mail:* brass@informatik.uni-halle.de)

submitted 14 February 2014; revised —; accepted —

Abstract

This paper introduces a framework for the bottom-up simulation of SLD-resolution based on partial evaluation. The main idea is to use database facts to represent a set of SLD goals. For deductive databases it is natural to assume that the rules defining derived predicates are known at "compile time", whereas the database predicates are known only later at runtime. The framework is inspired by the author's own SLDMagic method, and a variant of Earley deduction recently introduced by Heike Stephan and the author. However, it opens a much broader perspective.

KEYWORDS: deductive databases, Datalog, bottom-up evaluation, partial evaluation, optimization

1 Introduction

Deductive databases use logic programming for data intensive applications. For example, database queries are written in a Prolog-like language called Datalog. Basic Datalog is pure Prolog without structured terms. The data stored e.g. in a relational database can be seen as a large set of facts. Of course, many extensions of this basic language have been investigated and implemented in prototype systems. While in the beginning, the main achievement of deductive databases was seen in the possibility to write recursive queries, e.g. for hierarchical and graph-structured data, the more general goal is to support database queries and application programming in one declarative language.

For efficient query evaluation, it is important to distinguish between predicates defined by rules in a logic program, and predicates defined by facts in the database. The logic program is known at "compile time", while the facts are known only at "runtime". I.e. the database facts form the input to the logic program. Because the program might be executed several times with different database states, it pays off to invest time for optimization by precomputing as much as possible given the logic program, while the database facts are not yet known. Time might be saved even in a single execution of the program, because the logic program is usually small, while the database state is big.

Deductive databases use bottom-up evaluation, i.e. apply the T_P-operator to derive facts from already known facts to get logically implied instances of the query. Of course, many optimizations are added to this basic method. Especially, there are a lot of methods for making bottom-up evaluation goal-oriented, i.e. to derive only facts that are in some

sense needed for computing answers to the query. Most well-known in this area is the "magic set" method (Bancilhon et al. 1986), where magic sets simply encode subqueries.

François Bry had the idea to explain magic sets with a meta-interpreter, which describes top-down evaluation, but runs on a bottom-up machine (Bry 1990). When this meta-interpreter is partially evaluated with respect to the given rules, one gets exactly the result of the magic set transformation. This might be the first case of partial evaluation for strict bottom-up evaluation, other applications of partial evaluation for deductive databases are, e.g. (Sakama and Itoh 1988; Lei et al. 1990; Han 1995). Of course, partial evaluation for top-down evaluation (Prolog) was well investigated (see, e.g., (Gallagher 1993)), but the methods for partial evaluation depend crucially on the execution model.

It turned out that magic sets do not exactly correspond to SLD-resolution, and that in the case of tail recursions, SLD-resolution has a big advantage, because it does not need to materialize every proven "lemma" (Ross 1991; Brass 1995).

The author then proposed a meta-interpreter that describes SLD-resolution exactly and runs on a bottom-up machine. In this way, set-oriented evaluation techniques can be used, and termination can be guaranteed for Datalog, e.g. a rule like $\mathsf{p(X)} \leftarrow \mathsf{p(X)}$ does not cause an infinite loop. In contrast to magic sets and tabling techniques, tail-recursion runs much faster, e.g. consider the following logic program P:

$$\mathsf{path(X,Y)} \leftarrow \mathsf{edge(X,Y)}\cdot$$
$$\mathsf{path(X,Z)} \leftarrow \mathsf{edge(X,Y)} \wedge \mathsf{path(Y,Z)}\cdot$$

Suppose the database is $\mathsf{D} := \{\mathsf{edge}(i-1,i) \mid 1 \leqslant i \leqslant n\}$, i.e. the graph is a single path of length n. For the query $\mathsf{path}(0,\mathsf{X})$, the SLD-tree has $4n+3$ nodes, i.e. a number that is linear in n, whereas magic sets derive all facts of the form $\mathsf{path}(i,j)$ with $0 \leqslant i < j \leqslant n$, which is quadratic in n. While a tail-recursion optimization for magic sets has already been studied in (Ross 1991), our "SLDMagic"-method (Brass 2000) had other advantages as well by simulating SLD-resolution bottom-up. It passes also non-equality conditions on parameters to called predicates, and avoids joins when a predicate "returns".

However, the possibilities for bottom-up simulation of SLD-resolution extend much farther if we allow a single fact to represent multiple nodes in the SLD-tree. Already when the SLDMagic method was implemented, it was noted that it produces a lot of "copy" rules, which only copy tuples from one predicate to another predicate. These rules were then eliminated by a postprocessing step, which merged the predicates. This can be understood as allowing facts to represent a set of goals in the SLD tree.

Recently, Heike Stephan and the author developed a variant of Earley deduction (Pereira and Warren 1983; Porter III 1986), also tabling (Tamaki and Sato 1986; Chen and Warren 1996; Nguyen and Cao 2012) can be seen as developing the Earley method further. Our Earley deduction variant uses states describing a relatively big part of deduction using only program rules, but no database facts (Brass and Stephan 2013). While there are differences in the technical details, this can be seen as representing a whole set of SLD goals in a single "state", which is encoded a a database fact.

The purpose of this paper is to introduce an abstract framework for simulating SLD-resolution on a bottom-up machine, improve the understanding of the mentioned methods and their similarities, and discuss options for improving the efficiency of program execution in deductive databases.

2 Basic Definitions

A logic program P is a finite set of rules of the form $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$ where A and B_i, $i = 1,\ldots,n$ (with $n \geqslant 0$) are positive literals, i.e. have the form $\mathsf{p}(\mathsf{t}_1,\ldots,\mathsf{t}_m)$ with a predicate p and terms t_j, $j = 1,\ldots,m$. Terms are variables or constants. We assume that the rules are range-restricted (safe), i.e. every variable appearing in the head A also appears in the body $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$. In this paper, we do not consider negation.

A subset of the predicates are selected as EDB predicates ("extensional database"). These are the database relations. The EDB predicates can appear in the logic program only in the body (i.e. in the B_i), but not in the head (A). Besides the logic program, a database state D is given, which is a finite set of facts (positive ground literals) in which only EDB predicates appear (it follows that P and D are disjoint). We write $\mathscr{B}$ for the set of all positive ground literals with EDB predicate (the Herbrand base restricted to EDB predicates). This set will usually be infinite (because it permits arbitrary strings, integers, etc. as arguments). Of course, $\mathsf{D} \subseteq \mathscr{B}$.

The predicates which do appear in rule heads are called IDB predicates ("intensional database"). These predicates are defined by means of rules, not by enumerating facts. It is possible that an IDB predicate has only rules with empty body (i.e. facts), then the only difference to an EDB predicate is that we assume that the program is given for partial evaluation ("compile time"), whereas the database is only known at runtime.

In practice, one also needs "built-in predicates" like $<$, which are defined by procedures inside the system. Such predicates raise interesting questions of range restriction and safety, see, e.g., (Ramakrishnan et al. 1987; Kifer et al. 1988; Brass 2009). However, to simplify the presentation, we exclude them here.

A query (goal) Q is a conjunction $\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$ of positive literals (like a rule body). The variables appearing in the query are called the answer variables. The purpose of query evaluation is to find ground substitutions for the answer variables such that the corresponding instance of the query is true in the minimal model of $\mathsf{P} \cup \mathsf{D}$.

To simplify the presentation, we assume the first literal selection rule for SLD-resolution (as applied in Prolog). In SLD-resolution, the computed answer substitution is normally defined by looking at an entire SLD-derivation leading to the empty clause:

$$Q \longrightarrow g_1 \longrightarrow g_2 \longrightarrow \cdots \longrightarrow g_n \longrightarrow \square$$

But it suffices to consider only single goals at a time, if we start with an "extended query", which has a literal with all answer variables and a special predicate answer at the very end: $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \mathsf{answer}(\mathsf{X}_1,\ldots,\mathsf{X}_m)$. The predicate answer is not otherwise used in the program or the database. It only does the bookkeeping of the current values of the answer variables (since all substitutions done during the SLD derivation are also applied to this literal). Therefore, if we reach a goal consisting of a single literal $\mathsf{answer}(c_1,\ldots,c_m)$, we know that the answer substitution $\{X_1/c_1,\ldots,X_m/c_m\}$ has been computed.

It might seem at first that it is a restriction that we assume a completely given query. One might want to consider also "parameterized queries", containing constants not yet known at "compile time". For instance, one might want to do query optimization (partial evaluation) for any query of the form $\mathsf{path}(c,\mathsf{X})$, with an arbitrary constant c. This corresponds to "binding pattern" bf (mode $+-$) as used in the magic set method. However, we can simply use the query $\mathsf{input}(\mathsf{C}) \wedge \mathsf{path}(\mathsf{C},\mathsf{X})$ with a database predicate input which is filled with the parameter value before query evaluation starts.

3 States Representing Sets of Goals

3.1 Definition of SLDDB-Systems

We first define an abstract system with states representing sets of goals in the SLD tree, and a transition relation between these states. The states will later be encoded as facts, and the transition relation corresponds to rules which permit to derive these facts.

A SLDDB-System $\mathscr{T}$ consists of

- $\mathscr{S}$, a (usually infinite) set of states,
- $\mathscr{G}(S)$, a non-empty set of goals for every $S \in \mathscr{S}$,
- $S_0 \in \mathscr{S}$, an initial state,
- $\mapsto_\epsilon \subseteq \mathscr{S} \times \mathscr{S}$, a relation between states (ϵ-transitions),
- $\mapsto_{\mathsf{F}} \subseteq \mathscr{S} \times \mathscr{S}$, a relation between states for every possible database fact $\mathsf{F} \in \mathscr{B}$.

Given an SLDDB-System $\mathscr{T}$ as above, and a database state $\mathsf{D} \subseteq \mathscr{B}$ (set of facts), an answer tuple $(c_1, \ldots, c_m)$ is computed iff there is a finite sequence $S_0 \ldots S_n \in \mathscr{S}^*$ of states, starting with the initial state S_0, containing the answer in the final state, i.e. $\mathsf{answer}(c_1, \ldots, c_m) \in \mathscr{G}(S_n)$, and such that for $i = 1, \ldots, n$,

- $S_{i-1} \mapsto_\epsilon S_i$ or
- $S_{i-1} \mapsto_{\mathsf{F}} S_i$ for some $\mathsf{F} \in \mathsf{D}$.

The states will later be encoded as facts, and the reachable states will be computed bottom-up, therefore repeated states in a state sequence can be detected in order to improve termination. Of course, if an analysis shows that this cannot happen, one can save the effort for duplicate detection. Another useful property is that for certain states, there is only one possible successor state (if D satisfies integrity constraints such as keys).

3.2 Correctness of SLD-Simulation

An SLDDB-System should correspond to SLD-resolution for a given program and query. First we define the correctness, i.e. that all goals occurring in a state sequence are really derivable from the query, the program, and the database facts used in state transitions.

Let a logic program P and a query Q be given. An SLDDB-System $\mathscr{T}$ is correct with respect to P and Q iff

- For every $g \in \mathscr{G}(S_0)$, there is an SLD-derivation of g from the extended query $Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)$ using rules in P (this includes the case that the derivation is empty, i.e. g is the extended query).
- Whenever $S_1 \mapsto_\epsilon S_2$, then for every $g_2 \in \mathscr{G}(S_2)$ there is $g_1 \in \mathscr{G}(S_1)$ such that there is a non-empty SLD-derivation $g_1 \longrightarrow g' \longrightarrow^* g_2$ of g_2 from g_1, using rules in P, and the first step g' in this derivation is contained in $\mathscr{G}(S_2)$, i.e. $g' \in \mathscr{G}(S_2)$.
- Whenever $S_1 \mapsto_{\mathsf{F}} S_2$, then for every $g_2 \in \mathscr{G}(S_2)$ there is $g_1 \in \mathscr{G}(S_1)$ such that there is a non-empty SLD-derivation $g_1 \longrightarrow g' \longrightarrow^* g_2$ of g_2 from g_1, where the first step uses the fact F, other steps use rules in P, and $g' \in \mathscr{G}(S_2)$.

If this condition is satisfied, then for every computed answer tuple $(c_1, \ldots, c_m)$ there is an SLD-derivation of $\mathsf{answer}(c_1, \ldots, c_m)$ from $Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)$. This obviously means that there is also an SLD-derivation of the empty clause from Q, where the substitution $\theta := \{\mathsf{X}_1/c_1, \ldots, \mathsf{X}_m/c_m\}$ is applied to the variables in Q. Therefore, by the correctness of SLD-resolution, it follows that $Q\,\theta$ is a logical consequence of $\mathsf{P} \cup \mathsf{D}$.

3.3 Completeness of SLD-Simulation

In the opposite direction, we need that every SLD-derivation is indeed represented in the states and the transition relation. Note that the SLDDB-System is independent of a concrete database state, i.e. it represents derivations using any possible database facts. Of course, when a state sequence is constructed to answer a query in a concrete database state D, only facts in D can be used.

Let a logic program P and a query Q be given. An SLDDB-System $\mathscr{T}$ is complete with respect to P and Q iff

- The extended query $Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)$ is contained in $\mathscr{G}(S_0)$.
- For every state $S \in \mathscr{S}$ and every goal $g \in \mathscr{G}(S)$ and every g' which can be derived from g and a rule in P by an SLD-resolution step, there is a variant g'' of g' with $g'' \in \mathscr{G}(S)$ or $g'' \in \mathscr{G}(S')$ for some state S' with $S \mapsto_\epsilon S'$.
- For every state $S \in \mathscr{S}$, every goal $g \in \mathscr{G}(S)$ and every g' which can be derived from g and a fact $\mathsf{F} \in \mathscr{B}$ by an SLD-resolution step, there is a variant g'' of g' and state S' with $S \mapsto_{\mathsf{F}} S'$ such that $g'' \in \mathscr{G}(S')$.

This condition ensures that every SLD-derivation of the empty clause from Q using rules in P and facts in DB can indeed be represented by a state sequence.

It would actually suffice to require the completeness not for all SLD resolution steps, but only for steps in successful derivations where the set of facts used in the state sequence satisfies given integrity constraints. In this way, "dead ends" could be cut off early.

3.4 Example: SLD-Resolution

Of course, one would expect that SLD-resolution itself fits in this framework.

If one wants to simulate SLD-resolution exactly, including the non-termination for $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{p}(\mathsf{X})$, one uses SLD-derivations as states. I.e., given a program P and a query Q, the set of states $\mathscr{S}$ is the set of SLD-derivations $Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m) \longrightarrow^* g_n$ and the set of goals for the above state S is $\mathscr{G}(S) := \{g_n\}$, i.e. a singleton set consisting of the last (or current) goal in the derivation. The transition relations extend the derivation by one goal, i.e. lead from S to the following state S':

$$Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m) \longrightarrow^* g_n \longrightarrow g_{n+1}.$$

If in the last SLD resolution step a program rule was used, $S \mapsto_\epsilon S'$. If instead a fact F with EDB-predicate was used, $S \mapsto_{\mathsf{F}} S'$.

3.5 Example: SLD-Resolution Without Duplicate Nodes

Of course, it is more in the spirit of bottom-up evaluation to eliminate duplicate nodes, and this is what SLDMagic (Brass 2000) did. In the above framework, this simply means that the states are single goals, i.e. conjunctions of the form

$$\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \mathsf{answer}(\mathsf{t}_1, \ldots, \mathsf{t}_m).$$

Furthermore, we have to exclude goals which differ only in a variable renaming. We do this by requiring that variables are named $\mathsf{V}_1, \mathsf{V}_2, \ldots$ in the order of first occurrence in the goal. Let $\mathsf{norm}(g)$ be a mapping from goals to goals which normalizes variables in

this way. Of course, when states are single goals, we can simply let $\mathcal{G}(S) := \{S\}$. The transition relation is simply SLD-resolution plus the normalization. I.e. $S \mapsto_\epsilon S'$ holds iff S' is derivable from S by a single SLD resolution step (using a rule in P), followed by variable normalization. Correspondingly, $S \mapsto_{\mathsf{F}} S'$ holds when fact $\mathsf{F} \in \mathcal{B}$ is used in the resolution step. Obviously, the set of computed answers is not changed by merging nodes in the SLD-tree with the same goal. The number of duplicate answers is changed (every distinct answer is computed only once). However, if one considers duplicates as important, one probably wants a more declarative specification. Duplicates in deductive databases have been considered e.g. in (Mumick et al. 1990).

Note also that this works only because in SLD-resolution there is no need to return to the "caller" — otherwise the same subgoal could appear in different contexts, and it might be important to distinguish between them. This was one of the difficulties in our variant of Earley deduction. In SLD-resolution, the entire continuation of the proof is built into the goal. With the answer-literal at the end of the goal, even for determining the answer substitution, we do not have to look at an entire path in the tree.

Termination can be guaranteed if duplicate states in state sequences are excluded and the program is at most tail-recursive, i.e. only the last literal of a rule can be a recursive call. This property ensures that the length of goals is bounded (Brass 2000).

3.6 Maximal States

So far, states contained only single goals. But we want to compute as much as possible at "compile time", i.e. when query and program are known, but the facts in the database are not yet known. Therefore, we do the following closure operation on sets of goals:

$$\mathsf{cl}_\mathsf{P}(G) := \{\mathsf{norm}(g') \mid \text{there is } g \in G \text{ and an SLD-derivation } g \longrightarrow^* g' \text{ using only rules in } \mathsf{P}\}.$$

Now, given program P and query Q with answer variables $\mathsf{X}_1, \ldots, \mathsf{X}_m$, the initial state is the closure of the extended version of Q:

$$S_0 := \mathsf{cl}_\mathsf{P}\Big(\{Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)\}\Big).$$

Given a state S and a fact F with EDB-predicate, the successor state S' (with $S \mapsto_{\mathsf{F}} S'$) is defined as follows:

$$S' := \mathsf{cl}_\mathsf{P}\Big(\{g' \mid \text{there is } g \in S \text{ such that a single SLD-resolution step of } g \text{ and } \mathsf{F} \text{ gives } g'\}\Big).$$

The other transition relation $\mapsto_\epsilon$ is empty, i.e. state transitions occur only when EDB-facts are used, other deductions (with program rules) are done within the states.

Of course, it is possible that states become infinite. For instance, consider the left recursive version of the transitive closure example:

$$\begin{array}{lcl} \mathsf{path}(\mathsf{X},\mathsf{Y}) & \leftarrow & \mathsf{edge}(\mathsf{X},\mathsf{Y})\cdot \\ \mathsf{path}(\mathsf{X},\mathsf{Z}) & \leftarrow & \mathsf{path}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{edge}(\mathsf{Y},\mathsf{Z})\cdot \end{array}$$

Let the query be $\mathsf{path}(0,\mathsf{X})$. Then the initial state contains

$$\begin{array}{l} \mathsf{path}(0,\mathsf{V}_1) \wedge \mathsf{answer}(\mathsf{V}_1)\cdot \\ \mathsf{edge}(0,\mathsf{V}_1) \wedge \mathsf{answer}(\mathsf{V}_1)\cdot \\ \mathsf{path}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{answer}(\mathsf{V}_2)\cdot \\ \mathsf{edge}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{answer}(\mathsf{V}_2)\cdot \\ \mathsf{path}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{edge}(\mathsf{V}_2,\mathsf{V}_3) \wedge \mathsf{answer}(\mathsf{V}_3)\cdot \\ \mathsf{edge}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{edge}(\mathsf{V}_2,\mathsf{V}_3) \wedge \mathsf{answer}(\mathsf{V}_3)\cdot \\ \ldots \end{array}$$

However, this is not necessarily a problem if one can work with finite representations of this infinite set. For instance, the left recursive transitive closure works well in our variant of Earley deduction (Brass and Stephan 2013), and the graphs of partially processed rules used there can be seen as encoding an infinite set of SLD goals (if there are cycles in the "called by" relation). Furthermore, we have the following theorem, which shows that for programs without left recursions, the closure operation will not lead to infinite states:

Theorem 1
Suppose that P contains no IDB facts (i.e. all rules have a non-empty body), and no left recursions, i.e. the predicate of the first body literal of each rule does not depend on the predicate defined by the rule (i.e. it does not call — possibly indirectly — that predicate). Then $\mathsf{cl}_\mathsf{P}(G)$ is finite for every finite G.

Suppose for the moment that we can precompute all states (with parameters for the constants from database facts only known at runtime). Then working with maximal states is in some sense as efficent as it can get (when we look only at a single, successful derivation): One of the previously unknown database facts is processed in each step. If none of them is redundant (that depends on the program, e.g. there should be no repeated subgoals), any other query evaluation method must touch the same facts. But methods like "magic sets" also generate a lot of rules which do not contain EDB literals in the body. These rules are applied in addition to the necessary deductions with EDB literals. Furthermore, only a comparison with the magic set version with "supplementary predicates" would be fair (otherwise there are repeated accesses to the same fact in a derivation of a single answer). But then even more intermediate literals derived.

So, where is the hitch? At the moment, we can do the precomputation of states only for certain programs. Extending this set of programs is an interesting research problem.

Furthermore, there is a fundamental difference between SLD resolution and magic sets, namely, SLD resolution proves IDB literals always in the context of a concrete call (the goal contains everything that has to be done after the literal is proven), whereas magic sets derive ID predicates in isolation. Both has sometimes advantages: SLD resolution saves joins to get the proven literal back into the context of the caller, and with a more interesting selection function, conditions on the result can be checked at the best moment, and sometimes this is before the call is fully finished, see (Brass 2000). However, when magic sets have proven an IDB literal, they can use it several times in different contexts. SLD resolution (and thus our approach) has to prove it repeatedly. In the computation of a single answer this probably does not occur often, but when all answers are needed as in deductive databases, this might sometimes lead to suboptimal behaviour. In (Brass 2000) we proposed to mix both approaches, by doing explicit subproofs for certain literals. The same technique would work here. Ideally, we would have an automatic decision which of the two methods is better for a concrete call. This remains a research problem, too.

4 Encoding States as Facts

Our goal now is to study possibilities for encoding states as facts, such that the transition relation can be computed with standard Datalog rules. In this way, the approach becomes a source-to-source transformation like magic sets. However, the resulting rules have a very

simple structure, such that other implementations, like a direct translation to C++, are an interesting option.

Such an encoding is also required because there is normally an infinite number of states: The SLDDB-system models deductions with all possible database facts, e.g. containing arbitrary strings as arguments. Often constants from the facts will be contained in the goals for continuing the proof, and therefore, the number of states is infinite. But for the precomputation at compile time, we need a finite representation of the set of states.

This is done by introducing parameterized states, which are mappings from a certain number of data values (the parameter values or arguments) to states. If we introduce a predicate for a parameterized state (with the arity equal to the number of parameters), we only have to define which concrete state is represented by a fact with this predicate.

Another way to see this is that at "compile time", we do not know the concrete constants from the database (only constants occurring in the program). Therefore, we represent these unknown values by "parameters". Later, at "runtime", we have values for the parameters, and can fill in the "holes" to get a fully specified state.

However, simply replacing parameters in the goals by constants is not the only way how states can be encoded. As explained below, the counting method can be understood as encoding the length of a part of a goal (of very regular structure) in a parameter value. This permits to handle (at least some) cases where the length of the occurring goals depends on the data, therefore, we cannot precompute them at compile time. Note that this is different from the left recursive version of the transitive closure discussed above: There, we had goals of arbitrary length in a single state, and therefore did not need to store any concrete length. In the programs, for which the counting method is made (especially the same generation example), the exact length is important.

4.1 Parameters for Constants Known Only at Runtime

Let us first consider the case where the parameters are simply replaced by constants. We assume that there are special variables $\mathsf{C}_1, \mathsf{C}_2, \ldots$ for the parameters (disjoint from the variables $\mathsf{V}_1, \mathsf{V}_2, \ldots$ used for normalization). Now a parameterized state with n parameters is defined by a set of goals in which the variables $\mathsf{C}_1, \ldots, \mathsf{C}_n$ can occur, and the variables $\mathsf{V}_1, \mathsf{V}_2, \ldots$, but no other variables. Furthermore, each goal must satisfy the normalization requirement, i.e. if the variable V_i, $i > 1$, occurs somewhere in a goal, all variables $\mathsf{V}_1, \ldots, \mathsf{V}_{i-1}$ must occur to the left in the same goal. Whereas the scope of standard variables $\mathsf{V}_1, \mathsf{V}_2, \ldots$ is only a single goal, i.e. they are a kind of "local variables", parameters are "global" in the parameterized state (set of goals). Therefore, a normalization is more difficult (we must use a standard order of goals), but of course, we do not construct distinct states which differ only in a renaming of the parameters. Now, if a parameterized state G with n parameters is encoded as predicate p, then the fact $\mathsf{p}(c_1, \ldots, c_n)$ represents $\{g\,\theta \mid g \in G,\ \theta = \{\mathsf{C}_1/c_1, \ldots, \mathsf{C}_n/c_n\}\}$.

The unification procedure must be changed in order to respect the parameters. We must keep in mind that at runtime, there will be constants for them. The first change is that if we need to unify a parameter and a standard variable, we replace the variable by the parameter. The second change is that when we need to unify two parameters, or a parameter and a constant, unification produces a condition for the parameter values, e.g. $\mathsf{C}_i = a$. Thus, the unification procedure does not only yield a substitution for the

standard variables, but also a (consistent) conjunction of conditions for the parameters. For each call to the unification procedure, we must make a case distinction. Either the actual parameter values satisfy the condition (e.g. $C_i = a$), and the unification succeeds, or they do not ($C_i \neq a$), and the unification fails. Note that when we work with sets of goals, some failed unifications do not necessarily lead to the empty result set. If the computation of the successor state needs k unifications, there could be 2^k cases to distinguish, but usually it will be much less, because we can stop as soon as the condition which describes the case becomes inconsistent. Each case might yield a different parameterized state. In the rule that describes the state transition, we have to check the conditions on the parameters, and also the non-equality conditions in order to avoid computing the same SLD derivation twice. See also (Brass and Stephan 2013).

4.2 Other Encodings: Counting

For general recursions, goals might become larger and larger depending on the data, thus it is not possible to precompute them explicitly at compile time (even if we replace unknown constants by parameters). E.g., this happens in the "same generation" example:

$$\mathsf{sg}(X,X) \leftarrow \mathsf{person}(X)\cdot$$
$$\mathsf{sg}(X,Y) \leftarrow \mathsf{parent}(X,X') \wedge \mathsf{sg}(X',Y') \wedge \mathsf{parent}(Y,Y')\cdot$$

However, other types of encodings are possible. For instance, when applying the counting method (Bancilhon et al. 1986; Greco and Zaniolo 1992) to the same generation example, one can view $\mathsf{c\_sg}(C,I)$ as representing

$$\mathsf{sg}(C,Y_1) \wedge \mathsf{parent}(Y_2,Y_1) \wedge \cdots \wedge \mathsf{parent}(Y_{I+1},Y_I) \wedge \mathsf{answer}(Y_{I+1})\cdot$$

5 Conclusions

This paper offers a different view on some previous methods for query evaluation, such as SLDmagic, counting, and a variant of Earley deduction. By introducing a common framework for them, one can compare and combine their features, and this also opens a space for thinking about new, improved methods.

Obviously, there are currently still many (interesting) research questions, and few readymade methods beyond what was already there. Nevertheless, the understanding ist improved, and the potential of the presented ideas seems promising.

Currently, a prototype implementation for the method sketched in Sections 3.6 and 4.1 is being developed (for the class of programs which are at most tail recursive). See

`http://www.informatik.uni-halle.de/~brass/slddb/`.

As a further generalization of the framework, one could start subproofs for certain literals and and possibly reuse their results multiple times, in order to get magic sets, see (Brass 2000). It might also be possible to split goals in pieces and link them together by references to previous states, somewhat similar to (Greco and Zaniolo 1992).

Acknowledgements

I would like to thank Heike Stephan for starting the research on Earley deduction, which inspired the abstraction presented here. I would also like to thank Marcus Lehmann for doing performance tests with the SLDmagic method.

References

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. 1986. Magic sets and other strange ways to implement logic programs. In *Proc. of the 5th ACM Symp. on Principles of Database Systems (PODS'86)*. ACM Press, 1–15.

Brass, S. 1995. Magic sets vs. SLD-resolution. In *Advances in Databases and Information Systems (ADBIS'95)*, J. Eder and L. A. Kalinichenko, Eds. Springer, 185–203.

Brass, S. 2000. SLDMagic — the real magic (with applications to web queries). In *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, W. Lloyd et al., Eds. Number 1861 in LNCS. Springer, Heidelberg, Berlin, 1063–1077.

Brass, S. 2009. Range restriction for general formulas. In *23rd Workshop on (Constraint) Logic Programming (WLP'09)*, A. Wolf and U. Geske, Eds. Universitätsverlag Potsdam, 125–137.

Brass, S. and Stephan, H. 2013. A variant of earley deduction with partial evaluation. In *Datalog ieasoning and Rule Systems - 7th International Conference, RR 2013*, W. Faber and D. Lembo, Eds. LNCS, vol. 7994. Springer-Verlag, 35–49.

Bry, F. 1990. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering 5*, 289–312.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*, 1, 20–74.

Gallagher, J. P. 1993. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'93)*. ACM, 88–98.

Greco, S. and Zaniolo, C. 1992. Optimization of linear logic programs using counting methods. In *Advances in Database Technology — EDBT'92, 3rd Int. Conf.*, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Number 580 in LNCS. Springer-Verlag, 72–87.

Han, J. 1995. Chain-split evaluation in deductive databases. *IEEE Transactions on Knowledge and Data Engineering 7*, 2, 261–273.

Kifer, M., Ramakrishnan, R., and Siberschatz, A. 1988. An axiomatic approach to deciding query safety in deductive databases. In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'88)*. 52–60.

Lei, L., Moll, G.-H., and Kouloumdjian, J. 1990. A deductive database architecture based on partial evaluation. *SIGMOD Record 19*, 3, 24–29.

Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *Proc. of the 16th International Conf. on Very Large Data Bases (VLDB'90)*, D. McLeod, R. Sacks-Davis, and H.-J. Schek, Eds. Morgan Kaufmann, 264–277.

Nguyen, L. A. and Cao, S. T. 2012. Query-subquery nets. In *Computational Collective Intelligence. Technologies and Applications. 4th International Conference, ICCCI 2012, Proceedings, Part I*, N.-T. Nguyen et al., Eds. Number 7653 in LNCS. Springer, 239–248.

Pereira, F. C. N. and Warren, D. H. D. 1983. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL)*. 137–144.

Porter III, H. H. 1986. Earley deduction. `http://web.cecs.pdx.edu/~harry/earley/`.

Ramakrishnan, R., Bancilhon, F., and Siberschatz, A. 1987. Safety of recursive horn clauses with infinite relations. In *Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*. 328–339.

Ross, K. A. 1991. Modular acyclicity and tail recursion in logic programs. In *Proc. of the Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Princ. of Database Systems (PODS'91)*. 92–101.

Sakama, C. and Itoh, H. 1988. Partial evaluation of queries in deductive databases. *New Generation Computing 6*, 249–258.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proc. Third Int. Conf. on Logic Programming (ICLP)*, E. Shapiro, Ed. Number 225 in LNCS. Springer, 84–98.

Customisable Handling of Java References in Prolog Programs

SERGIO CASTRO, KIM MENS and PAULO MOURA*
ICTEAM Institute, Université catholique de Louvain, Belgium
CRACS & INESC TEC, Faculty of Sciences, University of Porto
(*e-mail*: {sergio.castro,kim.mens}@uclouvain.be,pmoura@inescporto.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Integration techniques for combining programs written in distinct language paradigms facilitate the implementation of specialised modules in the best language for their task. In the case of Java-Prolog integration, a known problem is the proper representation of references to Java objects on the Prolog side. To solve it adequately, multiple dimensions should be considered, including reference representation, opacity of the representation, identity preservation, reference life span, and scope of the inter-language conversion policies. This paper presents an approach that addresses all these dimensions, generalising and building on existing representation patterns of foreign references in Prolog, and taking inspiration from similar inter-language representation techniques found in other domains. Our approach maximises portability by making few assumptions about the Prolog engine interacting with Java (e.g., embedded or executed as an external process). We validate our work by extending JPC, an open-source integration library, with features supporting our approach. Our JPC library is currently compatible with three different open source Prolog engines (SWI, YAP and XSB) by means of drivers.

KEYWORDS: Multi-Paradigm Programming, Language Interoperability, Logic Programming, Object-Oriented Programming, Prolog, Java

1 Introduction

Writing program modules in the language best suited for their task can greatly facilitate their implementation (Mernik et al. 2005). However, integrating modules written in different languages is not trivial when such languages belong to different paradigms (Gybels 2003). This is especially the case for Prolog programs integrated with an object-oriented language such as Java (Denti et al. 2005). One of the main problems of this integration is the proper representation of foreign language artefacts in the logic language, such that they can be conveniently manipulated and interpreted (Gybels 2003).

The scope of this work concerns a portable approach to simplify the management and representation of Java object references in Prolog. Studying existing solutions to this problem in Prolog, similar logic languages (e.g., SOUL (Roover et al. 2011)) and even inter-language conversion libraries in other domains (e.g., Google's GSON library (Google Inc. 2012)), we have identified the following dimensions to be tackled: 1) reference

* This work is partially funded by ERDF through the CMPETE Programme and by FCT within project FCOMP-01-0124-FEDER-037281).

representation; 2) opacity of the representation; 3) identity preservation; 4) reference life span and 5) scope of the inter-language conversion policies. To maximise portability, our approach does not make any simplifying assumption regarding the architecture of the Prolog engine (e.g., such as it being embedded in the JVM). We validate our work by extending our JAVA PROLOG CONNECTIVITY (JPC)[1] integration library (Castro et al. 2013) with customisable support for managing Java references in Prolog.

This paper is structured as follows. Section 2 discusses the main Java reference representation issues in Prolog. Section 3 presents an overview of JPC's architecture. JPC's approach for custom management of Java references in Prolog is discussed in section 4. Section 5 discusses related work. Section 6 summarizes our conclusions and future work.

2 The Problem of Representing Java References in Prolog

In this section, we identify the different dimensions to be taken into consideration when looking at the problem of representing Java references in Prolog (figure 1). These dimensions have been extracted and generalised from existing solutions to this problem both in Prolog and other inter-language representation domains.

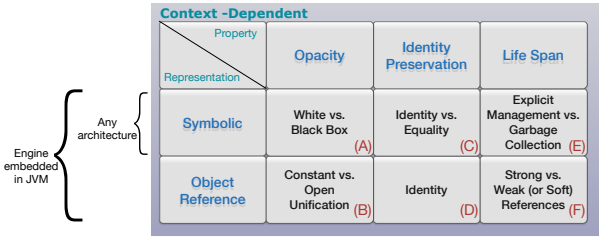


Fig. 1. Reference Management Dimensions

2.1 Reference Representation

A first important dimension is how Java objects are represented on the logic side. Several integration libraries allow to reify Java objects in Prolog using a symbolic term representation (Singleton et al. 2004; Carlsson et al. 1995; Calejo 2004). As show in figure 1, such approach has the advantage of not relying on any specific Prolog engine architecture.

Alternatively, Prolog implementations running in the JVM may support the storage of direct object references (e.g., Jinni (Tarau 2004) and LeanProlog (Tarau 2011)). An advantage of this representation scheme is that there are no performance penalties associated to the marshalling/unmarshalling of Java objects to/from the Prolog engine.

2.2 Opacity of the Representation

A second important dimension is the degree of opacity of the representation (i.e., the degree of data exposed). For symbolic term representations *(A)*, frequently a fine-grained reification of the internal object structure (i.e., a white box representation) is desired. For

[1] https://github.com/java-prolog-connectivity

example, JTransformer (Kniesel et al. 2007) allows to reason over the structure of terms reifying objects modelling a Java abstract syntax tree. However, if inspecting the object's structure on the Prolog side is not required, having an opaque reference (i.e., a black box representation) to the corresponding Java object is preferable (e.g., an opaque reference to a GUI component on the Java side). In those cases, an automatic mechanism to generate opaque term representations of Java objects is desirable.

When the Prolog engine is embedded in a JVM, a more direct kind of reference to Java objects can be established *(B)*. In the simplest case, the object reference can be considered and unified as a special constant term. In spite of the more direct mapping (no automated mapping to generate the reference is required; the term wraps the object 'as is'), this case is conceptually equivalent to mapping the object reference to an opaque term representation. But we may want to combine the best of both worlds and have direct references to the actual Java objects, while still allowing Prolog programs to reason over the internal structure of such objects. Approaches such as Soul (Roover et al. 2011) have achieved this through the mechanism of *open unification* (Brichau et al. 2007). This approach consists in allowing the programmer to customise not the term representation of an object, but rather its unification mechanism. In a nutshell, the unification mechanism is opened up so that Java objects are not regarded as constants but can be unified with structured logic terms of the right form.

2.3 Object Identity Preservation

For logic engines running in the JVM *(D)* object references are preserved automatically since the term wraps the object 'as is'. For engines not embedded in the JVM, a programmer needs to decide if an object reified as a term should preserve its identity when the term is translated back to a Java object *(C)*. In many situations, it is not important to preserve such identity (e.g., instances of `String`) and a different reference, considered equivalent to the original object (e.g., by means of the `equals` method), is acceptable. However, in certain cases, keeping track of the original reference is required to guarantee the expected behaviour of the program (e.g., if the reference points to a GUI component). Furthermore, passing around symbolic representations of object references is often more efficient than marshalling and unmarshalling large Java objects. Note that the need for preserving the original object identity is orthogonal to the required opacity of the representation. I.e., independently if the reference should be preserved or not, the programmer should still be able to decide on the best representation of the object on the Prolog side.

2.4 Reference Life Span

A fourth dimension is the life span of Prolog references to Java objects. For a symbolic term representation, a programmer should decide on a mechanism for delimiting the life span of a mapping between a Java reference and a Prolog term *(E)*. This mechanism can be explicit (e.g., an API allowing to request to 'forget' a mapping) or rely on JVM garbage collection mechanisms. An explicit mechanism enables a fine-grained control over the life span of a reference. For example, a symbolic term representation of an object that is not explicitly referenced in a program (i.e., normally to be scheduled for garbage collection) can still remain valid until explicitly discarded. Alternatively, a reference life span may be automatically delimited by the JVM garbage-collection mechanism (e.g., a reference to the application main window). For an object reference representation *(F)*, the programmer

may want to keep the reference alive as long as it is present in the Prolog database (i.e., a strong reference). However, in certain scenarios a Java reference stored in Prolog should not prevent it from being garbage collected (e.g., the reference points to a disposed GUI component). In that case, the reference should be invalidated when it is reclaimed by the garbage collector. A programmer may also want to define customisable cleaning tasks to be automatically executed when a reference is garbage collected. For example, clauses containing dead references may be automatically retracted from the Prolog database to avoid unexpected behaviours (e.g., null pointer exceptions). Furthermore, references that may be reclaimed by the garbage collector should be classified according to the Java (garbage-collected) reference types: *Weak* for eagerly collected references (discarded at the next garbage collection cycle) and *Soft* for references not aggressively reclaimed (only collected when the memory is tight).[2]

2.5 Scope of the Inter-Language Conversion Policies

We claim that it is useful for a programmer to be able to choose different reference management policies in different parts of the program. To achieve that, it is needed a simple mechanism for scoping and encapsulating the best reference handling policy for certain objects. Besides greater flexibility, this facilitates performance tuning and testing (e.g., generating mocking representations of references). Next, we will introduce the architecture of a library that supports a customisable management of all these dimensions.

3 Architecture

JPC is an integration library supporting the development of hybrid Java–Prolog programs. It provides different levels of abstractions, simplifying the implementation of common inter-operability tasks. To set the ground for discussing the JPC features for Java reference management in Prolog, this section overviews its main components (figure 2).

3.1 Prolog VM Abstraction

Several integration libraries rely on the notion of a Prolog engine as a convenient abstraction for interacting with a Prolog virtual machine from Java (Tarau 2004; Rho et al. 2004; Calejo 2004). In JPC, a programmer interacts with a Prolog engine abstraction that communicates with concrete Prolog engines using drivers. With portability in mind, when modelling such an abstract Prolog engine we tried to find a compromise between (1) offering convenient features facilitating the interaction from Java programs and (2) not assuming a specific implementation architecture of the underlying Prolog engine. Our Prolog engine abstraction provides a general purpose API for interacting with Prolog. However, as illustrated in section 4, JPC also supplies a higher level API that simplifies certain tasks (e.g., inter-language conversions). JPC defines a set of classes reifying Prolog data types: `Term`, `Atom`, `Compound`, `IntegerTerm`, `FloatTerm`, `Var`, `JRef` (a Java reference term; a special kind of term wrapping a Java reference).

3.2 Embedded Prolog Database

JPC uses an embedded Prolog database running on the JVM and supporting the storage of Java object references in addition to standard Prolog terms. Several JPC interoperability

[2] http://docs.oracle.com/javase/7/docs/api/java/lang/ref/Reference.html

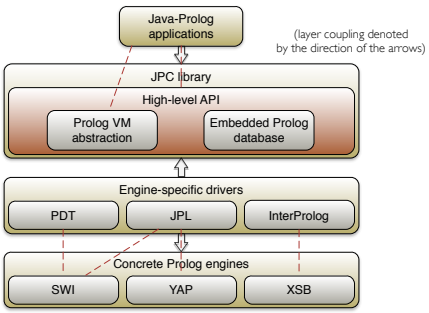


Fig. 2. The JPC architecture

features rely on this component, which maintains mappings between Prolog terms and arbitrary Java objects (represented as `JRef` terms).

4 Reference Management with JPC

This section describes JPC's support for the different dimensions related to the management of Java references in Prolog (figure 1).

4.1 Symbolic Representation

To illustrate the properties of symbolic references (identified by the first row of figure 1), we start by defining a `Person` class (listing 1) declaring `name` as its only instance variable.

```
1 public  class Person implements Serializable {
2     private final String name;
3     public Person(String name) {this.name = name;}
4     ...
5     @Override
6     public boolean equals(Object obj) {
7         ... return ((Person)obj).name.equals(name); //simplified implementation
8     }
9 }
```

Listing 1. The `Person` class

The `PersonConverter` class (listing 2) defines how instances of class `Person` are translated to a Prolog compound term (lines 5–7) and back (lines 8–10). According to our classification in section 2.2, the term reification of a person, according to this converter, corresponds to a white box representation since it exposes its internal data.

```
1  public class PersonConverter implements FromTermConverter<Compound, Person>,
2          ToTermConverter<Person, Compound> {
3      public static final String PERSON_FUNCTOR_NAME = "person";
4
5      @Override public Compound toTerm(Person person, Class<Compound> termClass, Jpc context) {
6          return new Compound(PERSON_FUNCTOR_NAME, asList(new Atom(person.getName())));
7      }
8      @Override public Person fromTerm(Compound personTerm, Type targetType, Jpc context) {
9          return new Person(((Atom)((Compound)personTerm).arg(1)).getName());
10     }
11 }
```

Listing 2. The `PersonConverter` class

Listing 3 illustrates a white box term representation of a Java object, without object identity preservation (the first three lines are common to most examples; we will not repeat them). A central artefact in our approach is a *conversion context*, instantiated in line 4 using a builder class and configured with the `PersonConverter` converter. With this context we obtain the conversion of a person in line 5 (`person(mary)`). Next, we assert the fact `student(person(mary))` (line 6). A `student(A)` goal is instantiated in line 7 passing the context defined before. A person is queried in line 8 using a deterministic query. The `selectObject()` method adapts each solution to the query as an object whose term reification is given as a string. This adaptation corresponds to the conversion as a Java object of the term that has been bound to the `Person` variable in the solution. Lines 9 and 10 verify that the queried and the original persons are equal, although with different identities.

```
1  final String STUDENT_FUNCTOR_NAME = "student";
2  PrologEngine prologEngine = getPrologEngine();
3  Person mary = new Person("Mary");
4  Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
5  Term personTerm = ctx.toTerm(mary);
6  prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
7  Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx);
8  Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
9  assertEquals(mary, queriedPerson);
10 assertFalse(mary == queriedPerson);
```

Listing 3. White Box without Identity Preservation

Listing 4 illustrates the mapping of a reference to a term representation (line 2) in the scope of a context. The `newRefTerm()` method associates a person reference (first argument) to an arbitrary (compound) term representation (second argument). In this example, the term corresponds to the term conversion of the reference according to a given conversion context (obtained by the `toTerm()` method of the context instance). We verify that this time the queried person corresponds to the original person reference in line 6.

```
1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newRefTerm(person, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx);
5 Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
6 assertTrue(mary == queriedPerson);
```

Listing 4. White Box and Identity Preservation

An example of a black box representation is shown in listing 5. Here, we assert a term of the form `student(serialisation)`, where the compound argument corresponds to the term representation of the serialisation of a `Person` instance. No converter is passed to the query in line 2. This is because the default conversion context (employed by the query if no context is explicitly passed) includes a converter able to deserialize a Java object from the term representation of its serialisation. Finally, we verify that our queried person is equal to the original person (line 4) although having different identities (line 5).

Although in the context of this example we have presented this term reification as a black box representation, note that in other contexts this may be considered as a white box. This would be the case if the Prolog side is intended to interpret such representation (e.g., if it reasons over the serialised bytes of the object (Calejo 2004)).

```
1 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(SerializedTerm.serialize(mary))));
2 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))));
3 Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
4 assertEquals(mary, queriedPerson);
5 assertFalse(mary == queriedPerson);
```

Listing 5. Black Box without Identity Preservation

A programmer can also associate an automatically generated term to a reference. An example is given in listing 6. This time we invoke the method `newRefTerm()` passing as only argument the reference to reify as a term (line 2). A (black box) term representation is generated behind the curtains. Our library guarantees that such generated term representations are identical for the same object even across different contexts.

```
1 Jpc ctx = JpcBuilder.create().build();
2 Term personTerm = ctx.newRefTerm(mary);
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx);
5 Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
6 assertTrue(mary == queriedPerson);
```

Listing 6. Black Box and Identity Preservation

As discussed in section 2.4, a programmer should also be able to control the life span of term–reference mappings. Listing 7 shows an example. We use the `newRefTerm()` method (line 2) to associate a reference to its (context dependent) term reification. But afterwards we delete this association using the `forgetRefTerm()` method (line 5). Thus, although the queried person is equal to the original person (line 7) since the term is translated according to the conversion context (line 1), they do not have the same identity (line 8) as the association between the term and the original reference was eliminated.

```
1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newRefTerm(person, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 assertTrue(mary == prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))),
      ctx).selectObject("Person").oneSolutionOrThrow());
5 ctx.forgetRefTerm((Compound)personTerm);
6 Person queriedPerson = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new
      Var("Person"))), ctx).<Person>selectObject("Person").oneSolutionOrThrow();
7 assertEquals(mary, queriedPerson);
8 assertFalse(mary == queriedPerson);
```

Listing 7. Explicit Management of Associations Life Span

A programmer can also rely on the Java garbage collection mechanism for delimiting the life span of an association as shown in listing 8. The `newWeakRefTerm()` method (line 2) is equivalent to the `newRefTerm()` method discussed earlier. But in this case the association between a term and a reference persists as long as the reference is not reclaimed in the next garbage collection cycle. To prove it, we assign `null` to the only variable keeping a reference to the person (line 4) and give a hint to the garbage collector to start a cycle (line 5). Note that the query is not instantiated with a conversion context (line 7). Thus, an exception is raised when we try to convert the term (bound to the variable `Person`) to an object as no converter is found and no reference is associated to such term. Our framework also provides the `newSoftRefTerm()` method with similar semantics than `newWeakRefTerm()`, with

the only difference that an association between a term and a reference may persist some time after a garbage collection cycle, and will be deleted only if the memory gets tight.

```
1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newWeakRefTerm(mary, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 mary = null;
5 System.gc();
6 try {
7     prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new
          Var("Person")))).<Person>selectObject("Person").oneSolutionOrThrow();
8     fail();
9 } catch(ConversionException e) {}
```

Listing 8. Garbage Collection Management of Associations Life Span

4.2 Object Reference Representation

This section focuses on the properties of object references (identified by the second row of figure 1). Although our library currently only has drivers for non-embedded Prolog engines, as a proof of concept we implement the examples in this section using the JPC embedded Prolog database described in section 3.1. With the exception of open unification, all the other properties are supported by our implementation.

We start with an example of constant unification of references in listing 9. As mentioned in section 3.1, a JPC`JRef` term wrapps an object reference. In our current version, they are unified as constants (i.e., unifying tow `JRef` terms succeeds if their referred objects are equal). In line 1 we assert that *mary* (wrapped in a `JRef` term) is a student. In line 2 we query if a different person object with the same name is a student, which succeeds.

```
1 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(mary))));
2 assertTrue(prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(new
      Person("mary"))))).hasSolution());
3 Solution solution = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new
      Var("X")))).oneSolutionOrThrow();
4 JRef<Person> jRef = (JRef<Person>) solution.get("X");
5 assertTrue(mary == jRef.getReferent());
```

Listing 9. Constant Unification of `JRef` terms

Thanks to our embedded Prolog database, the identity of a reference is trivially preserved. To illustrate this, we execute a deterministic query (line 3) with goal `student(X)`. We verify that the obtained referent has the same identity as *mary* in line 5.

Listing 10 shows how to create `JRef` instances that may be garbage collected. We first create two objects equal to *mary* and assert them, using two kind of references: *strong* (line 3) and *weak* (line 4). When we query for students unifying with *mary* (line 5) using a strong reference, we get two results instead of one. This is because the unification semantics of `JRef` terms evaluates the referents, not the actual `JRef` term wrapper. Afterwards we assign to `null` the variable `person2` (line 6) and give a hint to the garbage collector to execute a cycle (line 7). Since the referent of the `JRef` term asserted in line 4 has been invalidated, the number of students unifying with *mary* is now only 1 (line 8). Note that weak or soft references should be used with care: they may require non-monotonic reasoning as the referent of a `JRef` term may be invalidated during the query execution.

```
1 Person person2 = new Person("Mary");
2 Person person3 = new Person("Mary");
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(mary))));
4 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.weakJRef(person2))));
5 assertEquals(2, prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME,
      asList(JRef.jRef(mary)))).allSolutions().size());
6 person2 = null;
7 System.gc();
8 assertEquals(1, prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME,
      asList(JRef.jRef(mary)))).allSolutions().size());
```

Listing 10. Life Span of `JRef` terms

The previous example motivates the need of a cleaning mechanism. Listing 11 illustrates such mechanism using a user-defined cleaning task. To keep our example simple, this cleaning task retracts all the asserted students (lines 1–5) when a reference is invalidated. A more sophisticated example would retract only the invalidated reference. Our cleaning task is associated with a weak reference in line 6. In line 9 we verify that no students are in the database after the reference to *mary* has been invalidated (lines 7–8).

```
1 Runnable cleaningTask = new Runnable() {
2     @Override public void run() {
3         prologEngine.retractAll(new Compound(STUDENT_FUNCTOR_NAME, asList(Var.ANONYMOUS_VAR)));
4     }
5 };
6 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.weakJRef(mary, cleaningTask))));
7 mary = null;
8 System.gc();
9 assertFalse(prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME,
      asList(Var.ANONYMOUS_VAR))).hasSolution());
```

Listing 11. Cleaning Tasks

5 Related Work

Most related work has already been overviewed in sections 2 and 3.1 so we do not repeat it here. InterProlog inspired the serialisation mechanism illustrated in listing 5. It provides a more structured representation of a serialised object on the Prolog side using a definite clause grammar. Currently we represent serialised bytes as an atom using a raw base-64 encoding. InterProlog has limited support, however, for customising the reification as a term of arbitrary Java objects (even not serialisable ones) as in our approach. Concerning our mechanisms for custom two-way conversions between inter-language artefacts, this was inspired by Google's Gson library, which aims to provide a high-level tool for conversions between Java objects and their JSON representation.

6 Conclusions and Future Work

This work discusses different dimensions that should be taken into consideration when dealing with Java references in Prolog programs. These dimensions have been extracted from many sources, including our own experience, a study of existing approaches, and even existing solutions in other domains. At the moment, JPC does not implement a mechanism for interacting with Java from the Prolog side. In line with our portability goal, we plan to implement our Prolog side API using Logtalk (Moura 2003), a portable object-oriented layer for Prolog. As in the current Java side API, we expect to prototype a first version by

reusing existing bridge libraries. We will also continue improving our embedded Prolog database so that it can be released as a stand-alone embedded Prolog engine. We hope that our work will benefit not only implementors of Java–Prolog integration libraries, but also integrators of similar object-oriented and logic languages.

References

Brichau, J., De Roover, C., and Mens, K. 2007. Open Unification for Program Query Languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*.

Calejo, M. 2004. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. In *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, José Júlio Alferes and João Alexandre Leite, Ed. Lecture Notes in Computer Science, vol. 3229. Springer, 714–717.

Carlsson, M. et al. 1995. *SICStus Prolog User's Manual*, Release 3 ed. Swedish Institute of Computer Science. ISBN 91-630-3648-7.

Castro, S., Mens, K., and Moura, P. 2013. JPC: A Library for Modularising Inter-Language Conversion Concerns between Java and Prolog. In *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*.

Denti, E., Omicini, A., and Ricci, A. 2005. Multi-paradigm Java–Prolog Integration in tuProlog. *Science of Computer Programming 57,* 2, 217 – 250.

Google Inc. 2012. Gson 2.2.2: A Java library to convert JSON strings to Java objects and vice-versa. `http://code.google.com/p/google-gson/`.

Gybels, K. 2003. SOUL and Smalltalk — Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*.

Kniesel, G., Hannemann, J., and Rho, T. 2007. A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*. LATE'07. ACM, New York, NY, USA.

Mernik, M., Heering, J., and Sloane, A. M. 2005. When and How to Develop Domain-specific Languages. *ACM Comput. Surv. 37,* 4 (Dec.), 316–344.

Moura, P. 2003. Logtalk – Design of an Object-Oriented Logic Programming Language. Ph.D. thesis, Department of Computer Science, University of Beira Interior, Portugal.

Rho, T., Degener, L., Günter Kniesel, Frank Mühlschlegel, Eva Stöwe, Noth, F., Becker, A., and Alyiev, I. 2004. The Prolog Development Tool – A Prolog IDE for Eclipse. `http://sewiki.iai.uni-bonn.de/research/pdt/`.

Roover, C. D., Noguera, C., Kellens, A., and Jonckers, V. 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *International Conference on Principles and Practices of Programming on the Java Platform*. 71–80.

Singleton, P., Dushin, F., and Wielemaker, J. 2004. JPL 3.0: A Bidirectional Interface Between Prolog and Java. `http://www.swi-prolog.org/packages/jpl/java_api/`.

Tarau, P. 2004. Agent Oriented Logic Programming Constructs in Jinni 2004. In *International Conference of Logic Programming*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 477–478.

Tarau, P. 2011. Integrated Symbol Table, Engine and Heap Memory Management in Multi-engine Prolog. In *Proceedings of the 10th International Symposium on Memory Management*. ACM, 129–138.

*Analysis and Transformation Tools for Constrained Horn Clause Verification**

John P. Gallagher
Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain
(*e-mail:* jpg@ruc.dk)

Bishoksan Kafle
Roskilde University, Denmark
(*e-mail:* kafle@ruc.dk)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Several techniques and tools have been developed for verification of properties expressed as Horn clauses with constraints over a background theory (CHC). Current CHC verification tools implement intricate algorithms and are often limited to certain subclasses of CHC problems. Our aim in this work is to investigate the use of a combination of off-the-shelf techniques from the literature in analysis and transformation of Constraint Logic Programs (CLPs) to solve challenging CHC verification problems. We find that many problems can be solved using a combination of tools based on well-known techniques from abstract interpretation, semantics-preserving transformations, program specialisation and query-answer transformations. This gives insights into the design of automatic, more general CHC verification tools based on a library of components.

KEYWORDS: Constraint Logic Program, Constrained Horn Clause, Abstract Interpretation, Software Verification.

1 Introduction

CHCs provide a suitable intermediate form for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). As a result it has been used as a target language for software verification. Recently there is a growing interest in CHC verification from both the logic programming and software verification communities, and several verification techniques and tools have been developed for CHC.

Pure CLPs are syntactically and semantically the same as CHC. The main difference is that sets of constrained Horn clauses are not necessarily intended for execution, but rather as specifications. From the point of view of verification, we do not distinguish

* The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement no. 318337, ENTRA - Whole-Systems Energy Transparency and the Danish Natural Science Research Council grant NUSA: Numerical and Symbolic Abstractions for Software Model Checking.

between CHC and pure CLP. Much research has been carried out on the analysis and transformation of CLP programs, typically for synthesising efficient programs or for inferring run-time properties of programs for the purpose of debugging, compile-time optimisations or program understanding. In this paper we investigate the application of this research to the CHC verification problem.

In Section 2 we define the CHC verification problem. In Section 3 we define basic transformation and analysis components drawn from or inspired by the CLP literature. Section 4 discusses the role of these components in verification, illustrating them on an example problem. In Section 5 we construct a tool-chain out of these components and test it on a range of CHC verification benchmark problems. The results reported represent one of the main contributions of this work. In Section 6 we propose possible extensions of the basic tool-chain and compare them with related work on CHC verification tool architectures. Finally in Section 7 we summarise the conclusions from this work.

2 Background: The CHC Verification Problem

A CHC is a first order predicate logic formula of the form $\forall(\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k) \rightarrow H(X))$ $(k \geqslant 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $B_1, \ldots, B_k, H$ are predicate symbols, $H(X)$ is the head of the clause and $\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k)$ is the body. Sometimes the clause is written $H(X) \leftarrow \phi \wedge B_1(X_1), \ldots, B_k(X_k)$ and in concrete examples it is written in the form `H :-` ϕ`,` $\mathtt{B}_1\mathtt{(X}_1\mathtt{)}$`,...,`$\mathtt{B}_k\mathtt{(X}_k\mathtt{)}$. In examples, predicate symbols start with lowercase letters while we use uppercase letters for variables.

We assume here that the constraint theory is linear arithmetic with relation symbols $\leqslant$, $\geqslant$, $>$, $<$ and $=$ and that there is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$ is equivalent to the formula $\mathsf{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$. The latter might not be a CHC but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula `X=Y :- p(X,Y)` is equivalent to the set of CHCs `false :- X>Y, p(X,Y)` and `false :- X<Y, p(X,Y)`. Integrity constraints can be viewed as safety properties. If a set of CHCs encodes the behaviour of some system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses.

The CHC verification problem. To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. We restate this property in terms of the derivability of the predicate false.

Lemma 2.1

P has a model if and only if $P \not\models \mathsf{false}$.

Proof
Let us write $I(F)$ to mean that interpretation I satisfies F (I is a model of F).

$$\begin{aligned} P \not\models \mathsf{false} &\equiv \exists I.(I(P) \text{ and } \neg I(\mathsf{false})) \\ &\equiv \exists I.I(P) \quad (\textit{since } \neg I(\mathsf{false}) \textit{ is true by defn. of } \mathsf{false}) \\ &\equiv P \text{ has a model.} \end{aligned}$$

□

Obviously any model of P assigns false to the bodies of integrity constraints.

The verification problem can be formulated deductively rather than model-theoretically. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure. If the proof procedure is sound and complete then $P \not\models A$ if and only if $P \not\vdash A$. So the verification problem is to show (using CLP terminology) that the computation of the goal $\leftarrow \mathsf{false}$ in program P does not succeed using a complete proof procedure. Although in this work we follow the model-based formulation of the problem, we exploit the equivalence with the deductive formulation, which underlies, for example, the query-answer transformation and specialisation techniques to be presented.

2.1 Representation of Interpretations

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathscr{C}$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and $\mathscr{C}$ is a constraint over $Z_1, \ldots, Z_n$. If $\mathscr{C}$ is true we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \mathscr{C}$ is shorthand for the set of variable-free facts $A\theta$ such that $\mathscr{C}\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator, $T_P^{\mathscr{C}}$, which is an extension of the standard T_P operator from logic programming, extended to handle constraints (Jaffar and Maher 1994). Furthermore $\mathsf{lfp}(T_P^{\mathscr{C}})$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, T_P^{\mathscr{C}}(\emptyset), T_P^{\mathscr{C}}(T_P^{\mathscr{C}}(\emptyset)), \ldots$. For more details, see (Jaffar and Maher 1994). This sequence provides a basis for abstract interpretation of CHC clauses.

Proof by over-approximation of the minimal model. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(v_1, \ldots, v_n)$ if and only if $p(v_1, \ldots, v_n) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathsf{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathsf{false} \notin M'$. This technique is called proof by over-approximation of the minimal model.

3 Relevant tools for CHC Verification

In this section, we give a brief description of some relevant tools borrowed from the literature in analysis and transformation of CLP.

Unfolding. Let P be a set of CHCs and $c_0 \in P$ be $H(X) \leftarrow \mathscr{B}_1, p(Y), \mathscr{B}_2$ where $\mathscr{B}_1, \mathscr{B}_2$ are possibly empty conjunctions of atomic formulas and constraints. Let $\{c_1, \ldots, c_m\}$ be the set of clauses of P that have predicate p in the head, that is, $c_i = p(Z_i) \leftarrow \mathscr{D}_i$, where the variables of these clauses are standardised apart from the variables of c_0 and from each other. Then the result of unfolding c_0 on $p(Y)$ is the set of CHCs $P' = P \setminus \{c_0\} \cup \{c'_1, \ldots, c'_m\}$ where $c'_i = H(X) \leftarrow \mathscr{B}_1, Y = Z_i, \mathscr{D}_i, \mathscr{B}_2$. The equality $Y = Z_i$ stands for the conjunction of the equality of the respective elements of the vectors Y and Z_i. It is a standard result that unfolding a clause in P preserves P's minimal model (Pettorossi and Proietti 1999). In particular, $P \models \mathsf{false} \equiv P' \models \mathsf{false}$.

Specialisation. A set of CHCs P can be specialised with respect to a query. Assume A is an atomic formula; then we can derive a set P_A such that $P \models A \equiv P_A \models A$. P_A could be simpler than P, for instance, parts of P that are irrelevant to A could be omitted in P_A. In particular, the CHC verification problem for P_{false} and P are equivalent. There are many techniques in the CLP literature for deriving a specialised program P_A. Partial evaluation is a well-developed method (Gallagher 1993; Leuschel 1999).

We make use a form of specialisation know as forward slicing, more specifically redundant argument filtering (Leuschel and Sørensen 1996), in which predicate arguments can be removed if they do not affect a computation. Given a set of CHCs P and a query A, denote by P_A^{raf} the program obtained by applying the RAF algorithm from (Leuschel and Sørensen 1996) with respect to the goal A. We have the property that $P \models A \equiv P_A^{\mathsf{raf}} \models A$ and in particular that $P \models \mathsf{false} \equiv P_{\mathsf{false}}^{\mathsf{raf}} \models \mathsf{false}$.

Query-answer transformation. Given a set of CHCs P and an atomic query A, the query-answer transformation of P with respect to A is a set of CHCs which simulates the computation of the goal $\leftarrow A$ in P, using a left-to-right computation rule. Query-answer transformation is a generalisation of the magic set transformations for Datalog. For each predicate p, two new predicates p_{ans} and p_{query} are defined. For an atomic formula A, A_{ans} and A_{query} denote the replacement of A's predicate symbol p by p_{ans} and p_{query} respectively. Given a program P and query A, the idea is to derive a program P_A^{qa} with the following property $P \models A$ iff $P_A^{\mathsf{qa}} \models A_{ans}$. The A_{query} predicates represent calls in the computation tree generated during the execution of the goal. For more details see (Debray and Ramakrishnan 1994; Gallagher and de Waal 1993; Codish and Demoen 1993). In particular, $P_{\mathsf{false}}^{\mathsf{qa}} \models \mathsf{false}_{ans} \equiv P \models \mathsf{false}$, so we can transform a CHC verification problem to an equivalent CHC verification problem on the query-answer program generated with respect to the goal $\leftarrow \mathsf{false}$.

Predicate splitting. Let P be a set of CHCs and let $\{c_1, \ldots, c_m\}$ be the set of clauses in P having some given predicate p in the head, where $c_i = p(X) \leftarrow \mathscr{D}_i$. Let $C_1, \ldots, C_k$ be some partition of $\{c_1, \ldots, c_m\}$, where $C_j = \{c_{j_1}, \ldots, c_{j_{n_j}}\}$. Define k new predicates $p_1 \ldots p_k$, where p_j is defined by the bodies of clauses in partition C_j, namely $Q^j = \{p_j(X) \leftarrow \mathscr{D}_{j_1}, \ldots, p_j(X) \leftarrow \mathscr{D}_{j_{n_j}}\}$. Finally, define k clauses $C_p = \{p(X) \leftarrow p_1(X), \ldots, p(X) \leftarrow p_k(X)\}$. Then we define a splitting transformation as follows.

1. Let $P' = P \setminus \{c_1, \ldots, c_m\} \cup C_p \cup Q^1 \cup \ldots \cup Q^k$.
2. Let P^{split} be the result of unfolding every clause in P' whose body contains $p(Y)$ with the clauses C_p.

In our applications, we use splitting to create separate predicates for clauses for a given predicate whose constraints are mutually exclusive. For example, given the clauses `new3(A,B) :- A=<99, new4(A,B)` and `new3(A,B) :- A>=100, new5(A,B)`, we produce two new predicates, since the constraints `A=<99` and `A>=100` are disjoint. The new predicates are defined by clauses $\texttt{new3}_1\texttt{(A,B) :- A=<99, new4(A,B)}$ and $\texttt{new3}_2\texttt{(A,B) :- A>=100, new5(A,B)}$, and all calls to `new3` throughout the program are unfolded using these new clauses. Splitting has been used in the CLP literature to improve the precision of program analyses, for example in (Serebrenik and De Schreye 2001). In our case it improves the precision of the convex polyhedron analysis discussed below, since separate polyhedra will be maintained for each of the disjoint cases. The correctness of splitting can be shown using standard transformations that preserve the minimal model of the program (with respect to the predicates of the original program) (Pettorossi and Proietti 1999). Assuming that the predicate false is not split, we have that $P \models \mathsf{false} \equiv P^{\mathsf{split}} \models \mathsf{false}$.

Convex polyhedron approximation. Convex polyhedron analysis (Cousot and Halbwachs 1978) is a program analysis technique based on abstract interpretation (Cousot and Cousot 1977). When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \mathscr{C}$ for each predicate p. The constraint $\mathscr{C}$ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by Benoy and King (1996). Since the domain of convex polyhedra contains infinite increasing chains, the use of a widening operator is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds (Halbwachs et al. 1994).

Recently, a technique for deriving more effective thresholds was developed (Lakhdar-Chaouch et al. 2011), which we have adapted and found to be effective in experimental studies. The thresholds are computed by the following method. Let $T_P^{\mathscr{C}}$ be the standard immediate consequence operator for CHCs, that is, $T_P^{\mathscr{C}}(I)$ is the set of constrained facts that can be derived in one step from a set of constrained facts I. Given a constrained fact $p(\bar{Z}) \leftarrow \mathscr{C}$, define $\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow \mathscr{C})$ to be the set of constrained facts $\{p(\bar{Z}) \leftarrow C_i \mid \mathscr{C} = C_1 \wedge \ldots \wedge C_k, 1 \leqslant i \leqslant k)\}$. The function atomconstraints is extended to interpretations by $\mathsf{atomconstraints}(I) = \bigcup_{p(\bar{Z}) \leftarrow \mathscr{C} \in I} \{\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow \mathscr{C})\}$.

Let $I_\top$ be the interpretation consisting of the set of constrained facts $p(\bar{Z}) \leftarrow \mathsf{true}$ for each predicate p. We perform three iterations of $T_P^{\mathscr{C}}$ starting with $I_\top$ (the first three elements of a "top-down" Kleene sequence) and then extract the atomic constraints. That is, thresholds is defined as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(T_P^{\mathscr{C}(3)}(I_\top))$$

A difference from the method in (Lakhdar-Chaouch et al. 2011) is that we use the concrete semantic function $T_P^{\mathscr{C}}$ rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. See (Lakhdar-Chaouch et al. 2011) for further details. Threshold constraints that are not invariants are simply discarded during widening.

```
new6(A,B) :- B=<99.                    new4(A,B) :- C=1+A,D=1+B,A>=50,new3(C,D).
new5(A,B) :- B>=101.                   new3(A,B) :- A=<99, new4(A,B).
new5(A,B) :- B=<100, new6(A,B).        new3(A,B) :- A>=100, new5(A,B).
new4(A,B) :- C=1+A, A=<49, new3(C,B).  false :- A=0, B=50, new3(A,B).
```

Fig. 1. The example program `MAP-disj.c.map.pl`

4 The role of CLP tools in verification

The techniques discussed in the previous section play various roles. The convex polyhedron analysis, together with the helper tool to derive threshold constraints, constructs an approximation of the minimal model of a CHC theory. If false (or false_{ans}) is not in the approximate model, then the verification problem is solved. Otherwise the problem is not solved; in effect a "don't know" answer is returned. We have found that polyhedron analysis alone is seldom precise enough to solve non-trivial CHC verification problems; in combination with the other tools, it is very effective.

Unfolding can improve the structure of a program, removing some cases of mutual recursion, or propagating constraints upwards towards the integrity constraints, and can improve the precision and performance of convex polyhedron analysis.

Specialisation can remove parts of theories not relevant to the verification problem, and can also propagate constraint downwards from the integrity constraints. Both of these have a beneficial effect on performance and precision of polyhedron analysis.

Analysis of a query-answer program (with respect to false) is in effect the search for a derivation tree for false. Its effectiveness in CHC verification problems is variable. It can sometimes worsen performance since the query-answer transformed program is larger and contains more recursive dependencies than the original. On the other hand, one seldom loses precision and it is often more effective in allowing constraints to be propagated upwards (through the *ans* predicates) and downwards (through the *query* predicates).

4.1 Application of the tools

We illustrate the tools on a running example (Figure 1), one of the benchmark suite of the VeriMAP system De Angelis et al. (2014). The result of applying unfolding is shown in Figure 2 (omitting the definitions of the unfolded predicates `new4, new5` and `new6`, which are no longer reachable from `false`). The unfolding strategy we adopt is the following: the predicate dependency graph of a program consists of the set of edges (p, q) such that there is clause where p is the predicate of the head and q is a predicate occurring in the body. We perform a depth-first search of the predicate dependency graph, starting from `false`, and identify the backward edges, namely those edges (p, q) where q is an ancestor of p in the depth-first search. We then unfold every body call whose predicate is not at the end of a backward edge. In Figure 1, we thus unfold calls to `new4, new5` and `new6`.

The query-answer transformation is applied to the program in Figure 2, with respect to the goal `false` resulting in the program shown in Figure 3. The model of the predicate `new3_query` corresponds to those calls to `new3` that are reachable from the call in the integrity constraint. Explicit representation of the query predicates permits more effective propagation of constraints from the integrity clauses during model approximation.

The splitting transformation is now applied to the program in Figure 3. We do not

```
false :- A=0, B=50, new3(A,B).
new3(A,B) :- A=<99, C = 1+A, A=<49, new3(C,B).
new3(A,B) :- A=<99, C = 1+A, D = 1+B, A>=50, new3(C,D).
new3(A,B) :- A>=100, B>=101.
new3(A,B) :- A>=100, B=<100, B=<99.
```

Fig. 2. Result of unfolding `MAP-disj.c.map.pl`

```
false_ans :- false_query, A=0, B=50, new3_ans(A,B).
new3_ans(A,B) :- new3_query(A,B), A=<99, C = 1+A, A=<49, new3_ans(C,B).
new3_ans(A,B) :- new3_query(A,B),A=<99,C is 1+A,D is 1+B, A>=50, new3_ans(C,D).
new3_ans(A,B) :- new3_query(A,B), A>=100, B>=101.
new3_ans(A,B) :- new3_query(A,B), A>=100, B=<100, B=<99.
new3_query(A,B) :- false_query, A=0, B=50.
new3_query(A,B) :- new3_query(C,B), C=<99, A = 1+C, C=<49.
new3_query(A,B) :- new3_query(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
false_query.
```

Fig. 3. The query-answer transformed program for program of Figure 2

show the complete program, which contains 30 clauses. Figure 4 shows the split definition of `new3_query`, which is split since the last two clauses for `new3_query` in Figure 3 have mutually disjoint constraints, when projected onto the head variables.

A convex polyhedron approximation is then computed for the split program, after computing threshold constraints for the predicates. The resulting approximate model is shown in Figure 5 as a set of constrained facts. Since the model does not contain any constrained fact for `false_ans` we conclude that `false_ans` is not a consequence of the split program. Hence, applying the various correctness results for the unfolding, query-answer and splitting transformations, `false` is not a consequence of the original program.

Discussion of the example. Application of the convex polyhedron tool to the original, or the intermediate programs, does not solve the problem; all the transformations are needed in this case, apart from redundant argument filtering, which only affects efficiency. The ordering of the tool-chain can be varied somewhat, for instance switching query-answer transformation with splitting or unfolding. In our experiments we found the ordering in Figure 6 to be the most effective.

The model of the query-answer program is finite for this example. However, the problem is essentially the same if the constants are scaled; for instance we could replace 50 by 5000, 49 by 4999, 100 by 10000 and 101 by 10001, and the problem is essentially unchanged. We noted that some CHC verification tools applied to this example solve the problem, but essentially by enumeration of the finite set of values encountered in the search. Such

```
new3_query___1(A,B) :- false_query___1, A=0, B=50.
new3_query___1(A,B) :- new3_query____1(C,B), C=<99, A = 1+C, C=<49.
new3_query___1(A,B) :- new3_query____2(C,B), C=<99, A = 1+C, C=<49.
new3_query___2(A,B) :- new3_query____1(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
new3_query___2(A,B) :- new3_query____2(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
```

Fig. 4. Part of the split program for the program in Figure 3

```
false_query__1 :- []
new3_query__1(A,B) :- [1*A>=0,-1*A>= -50,1*B=50]
new3_query__2(A,B) :- [1*A>=51,-1*A>= -100,1*A+ -1*B=0]
```

Fig. 5. The convex polyhedral approximate model for the split program

a solution does not scale well. On the other hand the polyhedral abstraction shown above is not an enumeration; an essentially similar polyhedron abstraction is generated for the scaled version of the example, in the same time. The VeriMAP tool (De Angelis et al. 2014) also handles the original and scaled versions of the example in the same time.

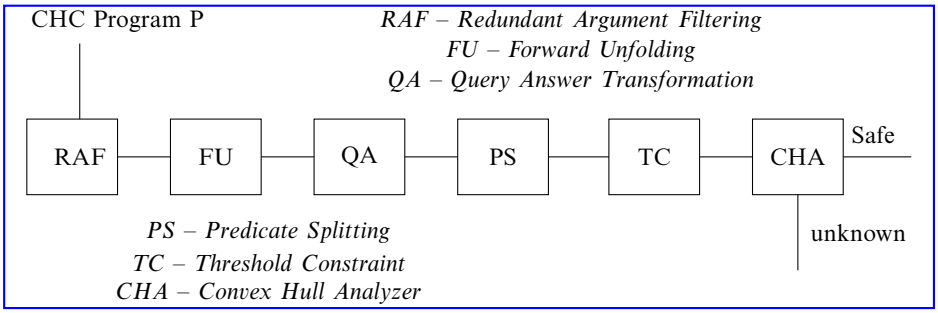


Fig. 6. *The basic tool chain for CHC verification.*

5 Combining off-the-shelf tools: Experiments

The motivation for our tool-chain, summarised in Figure 6, comes from our example program, which is a simple yet challenging program. We applied the tool-chain to a number of benchmarks from the literature, taken mainly from the repository of Horn clause benchmarks in SMT-LIB2 (https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/) and other sources including (Gange et al. 2013) and some of the VeriMap benchmarks (De Angelis et al. 2014). We selected these examples because many of them are considered challenging because they cannot be solved by one or more of the state-of-the-art-verification tools discussed below. Programs taken from the SMT-LIB2 repository are first translated to CHC form. The results are summarised in Table 1.

In Table 1, columns Program and Result respectively represent the benchmark program and the results of verification using our tool combination. Problems marked with (*) could not be handled by our tool-chain since they contain numbers which do not fit in 32 bits, the limit of our Ciao Prolog implementation. whereas problems marked with (**) are solvable by simple ad hoc modification of the tool-chain, which we are currently investigating (see Section 7). Problems such as systemc-token-ring.01-safeil.c contain complicated loop structure with large strongly connected components in the predicate dependency graph and our convex polyhedron analysis tool is unable to derive the required invariant. However overall results show that our simple tool-chain begins to compete with advanced tools like HSF (Grebenshchikov et al. 2012), VeriMAP (De Angelis et al. 2014), TRACER (Jaffar et al. 2012), *etc.* We do not report timings, though all these results are obtained in a

matter of seconds, since our tool-chain is not at all optimised, relying on file input-output and the individual components are often prototypes.

Table 1. *Experiments results on CHC benchmark program*

SN	Program	Result	SN	Program	Result
1	MAP-disj.c.map.pl	verified	17	MAP-forward.c.map.pl	verified
2	MAP-disj.c.map-scaled.pl	verified	18	tridag.smt2	verified
3	t1.pl	verified	19	qrdcmp.smt2	verified
4	t1-a.pl	verified	20	choldc.smt2	verified
5	t2.pl	verified	21	lop.smt2	verified
6	t3.pl	verified	22	pzextr.smt2	verified
7	t4.pl	verified	23	qrsolv.smt2	verified
8	t5.pl	verified	24	INVGEN-apache-escape-absolute	verified
9	pldi12.pl	verified	25	TRACER-testabs15	verified
10	INVGEN-id-build	verified	26**	amebsa.smt2	verified
11	INVGEN-nested5	verified	27**	DAGGER-barbr.map.c	verified
12	INVGEN-nested6	verified	28*	sshsimpl-s3-srvr-1a-safeil.c	NOT
13	INVGEN-nested8	verified	29	sshsimpl-s3-srvr-1b-safeil.c	NOT
14	INVGEN-svd-some-loop	verified	30*	bandec.smt2	NOT
15	INVGEN-svd1	verified	31	systemc-token-ring.01-safeil.c	NOT
16	INVGEN-svd4	verified	32*	crank.smt2	NOT

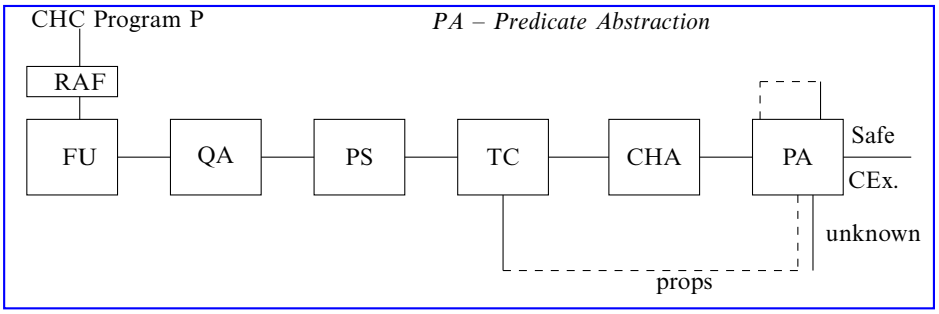


Fig. 7. *Future extension of our tool-chain.*

6 Discussion and Related Work

The most similar work to ours is by De Angelis et al. (2013) which is also based on CLP program transformation and specialisation. They construct a sequence of transformations of P, say, $P, P_1, P_2, \ldots, P_k$; if P_k contains no clause with head false then the verification problem is solved. A proof of unsafety is obtained if P_k contains a clause false $\leftarrow$. Both our approach and theirs repeatedly apply specialisations preserving the property to be proved. However the difference is that their specialisation techniques are based on unfold-fold transformations, with a sophisticated control procedure controlling unfolding

and generalisation. Our specialisations are restricted to redundant argument filtering and the query-answer transformation, which specialises predicate answers with respect to a goal. Their test for success or failure is a simple syntactic check, whereas ours is based on an abstract interpretation to derive an over-approximation. Informally one can say that the hard work in their approach is performed by the specialisation procedure, whereas the hard work in our approach is done by the abstract interpretation. We believe that our tool-chain-based approach gives more insight into the role of each transformation.

Work by Gange et al. (2013) is a top-town evaluation of CLP programs which records certain derivations and learns only from failed derivations. This helps to prune further derivations and helps to achieve termination in the presence of infinite executions. Duality (http://research.microsoft.com/en-us/projects/duality/) and HSF(C) (Grebenshchikov et al. 2012) are examples of the CEGAR approach (Counter-Example-Guided Abstraction Refinement). This approach can be viewed as property-based abstract interpretation based on a set of properties that is refined on each iteration. The refinement of the properties is the key problem in CEGAR; an abstract proof of unsafety is used to generate properties (often using interpolation) that prevent that proof from arising again. Thus, abstract counter-examples are successively eliminated. The relatively good performance of our tool-chain, without any refinement step at all, suggests that finding the right invariants is aided by a tool such as the convex polyhedron solver and the pre-processing steps we applied. In Figure 7 we sketch possible extensions of our basic tool-chain, incorporating a refinement loop and property-based abstraction.

It should be noted that the query-answer transformation, predicate splitting and unfolding may all cause an blow-up in the program size. The convex polyhedron analysis becomes more effective as a result, but for scalability we need more sophisticated heuristics controlling these transformations, especially unfolding and splitting, as well as lazy or implicit generation of transformed programs, using techniques such as a fixpoint engine that simulates query-answer programs (Codish 1999).

7 Concluding remarks and future work

We have shown that a combination of off-the-shelf tools from CLP transformation and analysis, combined in a sensible way, is surprisingly effective in CHC verification. The component-based approach allowed us to experiment with the tool-chain until we found an effective combination. This experimentation is continuing and we are confident of making improvements by incorporating other standard techniques and by finding better heuristics for applying the tools. Further we would like to investigate the choice of chain suitable for each example since more complicated problems can be handled just by altering the chain. We also suspect from initial experiments that an advanced partial evaluator such as ECCE (Leuschel et al. 2006) will play a useful role. Our results give insights for further development of automatic CHC verification tools. We would like to combine our program transformation techniques with abstraction refinement techniques and experiment with the combination.

References

Benoy, F. and King, A. 1996. Inferring argument size relationships with CLP(R). In *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, J. P. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer, 204–223.

Codish, M. 1999. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program. 38,* 3, 355–370.

Codish, M. and Demoen, B. 1993. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*, D. Miller, Ed. MIT Press.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 238–252.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2013. Verifying programs via iterated specialization. In *PEPM*, E. Albert and S.-C. Mu, Eds. ACM, 43–52.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014. Verimap: A tool for verifying programs through transformations. In *TACAS*, E. Ábrahám and K. Havelund, Eds. Lecture Notes in Computer Science, vol. 8413. Springer, 568–574.

Debray, S. and Ramakrishnan, R. 1994. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming 18*, 149–176.

Gallagher, J. P. 1993. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, Copenhagen, 88–98.

Gallagher, J. P. and de Waal, D. 1993. Deletion of redundant unary type predicates from logic programs. In *Logic Program Synthesis and Transformation*, K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, 151–167.

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. 2013. Failure tabled constraint logic programming by interpolation. *TPLP 13,* 4-5, 593–607.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. LNCS, vol. 7214. Springer, 549–551.

Halbwachs, N., Proy, Y. E., and Raymound, P. 1994. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis*. Lecture Notes in Computer Science, vol. 864. Springer, 223–237.

Jaffar, J. and Maher, M. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *CAV*, P. Madhusudan and S. A. Seshia, Eds. Lecture Notes in Computer Science, vol. 7358. Springer, 758–766.

Lakhdar-Chaouch, L., Jeannet, B., and Girault, A. 2011. Widening with thresholds for programs with complex control graphs. In *ATVA 2011*, T. Bultan and P.-A. Hsiung, Eds. Lecture Notes in Computer Science, vol. 6996. Springer, 492–502.

Leuschel, M. 1999. Advanced logic program specialisation. In *Partial Evaluation - Practice and Theory*, J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1706. Springer, 271–292.

Leuschel, M., Elphick, D., Varea, M., Craig, S.-J., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *PEPM 2006*, J. Hatcliff and F. Tip, Eds. ACM, 88–94.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, J. P. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer, 83–103.

Pettorossi, A. and Proietti, M. 1999. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program. 41,* 2-3, 197–230.

Serebrenik, A. and De Schreye, D. 2001. Inference of termination conditions for numerical loops in Prolog. In *LPAR 2001*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer, 654–668.

Multi-Criteria Optimal Planning for Energy Policies in CLP

MARCO GAVANELLI
EnDiF - Università di Ferrara, Italy
(*e-mail:* marco.gavanelli@unife.it)

STEFANO BRAGAGLIA
Department of Computer Science, University of Bristol, UK
(*e-mail:* stefano.bragaglia@bristol.ac.uk)

MICHELA MILANO, FEDERICO CHESANI
DISI - Università di Bologna, Italy
(*e-mail:* {michela.milano | federico.chesani}@unibo.it)

ELISA MARENGO
Faculty of Computer Science - Free University of Bozen-Bolzano
(*e-mail:* elisa.marengo@unibz.it)

PAOLO CAGNOLI
ARPA Emilia-Romagna, Italy
(*e-mail:* PCagnoli@arpa.emr.it)

submitted 14 February 2014; revised 18 April 2014; accepted 15 May 2014

Abstract

In the policy making process a number of disparate and diverse issues such as economic development, environmental aspects, as well as the social acceptance of the policy, need to be considered. A single person might not have all the required expertises, and decision support systems featuring optimization components can help to assess policies.

Leveraging on previous work on Strategic Environmental Assessment, we developed a fully-fledged system that is able to provide optimal plans with respect to a given objective, to perform multi-objective optimization and provide sets of Pareto optimal plans, and to visually compare them. Each plan is environmentally assessed and its footprint is evaluated. The heart of the system is an application developed in a popular Constraint Logic Programming system on the Reals sort. It has been equipped with a web service module that can be queried through standard interfaces, and an intuitive graphic user interface.

KEYWORDS: CLP applications, Strategic Environmental Assessment, Regional Energy Planning

1 Introduction

Policy making, in the current connected world, has to consider such a number of issues that a single person cannot possibly consider without introducing vast approximations. For example European regions should provide Regional Energy Plans to define strategic objectives and political actions for the energy sector, considering:

- the current energy balance in the region (produced/consumed energy, imported/exported, electrical/thermal, etc.)
- forecasts for the following years, about energy request or production costs;
- existing and new directives, e.g. the EU 20-20-20 initiative that poses three challenging targets for 2020: 20% improvement of energy efficiency, 20% of the energy produced from renewable sources, and 20% reduction of greenhouse gas emissions.

The policy contains strategic objectives on the energy share and energy efficiency, measures and activities to cope with the increased energy needs, new regulations, etc. Regional plans in particular are typically very high-level: they include activities such as building new power plants for some total output power, the share of each fuel type (nuclear, fossil fuels, biomasses, etc.) and the type of produced energy (electric or thermal); but they lack information about, for example, the actual placement of the plants in the region, since more detailed plans will be done at lower scale, like the province or municipality levels. By EU directives, regional policies on the energy sector should also include an environmental assessment of the plan. Being the plan so high-level, usually the assessment is done only in a qualitative way.

In a previous work (Gavanelli et al. 2010), we proposed and compared two alternative logic programming formulations for the strategic environmental assessment of regional plans; one was based on probabilistic logic programming, the other on **CLP!** (CLP!) (Jaffar and Maher 1994). We also developed four fuzzy-logic formulations of the assessment problem (Gavanelli et al. 2011). All these programs consider a regional plan, given in input, and provide its environmental assessment. In a following work (Gavanelli et al. 2013), the CLP! program was extended to generate plans together with their assessment, and it was used during the definition of the Regional Energy Plan 2011-2013 of the Emilia-Romagna region (Pilolli et al. 2011).

In this work, we show how the first prototype of the planner was extended to a fully-fledged application. In particular, the current version of the software supports:

- plans that consider decommissioning obsolete power plants;
- computation of emissions of the power plants for various types of pollutants, in a quantitative way;
- quantitative assessment of the effect of the plan on human health, global warming, and acidification potential;
- multi-criteria optimization considering a variety of objective functions based on qualitative and quantitative information;
- computation of the Pareto front, for two or more objective functions;
- a web service, providing access through a **GUI!** (GUI!) and APIs.

This work is one of the components of the EU ePolicy project[1]. The final application will include also an opinion mining component, to assess the acceptance of the policies from the public considering information coming from blogs and social networks; a social simulator component, that will simulate how the population will react to the policies adopted by the Region; a mechanism design component, that will include information

[1] http://www.epolicy-project.eu

from game theory to provide the best allocation schemes of regional subsidies to the stakeholders; and an integrated visualization component.

The rest of the paper is organized as follows. We first introduce the planning and environmental assessment as they are currently done by experts in the Emilia-Romagna region of Italy, and recap the basic **CLP!** program of the first prototype (Section 2). In Section 3, we extend it with new features. We show the design and features of the web service and **GUI!** in Section 4. Finally, we conclude in Section 5.

2 Problem considered and CLP solution

The strategic environmental assessment, in the Emilia-Romagna region of Italy, is currently performed by considering two matrices, called *coaxial matrices* (Cagnoli 2010). They are a development of the network method (Sorensen and Moss 1973), and they contain qualitative relations.

The first matrix, $\mathscr{M}$, considers the *activities* that can be undertaken in a plan, and links them with the *environmental pressures*. Pressures can be positive or negative, and they account for the impact on the environment of human activities. Each element m_j^i of the matrix $\mathscr{M}$ can take values $\{high, medium, low, null\}$, and defines a qualitative dependency between the activity i and the negative or positive pressure j.

The second matrix, $\mathscr{N}$, relates the pressures with the *environmental receptors*, that register the effect of the pressures on the environment. For example, the activity *"coal-fueled power plant"* generates the pressure *"emission of pollutants in the atmosphere"*; on its turn, this influences the receptor *"air quality"* (as well as other receptors, like e.g. *"human wellbeing"*). Each element n_j^i of the matrix can take the qualitative values: *high*, *medium*, *low* or *null*, and defines the dependency between pressure i and receptor j.

Currently, the matrices relate 115 activities with 29 negative and 19 positive pressures, and 23 receptors. They can be used to assess a variety of regional plans, including Agriculture, Forest, Fishing, Energy, Industrial, Transport, Waste, Water, Telecommunications, Tourism, Urban plans. The environmental assessment is usually done using a spreadsheet and deleting (by hand) those activities that do not belong to the given type of plan; then pressures and receptors that are not influenced by remaining activities are removed accordingly. The "reduced" matrices are evaluated by environmental experts, that state which parts are most important, mainly considering clusters of *High* values.

Clearly, this process is very slow, experts might overlook important combinations of *medium* or *low* values, and, most importantly, it can be done only after the plan has been provided by the policy maker. At this stage, usually only minor modifications can be back-propagated to the plan, and comparing a plan's effects with alternative plans is not possible without starting another planning phase.

2.1 A CLP solution

To overcome the limitations and improve on current practices, we devised a **DSS!** (**DSS!**) able to provide optimal plans and environmental assessment (Gavanelli et al. 2013): the planning problem was modelled as a linear program in **CLP!** on the Reals sort (CLP($\mathscr{R}$)).

Given a number N_a of activities, we consider a vector $\mathbf{A} = (a_1, \ldots, a_{N_a})$ in which we associate to each activity a variable a_i that defines its magnitude. The domain of a_i

depends on the availability of the resource on the given Region; for example some regions are very windy, while others can exploit better biomasses or solar energy.

We distinguish primary from secondary activities: primary activities are directly related to the given type of plan, while secondary ones are those supporting the primary activities by providing the needed infrastructures. E.g. in an energy plan, primary activities are those producing energy (e.g., power plants), and they may require other activities (e.g., power lines, waste stocking, streets, etc.) that have an environmental impact too. Let A^P be the set of indexes of primary activities and A^S that of secondary activities. The dependencies between primary and secondary activities are considered by the constraint:

$$\forall j \in A^S \quad a_j = \sum_{i \in A^P} d_{ij} a_i \tag{1}$$

Each activity a_i has a cost c_i; given a budget B_{Plan} available for a given plan, we have:

$$\sum_{i=1}^{N_a} a_i \; c_i \leqslant B_{Plan} \tag{2}$$

Given an expected outcome out_{Plan} of the plan, we also have:

$$\sum_{i=1}^{N_a} a_i \; out_i \geqslant out_{Plan}. \tag{3}$$

E.g. an energy plan outcome can be to increase available energy, so out_{Plan} could be the added availability of electrical power (in kilo-TOE, Tonnes of Oil Equivalent). Other outcomes can be considered, e.g. increasing only renewable energies.

Concerning the impacts of the regional plan, an environmental expert suggested to convert the qualitative values in the matrices into coefficients ranging from 0 to 1; we sum up the contributions of all the activities to estimate the impact on each pressure:

$$\forall j \in \{1, \ldots, N_p\} \quad p_j = \sum_{i=1}^{N_a} m_j^i \; a_i. \tag{4}$$

Similarly, given the pressures $\mathbf{P} = (p_1, \ldots, p_{N_p})$, the influence on the environmental receptor r_i is estimated through the matrix $\mathcal{N}$, relating pressures with receptors:

$$\forall j \in \{1, \ldots, N_r\} \quad r_j = \sum_{i=1}^{N_p} n_j^i p_i. \tag{5}$$

Possible objective functions include maximizing/minimizing the produced energy, the cost, or one of the receptors (e.g., *"air quality"*), or a linear combination of the above.

3 Extended solution

The CLP($\mathcal{R}$) program described in Section 2.1 was used in the development of the 2011-13 Regional Energy plan of the Emilia-Romagna region of Italy: the plan objective was to increase the share of renewable energy in the energy mix, and to fulfil the 20-20-20 directive. For the next years experts foresee the decommissioning of carbon-based power plants, with a residual utilisation when renewable energy is unavailable or in peak hours. Region experts asked us to extend the **DSS!** to consider also the closing of power plants.

Power plants decomissioning implies that some activities have a negative magnitude: e.g., the magnitude, in MW, of oil-based power plants could be reduced w.r.t. the previous years. However, negative activities introduce non-linearities. For example, if building a new plant i has a cost c_i in $€/MW$, decommisioning it will not give a *profit* of c_i $€/MW$.

Our implementation is based on the ECL^{i}PS^e CLP language (Apt and Wallace 2007; Schimpf and Shen 2012), using the `eplex` library (Shen and Schimpf 2005). The `eplex` library uses very fast solvers using linear programming or mixed-integer linear programming algorithms, allowing the use of linear constraints on variables ranging either on continuous or on integer domains. It is well known that linear programming is polynomially solvable, while (mixed) integer linear programming is NP-hard; thus the efficiency of the solution depends on whether there are integer variables or not. To address the non-linearity, we introduced, for each activity a_i that has negative values in its domain, a real variable Pos_i defined as:

$$Pos_i = \begin{cases} a_i & \text{if} \quad a_i \geqslant 0 \\ 0 & \text{if} \quad a_i < 0 \end{cases}$$

The cost constraint (2) is now rewritten as:

$$\sum_{i=1}^{N_a} Pos_i \; c_i \leqslant B_{Plan}. \tag{6}$$

Similarly, secondary activities should not be decommisisoned together with primary ones; so we impose their relationship only with the positive part of primary activities.

Concerning the environmental assessment, we may notice that any new activity has different types of impacts, some related to its initial implementation (e.g., land use for building a coal power plant), and others due to the activity functioning (e.g., air pollution for burning fuel). Equation (4) correctly accounts for both when dealing with "positive" activities, while would be incorrect w.r.t. "negative" activities.

To cope with the activities decommissioning, the co-axial matrices have been extended with new activities (e.g., *"Reduced use of fossil fuelled power plants"*). All the pressures are now computed on positive activities only, and Equation (4) is substituted with

$$\forall j \in \{1, \ldots, N_p\} \quad p_j = \sum_{i=1}^{N_a} m_j^i \; Pos_i. \tag{7}$$

We considered the new activities as a new type of secondary activities: the *"Reduced use of fossil fuelled power plants"* is a secondary activity that becomes positive only when activities like *"Coal-based power plant"*, etc., has a negative value (i.e., in case of decommissions). Hence, we now have two matrices of dependencies between activities: a $N_a \times N_a$ square matrix $\mathscr{D}^+$ where each element d_{ij}^+ represents the magnitude of activity j per unit of activity i; and another $N_a \times N_a$ square matrix $\mathscr{D}^-$ where each element d_{ij}^- represents the magnitude of activity j per unit of reduction of activity i. Equation (1) is updated with

$$\forall j \in A^S \quad a_j = \textstyle\sum_{i \in A^P} K_{ij} \qquad K_{ij} = \begin{cases} d_{ij}^+ \cdot a_i & \text{if} \quad a_i \geqslant 0 \\ d_{ij}^- \cdot (-a_i) & \text{if} \quad a_i < 0 \end{cases} \tag{8}$$

3.1 Computing emissions

A further extension to the model presented in Section 2.1 has been about the evaluation of emissions in quantitative terms. To this end, we rely on the data provided by two databases: INEMAR (Caserini et al. 2002) and ISPRA (ISPRA): both databases provide the various types of pollutants [2] emitted per fuel unit (in GJ). While ISPRA provides the average emission for each plant type, INEMAR provides fine grained information, in which also the type of boiler and the size of the plant (in MW) are considered.

Let N_B the number of boiler types, and $\mathbf{B} = (b_1, \ldots, b_{N_B})$ a vector of constrained variables where b_i is the total output power of plants using boiler type i. Let $\mathcal{O}$ be the matrix that relates power plants and the different kinds of boiler: each element o^i_j of the matrix is set to if the boiler $b_j \in \mathbf{B}$ can be used for the power plant $a_i \in \mathbf{A}$, and zero otherwise. The output power of each plant type is the sum of the power of its boilers:

$$\forall i \in \{1, \ldots, N_a\} \quad a_i = \sum_{j \in N_B} o^i_j b_j \tag{9}$$

Let $\mathbf{E} = (e_1, \ldots, e_{N_e})$ be the vector of emissions and $\mathcal{T}$ the matrix relating them with the boilers. An element $t^i_j \in \mathcal{T}$ represents the grams of pollutant $e_i \in \mathbf{E}$ emitted when 1GJ of fuel is provided to the boiler $b_j \in \mathbf{B}$. To calculate the emissions, we have to compute the input energy for each boiler type j, provided the output power b_j:

$$\forall i \in \{1, \ldots, N_e\} \quad e_i = \sum_{j \in N_B} t^i_j \left(\frac{T^U}{\eta} b_j \right). \tag{10}$$

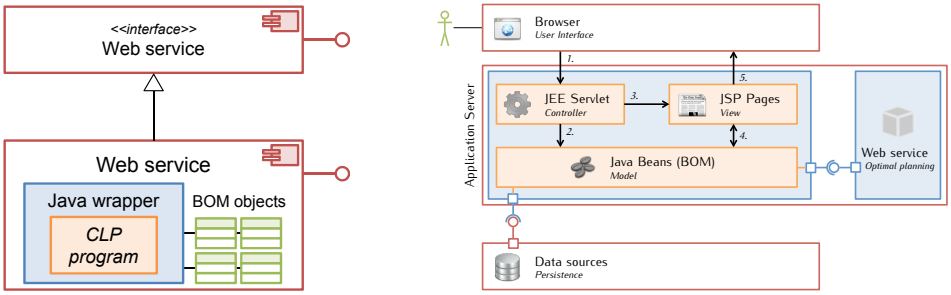
T^U is the average running time of a power plant per year (necessary to convert energy into power) and η is the average efficiency (output power/input power) of power plants, which is prescribed by law as 39% (Autorità per l'Energia Elettrica e il Gas 2008).

3.2 Indicators

Thanks to the extension presented in Section 3.1 it is possible to evaluate quantitatively the emissions of some gases, metals, etc. Such data can be exploited to consider plans aiming to minimize them; however, it is not clear how to compare the emissions. E.g., a policy maker could know that **NOx!** are toxic for humans, but how does that compare with heavy metal emissions?

The European Commission (2006) published a set of indicators quantifying the effect of various substances on *human toxicity*, *global warming* and *acidification*: e.g., the Annex 1 contains 100 chemicals with their human toxicity factor, defined as the toxicity of the substance compared to that of lead (Pb). By using the weights in the tables, one can provide, e.g., the total human toxicity (in kg of *equivalent emitted Pb*), the global warming effect (kg of *equiv.* CO_2) and the acidification of the plan (kg of *equiv.* SO_2). Moreover, a policy maker may want to optimize on these indicators (by directly minimizing them or any weighted sum).

[2] Considered types of pollutants include **SOx!** (**SOx!**), **NOx!** (**NOx!**), methane, CO, CO_2, N_2O, ammonia, **HCB!** (**HCB!**), various metals (Arsenic, Cadmium, Chromium, Copper, Mercury, Nickel, lead, Selenium, Zinc), particulate matter (PM10), Dioxins, and some families of compounds, like **PAH!** (**PAH!**), **PCB!** (**PCB!**), and **NMVOC!** (**NMVOC!**).



(a) The Web service. (b) The Web application.

Fig. 1: Software stack to deploy the **CLP!** program as a Web service and the typical **MVC!** pattern to exploit it as a Web application.

However, the tables provided by the EC do not always have the same granularity of the information available for emissions. For example, for each plant type we know the emissions of **NOx!**, while in the EU report there are the single toxicity values of NO and NO_2 (and they are quite different: respectively, 95 and 300 times that of Pb). Environmental experts suggested to provided as output, for each indicator, the best, worst, and average cases, considering respectively the highest toxicity in the compound class, the lowest and an average. If one of the indicators is in the objective function (e.g., one wants to find the plan with minimum human toxicity), we optimize the worst case.

3.3 Computing the Pareto front

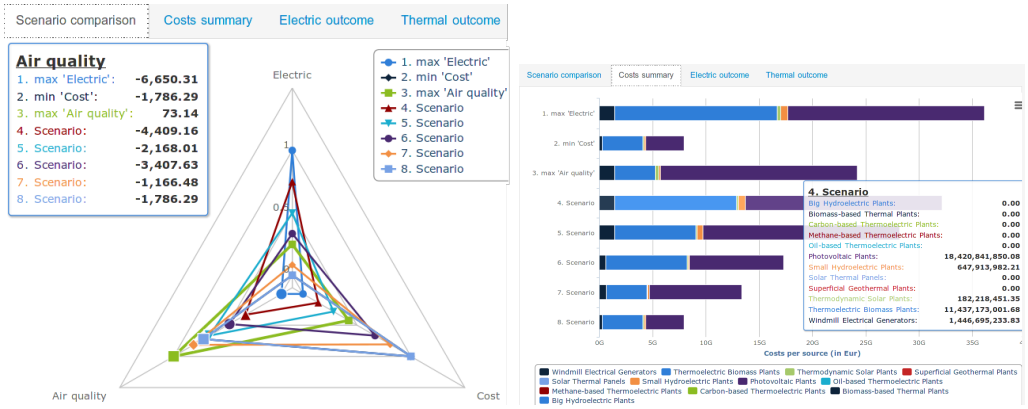
The approach presented in Section 2 allows to optimize w.r.t. a single function. However, in the case of regional planning it is very hard (if not impossible) to devise a unique function that includes all the objectives that are important for the user. Hence, we added a further extension towards multi-objective optimization.

In a multi-objective optimization problem, a solution is *Pareto optimal* if it is not possible to improve the result for one objective function, without worsening at least another objective function. More precisely, in a multi-objective problem with n functions to minimize, a solution μ^* is *Pareto-optimal* if there does not exist another solution μ such that $\mu_j \leqslant \mu_j^*$ for $1 \leqslant j \leqslant n$ and there exists at least one i, $1 \leqslant i \leqslant n$ such that $\mu_i < \mu_i^*$. The set of Pareto points is distributed on the so-called Pareto frontier.

We implemented the *normalized normal constraint method* (Messac et al. 2003), an algorithm that works with any type of constraints (linear and nonlinear) and variables (continuous and discrete), and that is able to find an *evenly distributed* set of Pareto solutions. In this way, the policy maker is provided with a set of solutions that are a good representation of the whole space of the Pareto frontier.

4 Graphical User Interface

Usually, policy makers are not IT-experts. To ease the access to the DSS, we deployed the **CLP!** planner as a stateless Web service and access it by means of a stateful Web application. The **CLP!** program is embedded inside a Java wrapper (Fig. 1a) that encodes the requests in



(a) Scenario comparison. (b) Costs summary.

Fig. 2: The views associated with the *General overview* entry for *Scenarios comparison*.

CLP! terms and decodes the results. This component provides a plethora of Java classes that represent the **BOM!** (**BOM!**) of this domain. Any query addressed to this component and all the returned results are expressed in terms of these objects. We adopted the Apache CXF framework to support the Web Service imlpementation. The Web application that stands as a **GUI!** for the Web service is a standard Java servlet (Fig. 1b) following the **MVC!** (**MVC!**) pattern and can be accessed at: `http://globalopt.epolicy-project.eu/Pareto/`.

After a welcome page that introduces the software, there are an input page, and a results page. As input the user can provide minimum and maximum bounds for each energy source, constraints, and objective functions for the Pareto optimization (together with a desired number of Pareto points). Constraints and objectives can include linear combinations of cost, produced power, receptors, emissions, or indicators.

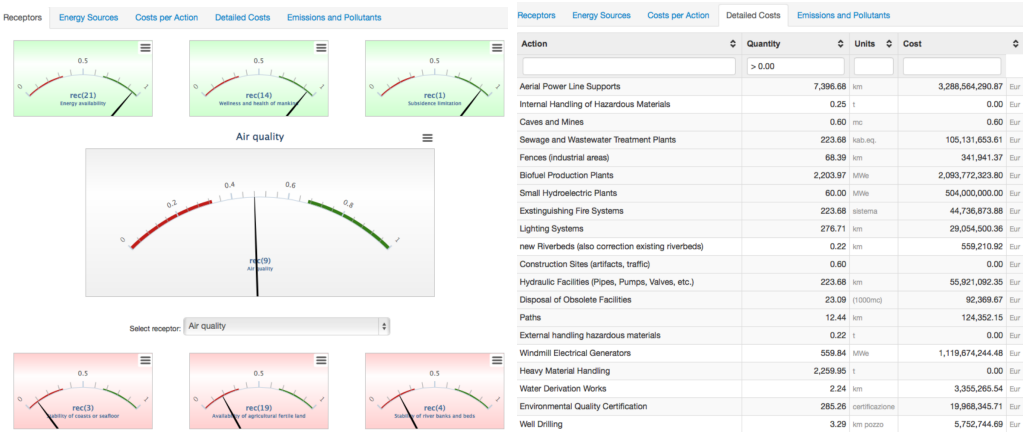
As a result, a set of graphs allow to inspect details on a specific plan (scenario), and/or to compare the computed plans. Scenarios are divided into *boundary scenarios*, that are those that optimize one of the objective functions, and *intermediate scenarios*, that try to balance the various objectives.

Scenarios comparison. Scenarios can be compared through a *spiderweb chart* (Fig. 2a) that has an axis for each objective function. Along each axis, the optimal values are far from the origin, and each scenario is represented by a polygon. Roughly speaking, a bigger polygon implies a better scenario (note that these solutions are Pareto optimal, so one polygon cannot be completely included into another polygon).

Scenarios can also be compared through *stacked bar chart*, showing, for each scenario, the distribution of costs per energy source (Fig. 2b), or the amount of electric/thermal energy per source. Moreover, a further view provides scenarios comparison in terms of pollutants (*Heavy metals*, *Greenhouse gases,* and *Other pollutants*), by means of basic column charts.

Scenarios Details. For each scenario, the following views are available:

- *Receptors*. This composite view uses 7 *VU-meter charts* (Fig. 3a). The top part shows the 3 receptors with the best normalised value, while the bottom one the 3 with the



(a) Receptors. (b) A tabular view.

Fig. 3: Details views for each scenario.

worst normalised value. The main chart allows the user to select any receptor and appraise its normalised value. This specific view ensures fast access to the best and worst receptors for the specific scenario.

- *Other views.* There are four interactive *tabular* views (Fig. 3b) showing respectively, for the chosen scenario, the amount of produced energy per source, the total cost for each energy source to be spent in primary and secondary activities, the detailed costs for each activity, and the list of emissions.

5 Conclusions and Future Work

We presented a decision support system with optimization based on **CLP!** for the regional planning, with particular emphasis on the environmental aspects. The program was practically used to produce the energy plan 2011-2013 of the Emilia-Romagna region in Italy (Pilolli et al. 2011), and it is foreseen to use it also for the forthcoming plans. The **CLP!** program is included into a standard web service, and it has been equipped with an intuitive GUI. The **CLP!** program will be the heart of the platform of the EU FP7 ePolicy project, that will also include components like a social simulator, an opinion miner, and a mechanism designer, all governed by the described **CLP!** program. Preliminary work has been done on its integration with the mechanism designer (Milano et al. 2012), and a social simulator (Borghesi et al. 2013).

Future work will be on extending the model at a more detailed level, e.g. taking into account decommissioning fixed costs.

Acknowledgements. This work was partially supported by EU project ePolicy, FP7-ICT-2011-7, grant agreement 288147.

References

Apt, K. R. and Wallace, M. 2007. *Constraint logic programming using Eclipse.* Cambridge University Press.

Autorità per l'Energia Elettrica e il Gas. 2008. Aggiornamento del fattore di conversione dei kWh in tonnellate equivalenti di petrolio connesso al meccanismo dei titoli di efficienza energetica. Gazzetta Ufficiale n. 100 del 29.4.08 - SO n.107.

Borghesi, A., Milano, M., Gavanelli, M., and Woods, T. 2013. Simulation of incentive mechanisms for renewable energy policies. In *Proceedings of the 27th European Conference on Modeling and Simulation*, W. Rekdalsbakken, R. T. Bye, and H. Zhang, Eds. European Council for Modeling and Simulation, 32–38.

Cagnoli, P. 2010. *VAS valutazione ambientale strategica – Fondamenti teorici e tecniche operative*, Terza edizione ed. Dario Flaccovio, Palermo, Italy.

Caserini, S., Fraccaroli, A., Monguzzi, A. M., Moretti, M., Giudici, A., and Volpi, G. 2002. The INEMAR database: a tool for regional atmospheric emission inventory. In *iEMSs 2002 International Congress: "Integrated Assessment and Decision Support". Proceedings of the 1st biennial meeting of the International Environmental Modelling and Software Society*, A. Rizzoli and A. Jakeman, Eds. Lugano, Switzerland.

European Commission. 2006. Integrated pollution prevention and control reference document on economics and cross-media effects. `http://eippcb.jrc.ec.europa.eu/reference/`.

Gavanelli, M., Riguzzi, F., Milano, M., and Cagnoli, P. 2010. Logic-Based Decision Support for Strategic Environmental Assessment. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue 10,* 4-6 (July), 643–658.

Gavanelli, M., Riguzzi, F., Milano, M., and Cagnoli, P. 2013. Constraint and optimization techniques for supporting policy making. In *Computational Intelligent Data Analysis for Sustainable Development*, T. Yu, N. Chawla, and S. Simoff, Eds. Data Mining and Knowledge Discovery Series. Chapman & Hall/CRC, Boca Raton, FL, USA, Chapter 12, 361–381.

Gavanelli, M., Riguzzi, F., Milano, M., Sottara, D., Cangini, A., and Cagnoli, P. 2011. An application of fuzzy logic to strategic environmental assessment. In *Artificial Intelligence Around Man and Beyond - XIIth AIxIA Intl. Conf.*, R. Pirrone and F. Sorbello, Eds. LNCS, vol. 6934. Springer, Berlin/Heidelberg, 324–335.

ISPRA. Inventario nazionale delle emissioni in atmosfera. Available at `http://www.sinanet.isprambiente.it/it/sia-ispra/serie-storiche-emissioni/fattori-di-emissione-per-le-sorgenti-di-combustione-stazionarie-in-italia`.

Jaffar, J. and Maher, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Messac, A., Ismail-Yahaya, A., and Mattson, C. A. 2003. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization 25,* 2, 86–98.

Milano, M., Gavanelli, M., O'Sullivan, B., and Holland, A. 2012. What-if analysis through simulation-optimization hybrids. In *Proc. of European Conference on Modelling and Simulation (ECMS)*. European Council for Modelling and Simulation, Dudweiler, Germany.

Pilolli, D., Raimondi, A., Scapinelli, D., Calò, C., and Cancila, E. 2011. *Piano Energetico Regionale, secondo piano attuativo 2011-2013.* Regione Emilia-Romagna.

Schimpf, J. and Shen, K. 2012. ECL^{i}PS^{e} - from LP to CLP. *Theory and Practice of Logic Programming 12,* 1-2, 127–156.

Shen, K. and Schimpf, J. 2005. Eplex: Harnessing mathematical programming solvers for constraint logic programming. In *Principles and Practice of Constraint Programming - CP 2005*, P. van Beek, Ed. LNCS, vol. 3709. Springer-Verlag, Berlin/Heidelberg, 622–636.

Sorensen, J. C. and Moss, M. L. 1973. Procedures and programs to assist in the impact statement process. Tech. rep., Univ. of California, Berkely.

Clingo = ASP + Control: Preliminary Report

Martin Gebser[1,2], Roland Kaminski[2], Benjamin Kaufmann[2], and Torsten Schaub[2]*
[1]*Aalto University, Finland* [2]*University of Potsdam, Germany*

submitted [n/a]; revised [n/a]; accepted [n/a]

Abstract

We present the new ASP system *clingo* 4. Unlike its predecessors, being mere monolithic combinations of the grounder *gringo* with the solver *clasp*, the new *clingo* 4 series offers high-level constructs for realizing complex reasoning processes. Among others, such processes feature advanced forms of search, as in optimization or theory solving, or even interact with an environment, as in robotics or query-answering. Common to them is that the problem specification evolves during the reasoning process, either because data or constraints are added, deleted, or replaced. In fact, *clingo* 4 carries out such complex reasoning within a single integrated ASP grounding and solving process. This avoids redundancies in relaunching grounder and solver programs and benefits from the solver's learning capacities. *clingo* 4 accomplishes this by complementing ASP's declarative input language by control capacities expressed via the embedded scripting languages Lua and Python. On the declarative side, *clingo* 4 offers a new directive that allows for structuring logic programs into named and parameterizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages. By strictly separating logic and control programs, *clingo* 4 also abolishes the need for dedicated systems for incremental and reactive reasoning, like *iclingo* and *oclingo*, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

1 Introduction

Standard Answer Set Programming (ASP; (Baral 2003)) follows a one-shot process in computing stable models of logic programs. This view is best reflected by the input/output behavior of monolithic ASP systems like *dlv* (Leone et al. 2006) and *clingo* (Gebser et al. 2011b). Internally, however, both follow a fixed two-step process. First, a grounder generates a (finite) propositional representation of the input program. Then, a solver computes the stable models of the propositional program. This rigid process stays unchanged when grounding and solving with separate systems. In fact, up to now, *clingo* provided a mere combination of the grounder *gringo* and the solver *clasp*. Although more elaborate reasoning processes are performed by the extended systems *iclingo* (Gebser et al. 2008) and *oclingo* (Gebser et al. 2011a) for incremental and reactive reasoning, respectively, they also follow a pre-defined control loop evading any user control. Beyond this, however, there is substantial need for specifying flexible reasoning processes, for instance, when it comes to interactions with an environment, as in assisted living, robotics, or with users, advanced search, as in multi-objective optimization, planning, theory solving, or heuristic search, or recurrent query answering, as in hardware analysis and testing or stream processing. Common to all these advanced forms of reasoning is that the problem specification evolves during the reasoning processes, either because data or constraints are added, deleted, or replaced.

* Affiliated with the Simon Fraser University, Burnaby, Canada, and Griffith University, Brisbane, Australia.

The new *clingo* 4 series offers novel high-level constructs for realizing such complex reasoning processes. This is achieved within a single integrated ASP grounding and solving process in order to avoid redundancies in relaunching grounder and solver programs and to benefit from the learning capacities of modern ASP solvers. To this end, *clingo* 4 complements ASP's declarative input language by control capacities expressed via the embedded scripting languages Lua and Python. On the declarative side, *clingo* 4 offers a new directive **`#program`** that allows for structuring logic programs into named and parameterizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages embedded via the **`#script`** directive. For exercising control, the latter benefit from a dedicated *clingo* library that does not only furnish grounding and solving instructions but moreover allows for continuously assembling the solver's program in combination with the directive **`#external`**. Hence, by strictly separating logic and control programs, *clingo* 4 abolishes the need for special-purpose systems for incremental and reactive reasoning, like *iclingo* and *oclingo*, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

2 Controlling grounding and solving in *clingo* 4

A key feature, distinguishing *clingo* 4 from its predecessors, is the possibility to structure (non-ground) input rules into subprograms. To this end, the directive **`#program`** comes with a name and an optional list of parameters. Once given in the *clingo* 4 input, it gathers all rules up to the next such directive (or the end of file) within a subprogram identified by the supplied name and parameter list. As an example, two subprograms `base` and `acid(k)` can be specified as follows:

```
1  a(1).
2  #program acid(k).
3  b(k).
4  #program base.
5  a(2).
```

Note that `base`, with an empty parameter list, is a dedicated subprogram that, in addition to rules in the scope of a directive like the one in Line 4, gathers all rules not preceded by a **`#program`** directive. Hence, in the above example, the `base` subprogram includes the facts `a(1)` and `a(2)`. Without further control instructions (see below), *clingo* 4 grounds and solves the `base` subprogram only, essentially yielding the standard behavior of ASP systems. The processing of other subprograms, such as `acid(k)` with the schematic fact `b(k)`, is subject to scripting control.

For a customized control over grounding and solving, a `main` routine (taking a control object representing the state of *clingo* 4 as argument) can be specified in either of the embedded scripting languages Lua and Python. For illustration, let us consider two Python `main` routines:

```
6  #script(python)
7  def main(prg):
8      prg.ground("base",[])
9      prg.solve()
10 #end.
```

```
6  #script(python)
7  def main(prg):
8      prg.ground("acid",[42])
9      prg.solve()
10 #end.
```

While the control program on the left matches the default behavior of *clingo* 4, the one on the right ignores all rules in the `base` program but rather, in Line 8, contains a `ground` instruction for `acid(k)`, where the parameter `k` is instantiated with the term `42`. Accordingly, the schematic fact `b(k)` is turned into `b(42)`, and the `solve` command in Line 9 yields a stable model consisting of `b(42)` only. Note that `ground` instructions apply to the subprograms given as

arguments, while `solve` triggers reasoning w.r.t. all accumulated ground rules. In fact, a `solve` command makes *clingo* 4 instantiate pending subprograms and then perform reasoning. That is, when Line 9 is replaced, e.g., by `print 'Hello!'`, *clingo* 4 merely writes out `Hello!` but does neither ground any subprogram nor compute stable models.

In order to accomplish more elaborate reasoning processes, like those of *iclingo* and *oclingo* or customized ones, it is indispensable to activate or deactivate ground rules on demand. For instance, former initial or goal state conditions need to be relaxed or completely replaced when modifying a planning problem, e.g., by extending its horizon. While the predecessors of *clingo* 4 relied on a **`#volatile`** directive to provide a rigid mechanism for the expiration of transient rules, *clingo* 4 captures the respective functionalities and customizations thereof in terms of the directive **`#external`**. This directive goes back to *lparse* (Syrjänen) and was also supported by the predecessors of *clingo* 4 to exempt (input) atoms from simplifications fixing them to false. The **`#external`** directive of *clingo* 4 provides a generalization that, in particular, allows for a flexible handling of yet undefined atoms.

For continuously assembling ground rules evolving at different stages of a reasoning process, **`#external`** directives declare atoms that may still be defined by rules added later on. As detailed in (Gebser et al. 2014), such atoms correspond to inputs in terms of module theory (Oikarinen and Janhunen 2006), which (unlike undefined output atoms) must not be simplified by fixing their truth value to false. In order to facilitate the declaration of input atoms, *clingo* 4 supports schematic **`#external`** directives that are instantiated along with the rules of their respective subprograms. To this end, a directive like

```
#external p(X,Y) : q(X,Z), r(Z,Y).
```

is treated similar to the (virtual) rule `p(X,Y) :- q(X,Z), r(Z,Y)` during grounding. However, the head atoms of resulting ground instances are merely collected as (external) inputs, whereas the ground rules as such are discarded. Given this, a subprogram from the *clingo* 4 input consists of all rules within the scope of **`#program`** directives with the same name and number of parameters, where `base` without parameters is used by default, along with virtual rules capturing **`#external`** directives in the same scope (see (Gebser et al. 2014) for details).

The instantiation of a subprogram R with a list $c_1,\ldots,c_k$ of parameters, such as `acid(k)` above, relies on a list $t_1,\ldots,t_k$ of terms to replace occurrences of $c_1,\ldots,c_k$ with, both in original rules and virtual rules capturing **`#external`** directives in R. The parameter replacement yields a subprogram $R(c_1/t_1,\ldots,c_k/t_k)$, which is instantiated relative to inputs. For instance, providing the term `42` for parameter `k` leads to `acid(k/42)` consisting of the fact `b(42)`. Control instructions guide the instantiation and assembly of subprograms, where `ground` instructions issued before the first or in-between two `solve` commands determine rules to instantiate and join with a module representing the previous state of *clingo* 4.

To sum up, schematic **`#external`** directives are embedded into the grounding process for a convenient declaration of input atoms from other subprogram instances. Given that they do not contribute ground rules, but merely qualify (undefined) atoms that should be exempted from simplifications, **`#external`** directives address the signature of subprograms' ground instances. Hence, it is advisable to condition them by domain predicates[1] (Syrjänen) only, as this precludes any interferences between signatures and grounder implementations. As long as input atoms remain undefined, their truth values can be freely picked and modified in-between `solve`

[1] Domain and built-in predicates have unique extensions that can be evaluated entirely by means of grounding.

```
1  #script(python)
2  from gringo import Fun, SolveResult

4  def init(val, default):
5      return val if val != None else default

7  def main(prg):
8      stop = str(init(prg.getConst("istop"), "SAT"))
9      step = int(init(prg.getConst("iinit"), 0))

11     prg.ground("base", [])
12     while True:
13         step += 1
14         prg.ground("cumulative", [step])
15         prg.assignExternal(Fun("query", [step]), True)
16         print 'STEP {0}'.format(step)
17         ret = prg.solve()
18         if (stop == "SAT"   and ret == SolveResult.SAT) or \
19            (stop == "UNSAT" and ret == SolveResult.UNSAT): break
20         prg.releaseExternal(Fun("query", [step]))
21 #end.
```

Listing 1: Python script implementing *iclingo* functionality in *clingo* (`iclingo.lp`)

commands via `assignExternal` instructions, which thus allow for configuring the inputs to modules representing *clingo* 4 states in order to select among their stable models. Unlike that, the predecessors *iclingo* and *oclingo* of *clingo* 4 always assigned input atoms to false, so that the addition of rules was necessary to accomplish switching truth values. However, for a well-defined semantics, *clingo* 4 like its predecessors builds on the assumption that the modules induced by subprograms' instantiations are compositional, which essentially requires definitions of (head) atoms and mutual positive dependencies to be local to evolving ground programs (cf. (Gebser et al. 2008)).

3 Using *clingo* 4 in practice

As mentioned above, *clingo* 4 fully supersedes its special-purpose predecessors *iclingo* and *oclingo*. To illustrate this, we give in Listing 1 a slightly simplified version of *iclingo*'s control loop in Python. The full control loop (included in the release) mainly adds handling of further *iclingo* options. Roughly speaking, *iclingo* offers a step-oriented, incremental approach to ASP that avoids redundancies by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem (as in iterative deepening search). To this end, a program is partitioned into a base part, describing static knowledge independent of the step parameter `t`, a cumulative part, capturing knowledge accumulating with increasing `t`, and a volatile part specific for each value of `t`. These parts were delineated in *iclingo* by the directives **`#base`**, **`#cumulative`** `t`, and **`#volatile`** `t`. In *clingo* 4, all three directives are captured by **`#program`** declarations along with **`#external`** for volatile rules.

We illustrate this by adapting the Towers of Hanoi encoding from (Gebser et al. 2012) in Figure 1. The problem instance in Figure 1(a) as well as Line 2 in 1(b) constitute static knowledge and thus belong to the base part. The transition function is described in the cumulative part in

```
1  #program base.
2  peg(a;b;c).
3  disk(1..4).
4  init_on(1..4,a).
5  goal_on(1..4,c).
```

(a) Towers of Hanoi instance

```
1  #program base.
2  on(D,P,0) :- init_on(D,P).

4  #program cumulative(t).
5  1 { move(D,P,t) : disk(D), peg(P) } 1.

7  move(D,t) :- move(D,P,t).
8  on(D,P,t) :- move(D,P,t).
9  on(D,P,t) :- on(D,P,t-1), not move(D,t).
10 blocked(D-1,P,t) :- on(D,P,t-1).
11 blocked(D-1,P,t) :- blocked(D,P,t), disk(D).

13 :- move(D,P,t), blocked(D-1,P,t).
14 :- move(D,t), on(D,P,t-1), blocked(D,P,t).
15 :- disk(D), not 1 { on(D,P,t) } 1.

17 #external query(t).
18 :- query(t), goal_on(D,P), not on(D,P,t).
```

(b) Towers of Hanoi incremental encoding

Fig. 1: Towers of Hanoi instance (`tohI.lp`) and incremental encoding (`tohE.lp`)

Line 5–15 of Figure 1(b). Finally, the query is expressed in Line 18; its volatility is realized by making the actual goal condition `goal_on(D,P), not on(D,P,t)` subject to the truth assignment to the external atom `query(t)`. Grounding and solving of the program in Figure 1(a) and 1(b) is controlled by the Python script in Listing 1. Line 4–9 fix the `stop` criterion and initial value of the `step` variable. Both can be supplied as constants `istop` and `iinit` when invoking *clingo* 4. Once the base part is grounded in Line 11, the script loops until the `stop` criterion is met in Line 18–19. In each iteration, the current value of `step` is used in Line 14 and 15 to instantiate the subprogram `cumulative(t)` and to set the respective external atom `query(t)` to true. If the `stop` condition is yet unfulfilled w.r.t. the result of solving the extended program, the current `query(t)` atom is permanently falsified (cf. Line 17–20), thus annulling the corresponding instances of the integrity constraint in Line 18 of Figure 1(b) before they are replaced in the next iteration.

Another innovative feature of *clingo* 4 is its incremental optimization. This allows for adapting objective functions along the evolution of a program at hand. A simple example is the search for shortest plans when increasing the horizon in non-consecutive steps. To see this, recall that literals in minimize statements (and analogously weak constraints) are supplied with a sequence of terms of the form $w@p,\vec{t}$, where w and p are integers providing a weight and a priority level and $\vec{t}$ is a sequence of terms (cf. (Calimeri et al. 2012)). As an example, consider the subprogram:

```
#program cumulativeObjective(t).
#minimize{ W@P,X,Y,t : move(X,Y,W,P,t) }.
% or :~ move(X,Y,W,P,t). [W@P,X,Y,t]
```

When grounding and solving `cumulativeObjective(t)` for successive values of `t`, the solver's objective function (per priority level `P`) is gradually extended with new atoms over `move/5`, and all previous ones are kept.

Moreover, for enabling the removal of literals from objective functions, we can use externals:

```
#program volatileObjective(t).
#external activateObjective(t).
#minimize{ W@P,X,Y,t : move(X,Y,W,P,t), activateObjective(t) }.
```

The subprogram `volatileObjective(t)` behaves like `cumulativeObjective(t)` as long as the external atom `activateObjective(t)` is true. Once it is set to false, all atoms over `move/5` with the corresponding term for `t` are dismissed from objective functions.

A reasoning process in *clingo* 4 is partitioned into a sequence of solver invocations. We have seen how easily the solver's logic program can be altered at each step. Sometimes it is useful to do this in view of a previously obtained stable model. For this purpose, the `solve` command can be equipped with an (optional) callback function `onModel`. For each stable model found during a call to `solve(onModel)`, an object encompassing the model is passed to `onModel`, whose implementation can then access and inspect the model. A typical example is the addition of constraints based on the last model that are then supplied to the solver before computing the next one. An application is theory solving by passing (parts of) the last model to a theory solver for theory-based consistency checking or for providing the value of an externally evaluated objective function. Moreover, *clingo* 4 also furnishes an asynchronous solving function `asolve` that launches an interruptable solving process in the background. This is particularly useful in reactive settings in order to stop solving upon the arrival of new external information.

Similarly, the configuration of *clasp* can be changed at each step via the function `setConf`, taking a string including command line options along with a flag indicating whether the previous configuration is updated or replaced as arguments. For instance, this allows for changing search parameters, reasoning modes, number of threads, etc. Changing search parameters is of interest when addressing computational tasks involving the generation of several models, like optimal planning, multi-criteria optimization, or heuristic search. Apart from analyzing the previous model via the `onModel` callback, one can also monitor the search progress by means of the function `getStats`, returning an object encapsulating up to 135 attributes of the previous search process. Furthermore, *clingo* 4 allows for customizing the heuristic values of variables, as described in (Gebser et al. 2013a). At a higher level, a user may simply want to explore the set of models, and decide to compute first one, then all, and then the intersection or union of all models. This can be interleaved with the addition of subprograms via the function `add`, which may in turn include **`#external`** directives to declare temporary hypotheses. The experienced reader may note that this can be done fully interactively by means of IPython. Practical examples for the mentioned features can be found in the releases at (potassco).

4 Related work

Although *clingo* 3 (Gebser et al. 2011c) already featured Lua as an embedded scripting language, its usage was limited to (deterministic) computations during grounding; neither were library functions furnished by *clingo* 3.

Of particular interest is *dlvhex* (Fink et al. 2013), an ASP system aiming at the integration of external computation sources. For this purpose, *dlvhex* relies on higher-order logic programs using external higher-order atoms for software interoperability. Such external atoms should not be confused with *clingo*'s **`#external`** directive because they are evaluated via procedural means during solving. Given this, *dlvhex* can be seen as an *ASP modulo Theory* solver, similar to SAT modulo Theory solvers (Nieuwenhuis et al. 2006). In fact, *dlvhex* uses *gringo* and *clasp* as backends and follows the design of the *ASP modulo CSP* solver *clingcon* (Ostrowski and Schaub

2012) in communicating with external "oracles" through *clasp*'s post propagation mechanism. In this way, theory solvers are tightly integrated into the ASP system and have access to the solver's partial assignments. Unlike this, the light-weighted theory solving approach offered by *clingo* 4 can only provide access to total (stable) assignments. It is thus interesting future work to investigate in how far *dlvhex* can benefit from lifting its current low-level integration into *clasp* to a higher level in combination with *clingo* 4. Clearly, the above considerations also apply to extensions of *dlvhex*, such as *acthex* (Fink et al. 2013). Furthermore, *jdlv* (Febbraro et al. 2012) encapsulates the *dlv* system to facilitate one-shot ASP solving in Java environments by providing means to generate and process logic programs, and to afterwards extract their stable models.

The procedural attachment to the *idp* system (De Pooter et al. 2013; De Cat et al. 2014) builds on interfaces to C++ and Lua. Like *clingo* 4, it allows for evaluating functions during grounding, calling the grounder and solver multiple times, inspecting solutions, and reacting to external input after search. The emphasis, however, lies on high-level control blending in with *idp*'s modeling language, while *clingo* 4 offers more fine-grained control over the grounding and solving process, particularly aiming at a flexible incremental assembly of programs from subprograms.

In SAT, incremental solver interfaces from low-level APIs are common practice. Pioneering work was done in *minisat* (Eén and Sörensson 2004), furnishing a C++ interface for solving under assumptions. In fact, the *clasp* library underlying *clingo* 4 builds upon this functionality to implement incremental search (see (Gebser et al. 2008)). Given that SAT deals with propositional formulas only, solvers and their APIs lack support for modeling languages and grounding. Unlike this, the SAT modulo Theory solver *z3* (de Moura and Bjørner 2008) comes with a Python API that, similar to *clingo* 4, provides a library for controlling the solver as well as language bindings for constraint handling. In this way, Python can be used as a modeling language for *z3*.

5 Discussion

The new *clingo* 4 system complements ASP's declarative input language by control capacities expressed by embedded scripting languages. This is accomplished within a single integrated ASP grounding and solving process in which a logic program may evolve over time. The addition, deletion, and replacement of programs is controlled procedurally by means of *clingo*'s dedicated library. The incentives for evolving a logic program are manifold and cannot be captured with the standard one-shot approach of ASP. Examples include unrolling a transition function, as in planning, interacting with an environment, as in assisted living, robotics, or stream reasoning, interacting with a user exploring a domain, theory solving, and advanced forms of search. Addressing these demands by embedded scripting languages provides us with a generic and transparent approach. Unlike this, previous systems, like *iclingo* and *oclingo*, had a dedicated purpose involving rigid control capacities buried in monolithic programs. Rather than that, the basic technology of *clingo* 4 allows us to instantiate subprograms in-between solver invocations in a fully customizable way. On the declarative side, the availability of program parameters and the embedding of `#external` directives into the grounding process provide great flexibility in modeling schematic subprograms. In addition, the possibility of assigning input atoms facilitates the implementation of applications such as query answering or sliding window reasoning, as truth values can now be switched without manipulating a logic program.

The semantic underpinnings of our framework in terms of module theory capture the dynamic combination of logic programs in a generic way. It is interesting future work to investigate how

dedicated change operations whose interest was so far mainly theoretic, like updating (Alferes et al. 2002) or forgetting (Zhang and Foo 2006), can be put into practice within this framework.

The input language of *clingo* 4 extends the *ASP-Core-2* standard (Calimeri et al. 2012). Although we have presented *clingo* 4 for normal logic programs, we mention that it accepts (extended) disjunctive logic programs, processed via the multi-threaded solving approach described in (Gebser et al. 2013b). In version 4.3, *clingo* moreover embeds *clasp* 3, featuring domain-specific heuristics (Gebser et al. 2013a) and optimization using unsatisfiable cores (Andres et al. 2012). *clingo* 4 is freely available at (potassco), and its releases include many best practice examples illustrating the aforementioned application scenarios.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9-1.

References

ALFERES, J., PEREIRA, L., PRZYMUSINSKA, H., AND PRZYMUSINSKI, T. 2002. LUPS: A language for updating logic programs. *Artificial Intelligence 138,* 1-2, 87–116.

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, A. Dovier and V. Santos Costa, Eds. Leibniz International Proceedings in Informatics, vol. 17. Dagstuhl Publishing, 212–221.

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2012. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.0.pdf`.

DE CAT, B., BOGAERTS, B., BRUYNOOGHE, M., AND DENECKER, M. 2014. Predicate logic as a modelling language: The IDP system. *CoRR abs/1401.6312*. Available at `http://arxiv.org/abs/1401.6312v1`.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, C. Ramakrishnan and J. Rehof, Eds. Lecture Notes in Computer Science, vol. 4963. Springer-Verlag, 337–340.

DE POOTER, S., WITTOCX, J., AND DENECKER, M. 2013. A prototype of a knowledge-based programming environment. In *Proceedings of the Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP'11) and the Twenty-fifth Workshop on Logic Programming (WLP'11)*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 7773. Springer-Verlag, 279–286.

DELGRANDE, J. AND FABER, W., Eds. 2011. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag.

EÉN, N. AND SÖRENSSON, N. 2004. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science, vol. 2919. Springer-Verlag, 502–518.

FEBBRARO, O., LEONE, N., GRASSO, G., AND RICCA, F. 2012. JASP: A framework for integrating answer set programming with Java. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, G. Brewka, T. Eiter, and S. McIlraith, Eds. AAAI Press, 541–551.

FINK, M., GERMANO, S., IANNI, G., REDL, C., AND SCHÜLLER, P. 2013. ActHEX: Implementing HEX programs with action atoms. In *Proceedings of the Twelfth International Conference on Logic*

Programming and Nonmonotonic Reasoning (LPNMR'13), P. Cabalar and T. Son, Eds. Lecture Notes in Artificial Intelligence, vol. 8148. Springer-Verlag, 317–322.

GEBSER, M., GROTE, T., KAMINSKI, R., AND SCHAUB, T. 2011a. Reactive answer set programming. See Delgrande and Faber (2011), 54–66.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND SCHNEIDER, M. 2011b. Potassco: The Potsdam answer set solving collection. *AI Communications 24,* 2, 107–124.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2008. Engineering an incremental ASP solver. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, M. Garcia de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, 190–205.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2014. Clingo = ASP + Control: Extended report. Available at `http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc14a.pdf`.

GEBSER, M., KAMINSKI, R., KÖNIG, A., AND SCHAUB, T. 2011c. Advances in gringo series 3. See Delgrande and Faber (2011), 345–351.

GEBSER, M., KAUFMANN, B., OTERO, R., ROMERO, J., SCHAUB, T., AND WANKO, P. 2013a. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, M. desJardins and M. Littman, Eds. AAAI Press, 350–356.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2013b. Advanced conflict-driven disjunctive answer set solving. In *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13)*, F. Rossi, Ed. IJCAI/AAAI Press, 912–918.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM 53,* 6, 937–977.

OIKARINEN, E. AND JANHUNEN, T. 2006. Modular equivalence for normal logic programs. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, Eds. IOS Press, 412–416.

OSTROWSKI, M. AND SCHAUB, T. 2012. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming 12,* 4-5, 485–503.

POTASSCO. `http://potassco.sourceforge.net`.

SYRJÄNEN, T. Lparse 1.0 user's manual. Available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

ZHANG, Y. AND FOO, N. 2006. Solving logic program conflict through strong and weak forgettings. *Artificial Intelligence 170,* 8-9, 739–778.

Transaction Logic with (Complex) Events

Ana Sofia Gomes and José Júlio Alferes*
CENTRIA - Dep. de Informática, Faculdade Ciências e Tecnologias
Universidade Nova de Lisboa

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

This work deals with the problem of combining reactive features, such as the ability to respond to events and define complex events, with the execution of transactions over general Knowledge Bases (KBs).

With this as goal, we build on Transaction Logic ($\mathcal{TR}$), a logic precisely designed to model and execute transactions in KBs defined by arbitrary logic theories. In it, transactions are written in a logic-programming style, by combining primitive update operations over a general KB, with the usual logic programming connectives and some additional connectives e.g. to express sequence of actions. While $\mathcal{TR}$ is a natural choice to deal with transactions, it remains the question whether $\mathcal{TR}$ can be used to express complex events, but also to deal simultaneously with the detection of complex events and the execution of transactions. In this paper we show that the former is possible while the latter is not. For that, we start by illustrating how $\mathcal{TR}$ can express complex events, and in particular, how SNOOP event expressions can be translated in the logic. Afterwards, we show why $\mathcal{TR}$ fails to deal with the two issues together, and to solve the intended problem propose Transaction Logic with Events, its syntax, model theory and executional semantics. The achieved solution is a non-monotonic extension of $\mathcal{TR}$, which guarantees that every complex event detected in a transaction is necessarily responded.

KEYWORDS: reactivity, complex events, transaction logic

1 Introduction

Reactivity stands for the ability to detect complex changes (also denoted as events) in the environment and react automatically to them according to some pre-defined rules. This is a pre-requisite of many real-world applications, such as web-services providing different services depending on external information, multi-agent systems adapting their knowledge and actions according to the changes in the environment, or monitoring systems reacting to information detected by their sensors and issuing actions automatically in response to it. In reactive systems, e.g. in those based on Event-Condition-Action (ECA) languages (Alferes et al. 2011; Bry et al. 2006; Chomicki et al. 2003), the reaction triggered by the detection of a complex event may itself be a complex action, formed e.g. by the sequencial execution of several basic actions. Moreover, we sustain that sometimes reactive systems are also required to execute *transactions* in response to events. For example, consider an airline web-service scenario where an external event arrives stating that a partner airline is on strike for a given time period. Then, the airline must address this event by e.g. rescheduling flights with alternative partners or refund tickets for passengers who do not accept

* The authors thank Michael Kifer for the valuable discussions in a preliminary version of this work. The first author was supported by the grant SFRH/BD/64038/2009 and by project ERRO (PTDC/EIA-CCO/121823/2010). The second author was supported by project ASPEN PTDC/EIA-CCO/110921/2009

the changes. Clearly, some transactional properties regarding these actions must be ensured: viz. it can never be the case that a passenger is simultaneously not refunded nor have an alternative flight; or that she is completely refunded and has a rescheduled flight.

Although the possibility of executing transactions is of crucial importance in many of today's systems, and a must e.g. in database systems, most reactive languages do not deal with it. Some exceptions exist, but are either completely procedural and thus lack from a clear declarative semantics (as e.g. in (Papamarkos et al. 2006)), or have a strong limitation on the expressivity of either the actions or events (as e.g. in (Zaniolo 1995; Lausen et al. 1998)).

In this paper we propose Transaction Logic with Events, $\mathcal{TR}^{ev}$, an extension of $\mathcal{TR}$ (Bonner and Kifer 1993) integrating the ability to reason and execute transactions over very general forms of KBs, with the ability to detect complex events. For this, after a brief overview of $\mathcal{TR}$, we show how it can be used to express and reason about complex events, and in particular, how it can express most SNOOP event operators (Adaikkalavan and Chakravarthy 2006) (Section 2). We proceed by showing why $\mathcal{TR}$ alone is not able to deal with both the detection of complex events and the execution of transactions, and, in particular, why it does not guarantee that all complex events detected during the execution of a transaction are responded within that execution. For solving this problem, we define $\mathcal{TR}^{ev}$, its language and model theory (Section 3.1), as well as its executional semantics (Section 3.2).

2 Using $\mathcal{TR}$ to express complex events

In this section we briefly recall $\mathcal{TR}$'s syntax and semantics with minor syntactic changes from the original, to help distinguish between actions and event occurrences, something that is useful ahead in the paper when extending $\mathcal{TR}$ to deal with reactive features and complex events.

Atoms in $\mathcal{TR}$ have the form $p(t_1,\ldots,t_n)$ where p is a predicate symbol and t_i's are terms (variables, constants, function terms). For simplicity, and without loss of generality (Bonner and Kifer 1998), we consider Herbrand instantiations, as usual. To build complex formulas, $\mathcal{TR}$ uses the classical connectives $\wedge, \vee, \neg, \leftarrow$ and the connectives $\otimes, \diamond$ denoting serial conjunction and hypothetical execution. Informally, $\phi \otimes \psi$ is an action composed of an execution of ϕ followed by an execution of ψ; and $\diamond\phi$ tests if ϕ can be executed without materializing the changes. In general, formulas are viewed as (the execution of) transactions, where, $\phi \wedge \psi$ is the simultaneous execution of ϕ and ψ; $\phi \vee \psi$ the non-deterministic choice of executing ϕ or ψ. $\phi \leftarrow \psi$ is a *rule* saying that one way to execute of ϕ is by executing ψ. As in classical logic, $\wedge$ and $\leftarrow$ can be written using $\vee$ and $\neg$ (e.g. $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$). Finally, we also use the connective ; as it is useful to express common complex events. $\phi;\psi$ says that ψ is true after ϕ but possibly interleaved with other occurrences, and it can be written in $\mathcal{TR}$ syntax as: $\phi \otimes \texttt{path} \otimes \psi$ where $\texttt{path} \equiv (\varphi \vee \neg\varphi)$ is a tautology that holds in paths of arbitrary size (Bonner and Kifer 1998).

For making possible the separation between the theory of states and updates, from the logic that combines them in transactions, $\mathcal{TR}$ considers a pair of oracles – $\mathcal{O}^d$ (data oracle) and $\mathcal{O}^t$ (transition oracle) – as a parameter of the theory. These oracles are mappings that assume a set of *state identifiers*. $\mathcal{O}^d$ is a mapping from state identifiers to a set of formulas that hold in that state, and $\mathcal{O}^t$ is a mapping from pairs of state identifiers to sets of formulas that hold in the transition of those states. These oracles can be instantiated with a wide variety of semantics, as e.g. relational databases, well-founded semantics, action languages, etc. (Bonner and Kifer 1993). For example, a relational database can be modeled by having states represented as sets of ground atomic formulas. Then, the data oracle simply returns all these formulas, i.e., $\mathcal{O}^d(D) = D$,

and for each predicate p in the KB, the transition oracle defines *p.ins* and *p.del*, representing the insertion and deletion of p, respectively. Formally, $p.ins \in \mathcal{O}^t(D_1, D_2)$ iff $D_2 = D_1 \cup \{p\}$ and, $p.del \in \mathcal{O}^t(D_1, D_2)$ iff $D_2 = D_1 \backslash \{p\}$. SQL-style bulk updates can also be defined by $\mathcal{O}^t$.

Example 1 (*Moving objects - $\mathcal{TR}$*)
As a $\mathcal{TR}$'s illustration, assume the prior relational database oracles and the action $move(O, X, Y)$ defining the relocation of object O from position X into position Y. In such a KB, states are defined using the predicates $location(O, P)$ saying that object O is in position P, and $clear(X)$ stating that X is clear to receive an object. In $\mathcal{TR}$, the move (trans)action can be expressed by:

$$move(O, X, Y) \leftarrow location(O, X) \otimes clear(Y) \otimes localUpdt(O, X, Y)$$
$$localUpdt(O, X, Y) \leftarrow location(O, X).del \otimes location(O, Y).ins \otimes clear(Y).del \otimes clear(X).ins$$

$\mathcal{TR}$'s theory is built upon the notion of sequences of states denoted as *paths*. Formulas are evaluated over paths, and truth in $\mathcal{TR}$ means *execution*: a formula is said to succeed over a path, if that path represents a valid execution for that formula. Although not part of the original $\mathcal{TR}$, here paths' state transitions are labeled with information about what (atomic occurrences) happen in the transition of states. Precisely, paths have the form $\langle D_0 \xrightarrow{O_1} D_2 \xrightarrow{O_2} \ldots \xrightarrow{O_k} D_k \rangle$, where D_i's are states and O_i's are labels (used later to annotate atomic event occurrences).

As usual, satisfaction of complex formulas is based on interpretations. These define what atoms are true in what paths, by mapping every path to a set of atoms. However, only the mappings compliant with the specified oracles are interpretations:

Definition 1 (*Interpretation*)
An interpretation is a mapping M assigning a set of atoms (or $\top$[1]) to every path, with the following restrictions (where D_is are states, and φ a formula):

1. $\varphi \in M(\langle D \rangle)$ if $\varphi \in \mathcal{O}^d(D)$
2. $\{\varphi, \mathbf{o}(\varphi)\} \subseteq M(\langle D_1 \xrightarrow{\mathbf{o}(\varphi)} D_2 \rangle)$ if $\varphi \in \mathcal{O}^t(D_1, D_2)$

In point 2 we additionally (i.e., when compared to the original definition) force $\mathbf{o}(\varphi)$ to belong to the same path where the primitive action φ is made true by the oracle, something that later (in Section 3) will help detect events associated with primitive actions, like "on insert/delete".

Next, we define operations on paths, and satisfaction of complex formulas over general paths.

Definition 2 (*Path Splits, Subpaths and Prefixes*)
Let π be a k-path, i.e. a path of length k of the form $\langle D_1 \xrightarrow{O_1} \ldots \xrightarrow{O_{k-1}} D_k \rangle$. A *split* of π is any pair of subpaths, π_1 and π_2, such that $\pi_1 = \langle D_1 \xrightarrow{O_1} \ldots \xrightarrow{O_{i-1}} D_i \rangle$ and $\pi_2 = \langle D_i \xrightarrow{O_i} \ldots \xrightarrow{O_{k-1}} D_k \rangle$ for some i ($1 \leqslant i \leqslant k$). In this case, we write $\pi = \pi_1 \circ \pi_2$.
A subpath π' of π is any subset of states and annotations of π where both the order of the states and their annotations is preserved. A prefix π_1 of π is any subpath of π sharing the initial state.

Definition 3 (*$\mathcal{TR}$ Satisfaction of Complex Formulas*)
Let M be an interpretation, π a path and ϕ a formula. If $M(\pi) = \top$ then $M, \pi \models_{\mathcal{TR}} \phi$; else:

1. **Base Case:** $M, \pi \models_{\mathcal{TR}} \phi$ iff $\phi \in M(\pi)$ for every event occurrence ϕ
2. **Negation:** $M, \pi \models_{\mathcal{TR}} \neg\phi$ iff it is not the case that $M, \pi \models_{\mathcal{TR}} \phi$
3. **Disjunction:** $M, \pi \models_{\mathcal{TR}} \phi \vee \psi$ iff $M, \pi \models_{\mathcal{TR}} \phi$ or $M, \pi \models_{\mathcal{TR}} \psi$.

[1] For not having to consider partial mappings, besides formulas, interpretations can also return the special symbol $\top$. The interested reader is referred to (Bonner and Kifer 1993) for details.

4. **Serial Conjunction:** $M, \pi \models_{\mathscr{TR}} \phi \otimes \psi$ iff there exists a split $\pi_1 \circ \pi_2$ of π s.t. $M, \pi_1 \models_{\mathscr{TR}} \phi$ and $M, \pi_2 \models_{\mathscr{TR}} \psi$
5. **Executional Possibility:** $M, \pi \models_{\mathscr{TR}} \diamond\phi$ iff π is a 1-path of the form $\langle D \rangle$ for some state D and $M, \pi' \models_{\mathscr{TR}} \phi$ for some path π' that begins at D.

Models and logical entailment are defined as usual. An interpretation models/satisfies a set of rules if each rule is satisfied in every possible path, and an interpretation models a rule in a path, if whenever it satisfies the antecedent, it also satisfies the consequent.

Definition 4 (*Models, and Logical Entailment*)
An interpretation M is a *model* of a formula ϕ iff for every path π, $M, \pi \models_{\mathscr{TR}} \phi$. M is a model of a set of rules P (denoted $M \models_{\mathscr{TR}} P$) iff it is a model of every rule in P.
ϕ is said to logically entail another formula ψ iff every model of ϕ is also a model of ψ.

Logical entailment is useful to define general equivalence and implication of formulas that express properties like "transaction ϕ is equivalent to transaction ψ" or "whenever transaction ψ is executed, ψ' is also executed". Moreover, if instead of transactions, we view the propositions as representing event occurrences, this entailment can be used to express complex events. For instance, imagine we want to state a complex event *alarm*, e.g. triggered whenever event ev_1 occurs after both ev_2 and ev_3 occur simultaneously. This can be expressed in $\mathscr{TR}$ as:

$$\mathbf{o}(alarm) \leftarrow (\mathbf{o}(e_2) \wedge \mathbf{o}(e_3)); \mathbf{o}(e_1) \tag{1}$$

In every model of this formula, whenever there is a (sub)path where both $\mathbf{o}(e_2)$ and $\mathbf{o}(e_3)$ are true, followed by a (sub)path where $\mathbf{o}(e_1)$ holds, then $\mathbf{o}(alarm)$ is true in the *whole* path.

Other complex event definitions are possible, and in fact we can encode most of SNOOP (Adaikkalavan and Chakravarthy 2006) operators in $\mathscr{TR}$. This is shown in Theorem 1 where, for a given history of past event occurrences, we prove that if an event expression is true in SNOOP, then there is a translation into a $\mathscr{TR}$ formula which is also true in that history. Since a SNOOP history is a set of atomic events associated with discrete points in time, the first step is to build a $\mathscr{TR}$ path expressing such history. We construct it as a sequence of state identifiers labeled with time, where time point i takes place in the transition of states $\langle s_i, s_{i+1} \rangle$, and only consider interpretations M over such a path that are *compatible* with SNOOP's history, i.e. such that, for every atomic event that is true in a time i, M makes the same event true in the path $\langle s_i, s_{i+1} \rangle$.

Theorem 1 (*SNOOP Algebra and $\mathscr{TR}$*)
Let E be a *SNOOP* algebra expression without periodic and aperiodic operators, H be a history containing the set of all SNOOP primitive events $e^i_j[t_1]$ that have occurred over the time interval t_1, t_{max}, and $\langle s_1, \ldots s_{max+1} \rangle$ be a path with size $t_{max} - t_1 + 1$. Let τ be the following function:

Primitive: $\tau(E) = \mathbf{o}(E)$ where E is a primitive event
Sequence: $\tau(E_1; E_2) = \tau(E_1) \otimes \mathtt{path} \otimes \tau(E_2)$
Or: $\tau(E_1 \triangledown E_2) = \tau(E_1) \vee \tau(E_2)$
AND: $\tau(E_1 \triangle E_2) = [(\tau(E_1) \otimes \mathtt{path}) \wedge (\mathtt{path} \otimes \tau(E_2))] \vee [(\tau(E_2) \otimes \mathtt{path}) \wedge (\mathtt{path} \otimes \tau(E_1))]$
NOT: $\tau(\neg(E_3)[E_1, E_2]) = \tau(E_1) \otimes \neg\tau(E_3) \otimes \tau(E_2)$

Then, $[t_i, t_f] \in E[H] \Rightarrow \forall M$ *compatible with* H, $M, \langle s_{t_i}, \ldots, s_{t_{f+1}} \rangle \models_{\mathscr{TR}} \tau(E)$, where, cf. (Adaikkalavan and Chakravarthy 2006), $E[H]$ is the set of time intervals (t_i, t_f) where E occurs over H in an unrestricted context, and where M is compatible with H if, for each $e^i_j[t_i] \in H$: $M, \langle s_{t_i}, s_{t_{i+1}} \rangle \models_{\mathscr{TR}} \mathbf{o}(e_j)$.

Besides the logical entailment, $\mathscr{TR}$ also provides the notion of executional entailment for reasoning about properties of a *specific* execution path.

Definition 5 (*Executional Entailment*)
Let P be a set of rules, ϕ a formula, and $D_0 \xrightarrow{O_1} \ldots \xrightarrow{O_n} D_n$ a path.
$P, (D_0 \xrightarrow{O_1} \ldots \xrightarrow{O_n} D_n) \models \phi$ ($\star$) iff for every model M of P, $M, \langle D_0 \xrightarrow{O_1} \ldots \xrightarrow{O_n} D_n \rangle \models \phi$.
Additionally, $P, D_0 \models \phi$ holds, if there is a path $D_0 \xrightarrow{O_1} \ldots \xrightarrow{O_n} D_n$ that makes ($\star$) true.

$P, (D_0 \xrightarrow{O_1} \ldots \xrightarrow{O_n} D_n) \models \phi$ says that a successful execution of transaction ϕ respecting the rules in P, can change the KB from state D_0 into D_n with a sequence of occurrences $O_1, \ldots, O_n$. E.g., in the Example 1 (with obvious abbreviations), the statement $P, (\{cl(t), l(c,o)\} \xrightarrow{\mathbf{o}(l(c,o).del)} \{cl(t)\} \xrightarrow{\mathbf{o}(l(c,t).ins)} \{cl(t), l(c,t)\} \xrightarrow{\mathbf{o}(cl(t).del)} \{l(c,t)\} \xrightarrow{\mathbf{o}(cl(o).ins)} \{l(c,t), cl(o)\}) \models move(c,o,t)$ means that a possible result of executing the transaction $move(c, oven, table)$ starting in the state $\{clear(table), loc(c, oven)\}$ is the path with those 5 states, ending in $\{loc(c, table), clear(oven)\}$.

This entailment has a corresponding proof theory (Bonner and Kifer 1993) which, for a subset of $\mathscr{TR}$, is capable of *constructing* such a path given a program, a $\mathscr{TR}$ formula, and an initial state. I.e. a path where the formula can be executed. If no such path exists, then the transaction fails, and nothing is built after the initial state.

3 $\mathscr{TR}^{ev}$: combining the execution of transactions with complex event detection

Reactive languages need to express behaviors like: "on *alarm* do action a_1 followed by action a_2", where the actions $a_1 \otimes a_2$ may define a transaction, and *alarm* is e.g. the complex event in (1). Clearly, $\mathscr{TR}$ can individually express and reason about transaction $a_1 \otimes a_2$, and its complex event. So, the question is whether it can deal with both simultaneously. For that, two important issues must be tackled: 1) how to model the triggering behavior of reactive systems, where the occurrence of an event drives the execution of a transaction in its response; 2) how to model the transaction behavior that prevents transactions to commit until all occurring events are responded.

Regarding 1), (Bonner et al. 1993) shows that simple events can be triggered in $\mathscr{TR}$ as:

$$\begin{array}{l} p \leftarrow body \otimes ev \\ ev \leftarrow \mathbf{r}(ev) \end{array} \tag{2}$$

With such rules, in all paths that make p true (i.e., in all executions of transaction p) the event ev is triggered/fired (after the execution of some arbitrary $body$), and ev's response, $\mathbf{r}(ev)$, is executed. Note that, both $\mathbf{r}(ev)$ and $body$ can be defined as arbitrary formulas.

But, this is just a very simple and specific type of event: atomic events that are explicitly triggered by a transaction defined in the program. In general, atomic events can also arrive as external events, or because some primitive action is executed in a path (e.g. as the database triggers - "on insert/on delete"). Triggering external events in $\mathscr{TR}$ can be done by considering the paths that make the external event true. E.g., if one wants to respond to an external event ev from an initial state, all we need to do is find the paths π starting in that state, s.t. $P, \pi \models ev$, where P includes the last rule from (2) plus the rules defining ev's response.

The occurrences of primitive actions can be tackled by Point 2 of Def. 1, and the occurrence of complex events can be defined as prescribed in Section 2. However, the above approach of (Bonner et al. 1993) does not help for driving the execution of an event response when such occurrences become true. For instance, the ECA-rule before could be stated as:

$$\begin{array}{l} \mathbf{o}(alarm) \leftarrow (\mathbf{o}(e_2) \wedge \mathbf{o}(e_3)); \mathbf{o}(e_1) \\ \mathbf{r}(alarm) \leftarrow a_1 \otimes a_2 \end{array}$$

But this does not drive the execution of $\mathbf{r}(alarm)$ when $\mathbf{o}(alarm)$ holds; one has further to force that whenever $\mathbf{o}(alarm)$ holds, $\mathbf{r}(alarm)$ must be made true subsequently. Of course, adding a rule $\mathbf{r}(alarm) \leftarrow \mathbf{o}(alarm)$ would not work: such rule would only state that, one alternative way

to satisfy the response of alarm is to make its occurrence true. And for that, it would be enough to satisfy $\mathbf{o}(alarm)$ to make $\mathbf{r}(alarm)$ true, which is not what is intended.

Clearly, this combination implies two different types of formulas with two very different behaviors: the *detection* of events which are tested for occurrence w.r.t. a past history; and the *execution* of transactions as a response to them, which intends to construct paths where formulas can succeed respecting transactional properties. This has to be reflected in the semantics and these formulas should be evaluated differently accordingly to their nature.

Regarding 2), as in database triggers, transaction's execution must depend on the events triggered. Viz., an event occurring during a transaction execution can delay that transaction to commit/succeed until the event response is successfully executed, and the failure of such response should imply the failure of the whole transaction. Encoding this behavior requires that, if an event occurs during a transaction, then its execution needs to be *expanded* with the event response. Additionally, this also precludes transactions to succeed in paths where an event occurs and is not responded (even if the transaction would succeed in that path if the event did not existed).

For addressing these issues, below we define $\mathscr{TR}^{ev}$. This extension of $\mathscr{TR}$ evaluates event formulas and transaction formulas differently, using two distinct relations (respectively $\models_{\mathscr{TR}}$ and $\models$), and occurrences and responses are syntactic represented w.r.t. a given event name e, as $\mathbf{o}(e)$ and $\mathbf{r}(e)$, respectively. In this context, $\models$ requires transactions to be satisfied in expanded paths, where every occurring event (made true by $\models_{\mathscr{TR}}$) is properly responded.

3.1 $\mathscr{TR}^{ev}$ Syntax and Model Theory

To make possible a different evaluation of events and transactions, predicates in $\mathscr{TR}^{ev}$ are partitioned into transaction names ($\mathscr{P}_t$), event names ($\mathscr{P}_e$), and oracle primitives ($\mathscr{P}_{\mathscr{O}}$) and, as with $\mathscr{TR}$, we work with the Herbrand instantiation of the language.

Formulas in $\mathscr{TR}^{ev}$ are partitioned into transaction formulas and event formulas. *Event formulas* denote formulas meant to be *detected* and are either an event occurrence, or an expression defined inductively as $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \otimes \psi$, or $\phi ; \psi$ where ϕ and ψ are event formulas. An *event occurrence* is of the form $\mathbf{o}(\varphi)$ s.t. $\varphi \in \mathscr{P}_e$ or $\varphi \in \mathscr{P}_{\mathscr{O}}$. Note that, we preclude the usage of $\diamond$ in event formulas, as it would make little sense to detect occurrences based on what could possibly be executed.

Transaction formulas are formulas that can be *executed*, and are either a transaction atom, or an expression defined inductively as $\neg\phi$, $\diamond\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, or $\phi \otimes \psi$. A *transaction atom* is either a transaction name (in $\mathscr{P}_t$), an oracle defined primitive (in $\mathscr{P}_{\mathscr{O}}$), the response to an event ($\mathbf{r}(\varphi)$ where $\varphi \in \mathscr{P}_{\mathscr{O}} \cup \mathscr{P}_e$), or an event name (in $\mathscr{P}_e$) The latter corresponds to the (trans)action of *explicitly* triggering an event directly in a transaction as in (2) or as an external event. As we shall see (Def. 7) explicitly triggering an event changes the path of execution (by asserting the information that the event has happened in the current state) and, as such, is different from simply inferring (or detecting) what events hold given a past path.

Finally, rules have the form $\varphi \leftarrow \psi$ and can be transaction or (complex) event rules. In a transaction rule φ is a transaction atom and ψ a transaction formula; in an event rule φ is an event occurrence and ψ is a event formula. A *program* is a set of transaction and event rules.

Importantly, besides the data and transition oracles, $\mathscr{TR}^{ev}$ is also parametric on a *choice* function defining what event should be selected at a given time in case of conflict. Since defining what event should be picked from the set of occurring events depends on the application in mind, $\mathscr{TR}^{ev}$ does not commit to any particular definition, encapsulating it in function *choice*.

As a reactive system, $\mathscr{TR}^{ev}$ receives a series of external events which may cause the execution of transactions in response. This is defined as $P, D_0 \models e_1 \otimes \ldots \otimes e_k$, where D_0 is the initial KB state and $e_1 \otimes \ldots \otimes e_k$ is the sequence of external events that arrive to the system. Here, we want to find the path $D_0 \overset{O_1}{\hookrightarrow} \ldots \overset{O_n}{\hookrightarrow} D_n$ encoding a KB evolution that responds to $e_1 \otimes \ldots \otimes e_k$.

As mentioned, triggering explicit events is a transaction formula encoding the *action* of making an occurrence explicitly true. This is handled by the definition of interpretation, in a similar way to how atomic events defined by oracles primitives are made true:

Definition 6 ($\mathscr{TR}^{ev}$ *interpretations*)
A $\mathscr{TR}^{ev}$ interpretation is a $\mathscr{TR}$ interpretation that additionally satisfies the restriction: 3) $\mathbf{o}(e) \in M(\langle D \overset{\mathbf{o}(e)}{\hookrightarrow} D \rangle)$ if $e \in \mathscr{P}_e$

We can now define the satisfaction of complex formulas, and then models of a program. Event formulas are evaluated w.r.t. the relation $\models_{\mathscr{TR}}$ specified in Def. 3. Transaction formulas are evaluated w.r.t. the relation $\models$ which requires formulas to be true in *expanded paths*, in which every occurring event is responded (something dealt by $\mathtt{exp}_M(\pi)$, defined below).

Definition 7 (*Satisfaction of Transaction Formulas and Models*)
Let M be an interpretation, π a path, ϕ transaction formula. If $M(\pi) = \top$ then $M, \pi \models \phi$; else:

1. **Base Case:** $M, \pi \models p$ iff $\exists \pi'$ prefix of π s.t. $p \in M(\pi')$ and $\pi = \mathtt{exp}_M(\pi')$, for every transaction atom p where $p \notin \mathscr{P}_e$.
2. **Event Case:** $M, \pi \models e$ iff $e \in \mathscr{P}_e$, $\exists \pi'$ prefix of π s.t. $M, \pi' \models_{\mathscr{TR}} \mathbf{o}(e)$ and $\pi = \mathtt{exp}_M(\pi')$.
3. **Negation:** $M, \pi \models \neg\phi$ iff it is not the case that $M, \pi \models \phi$
4. **Disjunction:** $M, \pi \models \phi \vee \psi$ iff $M, \pi \models \phi$ or $M, \pi \models \psi$.
5. **Serial Conjunction:** $M, \pi \models \phi \otimes \psi$ iff $\exists \pi'$ prefix of π and some split $\pi_1 \circ \pi_2$ of π' such that $M, \pi_1 \models \phi$ and $M, \pi_2 \models \psi$ and $\pi = \mathtt{exp}_M(\pi')$.
6. **Executional Possibility:** $M, \pi \models \diamond\phi$ iff π is a 1-path of the form $\langle D \rangle$ for some state D and $M, \pi' \models \phi$ for some path π' that begins at D.

An interpretation M is a *model* of a transaction formula (resp. event formula) ϕ iff for every path π, $M, \pi \models \phi$ (resp. $M, \pi \models_{\mathscr{TR}} \phi$). M is a model of a program P (denoted $M \models P$) iff it is a model of every (transaction and complex event) rule in P.

$\mathtt{exp}_M(\pi)$ is a function that, given a path with possibly unanswered events, expands it with the result of responding to those events. Its definition must perforce have some procedural nature: it must start by detecting which are the unanswered events; pick one of them, according to a given *choice* function; then expand the path with the response of the chosen event. The response to this event, computed by operator $\mathscr{R}_M$ defined below, may, in turn, generate the occurrence of further events. So, $\mathscr{R}_M$ must be iterated until no more unanswered events exist.

Definition 8 (*Expansion of a Path*)
For a path π_1 and an interpretation M, the response operator $\mathscr{R}_M(\pi_1)$ is defined as follows:

$$\mathscr{R}_M(\pi_1) = \begin{cases} \pi_1 \circ \pi_2 & \textbf{if } \mathit{choice}(M, \pi_1) = e \text{ and } M, \pi_2 \models \mathbf{r}(e) \\ \pi_1 & \textbf{if } \mathit{choice}(M, \pi_1) = \epsilon \end{cases}$$

The expansion of a path π is $\mathtt{exp}_M(\pi) = \uparrow \mathscr{R}_M(\pi)$.

In general it may not be possible to address all events in a finite path, and thus, $\mathscr{R}_M$ may not have a fixed-point. In fact, non-termination is a known problem of reactive systems, and is often undecidable for the general case (Bailey et al. 2004). However, if termination is possible, then a fixed-point exists and each iteration of $\mathscr{R}_M$ is an approximation of the expansion operator $\mathtt{exp}_M$.

This definition leaves open the *choice* function, that is taken as a further parameter of $\mathscr{TR}^{ev}$, and specifies how to choose the next unanswered event to respond to. For its instantiation one needs to decide: 1) in which order should events be responded and 2) how should an event be responded. The former defines the handling order of events in case of conflict, e.g. based on when events have occurred (temporal order), on a priority list, or any other criteria. The latter defines the response policy of an ECA-language, i.e. when is an event considered to be responded. E.g., if an event occurs more than once before the system can respond to it, this specifies if such response should be issued only once or equally to the amount of occurrences. Choosing the appropriate operational semantics depends on the application in mind. In the following definition we exemplify how this *choice* function can be instantiated, for a case when events are responded in the (temporal) order in which they occurred, and events for which there was already a response are not responded again.

Definition 9 (*Temporal choice function*)
Let M be an interpretation and π be a path. The temporal choice function is $choice(M,\pi) = firstUnans(M,\pi,order(M,\pi))$ where:

- $order(M,\pi) = \langle e_1,\ldots,e_n\rangle$ iff $\forall e_i\ 1 \leqslant i \leqslant n$, $\exists \pi_i$ subpath of π where $M,\pi \models_{\mathscr{TR}} \mathbf{o}(e_i)$ and $\forall e_j$ s.t. $i<j$ then e_j occurs after e_i
- e_2 occurs after e_1 w.r.t. π and M iff there exists π_1,π_2 subpaths of π such that $\pi_1 = \langle D_i \overset{O_i}{\hookrightarrow} \ldots \overset{O_{j-1}}{\hookrightarrow} D_j\rangle$, $\pi_2 = \langle D_n \overset{O_n}{\hookrightarrow} \ldots \overset{O_{m-1}}{\hookrightarrow} D_m\rangle$, $M,\pi_1 \models \mathbf{o}(e_1)$, $M,\pi_2 \models \mathbf{o}(e_2)$ and $D_j \leqslant D_m$ w.r.t. the ordering in π.
- $firstUnans(M,\pi,\langle e_1,\ldots,e_n\rangle) = e_i$ iff e_i is the first event in $\langle e_1,\ldots,e_n\rangle$ where given π' subpath of π and $M,\pi' \models_{\mathscr{TR}} \mathbf{o}(e)$ then $\neg\exists\pi''$ s.t. π'' is also a subpath of π, π'' is after π' and $M,\pi'' \models \mathbf{r}(e)$.

We continue by exemplifying the semantics in examples.

Example 2

$$\begin{array}{l} p \leftarrow a.ins \\ \mathbf{r}(e_1) \leftarrow c.ins \end{array} \quad (P_3) \qquad \begin{array}{ll} p \leftarrow a.ins & \\ \mathbf{r}(e_1) \leftarrow c.ins & (P_4) \\ \mathbf{o}(e_1) \leftarrow \mathbf{o}(a.ins) & \end{array}$$

Consider the programs[2] P_3 and P_4. In P_3, p holds in the path $\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle$. This is true since all interpretations must comply with the oracles and thus $\forall M$: $a.ins \in M(\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle)$ implying $M,\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle \models a.ins$. Assuming that M is a model of P_3, then it satisfies the rule $p \leftarrow a.ins$, which means that $p \in M(\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle)$ and $M,\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle \models p$.

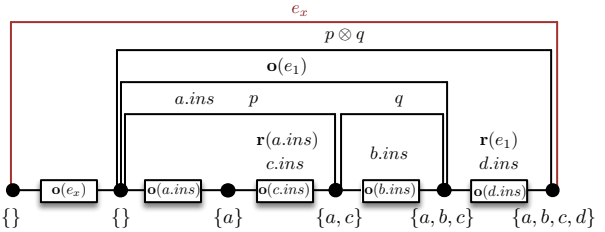
However, since $\mathbf{o}(e_1) \leftarrow \mathbf{o}(a.ins) \in P_4$ and $\forall M.\mathbf{o}(a.ins) \in M(\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle)$, for M to be a model of P_4, then $\mathbf{o}(e_1) \in M(\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle)$. Since e_1 has a response defined, then in path $\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle$ the occurrence e_1 is unanswered and both the transactions p and $a.ins$ cannot succeed in that path. Namely, $\mathbf{o}(e_1)$ constrains the execution of *every* transaction in the path $\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\}\rangle$ and, for transaction formulas to succeed, such path needs to be *expanded* with e_1's response. Since, $\mathtt{exp}_M(\langle\{\} \overset{a.ins}{\hookrightarrow} \{a\}\rangle) = \langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\} \overset{\mathbf{o}(c.ins)}{\hookrightarrow} \{a,c\}\rangle$ then, both transactions p and $a.ins$ succeed in the *longer* path $\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\} \overset{\mathbf{o}(c.ins)}{\hookrightarrow} \{a,c\}\rangle$, i.e. for an M model of P_4: $M,\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\} \overset{\mathbf{o}(c.ins)}{\hookrightarrow} \{a,c\}\rangle \models p$ and $M,\langle\{\} \overset{\mathbf{o}(a.ins)}{\hookrightarrow} \{a\} \overset{\mathbf{o}(c.ins)}{\hookrightarrow} \{a,c\}\rangle \models a.ins$. Notice the non-monotonicity of $\mathscr{TR}^{ev}$, viz. that adding a new event rule to P_3 falsifies the transaction formulas p and $a.ins$ in paths where they were previously true.

[2] For brevity, in this and the following examples we assume the rule $\mathbf{r}(p) \leftarrow \mathtt{true}$ to appear in every program for every primitive action p defined in the signature of the oracles, unless when stated otherwise. I.e., we assume the responses of events inferred from primitive actions to hold trivially whenever their rules do not appear explicitly in the program.

As in $\mathscr{TR}$, in $\mathscr{TR}^{ev}$ every formula that is meant to be executed, is meant to be executed as a transaction. As such, the primitive *a.ins* in example P_4 cannot succeed in the path $\langle \{\} \xrightarrow{\mathbf{o}(a.ins)} \{a\} \rangle$ since there are unanswered events in that path. However, note that *a.ins* belongs to every interpretation M of that path (due to the restrictions in Def. 1). Thus the primitive *a.ins* is true in $\langle \{\} \xrightarrow{\mathbf{o}(a.ins)} \{a\} \rangle$ although the transaction *a.ins* is not.

Example 3

$p \leftarrow a.ins$
$q \leftarrow b.ins$
$\mathbf{r}(e_x) \leftarrow p \otimes q$
$\mathbf{r}(e_1) \leftarrow d.ins$
$\mathbf{r}(a.ins) \leftarrow c.ins$
$\mathbf{o}(e_1) \leftarrow \mathbf{o}(a.ins) \otimes \mathbf{o}(b.ins)$



The right-hand side figure illustrates a satisfaction of the external event e_x. The occurrence of e_x forces the satisfaction of the transaction $p \otimes q$, which is true if both its "subformulas" (p and q) are satisfied over smaller paths. Note that, by definition of the relation $\models$, all occurrences detected over the independent paths that satisfy p and q are already responded in those paths. Thus, we need only to cater for the events triggered due to the serial conjunction. Here, for a model M of the program, $M, \langle \{\} \xrightarrow{\mathbf{o}(a.ins)} \{a\} \xrightarrow{\mathbf{o}(c.ins)} \{a,c\} \rangle \models p$ and $M, \langle \{a,c\} \xrightarrow{\mathbf{o}(b.ins)} \{a,b,c\} \rangle \models q$. Further, the rule $\mathbf{o}(e_1) \leftarrow \mathbf{o}(a.ins) \otimes \mathbf{o}(b.ins)$ defines one pattern for the occurrence of e_1 which constrains the execution of transaction $p \otimes q$ and forces the expansion of the path to satisfy $\mathbf{r}(e_1)$. Consequently, $M, \langle \{\} \xrightarrow{\mathbf{o}(a.ins)} \{a\} \xrightarrow{\mathbf{o}(c.ins)} \{a,c\} \xrightarrow{\mathbf{o}(b.ins)} \{a,b,c\} \xrightarrow{\mathbf{o}(d.ins)} \{a,b,c,d\} \rangle \models p \otimes q$, and $M, \langle \{\} \xrightarrow{\mathbf{o}(e_x)} \{\} \xrightarrow{\mathbf{o}(a.ins)} \{a\} \xrightarrow{\mathbf{o}(c.ins)} \{a,c\} \xrightarrow{\mathbf{o}(b.ins)} \{a,b,c\} \xrightarrow{\mathbf{o}(d.ins)} \{a,b,c,d\} \rangle \models e_x$

3.2 Entailment and Properties

The logical entailment defined in Def. 4 can be used to reason about properties of transaction and event formulas that hold for *every* possible path of execution. In $\mathscr{TR}^{ev}$, similarly to $\mathscr{TR}$, we further define executional entailment, to talk about properties of a *particular* execution path. But, to reason about the execution of transactions over a specific path, care must be taken since, as described above, the satisfaction of a new occurrence in a path may invalidate transaction formulas that were previously true.

To deal with a similar behavior, non-monotonic logics rely on the concept of minimal or preferred models: instead of considering all possible models, non-monotonic theories restrict to the most skeptical ones. Likewise, $\mathscr{TR}^{ev}$ uses the minimal models of a program to define entailment, whenever talking about a particular execution of a formula. As usual, minimality is defined by set inclusion on the amount of predicates that an interpretation satisfies, and a minimal model is a model that minimizes the set of formulas that an interpretation satisfies in a path.

Definition 10 (*Minimal Model*)
Let M_1 and M_2 be interpretations. Then $M_1 \leqslant M_2$ if $\forall \pi$: $M_2(\pi) = \top \vee M_1(\pi) \subseteq M_2(\pi)$
Let ϕ be a $\mathscr{TR}^{ev}$ formula, and P a program. M is a *minimal model* of ϕ (resp. P) if M is a model of ϕ (resp. P) and $M \leqslant M'$ for every model M' of ϕ (resp. P).

Thus, to know if a formula succeeds in a particular path, we need only to consider the event occurrences *supported* by that path, either because they appear as occurrences in the transition of states, or because they are a necessary consequence of the program's rules given that path. Because of this, executional entailment in $\mathscr{TR}^{ev}$ is defined w.r.t. minimal models (cf. Def. 5).

Definition 11 ($\mathscr{TR}^{ev}$ *Executional Entailment*)
Let P be a program, ϕ a transaction formula and $D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n$ a path. Then $P, (D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n) \models \phi$ ($\star$) iff for every minimal model M of P, $M, \langle D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n \rangle \models \phi$. $P, D_1 - \models \phi$ is said to be true, if there is a path $D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n$ that makes ($\star$) true.

Interestingly, as in logic programs, formulas satisfied by this entailment have some support.

Lemma 1 (*Support*)
Let P be a program, π a path, ϕ a transaction atom. Then, if $P, \pi \models \phi$ one of the following holds:

1. ϕ is an elementary action and either $\phi \in \mathscr{O}^d(\pi)$ or $\phi \in \mathscr{O}^t(\pi)$;
2. ϕ is the head of a transaction rule in P ($\phi \leftarrow body$) and $P, \pi \models body$;

As expected, $\mathscr{TR}^{ev}$ extends $\mathscr{TR}$. Precisely, if a program P has no complex event rules, and for every elementary action a defined by the oracles the only rule for $\mathbf{r}(a)$ in P is $\mathbf{r}(a) \leftarrow \texttt{true}$, then executional entailment in $\mathscr{TR}^{ev}$ can be recast in $\mathscr{TR}$ if, $\mathscr{TR}$ executional entailment is also restricted to minimal models. It is worth noting that, for a large class of $\mathscr{TR}$ theories, and namely for the so-called serial-Horn theories, executional entailment in general coincides with that only using minimal models (cf. (Bonner and Kifer 1993)). As an immediate corollary, it follows that if P is *event-free* and serial-Horn, then executional entailment in $\mathscr{TR}^{ev}$ and in $\mathscr{TR}$ coincide.

4 Discussion and Related Work

Several solutions exist to reason about complex events. Complex event processing (CEP) systems as (Adaikkalavan and Chakravarthy 2004; Wu et al. 2006) can reason efficiently with large streams of data and detect (complex) events. These support a rich specification of events based on event pattern rules combining atomic events with some temporal constructs. As shown in Theorem 1, $\mathscr{TR}$ and $\mathscr{TR}^{ev}$ can express most event patterns of SNOOP and, ETALIS (Anicic et al. 2012) CEP system even uses $\mathscr{TR}$'s syntax and connectives, although abandoning $\mathscr{TR}$'s model theory and providing a different satisfaction definition. However, in contrast to $\mathscr{TR}^{ev}$, CEP systems do not deal with the execution of actions in reaction to the events detected.

Extensions of Situation Calculus, Event Calculus, Action Languages, etc. exist with the ability to react to events, and have some transactional properties (Baral et al. 1997; Bertossi et al. 1998). However, as in database triggers, these events are restricted to detect simple actions like "on insert/delete" and thus have a very limited expressivity that fails to encode complex events, as defined in CEP systems and in $\mathscr{TR}^{ev}$. To simultaneously reason about actions and complex events, ECA (following the syntax "on *event* if *condition* do *action*") languages (Alferes et al. 2011; Bry et al. 2006; Chomicki et al. 2003) and logic programming based languages (Kowalski and Sadri 2012; Costantini and Gasperis 2012) exist. These languages normally do not allow the action component of the language to be defined as a transaction, and when they do, they lack from a declarative semantics as (Papamarkos et al. 2006); or they are based on active databases and can only detect atomic events defined as insertions/deletes (Zaniolo 1995; Lausen et al. 1998).

In contrast, $\mathscr{TR}^{ev}$ can deal with arbitrary atomic and complex events, and make these events trigger transactions. This is done by a logic-programming like declarative language. We have also defined a procedure to execute these reactive transactions, which is built upon the complex event detection algorithm of ETALIS and the execution algorithm of $\mathscr{TR}$, but is omitted for lack of space.

References

ADAIKKALAVAN, R. AND CHAKRAVARTHY, S. 2004. Formalization and detection of events over a sliding window in active databases using interval-based semantics. In *ADBIS*. 241–256.

ADAIKKALAVAN, R. AND CHAKRAVARTHY, S. 2006. Snoopib: Interval-based event specification and detection for active databases. *Data Knowl. Eng. 59,* 1, 139–165.

ALFERES, J. J., BANTI, F., AND BROGI, A. 2011. Evolving reactive logic programs. *Intelligenza Artificiale 5,* 1, 77–81.

ANICIC, D., RUDOLPH, S., FODOR, P., AND STOJANOVIC, N. 2012. Stream reasoning and complex event processing in etalis. *Semantic Web 3,* 4, 397–407.

BAILEY, J., DONG, G., AND RAMAMOHANARAO, K. 2004. On the decidability of the termination problem of active database systems. *Theor. Comput. Sci. 311,* 1-3, 389–437.

BARAL, C., LOBO, J., AND TRAJCEVSKI, G. 1997. Formal characterizations of active databases: Part ii. In *DOOD*. LNCS, vol. 1341. Springer, 247–264.

BERTOSSI, L. E., PINTO, J., AND VALDIVIA, R. 1998. Specifying active databases in the situation calculus. In *SCCC*. IEEE Computer Society, 32–39.

BONNER, A. J. AND KIFER, M. 1993. Transaction logic programming. In *ICLP*. 257–279.

BONNER, A. J. AND KIFER, M. 1998. Results on reasoning about updates in transaction logic. In *Transactions and Change in Logic Databases*. 166–196.

BONNER, A. J., KIFER, M., AND CONSENS, M. P. 1993. Database programming in transaction logic. In *DBPL*. 309–337.

BRY, F., ECKERT, M., AND PATRANJAN, P.-L. 2006. Reactivity on the web: Paradigms and applications of the language xchange. *J. Web Eng. 5,* 1, 3–24.

CHOMICKI, J., LOBO, J., AND NAQVI, S. A. 2003. Conflict resolution using logic programming. *IEEE Trans. Knowl. Data Eng. 15,* 1, 244–249.

COSTANTINI, S. AND GASPERIS, G. D. 2012. Complex reactivity with preferences in rule-based agents. In *RuleML*. 167–181.

KOWALSKI, R. A. AND SADRI, F. 2012. A logic-based framework for reactive systems. In *RuleML*. 1–15.

LAUSEN, G., LUDÄSCHER, B., AND MAY, W. 1998. On active deductive databases: The statelog approach. In *Transactions and Change in Logic Databases*. 69–106.

PAPAMARKOS, G., POULOVASSILIS, A., AND WOOD, P. T. 2006. Event-condition-action rules on rdf metadata in p2p environments. *Comp. Networks 50,* 10, 1513–1532.

WU, E., DIAO, Y., AND RIZVI, S. 2006. High-performance complex event processing over streams. In *SIGMOD Conference*. ACM, 407–418.

ZANIOLO, C. 1995. Active database rules with transaction-conscious stable-model semantics. In *DOOD*. 55–72.

*(Co)recursion in Logic Programming: Lazy vs Eager**

JÓNATHAN HERAS
School of Computing, University of Dundee, UK
(*e-mail:* jonathanheras@computing.dundee.ac.uk)

EKATERINA KOMENDANTSKAYA
School of Computing, University of Dundee, UK
(*e-mail:* katya@computing.dundee.ac.uk)

MARTIN SCHMIDT
Institute of Cognitive Science, University of Osnabrück, Germany
(*e-mail:* martisch@uos.de)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

CoAlgebraic Logic Programming (CoALP) is a dialect of Logic Programming designed to bring a more precise compile-time and run-time analysis of termination and productivity for recursive and corecursive functions in Logic Programming. Its second goal is to introduce guarded lazy (co)recursion akin to functional theorem provers into logic programming. In this paper, we explain lazy features of CoALP, and compare them with the loop-analysis and eager execution in Coinductive Logic Programming (CoLP). We conclude by outlining the future directions in developing the guarded (co)recursion in logic programming.

KEYWORDS: Logic Programming, Recursion, Corecursion, Termination, Productivity, Guardedness.

1 Introduction

Logic Programming (LP) was conceived as a *recursive* programming language for first-order logic. Prolog and various other implementations of LP feature *eager* derivations, and therefore *termination* has been central for logic programming (de Schreye and Decorte 1994). However, unlike e.g. functional languages, LP has not developed an operational semantics supporting explicit analysis of termination. In typed programming languages like Coq or Agda, it is possible to introduce syntactic (static) checks that ensure *structural* recursion, and hence termination of programs at run-time. In Prolog, there is no support of this kind.

Example 1.1 (BitList) *Consider the following recursive program that defines lists of bits.*

$$\begin{aligned} 1.&\texttt{bit(0)} &&\leftarrow \\ 2.&\texttt{bit(1)} &&\leftarrow \\ 3.&\texttt{bitlist([])} &&\leftarrow \\ 4.&\texttt{bitlist([X|Y])} &&\leftarrow \texttt{bit(X)}, \texttt{bitlist(Y)} \end{aligned}$$

* The work of the first two authors was supported by EPSRC Grant EP/K031864/1.

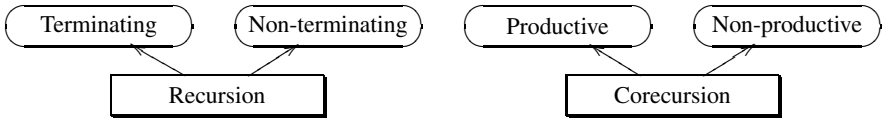


Fig. 1. *Distinguishing well-founded and non-well-founded cases of recursion and corecursion.*

It is a terminating program, however, if the order of clauses (3) and (4), or the order of atoms in clause (4) is accidentally swapped, the program would run into an infinite loop.

This example illustrates that non-terminating (co)recursion is distinguished only empirically at run-time in LP. This distinction is not always accurate, and may depend on searching strategies of the compiler, rather than semantic meaning of the program.

Coinductive Logic Programming (CoLP) (Gupta et al. 2007; Simon et al. 2007) has been introduced as a means of supporting *corecursion* in LP. A representative example of coinductive programming is to reason about an infinite data structure, for example an infinite stream of bits.

Example 1.2 (BitStream) *Given the definition of bits as in Example 1.1, an infinite stream of bits is defined as:*

$$1.\,\texttt{stream([X|Y])} \;\leftarrow\; \texttt{bit(X)}, \texttt{stream(Y)}$$

*Note that unlike **BitList**, we no longer have the base case for recursion on* `stream`*.*

The tradition (Coquand 1994) has a dual notion to termination for well-behaving *corecursion* – and that is *productivity*. If termination imposes the condition that any call to an inductively defined predicate like `bit` must terminate, then productivity requires that every call to a coinductive predicate like `stream` must *produce* some partially *observed* structure in a finite number of steps. E.g. calling `stream(X)?`, the program must compute an answer `[0|Y]` observing the component `0` in finite time. Moreover, the productivity imposes a second condition: the computation must be able to proceed corecursively, e.g. in our example, the condition is for `Y` to be an infinite productive datastructure. This situation is explained in e.g. (Abel et al. 2013; Bertot and Komendantskaya 2008).

CoLP deals with programs like **BitStream** by using a combination of eager evaluation, SLD-resolution and loop analysis. In simplified terms, for a goal `stream(X)?` the resolvent loop detection would allow to return an answer `X=[0|X]`; by observing the *"regular"* pattern in resolvents involving Clause (1) in the derivations. Similarly to standard (recursive) LP, non-terminating cases of corecursion (where no regular loop can be found) are not formally analysed in CoLP.

Example 1.3 (BadStream) ***BadStream*** *is not productive; that is, it would be executed infinitely without actually constructing a stream.*

$$1.\,\texttt{badstream([X|Y])} \;\leftarrow\; \texttt{badstream([X|Y])}$$

A different case of corecursion is the below example, which is productive, but cannot be handled by CoLP loop detector, as the stream it defines is not regular.

Example 1.4 (TakeFirstN) *The program* `TakeFirstN` *defines the stream of natural numbers,*

and allows to construct a list with the first n elements of the stream by calling `taken`.

$$\begin{aligned} 1.\,\texttt{from(X,[X|Y])} &\leftarrow \texttt{from(s(X),Y)} \\ 2.\,\texttt{take(0,Y,[])} &\leftarrow \\ 3.\,\texttt{take(s(X),[Y|Z],[Y|R])} &\leftarrow \texttt{take(X,Z,R)} \\ 4.\,\texttt{taken(N,X)} &\leftarrow \texttt{from(0,Y)}, \texttt{take(N,Y,X)} \end{aligned}$$

In CoLP, calls to e.g. `taken(s(s(0)),X)?` fall into infinite computations that are not handled by the loop detection procedure. Similar to how Prolog would be unable to handle **BitList** with swapped atoms in clause (4) though in principle the program describes a well-founded inductive structure, CoLP would not be able to handle **TakeFirstN** although it is a perfectly productive stream. For the query `taken`, it is intuitively clear that, the construction of the first n elements of the stream should take a finite number of derivation steps.

Coalgebraic Logic Programming (CoALP) (Komendantskaya and Power 2011; Komendantskaya et al. 2014a) gives a new (coalgebraic) operational semantics for LP; and in particular it offers new methods to analyse termination and productivity of logic programs. Using CoALP, we present here a coherent operational treatment of recursion and corecursion in LP, and discuss new methods to distinguish well-founded and non-well-founded cases of (co)recursion in LP, as outlined in Figure 1. Unlike Prolog or CoLP, CoALP is a first lazy dialect of logic programming; and it features guarded (co)recursion akin to structural recursion and guarded corecursion in e.g. Coq or Agda (Coquand 1994; Abel et al. 2013). The current implementation of CoALP in the parallel language Go is available in (Komendantskaya et al. 2014b); and is tested on a few benchmarks in this paper. Here, we abstract from some of the technical details available in (Komendantskaya et al. 2014a) and from implementation details available in (Komendantskaya et al. 2014c) and give a higher-level discussion of the issues of termination and productivity in LP.

The rest of the paper is structured as follows. In Section 2, we explain the role of laziness in semantics and implementation of CoALP; in Section 3, we discuss the effect of guarded corecursion. Section 4 is devoted to discussion of our current work on soundness properties for corecursive logic programming.

2 Lazy Corecursion in Logic Programming

CoALP uses the standard syntax of Horn-clause logic programming (Lloyd 1987), but offers a new derivation algorithm in place of the SLD-resolution. One of the main distinguishing features of CoALP is its laziness. To our knowledge, it is the first lazy dialect of logic programming. The issue is best explained using the following example:

Example 2.1 *Given the program* ***BitList*** *and the query* `bitlist([X|Y])`*, the standard algorithm of SLD-resolution (Lloyd 1987) will eagerly attempt to find a derivation, e.g.:*

$$\texttt{bitlist([X|Y])} \longrightarrow \texttt{bit(X)}, \texttt{bitlist(Y)} \xrightarrow{X=0} \texttt{bitlist(Y)} \xrightarrow{Y=[]} \Box$$

For the program ***BitStream*** *this will give rise to an infinite SLD-derivation:*

$$\texttt{stream([X|Y])} \longrightarrow \texttt{bit(X)}, \texttt{stream(Y)} \xrightarrow{X=0} \texttt{stream(Y)} \xrightarrow{Y=[X1|Y1]} \texttt{stream([X1|Y1])}\ldots$$

In the above setting, there is no natural place for laziness, as ultimately the strong side of the procedure is a fully automated proof search. Fibrational coalgebraic operational semantics of LP

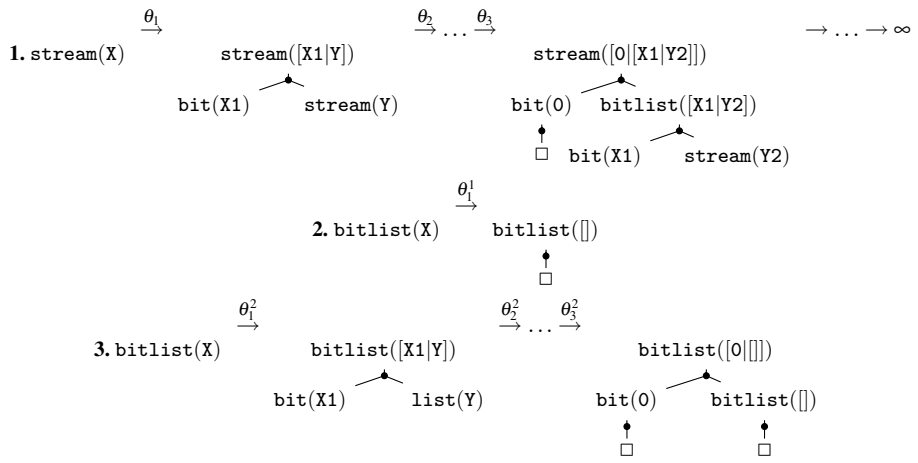


Fig. 2. ***1:*** *Three coinductive trees representing a coinductive derivation for the goal* $G = \texttt{stream(X)}$ *and the program* ***BitStream****, with* $\theta_1 = \texttt{X}/[\texttt{X1}|\texttt{Y}]$, $\theta_2 = \texttt{X1}/0$ *and* $\theta_3 = \texttt{Y}/[\texttt{X1}|\texttt{Y2}]$. ***2-3:*** *Coinductive trees representing two coinductive derivations for the goal* $G = \texttt{bitlist(X)}$ *and the program* ***BitList****, with* $\theta_1^1 = \texttt{X}/[]$, $\theta_1^2 = \texttt{X}/[\texttt{X1}|\texttt{Y}]$, $\theta_2^2 = \texttt{X1}/0$, *and* $\theta_3^2 = \texttt{Y}/[]$.

presented in (Komendantskaya et al. 2014a) inspired us to introduce a structure which we call *coinductive tree*; we use it as a measure for the size of lazy steps in derivations:

Definition 2.1 *Let P be a logic program and* $G = \leftarrow A$ *be an atomic goal. The* coinductive tree *for A is a (possibly infinite) tree T satisfying the following properties.*

- *A is the root of T.*
- *Each node in T is either an and-node (labelled by an atom) or an or-node (labelled by "*$\bullet$*"). The root of the tree is an and-node.*
- *For every and-node* A' *occurring in T, if there exist exactly* $m > 0$ *distinct clauses* $C_1, \ldots, C_m$ *in P (a clause* C_i *has the form* $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, *for some* n_i*), such that* $A' = B_1\theta_1 = \ldots = B_m\theta_m$, *for mgus* $\theta_1, \ldots, \theta_m$, *then* A' *has exactly m children given by or-nodes, such that, for every* $i \in \{1, \ldots, m\}$, *the ith or-node has* n_i *children given by and-nodes* $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.
  In such a case, we say C_i *and* θ_i *are* internal resolvents *of* A'.

Coinductive trees resemble *an-or parallel trees* (Gupta and Costa 1994), see (Komendantskaya et al. 2014a; Komendantskaya et al. 2014c) for discussion of their parallel features. However, they restrict mgus used to form nodes to term-matching. Given two first order atomic formulae A and B, an mgu θ for A and B is called a term-matcher if $A = B\theta$. In Definition 2.1, note the condition $A' = B_1\theta_1 = \ldots = B_m\theta_m$.

Example 2.2 *Figure 2 shows coinductive trees for various goals in* ***BitStream*** *and* ***BitList****; compare with SLD-derivations in Example 2.1. Note that each of those trees is finite by construction of Definition 2.1; and we do not impose any additional conditions. The size of coinductive trees varies, but it is automatically determined by construction of the definition.*

We now define derivations between coinductive trees – a lazy analogue of SLD-derivations.

Definition 2.2 *Let* $G = \langle A, T\rangle$ *be a goal given by an atom* $\leftarrow A$ *and the coinductive tree T*

induced by A, and let C be a clause $H \leftarrow B_1, \ldots, B_n$. *Then, the goal* G' *is* coinductively derived *from G and C using the mgu* θ *if the following conditions hold:*

★ $Q(\bar{t})$ *is a node in* T.
★★ θ *is an* mgu *of* $Q(\bar{t})$ *and* H.
★★★ G' *is given by the (coinductive) tree* $T\theta$ *with the root* $A\theta$.

Definition 2.3 *A* coinductive derivation *of* $P \cup \{G\}$ *consists of a sequence of goals* $G = G_0, G_1, \ldots$ *and a sequence* $\theta_1, \theta_2, \ldots$ *of mgus such that each* G_{i+1} *is derived from a node* $A \in T_i$ *(where* T_i *is the coinductive tree of* G_i*) and a clause C using a non-empty substitution* θ_{i+1}. *In this case,* $\langle A, C, \theta_{i+1} \rangle$ *is called a* resolvent.

Coinductive derivations resemble *tree rewriting*. They produce the "lazy" corecursive effect: derivations are given by potentially infinite number of steps, where each individual step is executed in finite time.

Example 2.3 *Figure 2 shows three possible coinductive derivations for* ***BitStream*** *and* ***BitList****. Note that two derivations for* ***BitList*** *terminate (with* □ *closing all branches). Note also, that this time, due to the and-or parallel nature of coinductive trees, changing the order of atoms or clauses in the program* ***BitList*** *will not change the result.*

For terminating coinductive derivations, we require at least one or-subtree of the coinductive tree to be closed (with □ leaves). We also say in such cases that the coinductive tree contains a *success subtree*. The last coinductive trees in the second and third derivation of Figure 2 are themselves success subtrees.

Due to its and-or parallel properties (Komendantskaya et al. 2014c), CoALP is more robust than eager sequential SLD-resolution when it comes to reflecting program's operational meaning; and mere change in the clause order would not place a terminating recursive function into a non-terminating class, cf. Figure 1. Yet more importantly, this new coinductive derivation procedure allows us to characterise productive and non-productive programs with better precision. In Introduction, we have seen that according to eager interpreter of CoLP, both programs **BadStream** and **TakeFirstN** are non-terminating; despite of one being productive, and another – non-productive. Next example shows that under lazy execution, productive programs with irregular pattern of resolvents can be handled more naturally.

Example 2.4 *Figure 3 shows the first steps in the derivation for the program* ***TakeFirstN*** *and the goal* `taken(s(s(0)),X)`. *Unlike CoLP, CoALP is able to compute the second element of the stream in finite time.*

There will be classes of non-terminating and non-productive programs for which coinductive trees grow infinite, and lazy derivations fail being "lazy". The program **BadStream** is one such example. We will consider this issue in the next section.

3 Guarding (Co)recursion

The previous section introduced coinductive trees, which allowed us to distinguish terminating and productive programs like **BitStream**, **BitList**, **TakeFirstN** from non-productive programs like **BadStream**, by simply observing that coinductive trees remain finite for the former, while

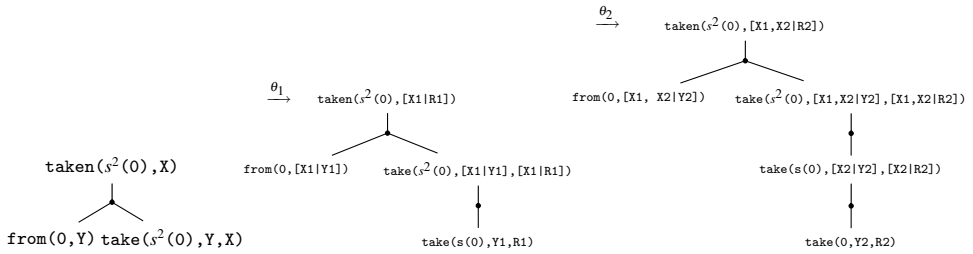


Fig. 3. *First steps of the derivation for the goal* `taken(s²(0),X)` – `s²(0)` *denotes* `s(s(0))` – *and the program* `TakeFirstN`, *with* $\theta_1 = \mathtt{Y}/[\mathtt{X1}|\mathtt{Y1}], \mathtt{X}/[\mathtt{X1}|\mathtt{R1}]$ *and* $\theta_2 = \mathtt{Y1}/[\mathtt{X2}|\mathtt{Y2}], \mathtt{R1}/[\mathtt{X2}|\mathtt{R2}]$. *As* `take` *is an inductive predicate, and* `from` *is coinductive; resolvents for* `take` *nodes are given priority.*

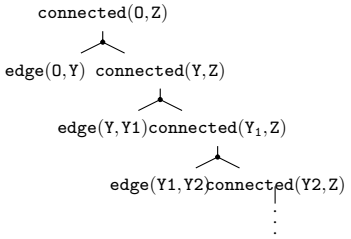


Fig. 4. *The infinite coinductive tree for the program GC from from Example 3.1, for the database* `edge(0,1)` ←.

growing infinite for the latter. It was especially significant that this new approach was, unlike Prolog, robust to permutations of clauses and atoms, and, unlike CoLP, was working with productive irregular streams. Curiously, the following logic program fails to produce finite coinductive trees:

Example 3.1 (GC) *Let* `GC` *(for graph connectivity) denote the logic program*

$$1.\ \mathtt{connected(X,X)} \leftarrow$$
$$2.\ \mathtt{connected(X,Y)} \leftarrow \mathtt{edge(X,Z)}, \mathtt{connected(Z,Y)}$$

It would be used with database of graph edges, like `edge(0,1)` ←.

The program gives rise to infinite coinductive trees, see Figure 4. It would terminate in LP, but, similarly to our discussion of ***BitList****, would lose the termination property if the order of clauses (1) and (2) changes, or if the order of the atoms in clause (2) changes.*

The reason behind infinity of coinductive trees for the above program is the absence of function symbols – "constructors" in the clause heads. The lazy nature of coinductive trees was in part due to the term-matching used to compute them. Term-matching loses its restrictive power in the absence of constructors. A very similar procedure of guarding recursion by constructors of types is used in e.g. Coq or Agda. This observation would suggest an easy way to fix the **GC** example, by introducing reducing dummy-constructors:

Example 3.2 (Guarded GC)

$$1.\ \mathtt{connected(X,cons(Y,Z))} \leftarrow \mathtt{edge(X,Y)}, \mathtt{connected(Y,Z)}$$
$$2.\ \mathtt{connected(X,nil)} \leftarrow$$

Considerations of this kind led us to believe that our lazy (co)recursive approach opens the way for a compile-time termination and productivity checks akin to respective checks in Coq or Agda (Coquand 1994; Abel et al. 2013). The programmer would be warned of non-terminating cases and asked to find a guarded reformulation for his functions. In Coq and Agda, different checks are imposed on recursive functions ("structural recursion" condition) and corecursive functions ("guardedness" checks). In logic programming terms, where types or predicate annotations are unavailable, we can formulate a uniform productivity property for recursive and corecursive programs, as follows:

Definition 3.1 *Let P be a logic program, P is* productive *if for any goal G, the coinductive tree for $P \cup \{G\}$ has a finite size.*

The above is a semantic property; syntactically, we need to introduce guardedness checks to ensure productivity. The intuitive idea is to ensure that every coinductive program behaves like `BitStream`: `BitStream` is guarded by the coinductive function symbol (or "guard") `scons` (denoted by `[.|.]`); and hence all coinductive trees for it are finite, see Figure 2. On the contrary, **Comember** lacks a guarding constructor.

Example 3.3 (Comember) *The predicate `comember` is true if and only if the element `X` occurs an infinite number of times in the stream `S`.*

```
1. drop(H,[H|T],T)   ←
2. drop(H,[H1|T],T1) ←  drop(H,T,T1)
3. comember(X,S)     ←  drop(X,S,S1), comember(X,S1)
```

`Comember` is un-productive for e.g. the coinductive tree arising from the query `comember(X,S)` contains a chain of alternating $\bullet$'s and atoms `comember(X,S1)`, `comember(X,S2)`, etcetera, yielding an infinite coinductive tree.

We will give a high-level formulation of guardedness checks here, for more technical discussion, see (Komendantskaya et al. 2014a).

Guardedness Check 1 (GC1): If the same predicate Q occurs in the head and in the body of a clause, then there must exist a function symbol f occurring among the arguments of Q; such that the number of its occurrences is reduced from head to body.

Example 3.4 (Guarded Comember) *We propose the following guarded definition of comember, thereby simplifying it and reducing an extra argument to `drop`.*

```
1. drop(H,[H|T])      ←
2. drop(X,[H|T])      ←  drop(X,T)
3. gcomember(X,[H|T]) ←  drop(X,[H|T]), gcomember(X,T)
```

In CoALP, the goal `gcomember(0,nats)` will lazily search for 0 *in an infinite stream of natural numbers, but it never falls into un-productive coinductive trees, as CoLP would do.*

GC1 would be sufficient for some programs, like `BitStream`, where there is only one (co)inductive clause; but not in the general case. LP in general is not compositional, that is, composing two programs may yield a program that has semantic properties not present in the initial programs.

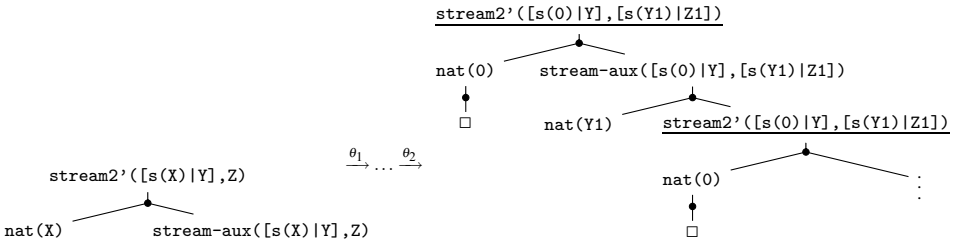


Fig. 5. *Coinductive derivation of* `stream2'([s(X)|Y],Z)` *and the program from Example 3.5 producing an infinite coinductive tree, with* $\theta_1 = $ `X/0` *and* $\theta_2 =$ `Z/[s(Y1)|Z1]`*. The figure also represents one GC-derivation generated during* ***GC3****.* ***GC3*** *detects the un-guarded loop; see the underlined atoms.*

Same rule applies in CoALP: if both P_1 and P_2 are productive programs, their composition is not guaranteed to be a productive program; the next check is imposed to cover the compositional cases.

Guardedness Check 2 (GC2): For every clause head A, construct a coinductive tree with the root A. If there are atoms $Q(\bar{t})$ and $Q(\bar{t'})$ in the coinductive tree such that $Q(\bar{t'})$ is a child of $Q(\bar{t})$, apply **GC1** to the clause $Q(\bar{t}) \leftarrow Q(\bar{t'})$.

GC1–GC2 handle some programs well, but they are still insufficient in the general case. The following program passes the checks **GC1-GC2**, but is not productive in the sense of Definition 3.1, see Figure 5.

Example 3.5 (Un-productive Program that passes GC1-GC2)

```
1.stream2'([s(X)|Y],Z)  ←  nat(X), stream-aux([s(X)|Y],Z)
2.stream-aux(X,[s(Y)|Z])  ←  nat(Y), stream2'(X,[s(Y)|Z])
```

Guardedness Check 3 (GC3): For every clause head A, start a coinductive derivation with the goal A imposing **GC2** condition to every coinductive tree in the derivation, and imposing the following termination conditions:

1. Terminate coinductive derivation if **GC2** fails for at least one tree.
2. Terminate coinductive derivation if all branches are either closed with □ or contain guarded loops only.

Note that the checks **GC1-GC3** we have introduced here are a pre-processing (compile-time) mechanism of CoALP. Once the program passed the guardedness checks, it does coinductive derivations lazily; and does not require any loop-detection procedures at run-time. If a program fails **GC1-GC3**, the programmer will be asked to re-formulate the definitions as we have seen in Examples 3.2 and 3.4. The first implementation of guardedness checks is available at (Komendantskaya et al. 2014b).

We finish this section with Table 1 comparing how SWI-Prolog, CoLP and CoALP handle various recursive and corecursive programs. For CoALP, we also benchmark guardedness checks.

For coinductive programs, CoLP can only handle coinductive programs that contain a regular pattern and fails otherwise (cf. Table 1); on the contrary, CoALP, in its lazy style, works for any program. This is illustrated, for instance, with the programs `TakeFirstN` and `TakeRepeat`. Table 1 shows that CoALP is slower than the CoLP interpreter and SWI-Prolog – note that SWI-Prolog is a fully-tuned mature programming language and the CoLP interpreter runs on top of SWI-Prolog, as opposed to our implementation of CoALP in Go from scratch.

	CoALP		CoLP	SWI-Prolog
TakeFirstN†	Yes	GC time: 0.0002s runtime: lazy execution	No	No
Takerepeat†	Yes	GC time: 0.0009s runtime: lazy execution	Yes (0.0001s)	No
Comember†	Not guarded		Yes$^?$ (0.0001s)	No
GComember†	Yes	GC time: 0.0011s runtime: lazy execution	Yes$^?$ (0.0001s)	No
SumFirstn†	Yes	GC time: 0.0013s runtime: lazy execution	No	No
`FibStream`†	Yes	GC time: 0.0006s runtime: lazy execution	No	No
Infinite Automata†	Yes	GC time: 0.0011s runtime: lazy execution	Yes (0.0001s)	No
Knights	Yes	GC time: 0.225s runtime: 3.002s	Yes (1.13s)	Yes (0.012s)
Finite Automata	Yes	GC time: 0.0011s runtime: 0.0023s	Yes (0.04s)	Yes (0.0005s)
Ackermann	Yes	GC time: 0.001s runtime: 13.23s	Yes (7.692s)	Yes (3.192s)

Table 1. *Execution of different programs in CoALP, CoLP and SWI-Prolog. Examples marked with † involve both inductive and coinductive predicates. In the table, "No" means that the system runs forever without returning an answer, and "Yes$^?$" indicates that the program succeeds if it contains a regular pattern and fails otherwise.*

4 Work-in-Progress: Soundness for Corecursion

There are two main directions for CoALP's development, both related to soundness:

(I) We are in the process of establishing soundness of **GC1-GC3** that is, the property that, *if a program P is guarded by **GC1-GC3**, then it is productive in CoALP*.

Proving this property in the general case is a challenge; and involves pattern analysis on resolvents and also a proof of termination of **GC1-GC3**. Example 3.5 and Figure 5 give a flavour of the complicated cases the guardedness checks need to cover. Note that **GC1-GC3** provide the guarding property only in the CoALP setting, and the same idea of guarding (co)recursion by constructors would fail for standard LP or CoLP, as many examples of this paper show.

(II) Soundness of coinductive derivations needs to be established. This challenge is best illustrated by the following example.

Example 4.1 (Soundness for Comember) *To check the validity of a query in* `Comember` *(Example 3.3) for an arbitrary stream, one needs to satisfy two conditions: 1) finding an element to* `drop` *in a finite time, 2) finding guarantees that this finite computation will be repeated an infinite number of times for the given stream. CoLP would handle such a case for all streams that consist of a regular finite repeating pattern and will not be able to handle cases when the input stream is not regular. CoLP would fail to derive true or falsity of e.g. the query* `comember(0,nats)`*, where* `nats` *is the stream of natural numbers, as CoLP falls into an infinite non-terminating computation and fails to produce any response to the query. CoALP in its current implementation will handle any case of corecursion, including* `comember(0,nats)`*, but in its lazy, and therefore partial, style.*

Similarly, `TakeFirstN` falls into an infinite loop with CoLP, but unfolds lazily with CoALP, see Figure 3. Laziness on its own, however, does not guarantee soundness.

For inductive programs and recursive functions, CoALP yields the same theorems of soundness and completeness as classical LP (Lloyd 1987); cf. (Komendantskaya et al. 2014a). The

only adaptation to the already described coindutive derivation procedure is the requirement that the derivation terminates and gives an answer whenever a *success subtree* is found. Thus, generalisation of standard soundness and completeness for induction in CoALP is not very surprising.

Soundness of CoALP for coinductive programs is conceptually more interesting: it has to include a number of guarantees that need to be checked at compile-time and run-time, that is:

1. Identification of the guarding pattern coming from sound guardedness checks;
2. Guarantee that the guarding pattern will be produced in a finite number of derivation steps;
3. Guarantee that the guarding pattern will be re-produced an infinite number of times.

Item 3. in particular may allow for a few different solutions. In its basic form, it can be a repeated regular pattern, as it is done in CoLP. In a more sophisticated form, it can cover irregular patterns, as long as more involved guarantees of infinite execution are be provided, cf. Example 1.4 and Figure 3.

To conclude, we have described a new method to analyse termination and productivity of logic programs by means of lazy guarded corecursion in CoALP, as outlined in Figure 1. We advocated a new style of programming in LP, where the programmer is in charge of providing termination or productivity measures for (co)recursive programs at compile-time, as it is done in some other declarative languages with recursion and corecursion. Finally, we outlined the main directions towards establishing soundness results for CoALP outputs.

References

ABEL, A. ET AL. 2013. Copatterns: programming infinite structures by observations. In *POPL'13*. ACM SIGPLAN Notices, vol. 48. 27–38.

BERTOT, Y. AND KOMENDANTSKAYA, E. 2008. Inductive and coinductive components of corecursive functions in Coq. *ENTSC 203,* 5, 25–47.

COQUAND, T. 1994. Infinite objects in type theory. In *TYPES'93*. LNCS, vol. 806. 62–78.

DE SCHREYE, D. AND DECORTE, S. 1994. Termination of logic programs: the never-ending story. *J. of Logic Programming 19–20, Supplement 1*, 199–260. Special Issue: Ten Years of Logic Programming.

GUPTA, G. ET AL. 2007. Coinductive logic programming and its applications. In *ICLP'07*. LNCS, vol. 4670. 27–44. Interpreter Available at http://www.utdallas.edu/~gupta/meta.html.

GUPTA, G. AND COSTA, V. 1994. Optimal implementation of and-or parallel prolog. In *PARLE'92*. 71–92.

KOMENDANTSKAYA, E. ET AL. 2014a. Coalgebraic logic programming: from semantics to implementation. *J. Logic and Computation*.

KOMENDANTSKAYA, E. ET AL. 2014b. CoALP webpage: software and supporting documentation. http://staff.computing.dundee.ac.uk/katya/CoALP/.

KOMENDANTSKAYA, E. ET AL. 2014c. Exploiting parallelism in coalgebraic logic programming. *ENTCS 303*, 121–148.

KOMENDANTSKAYA, E. AND POWER, J. 2011. Coalgebraic derivations in logic programming. In *CSL'11*. LIPIcs. Schloss Dagstuhl, 352–366.

LLOYD, J. 1987. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag.

SIMON, L. ET AL. 2007. Co-logic programming: Extending logic programming with coinduction. In *ICALP'07*. LNCS, vol. 4596. 472–483.

Logic and Constraint Logic Programming for Distributed Constraint Optimization

Tiep Le, Enrico Pontelli, Tran Cao Son, William Yeoh
Department of Computer Science, New Mexico State University
(*e-mail:* {tile,epontell,tson,wyeoh}@cs.nmsu.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The field of *Distributed Constraint Optimization Problems (DCOPs)* has gained momentum, thanks to its suitability in capturing complex problems (e.g., multi-agent coordination and resource allocation problems) that are naturally distributed and cannot be realistically addressed in a centralized manner. The state of the art in solving DCOPs relies on the use of ad-hoc infrastructures and ad-hoc constraint solving procedures. This paper investigates an infrastructure for solving DCOPs that is completely built on logic programming technologies. In particular, the paper explores the use of a general constraint solver (a *constraint logic programming system* in this context) to handle the agent-level constraint solving. The preliminary experiments show that logic programming provides benefits over a state-of-the-art DCOP system, in terms of performance and scalability, opening the doors to the use of more advanced technology (e.g., search strategies and complex constraints) for solving DCOPs.

KEYWORDS: DCOP, CLP, Implementation

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) are descriptions of constraint optimization problems where variables and constraints are distributed among a group of agents, and where each agent can only interact with agents that share a common constraint (Modi et al. 2005; Petcu and Faltings 2005; Yeoh and Yokoo 2012). Researchers have realized the importance of DCOPs, as they naturally capture real-world scenarios, where a collective tries to achieve optimal decisions, but without the ability to collect all information about resources and limitations into a central solver. For example, DCOPs have been successfully used to model domains like resource management and scheduling (Maheswaran et al. 2004; Farinelli et al. 2008; Léauté and Faltings 2011), sensor networks (Fitzpatrick and Meertens 2003; Jain and Ranade 2009; Zhang et al. 2005; Zivan et al. 2009; Stranders et al. 2009), and smart grids (Kumar et al. 2009; Gupta et al. 2013).

The DCOP field has grown at a fast pace in recent years. Several popular implementations of DCOP solvers have been created (Léauté et al. 2009; Sultanik et al. 2007; Ezzahir et al. 2007). The majority of the existing DCOP algorithms can be placed in one of three classes. *Search-based* algorithms perform a distributed search over the space of solutions to determine the optimum (Modi et al. 2005; Gershman et al. 2009; Zhang et al. 2005; Yeoh et al. 2010). *Inference-based* algorithms, on the other hand, make use of techniques

from dynamic programming to propagate aggregate information among agents (Petcu and Faltings 2005; Farinelli et al. 2008; Vinyals et al. 2009); these two classes provide a different balance between memory requirements and number of messages exchanged. Another class of methods includes approximated algorithms that rely on *sampling* (Ottens et al. 2012; Nguyen et al. 2013) applied to the overall search space.

The driving objective of the investigation discussed in this paper is to understand the role that logic programming can play in solving DCOPs. In particular, existing popular DCOP solvers (e.g., the frequently used FRODO platform (Léauté et al. 2009)) are ad-hoc systems, with a relatively closed structure, and making use of ad-hoc dedicated solvers for constraint handling within each agent. Thus, a question we intend to address with this paper is whether the use of a general infrastructure for constraint solving within each agent of a DCOP would bring benefits compared to the ad-hoc solutions of the existing implementations. We propose a general infrastructure (based on distributed dynamic programming) for the communication among agents, guaranteeing completeness of the system. The platform enables the use of a generic logic programming solver (e.g., a Constraint Logic Programming system) to handle the local constraints within each agent; the generality of the platform will also allow the use of distinct logic programming paradigms within each agent (e.g., Answer Set Programming).

The paper discusses the overall logic programming infrastructure, along with the details of the modeling of each agent using constraint logic programming. We provide some preliminary experimental results, validating the viability and effectiveness of this research direction for DCOPs. The results also highlight the potential offered by logic programming to provide an implicit representation of hard constraints in DCOPs, enabling a more effective pruning of the search space and reducing memory requirements.

2 Background

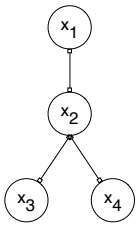
In this section, we provide a brief review of basic concepts from DCOPs. We assume that the readers have familiarity with logic and constraint logic programming; in particular, we will refer to the syntax of the clpfd library of SICStus Prolog (Carlsson et al. 2012).

2.1 Distributed Constraint Optimization Problems (DCOPs)

A *DCOP* (Modi et al. 2005; Petcu and Faltings 2005; Yeoh and Yokoo 2012) is described by a tuple $\mathscr{P} = (X, D, F, A, \alpha)$ where: **(i)** $X = \{x_1, \ldots, x_n\}$ is a set of *variables*; **(ii)** $D = \{D_{x_1}, \ldots, D_{x_n}\}$ is a set of finite *domains*, where each D_{x_i} is the domain of variable x_i; **(iii)** $F = \{f_1, \ldots, f_m\}$ is a set of *utility functions* (a.k.a. *constraints*), where each $f_j : D_{x_{j1}} \times D_{x_{j2}} \times \ldots \times D_{x_{jk}} \mapsto \mathbb{N} \cup \{-\infty, 0\}$ specifies the utility of each combination of values of variables in its *scope* $scp(f_j) = \{x_{j1}, \ldots, x_{jk}\} \subseteq X$; **(iv)** $A = \{a_1, \ldots, a_p\}$ is a set of *agents*; and **(v)** $\alpha : X \to A$ maps each variable to an agent.

We assume the domains D_x to be finite intervals of integer numbers. A *substitution* θ of a DCOP $\mathscr{P}$ is a value assignment for the variables in X s.t. $\theta(x) \in D_x$ for each $x \in X$. Its utility is $ut_{\mathscr{P}}(\theta) = \sum_{i=1}^{m} f_i(scp(f_i)\theta)$, i.e., the evaluation of all utility functions on it. A solution θ is a substitution such that $ut_{\mathscr{P}}(\theta)$ is maximal, i.e., there is no other substitution σ such that $ut_{\mathscr{P}}(\theta) < ut_{\mathscr{P}}(\sigma)$. $Soln_{\mathscr{P}}$ denotes the set of solutions of $\mathscr{P}$.

Each DCOP $\mathscr{P}$ is associated with a *constraint graph*, denoted with $G_{\mathscr{P}} = (X, E_{\mathscr{P}})$, where



x_i	x_j	*utility*
0	0	5
0	1	8
1	0	20
1	1	2

x_2	x_1	*utility*
0	0	5+20+20=45
0	1	8+20+20=48
1	0	20+8+8=36
1	1	2+8+8=18

Fig. 1. DCOP Example

X is a set of nodes which correspond to DCOP variables, and $E_{\mathscr{P}}$ is a set of edges which connect pairs of variables in the scope of the same utility function.

2.2 Distributed Pseudo-tree Optimization Procedure (DPOP)

DPOP (Petcu and Faltings 2005) is one of the most popular complete algorithms for the distribution resolution of DCOPs; as discussed in several works, it has several nice properties (e.g., it requires only a linear number of messages), and it has been used as the foundations for several more advanced algorithms (Petcu et al. 2006; Petcu and Faltings 2007; Petcu et al. 2007).

The premise of DPOP is the generation of a *DFS-Pseudo-tree*—composed of a subgraph of the constraint graph of a DCOP. The pseudo-tree has a node for each agent in the DCOP; edges meet the following conditions: **(a)** If an edge (a_1, a_2) is present in the pseudo-tree, then there are two variables x_1, x_2 s.t. $\alpha(x_1) = a_1$, $\alpha(x_2) = a_2$, and $(x_1, x_2) \in E_{\mathscr{P}}$; **(b)** The set of edges describes a rooted tree; **(c)** For each pair of variables x_i, x_j s.t. $\alpha(x_i) \neq \alpha(x_j)$ and $(x_i, x_j) \in E_{\mathscr{P}}$, we have that $\alpha(x_i)$ and $\alpha(x_j)$ appear in the same branch of the pseudo-tree. $\alpha(x_i)$ and $\alpha(x_j)$ are also called the pseudo-parent and pseudo-child of each other.

Algorithms exist (e.g., (Hamadi et al. 1998)) to support the distributed construction of a DFS-Pseudo-tree. Given a DCOP $\mathscr{P}$, we will refer to a DFS-Pseudo-tree of $\mathscr{P}$ by $\mathscr{T}_{\mathscr{P}} = (A, ET_{\mathscr{P}})$. We will also denote with $a \mapsto_{\mathscr{P}} b$ if there exists a sequence of edges $(a_1, a_2), (a_2, a_3), \ldots, (a_{r-1}, a_r)$ in $ET_{\mathscr{P}}$ such that $a = a_1$ and $b = a_r$; in this case, we say that b is reachable from a in $\mathscr{T}_{\mathscr{P}}$. Given an agent a, we denote with $\mathscr{S}_{\mathscr{P}}(a)$ the set of agents in $\mathscr{T}_{\mathscr{P}}$ in the subtree rooted at a (including a itself).

The DPOP algorithm operates in two phases:

- **UTIL Propagation:** During this phase, messages flow bottom-up in the tree, from the leaves towards the root. Given a node N, the UTIL message sent by N summarizes the maximum utility achievable within the subtree rooted at N for each combination of values of variables belonging to the separator set (Dechter 2003) of N. The agent does so by summing the utilities in the UTIL messages received from its children agents, and then projecting out its own variables by optimizing over them.
- **VALUE Propagation:** During this phase, messages flow top-down in the tree. Node N determines an assignment to its own variables that produces the maximum utility based on the assignments given by the ancestor nodes; this assignment is then propagated as VALUE messages to the children.

Let us consider a DCOP with $X = \{x_1, x_2, x_3, x_4\}$, each with $D_{x_i} = \{0, 1\}$ and with binary constraints described by the graph (and pseudo-tree) and utility table (assuming $i > j$) in Fig. 1 (left and middle). For simplicity, we assume a single variable per agent.

Node x_2 will receive two UTIL messages from its children; for example, the message from x_3 will indicate that the best utilities are 20 (for $x_2 = 0$) and 8 (for $x_2 = 1$). In turn, x_2 will compose the UTIL messages with its own constraint, to generate a new utility table, shown in Fig. 1 (right). This will lead to a UTIL meassage sent to x_1 indicating utilities of 45 for $x_1 = 0$ and 48 for $x_1 = 1$. In the VALUE phase, node x_1 will generate an assignment of $x_1 = 1$, which will be sent as a VALUE message to x_2; in turn, x_2 will trigger the assignment $x_2 = 0$ as a VALUE message to its children.

3 Logic-Programing-based DPOP (LP-DPOP)

In this section, we illustrate the *LP-DPOP* framework, designed to map DCOPs into logic programs that can be solved in a distributed manner using the DPOP algorithm.

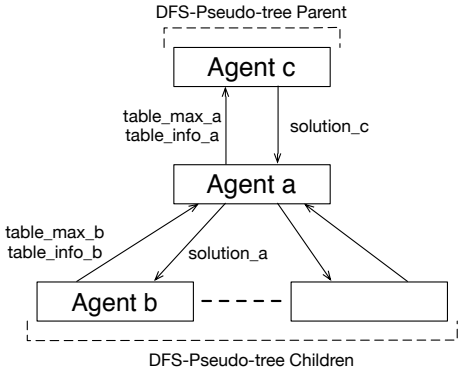


Fig. 2. Overall Communication Needs

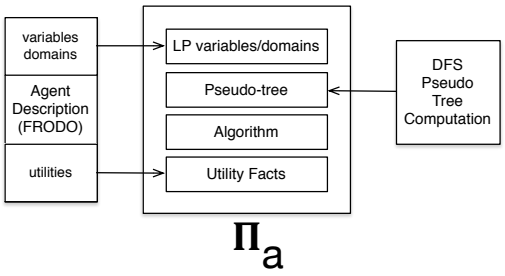


Fig. 3. Components of an Agent in LP-DPOP^{facts}

3.1 Overall Structure

The overall structure of LP-DPOP is summarized in Fig. 2. Intuitively, each agent a of a DCOP $\mathscr{P}$ is mapped to a logic program Π_a. Agents exchange information according to the communication protocol of DPOP. These exchanges are represented by collections of facts that are communicated between agents. In particular,

- UTIL messages from agent b to agent a are encoded as facts $table\_max\_b(L)$, where L is a list of $[u, v_1, \ldots, v_k]$. Each one is a row of the UTIL message, where u is the maximum utility for the combination of values $v_1, \ldots, v_k$. It is also necessary to transmit an additional message describing the variables being communicated: $table\_info\_b([v(x_1, low_1, high_1), \ldots, v(x_k, low_k, high_k)])$. This message identifies the names of the variables being communicated and their respective domains. It should be mentioned that the UTIL message from b to a can contain variables belonging to some ancestors of a.
- VALUE messages from agent c to agent a are encoded as facts $solution\_c(Var, Val)$, where Var is the name of a variable and Val is the value assigned to it.

3.2 LP-DPOP Execution Model

Computing DFS-PseudoTree: One can use existing off-the-shelf distributed algorithms to construct pseudo-trees. A commonly used algorithm is the distributed DFS protocol (Hamadi et al. 1998), that creates a DFS tree with the max-degree heuristic as the

variable-ordering heuristic. The max-degree heuristic favors variables with larger numbers of constraints to be higher up in the pseudo-tree.

Solving a DCOP: The actual agent a is implemented by a logic program Π_a. In the context of this paper, the logic program is a CLP program, whose entry point is a predicate called `agent`:

```
agent :- agent(ID),
         (\+is_leaf(ID) -> get_utils; true),
         (\+is_root(ID) -> compute_utils, send_utils, get_value; true),
         (\+is_leaf(ID) -> compute_value, send_value; compute_value).
```

The logic program implements the `compute_utils` and the `compute_value` predicates. They are described in the next section.

3.3 Modeling LP-DPOP as CLP

In this section, we illustrate the structure of the logic program that encodes each individual agent. We propose two alternative models. The first one follows the model illustrated in Fig 3: the input DCOP is described using the standardized format introduced by the FRODO DCOP platform (Léauté et al. 2009).

In the first model, referred to as LP-DPOP$^{\text{facts}}$, the FRODO model is literally translated into collections of logic programming facts. The second model, referred to as LP-DPOP$^{\text{rules}}$, follows the more "realistic" option of capturing the hard constraints present in the DCOP model explicitly as logical constraints, instead of forcing their mapping to explicit utility tables (as automatically done by FRODO).

3.3.1 LP-DPOP$^{\text{facts}}$

The logic program Π_a modeling an agent is composed of four primary modules, as illustrated in Fig. 3:

1. *Agent, Variables and Domains:* the core components of the agent variables and domains are encoded in Π_a by facts of the form:
   - A single fact $agent(a)$ describing the identity of the agent;
   - For each variable x_i with domain D_{x_i}, such that $\alpha(x_i) = a$ or $\alpha(x_j) = a$ for some variable x_j such that $(x_i, x_j) \in E_{\mathscr{P}}$: a fact $variable(x_i, min(D_{x_i}), max(D_{x_i}))$ and a fact $owner(\alpha(x_i), x_i)$.
2. *DFS-Pseudo-Tree:* the local position of a in the DFS-Pseudo-tree is described by:
   - facts of the form $child(b)$ where b is agent s.t. $(a, b) \in ET_{\mathscr{P}}$;
   - a fact $parent(c)$ where c is the (only) agent s.t. $(c, a) \in ET_{\mathscr{P}}$; and
   - a fact $ancestor(c)$ where c is any non-parent ancestor of a in the pseudo-tree, i.e., any agent c s.t. $(c, a) \notin ET_{\mathscr{P}}$ and $c \mapsto_{\mathscr{P}} a$.
3. *Utilities/Constraints:* the constraints are obtained as direct translation of the utility tables in the FRODO representation: for each constraint f_j, there is a fact of the form $constraint\_f_j(L)$, where L is a list containing lists $[f_j(v_1, \ldots v_r), v_1, \ldots, v_r]$ for each assignement $\{x_1/v_1, \ldots, x_r/v_r\}$ to the variables of $scp(f_j) = \{x_1, \ldots, x_r\}$ where $f_j(v_1, \ldots, v_r) \neq -\infty$. Each constraint is further described by the facts: *(i)* a fact $constraint(f_j)$, identifying the name of each constraint, *(ii)* a fact $scope(f_j, x_i)$ for each $x_i \in scp(f_j)$, identifying the variables contributing to the scope of the

constraint, and *(iii)* facts of the form *constraint_agent*(f_j, a_r), identifying agents that has variables in the scope of the constraint.

4. *Resolution Engine:* a collection of rules that implement the `compute_utils` and `compute_value`—these are described below.

The core of the computation of the UTIL message is implemented within the `compute_utils` predicate. Intuitively, the construction of the UTIL message is mapped to a CLP problem. Its construction and resolution can be summarized as follows:

```
... define_variables(L,Low,High),
    define_constraints(L,Util),
    generate_utils(Low,High,UTILITIES), ...
```

The steps can be summarized as follows:

- The `define_variables` predicate is used to collect the variables that belong to the agent and its ancestors (returned in the list `Low` and `High`, respectively), and for each variable generates a corresponding CLP domain variable. The collecting variables phase is based on the *variable* facts (describing all variables owned by the agent) and the variables indicated in the *table_info_b* messages received from the children; these may contain variables that belong to pseudo-parents in the tree and unknown to the agent *a*. To enable interpretation of the CLP variables, two facts `low_vars`(Low) and `high_vars`(High) are created in this phase. In the latter phase, for each X_i in the collection of variables collected from the former phase calls X_i `in` $\ell..m$ where ℓ and m are the minimum and maximum value of X_i's domain which are either known to the agent or given in received the *table_info_b* message.
- The predicate `define_constraints` creates CLP constraints capturing the utilities the agent has to deal with—these include the utilities described by each *table_max_b* message received from a child b and the utilities f_j of the agent a s.t. $scp(f_j)$ does not contain any variables in $\bigcup_{(a,b)\in ET_{\mathscr{P}}}\{x \in X \mid \alpha(x) = b\}$. For each utility f_i of these utilities (described by a list of lists), the predicate `define_constraints` introduces a constraint of the form:

  ```
  table([[Ui, X1, .., Xr]] , L, [order(id3), consistency(domain)])
  ```

  where:
  - $X1,\ldots,Xr$ are the CLP variables which were created by `define_variables` and correspond to the scope of this utility.
  - L is the list of lists given in *constraint_*$f_i(L)$;
  - Ui is a new variable introduced for each utility f_i.

  The final step of the `define_constraints` is to introduce the additional CLP constraint $Util\#= U_1 + U_2 + \ldots + U_s$ where U_i are the variables introduced in the `table` constraints and $Util$ is a brand new variable.
- The `generate_utils` predicate has the following general structure:

```
generate_utils(Lo, Hi, UTILITIES) :-
 findall([Util|Hi], (labeling([],Hi),find_max_util(Lo,Hi,Util)),UTILITIES).
find_max_util(Lo, Hi, Util) :-
 maximize(labeling([ff],Lo), Util), assert(agent_a_table_max(Lo,Hi)).
```

The core of the computation of the VALUE message takes advantage of the fact that the combination of variables producing the maximum values are asserted as `agent_a_table_max` facts during the UTILs phase, enabling a simple lookup to compute the solution. This can be summarized as follows:

```
... high_vars(H),
```

```
findall(Value,(member(Name,H),solution(Name,Value)), Sols),
agent_a_table_max(Low,Sols),
low_vars(Lo), length(Lo,Len), I in 1..Len,
findall(solution(Name,Value),
        (indomain(I), nth1(I,Lo,Name), nth1(I,Low,Value)), VALUES), ...
```

3.3.2 LP-DPOP$^{\text{rules}}$

An alternative encoding takes advantage of the fact that the utilities provided in the utility table of a FRODO encoding are the results of enumerating the solutions of *hard constraints*. A hard constraint captures a relation $f_j(x_1,\ldots,x_r) \oplus u$ where $\oplus$ is a relational operator, and u is an integer. This is typically captured in FRODO as a table, containing all tuples of values from $D_{x_1} \times \cdots \times D_{x_r}$ that satisfy the relation (with a utility value of 0), and the default utility value of $-\infty$ assigned to the remaining tuples.

This utility can be directly captured in CLP, thus avoiding the transition through the creation of an explicit table of solutions:

$$hard\_constraint\_f_j(X_1,\ldots,X_r) :- \widehat{f}_j(X_1,\ldots,X_r)\widehat{\oplus}u$$

where $\widehat{f}_j$ and $\widehat{\oplus}$ are the CLP operators corresponding to f_j and $\oplus$. For example, the smart grid problems used in the experimental section uses hard constraints encoded as

$$\texttt{hard\_constraint\_eq0}(\texttt{X}_{1,2},\texttt{X}_{2,1}) \;:-\; \texttt{X}_{1,2} + \texttt{X}_{2,1} \# = 0$$

The resulting encoding of the UTIL value computation will modify the encoding of LP-DPOP$^{\text{facts}}$ as shown below

```
constraint_f(L),
table([[U,X_1,...,X_r]],L,_)        ⇒   hard_constraint_f(X_1,...,X_r)
```

3.4 Some Implementation Details

The current implementation of LP-DPOP makes use of the Linda (Carriero et al. 1994) infrastructure of SICStus Prolog (Carlsson et al. 2012) to handle all the communication.

Independent agents can be launched on different machines and connect to a Linda server started on a dedicated host. Each agent has a main clause of the type

```
run_agent :- prolog_flag(argv, [Host,Port]),linda_client(Host:Port),agent.
```

The operations of sending a UTIL message from b to the parent a is simply realized by a code fragment of the type

```
send_util(Vars,Utils,To):- out(msg_to(To),[table_info_b(Vars),table_max_b(Utils)]).
```

The corresponding reception of UTIL message by a will use a predicate of the form

```
get_util(Vars,Utils,Me):- in(msg_to(Me), [table_info_b(Vars),table_max_b(Utils)]).
```

The communication of VALUE messages is analogous. `get_value` and `send_value` are simple wrappers of the predicates discussed above.

3.5 Some Theoretical Considerations

The soundness and completeness of the LP-DPOP system is a natural consequence of the soundness and completeness properties of the DPOP algorithm, along with the soundness and completeness of the CLP(FD) solver of SICStus Prolog. Since LP-DPOP emulates the computation and communication operations of DPOP, each Π_a program is a correct and complete implementation of the corresponding agent a.

In the worst case, each agent in LP-DPOP, like DPOP, needs to compute, store, and send a utility for each combination of values of the variables in the separator set of the agent. Therefore, like DPOP, LP-DPOP also suffers from an exponential memory requirement, i.e., the memory requirement per agent is $O(maxDom^w)$, where $maxDom = \text{argmax}_i |D_i|$ and w is the induced width of the pseudo-tree.

4 Experimental Results

We compare two implementations of the LP-DPOP framework, LP-DPOP^{facts} and LP-DPOP^{rules} with a publicly-available implementation of DPOP, which is available on the FRODO framework (Léauté et al. 2009). All experiments are conducted on a Quadcore 3.4GHz machine with 16GB of memory. The runtime of the algorithms are measured using the simulated runtime metric (Sultanik et al. 2007). The timeout is set to 10 minutes. Two domains, randomized graphs and smart grids, were used in the experiments.

Randomized Graphs: A randomized graph generated using the model in (Erdös and Rényi 1959) with the input parameters n (number of nodes) and M (number of binary edges) will be used as the constraint graph of a DCOP instance $\mathscr{P}$.

Each instance $\mathscr{P} = (X, D, F, A, \alpha)$ is generated using five parameters: $|X|$, $|A|$, the domain size d of all variables, the constraint density p_1 (defined as the ratio between the number of binary edges M and the maximum number of binary edges among $|X|$ nodes), and the constraint tightness p_2 (defined as the ratio between the number of infeasible value combinations, that is, their utility equals $-\infty$, and the total number of value combinations).

We conduct experiments, where we vary one parameter in each experiment. The "default" value for each experiment is $|A| = 5$, $|X| = 15$, $d = 6$, $p_1 = 0.6$, and $p_2 = 0.6$. As the utility tables of instances of this domain are randomly generated, the programs for LP-DPOP^{rules} and LP-DPOP^{facts} are very similar. Thus, we only compare FRODO with LP-DPOP^{facts}. Table 1 shows the percentage of instances solved and the average simulated runtime (in ms) for the solved instances; each data point is an average over 50 randomly generated instances. If an algorithm fails to solve more than 85% of instances in a specific configuration, then we consider that it fails to solve problems with that configuration.

The results show that LP-DPOP^{facts} is able to solve more problems and is faster than DPOP when the problem becomes more complex (i.e., increasing $|X|$, d, p_1, or p_2). The reason is that at a specific percentage of hard constraints (i.e., $p_2 = 0.6$), LP-DPOP^{facts} is able to prune a significant portion of the search space. Unlike DPOP, LP-DPOP^{facts} does not need to explicitly represent the rows in the UTIL table that are infeasible, resulting in lower memory usage and runtime needed to search through search space. The size of the search space pruned increases as the complexity of the instance grows, making the difference between the runtimes of LP-DPOP^{facts} and DPOP significant.

Smart Grids: A *customer-driven microgrid* (CDMG), one possible instantiation of the smart grid problem, has recently been shown to subsume several classical power system sub-problems (e.g., load shedding, demand response, restoration) (Jain et al. 2012). In this domain, each agent represents a node with consumption, generation, and transmission preference, and a global cost function. Constraints include the power balance and no power loss principles, the generation and consumption limits, and the capacity of the power line between nodes. The objective is to minimize a global cost function. CDMG optimization problems are well-suited to be modeled with DCOPs due to their distributed

Table 1. *Experimental Results on Random Graphs (%: Solved; Time: Runtime)*

$\|X\|$	DPOP %	DPOP Time	LP-DPOP^{facts} %	LP-DPOP^{facts} Time	d	DPOP %	DPOP Time	LP-DPOP^{facts} %	LP-DPOP^{facts} Time
5	100%	35	100%	30	4	100%	782	100%	74
10	100%	204	100%	264	6	90%	28,363	100%	539
15	86%	39,701	100%	1,008	8	14%	-	98%	22,441
20	0%	-	100%	1,263	10	0%	-	94%	85,017
25	0%	-	100%	723	12	0%	-	60%	-
30	0%	-	100%	255					
35	0%	-	100%	256					

p_1	DPOP %	DPOP Time	LP-DPOP^{facts} %	LP-DPOP^{facts} Time	p_2	DPOP %	DPOP Time	LP-DPOP^{facts} %	LP-DPOP^{facts} Time
0.3	100%	286	100%	2,629	0.4	86%	48,632	92%	155,089
0.4	100%	1,856	100%	2,038	0.5	94%	38,043	100%	23,219
0.5	100%	13,519	100%	938	0.6	90%	31,513	100%	844
0.6	94%	42,010	100%	706	0.7	90%	39,352	100%	84
0.7	56%	-	100%	203	0.8	92%	40,525	100%	61
0.8	20%	-	100%	176	0.9	96%	27,416	100%	60

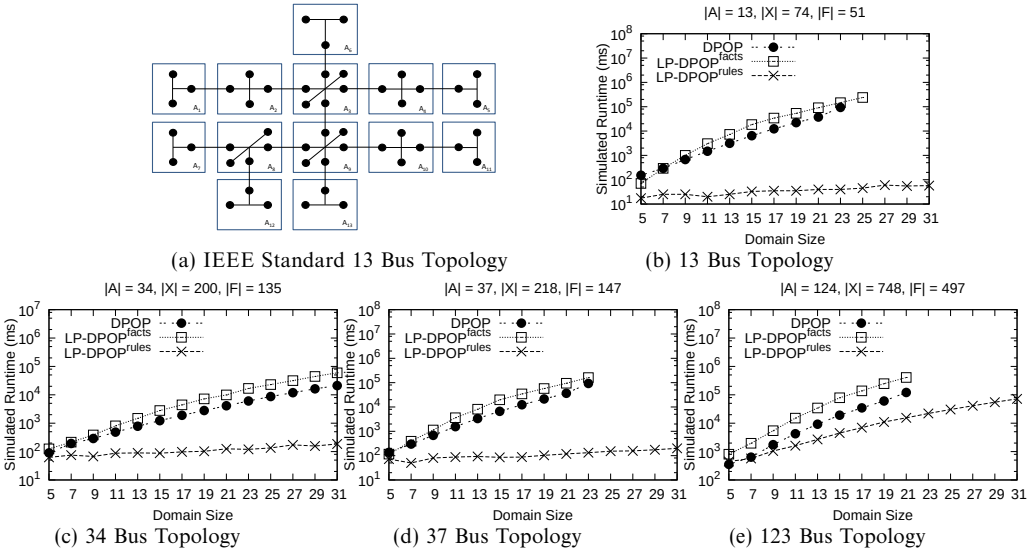
nature. Moreover, as some of the constraints in CDMGs (e.g., the power balance principle) can be described in functional form, they can be exploited by LP-DPOP^{rules}. For this reason, both LP-DPOP^{facts} and LP-DPOP^{rules} were used in this domain.

We conduct experiments on a range of CDMG problem instances generated using the four network topologies following the IEEE standards and varying the domain of the variables.[1] Fig. 4(a) displays the topology of the IEEE 13 Bus network, where rectangles represent nodes/agents, filled circles represent variables, and links between variables represent constraints. The initial configuration of the CDMG and the precise equations used in the generation of the problems can be found in (Jain et al. 2012). The experimental results for the four largest standards, the 13, 34, 37, and 123 Bus Topology,[2] are shown in Fig. 4(b), 4(c), 4(d), and 4(e), respectively. We make the following observations:

- LP-DPOP^{rules} is the best among the three systems both in terms of runtime and scalability in all experiments. LP-DPOP^{rules}'s memory requirement during its execution is significant smaller and increases at a much slower pace than other systems. This indicates that the rules used in expressing the constraints help the constraint solver to more effectively prune the search space resulting in a better performance.
- LP-DPOP^{facts} is slower than DPOP in all experiments in this domain. It is because LP-DPOP^{facts} often needs to backtrack while computing the UTIL message, and each backtracking step requires the look up of several related utility tables—some tables can contain many tuples (e.g., one agent in the 13 Bus problem with domain size of 23 could have $3,543,173$ facts). We believe that this is the source of the weak performance of LP-DPOP^{facts}.

[1] `www.ewh.ieee.org/soc/pes/dsacom/`

[2] In 123 Bus Topology's experiments, a multi-server version of LP-DPOP^{facts} and LP-DPOP^{rules} was used because of the limit on the number of concurrent streams supported by Linda and SICStus. FRODO cannot be run on multiple machines.



(a) IEEE Standard 13 Bus Topology (b) 13 Bus Topology

(c) 34 Bus Topology (d) 37 Bus Topology (e) 123 Bus Topology

Fig. 4. Experiment Results on Smart Grids

5 Conclusion and Future Work

In this paper, we presented a generic infrastructure built on logic programming to address problems in the area of DCOP. The use of a generic CLP solver to implement the individual agents proved to be a winning option, largely outperforming existing DCOP technology in terms of speed and scalability. The paper also makes the preliminary case for a different encoding of DCOPs w.r.t. existing technologies; the ability to explicitly model hard constraints provides agents with additional knowledge that can be used to prune the search space, further enhancing performance.

This is, in many regards, a preliminary effort that will be expanded in several directions. First, we believe that different types of DCOP problems may benefit from different types of local solvers within each agent; we currently explore the use of ASP as an alternative for the encoding the agents. The preliminary results are competitive and superior to those produced by DPOP. Classifying DCOP problems in such a way to enable the automated selection of what type of LP-based solver to use is an open research question to be addressed. The strong results observed in the use of implicit encodings of hard constraints also suggest the need of developing *DCOP description languages* that separate hard and soft constraints and do not require the explicit representation for all constraints.

On the other direction, we view this work as a feasibility study towards the development of distributed LP models (e.g., Distributed ASP). Paradigms like ASP are highly suitable to capture the description of individual agents operating in multi-agent environments; yet, ASP does not inherently provide the capability of handling a distributed ASP computation with properties analogous to those found in DCOP. We believe the models and infrastructure described in this paper could represent the first step in the direction of creating the foundations of DASP and other distributed logic programming models.

References

Carlsson et al., M. 2012. SICStus Prolog User's Manual. Tech. rep., Swedish Institute of Computer Science.

Carriero, N., Gelernter, D., Mattson, T., and Sherman, A. 1994. The Linda Alternative to Message Passing Systems. *Parallel Computing 20,* 4, 633–655.

Dechter, R. 2003. *Constraint processing*. Elsevier Morgan Kaufmann.

Erdös, P. and Rényi, A. 1959. On random graphs I. *Publicationes Mathematicae Debrecen 6*, 290.

Ezzahir, R., Bessiere, C., Belaissaoui, M., and Bouyakhf, E. H. 2007. DisChoco: A platform for distributed constraint programming. In *Proceedings of the Distributed Constraint Reasoning Workshop*. 16–27.

Farinelli, A., Rogers, A., Petcu, A., and Jennings, N. 2008. Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 639–646.

Fitzpatrick, S. and Meertens, L. 2003. Distributed coordination through anarchic optimization. In *Distributed Sensor Networks: A Multiagent Perspective*, V. Lesser, C. Ortiz, and M. Tambe, Eds. Kluwer, 257–295.

Gershman, A., Meisels, A., and Zivan, R. 2009. Asynchronous Forward-Bounding for distributed COPs. *Journal of Artificial Intelligence Research 34*, 61–88.

Gupta, S., Jain, P., Yeoh, W., Ranade, S., and Pontelli, E. 2013. Solving customer-driven micro-grid optimization problems as DCOPs. In *Proceedings of the Distributed Constraint Reasoning Workshop*. 45–59.

Hamadi, Y., Bessière, C., and Quinqueton, J. 1998. Distributed intelligent backtracking. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. 219–223.

Jain, P. and Ranade, S. 2009. Capacity discovery in customer-driven micro-grids. In *Proceedings of the North American Power Symposium (NAPS)*. 1–6.

Jain, P., Ranade, S., Gupta, S., and Pontelli, E. 2012. Optimum operation of a customer-driven microgrid: A comprehensive approach. In *Proceedings of International Conference on Power Electronics, Drives and Energy Systems (PEDES)*. 2012. 1–6.

Kiekintveld, C., Yin, Z., Kumar, A., and Tambe, M. 2010. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 133–140.

Kumar, A., Faltings, B., and Petcu, A. 2009. Distributed constraint optimization with structured resource constraints. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 923–930.

Léauté, T. and Faltings, B. 2011. Coordinating logistics operations with privacy guarantees. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2482–2487.

Léauté, T., Ottens, B., and Szymanek, R. 2009. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of the Distributed Constraint Reasoning Workshop*. 160–164.

Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., and Varakantham, P. 2004. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 310–317.

Modi, P., Shen, W.-M., Tambe, M., and Yokoo, M. 2005. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence 161,* 1–2, 149–180.

Nguyen, D. T., Yeoh, W., and Lau, H. C. 2013. Distributed Gibbs: A memory-bounded sampling-based DCOP algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 167–174.

Ottens, B., Dimitrakakis, C., and Faltings, B. 2012. DUCT: An upper confidence bound approach to distributed constraint optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 528–534.

Petcu, A. and Faltings, B. 2005. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1413–1420.

Petcu, A. and Faltings, B. 2007. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1452–1457.

Petcu, A., Faltings, B., and Mailler, R. 2007. PC-DPOP: A new partial centralization algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 167–172.

Petcu, A., Faltings, B., and Parkes, D. C. 2006. MDPOP: Faithful distributed implementation of efficient social choice problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1397–1404.

Prosser, P. 1996. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence 81,* 1-2, 81–109.

Stranders, R., Farinelli, A., Rogers, A., and Jennings, N. 2009. Decentralised coordination of mobile sensors using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 299–304.

Sultanik, E., Lass, R., and Regli, W. 2007. DCOPolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the Distributed Constraint Reasoning Workshop*.

Vinyals, M., Rodrguez-Aguilar, J. A., and Cerquides, J. 2009. Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 1239–1240.

Yeoh, W., Felner, A., and Koenig, S. 2010. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research 38*, 85–133.

Yeoh, W. and Yokoo, M. 2012. Distributed problem solving. *AI Magazine 33,* 3, 53–65.

Zhang, W., Wang, G., Xing, Z., and Wittenberg, L. 2005. Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence 161,* 1–2, 55–87.

Zivan, R., Yedidsion, H., Okamoto, S., Glinton, R., and Sycara, K. 2014. Distributed constraint optimization for teams of mobile sensing agents. *Autonomous Agents and Multi-Agent Systems*, 1–42.

Adaptive MCMC-Based Inference in Probabilistic Logic Programs

(Extended Abstract)

Arun Nampally, C. R. Ramakrishnan
Department of Computer Science, Stony Brook University, Stony Brook, NY 11794
{anampally,cram}@cs.stonybrook.edu

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

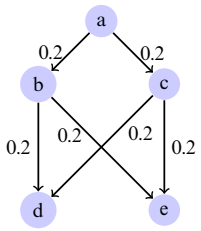
Probabilistic Logic Programming (PLP) languages enable programmers to specify systems that combine logical models with statistical knowledge. The inference problem, to determine the probability of query answers in PLP, is intractable in general. In this paper, we present a technique for approximate inference of conditional probabilities for PLP queries. It is an adaptive Markov Chain Monte Carlo (MCMC) technique, where the proposal distribution is modified as the Markov Chain is explored. In particular, the distribution is modified to increase the likelihood that a proposed sample is consistent with evidence. In our context, each sample is uniquely characterized by the outcomes of a set of random variables. Inspired by reinforcement learning, our technique propagates positive rewards to random variable/outcome pairs used in a consistent sample. The cumulative rewards of each outcome of a random variable is used to derive a new "adapted distribution" for the variable. For a query with "Markovian evaluation structure", we show that the adapted proposal distribution converges to the query's conditional probability distribution. We empirically evaluate the effectiveness of the adaptive sampling method.

1 Introduction

Probabilistic Logic Programming (PLP) covers a class of Statistical Relational Learning frameworks (Getoor and Taskar 2007) aimed at combining logical and statistical reasoning. Examples of languages and systems combining logical and statistical inference include ICL (Poole 1997), SLP (Muggleton et al. 1996), PRISM (Sato and Kameya 1997), LPAD (Vennekens and Verbaeten 2003) and ProbLog (De Raedt et al. 2007). In addition to standard statistical models, these languages allow reasoning over many models where logical and statistical knowledge is intricately combined, and cannot be expressed as standard statistical models.

An example problem with such a model is reachability over finite probabilistic graphs, i.e., graphs in which the presence or absence of edges is determined by a set of independent probabilistic processes. Fig. 1(a) shows a probabilistic graph, where labels on the edges denote the probability with which that edge is present. The logical relationship between reachability of e from a, and the underlying edges in the graph cannot be expressed concisely in traditional probabilistic frameworks, but is easily specified in a probabilistic logic framework. A PRISM program encoding this problem is shown in Fig. 1(b).

As illustrated in Fig. 1(b), PRISM adds *probabilistic facts* of the form $\texttt{msw}(s, i, t)$ where s is a term representing a random process called a *switch*, i a term representing its



(a)

```
% Possible edges
pedge(a,b).
pedge(a,c).
pedge(b,d).
pedge(b,e).
pedge(c,d).
pedge(c,e).
```

```
% Random processes
values(r(_,_), [t,f]).
% Distributions
:- set_sw(r(_,_), [0.2,0.8]).
% Edge:
edge(X, Y) :-
     pedge(X, Y),
     msw(r(X, Y), 1, t).
```

```
% Reachability
reach(X, Y) :-
    edge(X, Y).
reach(X, Y) :-
    edge(X, Z),
    reach(Z, Y).
```

(b)

Fig. 1. Example: (a) Probabilistic Graph; (b) Reachability over probabilistic graphs in PRISM

instance, and t its outcome. Instance i in an `msw` may be omitted when a switch has only one instance. The range of a switch is specified by "`values`" declarations; and its distribution by "`set_sw`" declarations. A *possible world* associates an outcome with each switch instance, and can be seen as a set of `msw` facts external to the program. In each possible world, the PRISM program, together with `msw` facts defining the world is a non-probabilistic program; the distribution over the possible worlds induces a distribution over the models of the PRISM program. Such a declarative *distribution semantics*, originally defined for ICL and PRISM, has been defined for other PLP languages such as LPAD and ProbLog as well.

The Problem. Inference of answer probabilities in PLPs is intractable in general. Of the several powerful sampling-based techniques developed for statistical reasoning, Markov Chain Monte Carlo (MCMC) techniques are especially suited for inference in PLPs, as shown by Cussens (2000) and Moldovan et al. (2013).

PLP queries for evaluating conditional probabilities are called as *conditional queries* and denoted as $\texttt{prob}(q \mid e)$, where q and e are ground atomic goals, called *query* and *evidence*, respectively. A conditional query $\texttt{prob}(q \mid e)$ denotes the suitably normalized distribution of q over all possible worlds where e holds. Existing PLP systems either provide efficient techniques that apply to a restricted class of q and e (e.g., *hindsight* in PRISM) or do not treat evidence specially, leading to poor performance especially when the likelihood of evidence is low. For instance, consider evaluating $\texttt{prob}(\texttt{reach}(\texttt{a},\texttt{d}) \mid \texttt{reach}(\texttt{a},\texttt{e}))$, over the probabilistic graph in Fig. 1(a). Techniques such as the one proposed by Moldovan et al. (2013) will generate a world and reject it if evidence does not hold in the world. Since the probability that e is reachable is about 0.08, a number of generated worlds will be inconsistent with the evidence, and hence unusable for computing the conditional probability.

We address the problem of efficiently estimating conditional probability by developing an *Adaptive* Markov Chain Monte Carlo (AMCMC) technique. Following adaptive MCMC techniques in statistical reasoning, we progressively modify the distribution from which samples are derived, so as to favor those samples that are consistent with evidence. The adaptive sampler reduces the number of generated samples needed to estimate the conditional probability to a given precision.

Approach Overview and Summary of Contributions. Our technical development starts with an MCMC technique where each state of the Markov chain is an *assignment* of values to a set of switch instances. An assignment at a state corresponds to a set of possible worlds such that the truth values of evidence and query are identical in all the

worlds in the set. Transitions are proposed on this chain by resampling one or more switch instances in the state and extending the resulting assignment to another state. A Metropolis-Hastings (Hastings 1970) sampler is used to accept or reject this proposal, yielding the next state in the chain. The procedure is described in Section 3. Since the states of the Markov chain are sets of possible worlds, this procedure is largely *independent of LP evaluation itself*. It can hence be used for approximate inference even in the presence of complex logical inference procedures such as tabling.

We introduce adaptation into this basic MCMC technique (see Section 4). For each switch instance/outcome pair, we maintain its *Q-value*: an estimate of the likelihood that an evaluation of the evidence goal using that switch instance/outcome will succeed. Q-values are computed by propagating rewards through the set of switch instance/outcome pairs used in the evidence goal's evaluation, where the reward depends on the success or failure of the evaluation. The adapted distribution of a switch instance is proportional to its original distribution weighted by the Q-values of each outcome.

Although motivated by conditional queries, the technique we describe is more generally applicable, even to unconditional queries. For a class of queries with "Markovian evaluation structure", the adapted distribution of a random variable coincides with its marginal. For such queries, we obtain an alternative adaptation procedure that can be used to obtain an adaptive independent sampler.

We describe the results of our preliminary experiments to evaluate the MCMC procedure as well as the adaptation procedure in Section 5. The rest of the paper begins with a brief overview of MCMC in Section 2. A more detailed description of related work and concluding remarks appears Section 6.

2 Preliminaries: Markov Chain Monte Carlo Techniques

A sequence of random variables $X^{(i)}, i \geqslant 0$ taking on values $x^{(i)}$ is called a *Markov chain* if $P(x^{(i)}|x^{(i-1)}, x^{(i-2)}, \ldots, x^{(1)}) = P(x^{(i)}|x^{(i-1)})$ (Andrieu et al. 2003). The values of the random variables are chosen from a fixed set called the state space of the Markov chain. When the state space is finite, the one step transition probabilities between various states are generally given as a matrix known as the *transition kernel*.

For certain Markov chains, irrespective of the initial distribution on $X^{(0)}$, the distribution on the of values $X^{(n)}$ converges as n increases, to its *limiting* or *stationary distribution*. More formally, a stationary distribution π with respect to a Markov chain with transition kernel A satisfies the condition $\pi = \pi A$.

Given a hard-to-sample *target* distribution, MCMC techniques solve the problem by constructing a Markov chain whose stationary distribution is the target distribution and drawing samples from it. **Metropolis-Hastings** (MH) is a popular MCMC-based sampling technique. Given a target distribution π and an irreducible, aperiodic Markov chain with transition kernel A, the MH sampler proposes a transition from state x to y according to $A(x, y)$, but then accepts or rejects this proposal according to the acceptance probability $min\{1, \frac{\pi(y)A(y,x)}{\pi(x)A(x,y)}\}$ (Hastings 1970).

Adaptive MCMC. Given a target distribution, MCMC algorithms construct a Markov chain whose stationary distribution matches the target. Several Markov chains may have matching stationary distribution, with different rates of convergence. Adaptive MCMC

algorithms tune a given transition kernel, effectively switching between different Markov chains as samples are drawn (Roberts and Rosenthal 2007).

The *total variation distance* between two distributions P and Q is defined as $\|P-Q\| = \frac{1}{2}\sum_x |P(x)-Q(x)|$ (Levin et al. 2009). An adaptive MCMC algorithm preserves the stationary distribution under the *ergodicity* conditions (Roberts and Rosenthal 2007) given below. In the following, $A(x,\cdot)$ denotes the x-th row of matrix A, and $\pi(\cdot)$ denotes the *row vector* π.

Ergodicity conditions. Given a family of transition kernels $\{P_{\Gamma_1}, P_{\Gamma_2}, \ldots\}$, with π as the common stationary distribution, adaptive MCMC algorithms choose kernel P_{Γ_i} at time i. The update rule of P_{Γ_i} is specified by the adaptive algorithm. Then ergodicity is preserved if all transition kernels have *simultaneous uniform ergodicity*, namely,

$$\forall \varepsilon > 0, \exists N \text{ such that } \|P_\gamma^N(x,.) - \pi(.)\| \leqslant \varepsilon \text{ for all } x \text{ and } \gamma$$

and the following *diminishing adaptation* condition is satisfied.

$$\lim_{n\to\infty} \sup_x \|P_{\Gamma_{n+1}}(x,.) - P_{\Gamma_n}(x,.)\| = 0 \text{ in probability}$$

3 MCMC for Probabilistic Logic Programs

The ability to treat a PRISM program as non-probabilistic in each world also helps us in designing sample-based query evaluation. Given a PRISM program P and a ground goal q, we lazily construct a set of worlds by sampling, such that q succeeds or fails in all worlds in the set. The set of worlds are represented by *assignments* described below.

Assignments. An *assignment* is denoted by partial function σ such that $\sigma(s,i)$ is the value of instance i of switch s. Note that σ represents a set of worlds; the set of worlds corresponding to σ is denoted by $worlds(\sigma)$.

Let σ be an assignment. Then $\sigma[(s,i) \to v]$ is an assignment that is identical to σ at every point except at (s,i) where it is v. We define a partial order "$\succeq$" over assignments: $\sigma' \succeq \sigma$ if $\sigma'(s,i) = \sigma(s,i)$ whenever $\sigma(s,i) \neq \bot$. We also say that σ' *extends* σ if $\sigma' \succeq \sigma$. Two assignments σ and σ' are *mutually exclusive*, denoted by $\sigma \parallel \sigma'$, if there is some switch instance (s,i) such that both σ and σ' are defined at (s,i), but $\sigma(s,i) \neq \sigma'(s,i)$. Two assignments are *compatible* (denoted by "$\nparallel$") if they are *not* mutually exclusive.

Given a switch s, we denote by $random(s)$ a value randomly drawn from the domain of s using the probability distribution defined for s. For looking up in an assignment or extending an assignment, we use function *pick_value* defined as follows:

$$pick\_value(\sigma,s,i) \quad = \quad \begin{cases} \langle v, \sigma\rangle & \text{if } \sigma(s,i) = v \neq \bot \\ \langle v, \sigma[(s,i) \mapsto v]\rangle & \text{if } \sigma(s,i) = \bot \text{ and } v = random(s) \end{cases}$$

Note that *pick_value* is non-decreasing in the sense that if $\langle v, \sigma'\rangle = pick\_value(\sigma,s,i)$, then $\sigma' \succeq \sigma$.

Sampling Evaluators. Our MCMC algorithm is parameterized with respect to a probabilistic query evaluation procedure called the *Sampling Evaluator*. Given an assignment σ and ground goal q, the sampling evaluator (probabilistically) generates an answer to q (*success*/*failure*), denoted by $ans(q)$, an assignment σ', and a sequence ρ of switch/instance/outcome triples $(s_1,i_1,v_1),(s_2,i_2,v_2),\ldots,(s_k,i_k,v_k)$, $k \geqslant 0$ such that the following conditions hold:

SE1. Consider a sequence of assignments $\sigma_0, \sigma_1, \ldots, \sigma_k$ such that $\sigma_0 = \emptyset$, and $\sigma_j = \sigma_{j-1}[(s_j, i_j) \mapsto v_j]$, $0 < j \leqslant k$. Then, $\sigma_k = \sigma'$, and σ' is compatible with σ, i.e. $\sigma' \nparallel \sigma$.

SE2. If $ans(q) = success$ (similarly, *failure*), then q is *true* (or *false*, resp.) in all worlds $w \in worlds(\sigma')$.

Moreover, let Σ denote the set of all σ' generated by the sampling evaluator. Then,

SE3. If w is a world s.t. q is true (similarly, false) in w, then $\exists \sigma' \in \Sigma$ such that $w \in worlds(\sigma')$ and $ans(q) = success$ (or *failure*, respectively).

SE4. Every distinct $\sigma_a, \sigma_b \in \Sigma$ are mutually exclusive: i.e. either $\sigma_a = \sigma_b$, or $\sigma_a \parallel \sigma_b$.

Properties **SE2** and **SE3** correspond to soundness and completeness, respectively. Property **SE4** ensures that a sampling evaluator can be used to define states in an MCMC algorithm. We can use Prolog-style evaluation, performed until the first derivation is found (if one exists) to construct a sampling evaluator satisfying the above requirements including **SE4**. The key idea is to implement *pick_value* to draw samples for switch instances and commit to the chosen values by representing assignments in the dynamic database. This evaluator is described in detail in (Nampally and Ramakrishnan 2014).

Initial State. When evaluating probabilities of unconditional queries, we generate an assignment corresponding to the initial state by invoking a sampling evaluator with an empty assignment. For conditional queries, a randomly constructed explanation for evidence is used to generate the initial state. We do this via a backtracking search for a derivation of evidence, and collect all the switches and outcomes used in that derivation into an initial assignment. The initial assignment is randomly constructed by randomizing the order in which clauses and switch values are selected during the backtracking search. We refer to this procedure as $InitialSample(P, e)$ in the MCMC algorithm shown in Fig. 2.

Transitions. Consider a state in the Markov Chain corresponding to assignment σ_j. We generate a successor state by (1) generating an alternative assignment σ' by assigning different outcomes for some switch instances in σ_j, and (2) invoking *SamplingEvaluator* with σ' to evaluate e and q to obtain the proposal for the next state, $\hat{\sigma}_{j+1}$. The switch instances to be resampled can be selected in many ways. We use one of the following:

1. Single Switch: We select a single (s, i) such that $\sigma_j(s, i) \neq \perp$ uniformly, and generate $\sigma' = \sigma_j[(s, i) \mapsto \perp]$, effectively forgetting (s, i).

2. Multi-Switch: This resampling mode is parameterized with a probability P. We generate σ' from σ_j by forgetting with probability P each (s, i) for which σ_j is defined. In Fig. 2, the resampling procedure is referred to as *Resample*.

```
1:  function MCMC
2:      Input: P: Program, q: Query, e: Evidence,
3:             N: Steps to simulate
4:      Output: p = prob(q | e)
        // Initialize
5:      σ_0 := InitialSample(P, e)
6:      (r_q, σ, _) := SamplingEvaluator(P, q, σ_0)
7:      N_q := 0
        // Generate a chain of length N
8:      for N times do
9:          σ' = Resample(σ)
10:         (r'_e, σ_e, ρ) = SamplingEvaluator(P, e, σ')
11:         if r'_e = success then
12:             (r'_q, σ_q, _) = SamplingEvaluator(P, q, σ_e)
13:             if accept(σ, σ_q) then
14:                 σ := σ_q
15:                 r_q := r'_q
16:         if r_q = success then N_q := N_q + 1
17:     return N_q/N
```

Fig. 2. MCMC Algorithm for Inferring Conditional Probabilities

Metropolis-Hastings. To draw samples from the target distribution $\texttt{prob}(q \mid e)$, we construct an MH sampler as follows. If the proposed state is inconsistent with evidence, it is rejected deterministically. If it is consistent, it is accepted/rejected based on *acceptance probability*. For single switch resampling strategy, the acceptance probability to go to state σ_2 from σ_1 is $\min\{1, \frac{|\sigma_1|}{|\sigma_2|}\}$. For multi-switch resampling strategy, the acceptance probability is 1, as derived in (Nampally and Ramakrishnan 2014).

4 Adaptive MCMC for Probabilistic Logic Programs

The rate at which samples are rejected deterministically based on the evidence (due to failure of condition in line 11 of Fig. 2) is called the *rejection rate*. We now present a technique to progressively adapt the proposal distribution based on the samples generated so far, in order to reduce the rejection rate.

The adaptation algorithm we present here is inspired by Q-learning, a reinforcement learning technique (Sutton and Barto 1998). For each switch s, instance i and outcome v used by the sampling evaluator, the Q-value $Q(s, i, v)$ is a real number in $[0, 1]$. Intuitively $Q(s, i, v)$ represents the probability of generating a consistent sample, when the sampling evaluator chooses v as the outcome of the i-th instance of switch s.

Initially, all Q-values are set to the same constant, essentially assuming that are equally likely to yield consistent samples. At each iteration of MCMC, adaptation is done after evidence is evaluated, by passing rewards to each switch/instance/outcome triple in ρ (computed in line 10 of Fig 2). We begin this processing with *reward* $= 0$ if $r'_e =$ *failure*, denoting an inconsistent sample, and *reward* $= 1$ otherwise. We work backwards through the sequence ρ so that the last switch/instance/outcome is given a reward of 0/1, which it then modifies and passes to the switch/instance/outcome preceding it in ρ. The Q-value of each random process/instance/outcome is computed as the average of the all rewards received by it. The algorithm for maintaining Q-values is given in Fig. 3.

```
1:  function ADAPT
2:      Input: ρ, r: Reward
3:      Global: Q: Q-values, c: counts,
4:              t: total Q-values.
5:      j := length(ρ)                          ▷ Initialize
6:      while j > 0 do
7:          let (s_j, i_j, v_j) = ρ[j]
8:          t(s_j, i_j, v_j) := (t(s_j, i_j, v_j) + r)
9:          c(s_j, i_j, v_j) := c(s_j, i_j, v_j) + 1
10:         Q(s_j, i_j, v_j) := t(s_j, i_j, v_j) ÷ c(s_j, i_j, v_j)
11:         j := j − 1
12:         r := Σ_{v ∈ values(s_j)} P(s_j, i_j, v) * Q(s_j, i_j, v)
```

Fig. 3. Adaptation of Q-values

The MCMC algorithm in Fig. 2 is modified for adaptive sampling as follows. First of all, function ADAPT is invoked after line 16. Secondly, *pick_value* function used in the sampling evaluator draws values for a switch instance (s, i) based on the normalized product of the original distribution and the Q-values of (s, i). Finally, the acceptance probability computation is modified to take the adapted distributions into account. Consider computing the acceptance probability to transition from state σ to σ'. We can partition the assignment σ into three non-overlapping functions: σ_1 for those (s, i)'s defined by σ but not by σ'; σ_2 for those defined by both σ and σ' but assigned different values; and finally, σ_3 for those defined by both σ and σ' and assigned same values. We can similarly partition σ' into σ'_1, σ'_2 and σ'_3.

For single-switch resampling strategy, the acceptance probability is given by

$$\min\left(1, \frac{P(\sigma_1')P(\sigma_2')P'(\sigma_1)P'(\sigma_2)1/|\sigma'|}{P(\sigma_1)P(\sigma_2)P'(\sigma_1')P'(\sigma_2')1/|\sigma|}\right)$$

where P is the original probability and P' is the adapted probability. For multi-switch strategy, the acceptance probability is given by

$$\min\left(1, \frac{P(\sigma_1')P(\sigma_2')P'(\sigma_1)P'(\sigma_2)}{P(\sigma_1)P(\sigma_2)P'(\sigma_1')P'(\sigma_2')}\right)$$

The derivations of these probabilities and the proof that the algorithm preserves ergodicity with respect to the target distribution are in (Nampally and Ramakrishnan 2014).

Beyond MCMC. It should be noted that the adapted distribution may not coincide with the conditional distribution $\texttt{prob}(q \mid e)$. This is not a problem for MCMC, since the adapted distribution is used as the proposal. However, the same adaptation scheme cannot be used, in general, with other sampling strategies such as independent sampling. Nevertheless, for a class of program/query pairs whose sampling evaluations is "Markovian", each switch instance's adapted distribution converges to its marginal distribution. For such programs, a modified adaptation can be used for independent sampling as well.

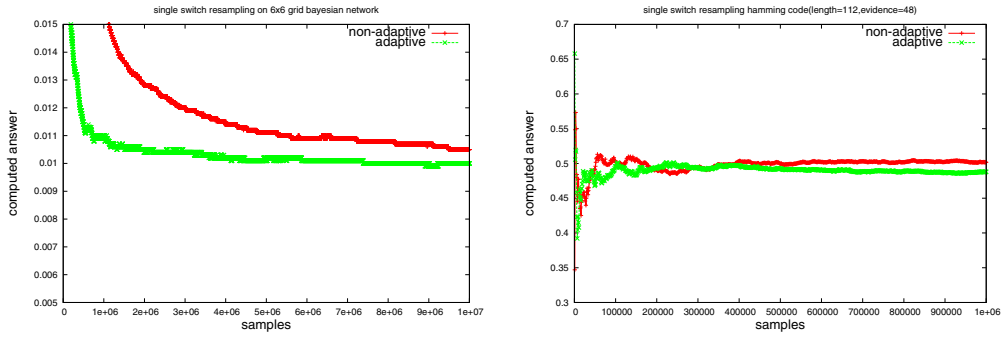
Consider the class of programs and queries for which the sequence ρ of random process/instance outcomes $(s_1, i_1, v_i), \ldots, (s_k, i_k, v_k)$ is such that the probability $P(ans(e)|(s_j, i_j, v_j))$ is independent of the triples $(s_l, i_l, v_l), l < j$. These program/query pairs are said to have a *Markovian Evaluation Structure*. We can redefine the Q-value to be to be the last propagated reward, ensuring that the rewards received by any switch instances will monotonically decrease due to adaptation. This allows us to adapt independent sampling as well as MCMC for such programs and queries.

5 Experimental Results

The MCMC algorithm was implemented in the XSB logic programming system (Swift et al. 2012). The sampling evaluator and the main control loop (Fig. 2) were implemented in Prolog. Lower level primitives managing the maintenance of assignments, resampling, computation of acceptance/rejection were implemented in C and invoked from Prolog. We evaluated the performance of this implementation on four synthetic examples: BN, Hamming, Grammar, and Reach. The experiments were run on a machine with 2.4GHz Opteron 280 processor and 4G RAM.

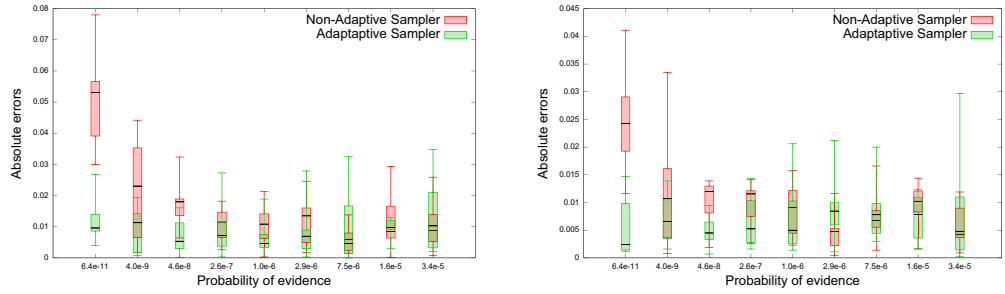
BN. This example consists of Bayesian networks whose Boolean-valued variables are arranged in the form of a 6×6 grid with each node having its left and top neighbors (if any) as parents. Evidence sets the outcome of 6 variables; and we query the outcome of one of the remaining variables. Fig 4(a) shows the conditional probability estimated by our algorithm plotted as a function of sample size. The time overhead for performing adaptation for this problem was small. This example clearly illustrates the benefit of adaptation.

Hamming. The Hamming code example is a PRISM program that generates a set of (4,3) Hamming codes. The evidence is a set of bits in the code with fixed values, and the query is the value of a non-evidence bit (Moldovan et al. 2013). The data bits in the code



(a) 6×6 grid Bayesian Network (b) 112-bit Hamming Code

Fig. 4. Computed probability vs. sample size for two examples



(a) sample size 40000 (b) sample size 80000

Fig. 5. Effectiveness of adaptation with varying probability of evidence and sample sizes

were independent random variables, while the parity bits were computed from the data bits' values. The answers computed by adaptive and non-adaptive samplers are given in Fig 4(b). In this example, the convergence of the adaptive MCMC is only a little better than that of the non-adaptive algorithm.

Reach. The final set of examples are reachability queries in probabilistic acyclic graphs, of the form shown in Fig. 1. For the graph shown in Introduction, while computing $\texttt{prob}(\texttt{reach}(\texttt{a},\texttt{d}) \mid \texttt{reach}(\texttt{a},\texttt{e}))$, the non-adaptive sampler rejects 8% of the samples, while the adaptive one rejects 1.5%. Similar rejection rates were observed for larger randomly generated graphs as well. However, since the rejection rate of the non-adaptive sampler is low, there is no significant difference between the convergence of adaptive and non-adaptive samplers.

Effect of probability of evidence. We compared the answers computed by the adaptive and non-adaptive samplers as the probability of evidence is varied. We used the 6x6 grid BN described earlier and varied the probability of evidence from 6.4e-11 to 3.4e-5 and used sample sizes of 40k and 80k. The absolute values of errors made by both the samplers are plotted in Fig 5. Observe from the figure that adaptive sampling exhibits significantly lower errors than non-adaptive sampling when the probability of evidence is low. This difference disappears when the evidence probability is high.

6 Discussion

An MCMC-based algorithm for approximate inference of stochastic logic programs (SLP)s (Muggleton et al. 1996) was presented by Cussens (2000). The technique uses derivations as states of the chain, and proposes transitions by selecting alternative branches using repeated Bernoulli trials. In contrast, our sampling technique is largely independent of the query evaluation process itself. More recently, Moldovan et al. (2013) describe an MCMC technique for inference in ProbLog that treats explanations as states. However, special handling is needed since explanations may not be mutually exclusive in general, resulting in memory and time overheads when traversing the chain. In contrast, we use Prolog-style evaluation to assure that the samples are pair-wise mutually exclusive. A more detailed comparison of these techniques appears in (Nampally and Ramakrishnan 2014).

Adaptation techniques have been used to focus other sampling algorithms as well. For instance, Mansinghka et al. (2009) presents an adaptive sequential rejection sampling algorithm. This algorithm requires knowledge of the factors in the distribution from which samples are drawn. Since PRISM programs represent logical as well as statistical knowledge, explicit knowledge may not even be available in our case. Consequently, our work does not rely on an explicit knowledge of factors.

Recent works have used sampling techniques for parameter estimation in PLP. Sato (2011) presents an MCMC-based algorithm for computing posterior distribution on the parameters of a PRISM program. At its core, that work builds a Markov chain over the space of parameters, assuming that answer probabilities can be computed efficiently. In contrast, our work deals with inference when answer probabilities are intractable. Cussens (2011) describes an application of Approximate Bayesian Computation (ABC) Sequential Monte Carlo (SMC) (Toni et al. 2009) technique to the problem of parameter estimation in PRISM programs. A scheme similar to SMC is used to maintain a weighted set of parameters (particles), and these particles are updated using a "perturbation kernel". The ABC scheme generates synthetic data using independent sampling and uses the notion of distance between synthetic data and observed data to weight the particles. An interesting open question is whether MCMC-based computation of answer probabilities can be effectively used by the ABC scheme.

This paper focused on a generic MCMC method and adaptation, and did not consider the effect of resampling strategies. The order in which random processes are sampled may affect the convergence and hence the quality of inference. For instance, Decayed MCMC (Marthi et al. 2002) samples processes based on a temporal order. As future work, we plan extend our sampler to use an order based on programmer annotation; whether such annotations can be inferred from the program is an open problem. Finally, while sampling-based inference may be generally deployed, exact inference may still be feasible for queries with short derivations. Hence, an interesting direction of future work is to develop a hybrid inference technique that can combine exact and approximate inference based on programmer annotation. Such an inference technique can be seen as an analogue of the Rao-Blackwellized Particle Filtering method developed for Dynamic Bayesian Networks (Doucet et al. 2000).

Acknowledgements: This work was supported in part by NSF Grants CCF-1018459, CCF-0831298, and ONR Grant N00014-07-1-0928.

References

Andrieu, C., De Freitas, N., Doucet, A., and Jordan, M. 2003. An introduction to MCMC for machine learning. *Machine learning 50,* 1, 5–43.

Cussens, J. 2000. Stochastic logic programs: Sampling, inference and applications. In *Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann, 115–122.

Cussens, J. 2011. Approximate Bayesian computation for the parameters of PRISM programs. In *Inductive Logic Programming*. Springer, 38–46.

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. Problog: A probabilistic Prolog and its application in link discovery. In *IJCAI*. 2462–2467.

Doucet, A., Freitas, N. d., Murphy, K. P., and Russell, S. J. 2000. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *Uncertainty in Artificial Intelligence (UAI)*. 176–183.

Getoor, L. and Taskar, B. 2007. *Introduction to Statistical Relational Learning*. MIT press.

Hastings, W. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika 57,* 1, 97–109.

Levin, D. A., Peres, Y., and Wilmer, E. L. 2009. *Markov chains and mixing times*. Amer. Mathematical Society.

Mansinghka, V. K., Roy, D. M., Jonas, E., and Tenenbaum, J. B. 2009. Exact and approximate sampling by systematic stochastic search. In *International Conference on Artificial Intelligence and Statistics*. 400–407.

Marthi, B., Pasula, H., Russell, S., and Peres, Y. 2002. Decayed MCMC filtering. In *Uncertainty in Artificial Intelligence (UAI)*. 319–326.

Moldovan, B., Thon, I., Davis, J., and Raedt, L. D. 2013. MCMC estimation of conditional probabilities in probabilistic programming languages. In *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*. 436–448.

Muggleton, S. et al. 1996. Stochastic logic programs. *Advances in inductive logic programming 32*, 254–264.

Nampally, A. and Ramakrishnan, C. R. 2014. Adaptive MCMC-based inference in probabilistic logic programs. *CoRR abs/1403.6036*.

Poole, D. 1997. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence 94,* 1, 7–56.

Roberts, G. O. and Rosenthal, J. S. 2007. Coupling and ergodicity of adaptive Markov chain Monte Carlo algorithms. *Journal of applied probability 44,* 2, 458–475.

Sato, T. 2011. A general MCMC method for Bayesian inference in logic-based probabilistic modeling. In *IJCAI*. AAAI Press, 1472–1477.

Sato, T. and Kameya, Y. 1997. PRISM: a language for symbolic-statistical modeling. In *IJCAI*. 1330–1335.

Sutton, R. S. and Barto, A. G. 1998. *Introduction to reinforcement learning*. MIT Press.

Swift, T., Warren, D. S., et al. 2012. The XSB logic programming system, Version 3.3. Tech. rep., Computer Science, SUNY, Stony Brook. `http://xsb.sourceforge.net`.

Toni, T., Welch, D., Strelkowa, N., Ipsen, A., and Stumpf, M. P. 2009. Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems. *Journal of the Royal Society Interface 6,* 31, 187–202.

Vennekens, J. and Verbaeten, S. 2003. A general view on probabilistic logic programming. In *Proceedings of BNAIC-03*.

Propagation Properties of Min-closed CSPs

Guy Alain Narboni
Implexe, France
(*e-mail:* r-d@implexe.fr)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Min-closed constraints are numerical relationships characterised by a simple property. Yet, with finite-domain variables, min-closed systems give rise to a polynomial class of Constraint Satisfaction Problems. Propagation alone checks them for satisfiability. Solving is therefore search-free. Can this result be generalized from a discrete to a continuous (or mixed) setting? In this paper, we investigate the use of interval solvers for handling constraints with real variables. We show that the completeness result observed in the discrete case gracefully degrades into a 'close approximation' property in the continuous case. When switching from finite to infinite domains, the pruning power of propagation remains intact in the sense that it provides a box enclosure whose lower bound cannot be further updated (even by domain splitting). Applications of this analysis to scheduling, rule-based reasoning and scientific simulation are briefly mentionned.

KEYWORDS: Interval solvers, Bounds-propagation, Min-closed (resp. Max-closed) constraints

1 Introduction

A constraint satisfaction problem (CSP) is a formal problem statement which involves a finite set of variables (or unknowns), together with their associated definition domains, and a finite conjunction of constraints (or requirements), interrelating the variables. Typically, in combinatorial problems, variables have a finite domain, represented as an interval over the integers. A binary variable for instance has a $\{0, 1\}$ domain, where 0 conventionally stands for false and 1 for true.

A numeric CSP can be viewed as a CSP in which the integrality constraint on a variable x is removed and replaced — if implicit — by an explicit constraint $x \in \mathbb{Z}$. By stating constraint systems on real numbers, we can this way express both continuous and discrete problems, as is the case with integer or mixed models (MIP) in the extensions of linear programming (LP).

In a numeric CSP, a constraint $C(\vec{x})$ therefore identifies a subset of $\mathbb{R}^n$. Domain constraints are specific unary constraints. They bind the problem variables to their definition domain. For instance, the condition $\vec{x} \geqslant \vec{0}$ states a conjunction of domain constraints that restricts the problem space to the positive orthant: $x_1 \geqslant 0 \wedge \ldots \wedge x_n \geqslant 0$. The other constraints set additional conditions for a solution. Altogether, the problem restrictions define a feasible region.

By far, the best-known continuous CSP is a linear program. A linear constraint $\vec{a}\vec{x} \leqslant b$ (where $\vec{a}\vec{x}$ denotes a scalar product) cuts down an entire half-space. As stated in Table 1 in matrix notation, a conjunction of linear inequalities delimits a convex polyhedron.

Table 1: *A linear constraint system viewed as a numeric CSP*

$A\vec{x} \leqslant \vec{b}$	$C_1(\vec{x}) \wedge \ldots \wedge C_m(\vec{x})$
$\vec{x} \geqslant \vec{0}$	$\vec{x} \in D = D_1 \times \ldots \times D_n$

Solving a CSP means finding a solution or proving that the feasible region is empty. Clearly, the method of choice for solving a linear program is to use a linear solver (the satisfiability problem is of polynomial complexity). In the CSP framework however, we can easily express mixed-integer problems (including NP-complete ones) as well as non-linear problems for which solution procedures may be lacking. Therefore, deciding — in reasonable time — whether a solution exists to the problem modeled is a task that no inference engine is capable of, in general.

We can relax these requirements, either by restricting the constraint language, or by approximating the solution process. Constraint programming (CP) favors the latter option, in order to preserve the richness of CSPs. The key idea behind interval solving (which generalizes finite-domain solving) is to quickly exclude from tests areas void of solutions, so to concentrate the search effort on promising areas.

In some special circumstances, those consistency checks can be proved sufficient. For instance, to guarantee the tractability of a combinatorial search problem, it is enough to observe that the constraints are min-closed (or, symmetrically, max-closed). These algebraic properties have been introduced by Jeavons and Cooper (1995) to the study of finite-domain CSPs. But their definition is relevant to numeric CSPs as well.

In this paper, we prove that, in the general continuous case, min-closed CSPs have a 'close approximation property' from which the completeness of the specific discrete case derives. In reference to the shaving procedure used for trimming variable domains, we call it the *close shaven property*. Indeed, the 'box relaxation' an interval solver computes cannot be better refined with respect to its bottom corner, so that shaving is of no use.

Identifying min-closed constraints allows us to spot 'easy to solve' qualitative or quantitative problems. We get a certificate of tractability or, at least, of good convergence.

This research note is organised as follows. After recalling the basics of interval solving, we focus on the outcome of propagation — the underlying procedure used for pre-solving a CSP. We first characterize the reduced domains computed in the CP relaxation. Then, combining this knowledge to the definition of min-closed systems, we put forward and prove the close-shaven property that the reduced domains exhibit. We finally present a few application examples, crossing the lines between different disciplines.

2 Interval approximations in Constraint Logic Programming

When seeking solutions to numerical constraints, exact methods of computer algebra often appear too specialized or too costly. Interval approximation methods provide an alternative means which, though usually weak, is general-purpose (Older and Vellino 1993; Van Hentenryck et al. 1997; Benhamou and Granvilliers 2006).

$$-\infty < -f_M < \cdots < -f^+ < -f < \cdots < 0 < \cdots < f < f^+ < \cdots < f_M < +\infty$$

The finite numerical scale $\mathbb{F}$.

2.1 Floating point intervals

Calculations on reals are performed using a slide rule having a finite set $\mathbb{F}$ of graduations. In practice, this set is given by the IEEE scale of floating point numbers. Small integers (together with the 2 infinities) are another option. Members of $\mathbb{F} \cup \{-\infty, +\infty\}$ have a direct machine representation. But most real numbers like 1/3 lie within two graduations. As a matter of consequence, operations on reals are substituted with operations on intervals.

Following the theory of Prolog IV (Colmerauer 1994), we shall distinguish between open and closed intervals. Historically, the need for expressing strict as well as non-strict inequalities derives from the universal role given in the language to the *not equal* constraint $x \neq y$ (aka *dif*). We'll call *approximation interval* any interval of reals whose bounds (when they exist) are members of $\mathbb{F}$. We'll use the notation [1, 2[equivalent to [1, 2) for the half-closed and half-open interval $\{x \in \mathbb{R} \mid 1 \leqslant x < 2\}$. The choice of $\mathbb{F}$ leads to a finite partition[1] of the real line $\mathbb{R}$ into *atomic intervals* of the following kind:

- degenerate closed intervals $\{f\}$ reduced to a point which is a member of $\mathbb{F}$
- open intervals $]f, f^+[$ between reals that are consecutive members of $\mathbb{F}$
- the open half-line $]f_M, +\infty[$ where f_M is the greatest real member of $\mathbb{F}$
- the open half-line $]-\infty, -f_M[$ where $-f_M$ is the least real member of $\mathbb{F}$.

Any non-empty approximation interval D therefore uniquely decomposes into a disjoint union of atomic intervals that forms a totally ordered sequence. We'll call the first element Δ of that sequence the *leftmost slice* of D (and the last, its *rightmost slice*).

2.2 Interval constraints

In the sequel, we'll make the non-restrictive assumption that variable domains are convex[2]. Thus, a *domain constraint* will be a relation of the form $x \in D$ where D is an approximation interval (possibly empty). Domain constraints are handled apart in CP.

Assuming a fixed number of unknowns, say n, the conjunction of domain constraints $(x_1 \in D_1) \wedge (x_2 \in D_2) \wedge \ldots \wedge (x_n \in D_n)$ is equivalent to $\vec{x} \in D_1 \times D_2 \times \ldots \times D_n$. The cross-product of the variable's domains defines a *box*, i.e., a hyper-rectangle aligned with the axes of the Cartesian coordinate system.

A numerical constraint solver comes equipped with a catalogue of *primitive constraints* (from arithmetic, trigonometry, ...) that can be used for model building.

For the sake of simplicity, we shall assume that all the constraints used in our model are n-ary[3]. Thus, a system of m simultaneous constraints can be condensed as:

$$C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in D_1 \times \ldots \times D_n)$$

where each C_i denotes a primitive constraint ($1 \leqslant i \leqslant m$) and each x_j refers to a numerical unknown ($1 \leqslant j \leqslant n$). A constraint system is satisfiable if it has solutions. Obviously, the domain product $D_1 \times \ldots \times D_n$ provides an enclosure of all of the numerical solutions sought in $\mathbb{R}^n$. But even with a tight enclosure, a solution set may remain infinite.

[1] If the approximation is restricted to closed intervals, as in (ECLiPSe), we have a covering instead.
[2] The rationale (and leftmost slice) is the same for domains made of unions of approximation intervals.
[3] A relation is always the projection of its cylindrical extension

The unary constraint 'is an integer' is of particular interest for making the connection with finite-domain contraint solving. By typing all of (or some of) the variables as integers, we are allowed to express purely combinatorial or mixed problems.

3 Interval solving

Basically, constraint solving proceeds by transforming a system of primitive constraints:

$$C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in D_1 \times \ldots \times D_n) \tag{1}$$

into an equivalent system:

$$C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in \underline{D}_1 \times \ldots \times \underline{D}_n) \tag{2}$$

where all the domains are reduced (i.e., their bounds are updated): $\underline{D}_j \subset D_j$ $(1 \leqslant j \leqslant n)$.

By reducing domains, interval *propagation* amounts to drawing valid inferences locally on variables' domains that are shared globally. Every consequence derived is correct, but not all expected consequences are derived. So, reasoning is sound but incomplete. Interval solving is akin to a pre-processing step: it often has to be complemented by search.

When domains are finite, a complete exploration of the search space is possible (at an exponential worst cost in the number of variables). When domains are infinite, search is inevitably limited by the precision of the grid being used for the calculations.

3.1 Properties of reduced domains

Since implementations of constraint solvers vary with respect to the *consistency checks* performed during propagation, we'll stick to the theory of Prolog IV to make things clear[4]. Formally, for every constraint C_i $(1 \leqslant i \leqslant m)$, the following property holds:

$$\underline{D}_1 \times \ldots \times \underline{D}_n = red_{C_i}(\underline{D}_1 \times \ldots \times \underline{D}_n) \tag{3}$$

where $red_C(D_1 \times \ldots \times D_n)$ is defined as the smallest box of $\mathbb{R}^n$ contained in $D_1 \times \ldots \times D_n$ and containing the graph of the relation C, i.e., $\{\vec{x} \in \mathbb{R}^n |\ C(\vec{x}) \wedge\ \vec{x} \in D_1 \times \ldots \times D_n\}$. The narrowing operator red_{C_i} attached to the primitive constraint C_i minimizes the size of the domains. Those are trimmed as much as possible by performing bounds' updates.

The fixed point equation (3) means interval solving is complete for a 1-constraint CSP: $C_i(\vec{x}) \wedge\ \vec{x} \in D_1 \times \ldots \times D_n$ (in case of inconsistency, the box becomes the empty set). Moreover, if the box reduces to a singleton $\{\vec{x}\}$, we have the guarantee that $C_i(\vec{x})$ holds. From this, we recover the well-known global properties of interval solving:

- There are no solutions outside the box determined by the reduced domains.
- When the enclosing box becomes empty, system (2) is equivalent to the constraint false. We thus have a proof that system (1) has no solution.
- When the enclosing box reduces to a single point, we know for sure that this point satisfies every constraint. It is therefore a solution to the whole constraint system (the unique one indeed in the original domains).

[4] Refer to Collavizza et al. 1999 or Chiu Wo Choi et al. 2006 for a comparison with stronger forms of inference, over infinite and finite domains respectively.

3.2 Bounds-consistency property

Let us examine in detail what happens when the system (2) is *locally consistent*, i.e., when none of the reduced domain is empty. Then, every $\underline{D}_j$ has a leftmost slice, Δ_j. According to (3), for every constraint i (taken independently), and for every variable j, the following formula holds — otherwise, the slice Δ_j would have been ruled out:

$$\exists \vec{x}\ C_i(\vec{x}) \wedge (\vec{x} \in \underline{D}_1 \times \cdots \times \Delta_j \times ... \times \underline{D}_n) \tag{4}$$

When all of the reduced domains are bounded and closed — as is the case with finite domains, we retrieve the classical definition of *bounds-consistency*, also known as 2B- (Lhomme 1993) or hull-consistency (Benhamou et al. 1999). We can always find in the box $\underline{D}_1 \times \cdots \times \underline{D}_n$ a solution to $C_i(\vec{x})$ extending the partial assignment $x_j = min(\underline{D}_j)$, since for every constraint i ($1 \leqslant i \leqslant m$) and every domain j ($1 \leqslant j \leqslant n$) we have:

$$\exists \vec{x}\ C_i(\vec{x}) \wedge (\vec{x} \in \underline{D}_1 \times \cdots \times \{min(\underline{D}_j\} \times ... \times \underline{D}_n) \tag{5}$$

A similar analysis applies to rightmost slices and maximum domain values.

3.3 Variants of primitive constraints

Quite often, quantitative relationships are specified using mathematical functions. We can define for instance two constraints over $\mathbb{R}^n$ out of a function f, from $\mathbb{R}^{n-1}$ into $\mathbb{R}$: $C(\vec{x}, y)$ if and only if $y = f(\vec{x})$ and $C'(\vec{x}, y)$ if and only if $y \geqslant f(\vec{x})$. The definitions of C and C' being fairly close, the following lemma shows we don't need to implement two narrowing operators, red_C and $red_{C'}$. We can define C' as a variant of the primitive constraint C, without loosing the native properties (3, 4) that propagation guarantees.

Lemma *Let C be a primitive constraint over $\mathbb{R}^n$ and Op an inequality symbol ($<$, $\leqslant$, $\geqslant$, or $>$). Define C' in terms of C by* $C'(\vec{x}, y) = \{(\vec{x}, y) \in \mathbb{R}^n \mid \exists z\ C(\vec{x}, z) \wedge (z\ Op\ y)\}$. *Then, the bounds-consistency property holds for the defined constraint C'.*

Proof
Consider a system made of the pair of constraints defining C', together with some domain restrictions (without loss of generality, we here assume the inequality stated is $z \leqslant y$):

$$C(\vec{x}, z) \wedge (z \leqslant y) \wedge (\vec{x} \in D_1 \times \ldots \times D_{n-1}) \wedge (y \in D_n) \wedge (z \in D_{n+1})$$

Let $\underline{D}_1 \times \ldots \times \underline{D}_n \times \underline{D}_{n+1}$ be the cross-product of the reduced domains. Assume the system is locally consistent and suppose $\underline{D}_1 \times \ldots \times \underline{D}_n$ is not the smallest box containing the solutions to the system $C'(\vec{x}, y) \wedge (\vec{x} \in D_1 \times \ldots \times D_{n-1}) \wedge (y \in D_n)$. Then, we can safely cut away a leftmost or rightmost slice, Δ_i, from a reduced domain $\underline{D}_i$ ($1 \leqslant i \leqslant n$). The constraint $C'(\vec{x}, y)$ having no solution in the box $\underline{D}_1 \times \ldots \times \Delta_i \ldots \times \underline{D}_n$, there exists no value of z for which the system $C(\vec{x}, z) \wedge (z \leqslant y)$ is satisfiable, and all the more so if z lies in the rightmost slice Δ_{n+1} of $\underline{D}_{n+1}$. Still, the bounds-consistency property holds for C in the conjunct, so $C(\vec{x}, z)$ remains satisfiable with $(\vec{x}, y, z)$ in $\underline{D}_1 \times \ldots \times \Delta_i \ldots \times \underline{D}_n \times \Delta_{n+1}$. We thus have a particular solution to C, say $(\vec{x}^*, z^*)$. The same is true for the inequality $z \leqslant y$. Observe that the leftmost slice of the reduced domain of y can only be equal to Δ_{n+1} or follow it, so that we can always choose a value $y^* \geqslant z^*$. We are thus able to construct a point of $\underline{D}_1 \times \ldots \times \Delta_i \ldots \times \underline{D}_n \times \Delta_{n+1}$ that is at the same time a solution to $z \leqslant y$ and a solution to $C(\vec{x}, z)$. Hence a contradiction. □

4 Min-closed constraint systems

4.1 Definitions

We say that a subset S of $\mathbb{R}^n$ is closed for the minimum operation (in short, *min-closed*) if, whenever $\vec{x}$ and $\vec{y}$ are in S, their minimum (defined component-wise) also lies in S:

$$\vec{x} \in S \wedge \vec{y} \in S \implies min(\vec{x}, \vec{y}) = (min(x_1, y_1), \ldots, min(x_n, y_n)) \in S \tag{6}$$

By definition, a constraint is min-closed if its solution set is min-closed.

A conjunction of min-closed constraints forms a min-closed system. Its solution set is min-closed, for the reason that the intersection of min-closed sets is min-closed. Similarly, the projection onto a subset of variables of a min-closed system is min-closed.

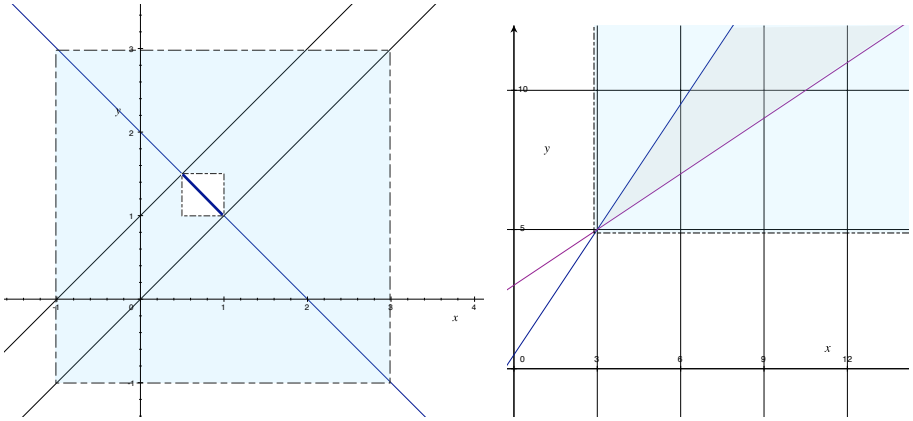
4.2 First catalogue of min-closed constraints

- Unary constraints are min closed. In particular, domain constraints are min-closed.
- Binary constraints of the form $y = f(x)$ are min-closed if f is a monotonically increasing function of x, since $f(min(x_1, x_2)) = min(f(x_1), f(x_2))$.

Table 2: *Min-closed primitive (or near-primitive) constraints.*

	Constraint		**Reified constraint**	
Comparison	$x = c$	•		
	$x \neq c$	•		
	$x \leqslant c$	•		
	$x < c$	•		
	$x \geqslant c$	•	$\delta \equiv (x \geqslant c)$	•
	$x > c$	•	$\delta \equiv (x > c)$	•
	$x = y$	•		
	$x \leqslant y$	•		
	$x < y$	•		
Integrity	$x \in \mathbb{Z}$	•		
Boolean	$\neg\beta$	•		
	$\beta \wedge \gamma$	•	$\delta \equiv (\beta \wedge \gamma)$	○
	$\beta \Rightarrow \gamma$	•		
Linear	$x + c = y$	•		
	$x - c = y$	•		
	$x + y \leqslant z$	○		
	$x + y < z$	○		
	$ax = y$	•		
Non-linear	$xy \leqslant z,\ x \geqslant 0,\ y \geqslant 0$	○		
	$y = sqrt(x)$	$\{(x, y) \in \mathbb{R}^2 \mid y = \sqrt{x},\ x \geqslant 0\}$	•	
	$y = log(x)$	$\{(x, y) \in \mathbb{R}^2 \mid y = \log x,\ x > 0\}$	•	
	...			

All of the constraints in this excerpt are min-closed. Some are max-closed too (• mark).
Notations: x, y, z are real variables; β, γ, δ binary variables, i.e., restricted to {0,1};
c is a numerical constant; a a non-negative coefficient.



(a) A counter-example (line segment) (b) A min-closed polyhedron

Fig. 1: Linear constraint systems and their CP relaxations (boxes)

5 The 'close-shaven' property

Interval solvers are notoriously bad at solving linear systems. Contrary to linear solvers, they miss a global view. Fig. 1a gives an example where propagation does not perform any pruning at all. The CSP is: $(-1 \leqslant x \leqslant 3) \wedge (-1 \leqslant y \leqslant 3) \wedge (x \leqslant y) \wedge (y \leqslant x+1) \wedge (x+y = 2)$. The *CP relaxation* $[-1,3] \times [-1,3]$ leads to a quite pessimistic approximation of the feasible region (highlighted line segment) whose exact projections are [0.5, 1] and [1, 1.5].

When the overall interval approximation is too loose, *shaving* heuristics can be used for tightening bounds without creating choice points (contrary to general search procedures). Shaving is therefore deterministic and attempts to exclude a leftmost (or rightmost) part of a domain as long as it is safe do so (i.e., when its cross-product with the other domains is void of solution). In the worst case, shaving may have to recursively examine all the atomic boxes that decompose a domain. So, shaving can be potentially costly too.

5.1 General continuous case

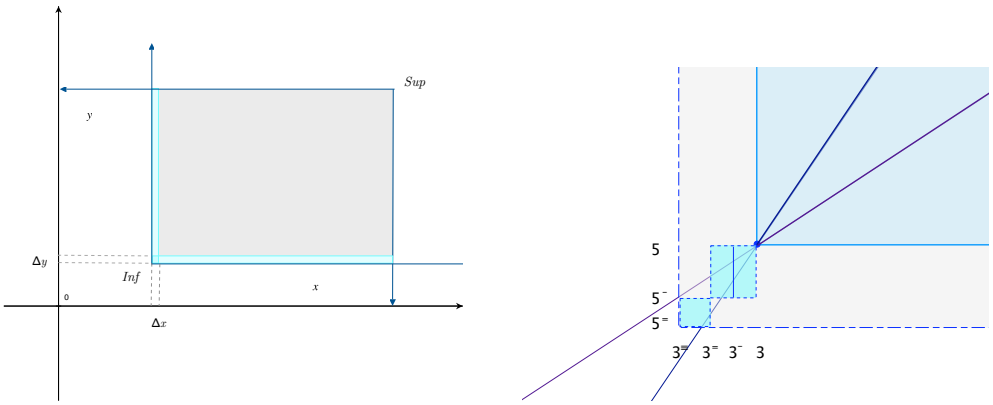
We are now at a point where we can state the approximation property that min-closed systems show for interval solving.

Proposition *Assume* $C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in \underline{D}_1 \times \ldots \times \underline{D}_n)$ *is a min-closed system that is locally consistent. Then, considering the leftmost slices* Δ_j *of the reduced domains* $\underline{D}_j$*, the system* $C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in \Delta_1 \times \ldots \times \Delta_n)$ *is also locally consistent.*

Proof
According to the bounds-consistency property (4), for each constraint C_i $(1 \leqslant i \leqslant m)$, we have a solution in $\Delta_1 \times \underline{D}_2 \times \ldots \times \underline{D}_n$, a solution in $\underline{D}_1 \times \Delta_2 \times \ldots \times \underline{D}_n$, ... and a solution in $\underline{D}_1 \times \ldots \times \underline{D}_{n-1} \times \Delta_n$. Their minimum is a point in $\Delta_1 \times \ldots \times \Delta_n$. Since the constraint is min-closed, it is a solution to C_i.

So, each constraint C_i is satisfiable within the cross-product $\Delta_1 \times \ldots \times \Delta_n$ which defines an atomic box and cannot be further reduced. It follows that the constraint system $C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in \Delta_1 \times \ldots \times \Delta_n)$ is locally consistent. □

Consequently, none of the lower bounds of the reduced domains obtained by propagation can be further improved by shaving: the reduced domains are all *close-shaven*. Still, the system may be globally inconsistent, but bisection then is of no help. The solver cannot disprove the formula: $\exists \vec{x}\ C_1(\vec{x}) \wedge ... \wedge C_m(\vec{x}) \wedge (\vec{x} \in \Delta_1 \times \cdots \times \Delta_j \times \cdots \times \Delta_n)$.



(a) The leftmost slices Δ_x and Δ_y in a 2D case. (b) Zoom in on $\Delta_x \times \Delta_y$ for Fig. 1b example.

Fig. 2: Search space decomposition into atomic slices and boxes

When constraints are min-closed, one should not observe gross over-approximations in the CP relaxation. By way of illustration, the cone of Fig. 1b is defined by two linear inequalities in the $\mathbb{R}^2$ plane: $(3x \geqslant 2y-1) \wedge (3y \geqslant 2x+9)$. This CSP is min-closed. It has a least element: $(x, y) = (3, 5)$. The close-up on Fig. 2b shows that the minimum, though not exactly computed, is precisely surrounded by the infimum of the reduced domains. $3^{=}$ denotes the third float preceding 3 and $5^{=}$ the second before 5. Atomic boxes that are filled indicate areas where the interval solver cannot exclude solutions. We see in particular that the leftmost slices of the two domains cannot be shaved off.

5.2 Specific discrete case

With finite domains, we derive from the close-shaven property a least element solution (minimum of the box). This witness point ensures satisfiability and restores decidability.

Corollary *Consider a min-closed system whose domains are bounded from below and whose variables are of integer type . If the reduced domains are not empty, then the CP relaxation $\underline{D}_1 \times \ldots \times \underline{D}_n$ has a least element which is a solution to the constraint system.*

Proof
Since the domains are closed and non empty, they have a minimum which is also their lowest slice. The atomic box $\Delta_1 \times \cdots \times \Delta_n$ is a singleton point of integer coordinates $min(D_1), \ldots, min(D_n)$. As it satisfies every constraint, it is a solution. □

We retrieve the fact that propagation (which is a polynomial-time procedure) is complete for min-closed finite-domain satisfiability checking[5]. Such CSPs are thus tractable.

[5] The original paper of (1995) proved that a min-closed CSP is solved by enforcing arc-consistency only, a stronger form of inference (Mackworth) than bounds-consistency. But obviously, the latter suffices.

6 Cross-connections and examples

6.1 Application to linear systems

The following result generalizes the case illustrated by Fig. 1b.

Property 1 *Every linear constraint of the form $a_1x_1 + \ldots + a_nx_n \leqslant cy + b$ where the a_i's are non-negative is min-closed.*

This 'meta-constraint' easily decomposes into a min-closed system, using the primitives of Table 2. If we define a *quasi-positive* matrix as a matrix having at most one negative entry per row, it follows that a linear system $A\vec{x} \leqslant \vec{b}$ is min-closed if A is quasi-positive. Then, if the feasible region is non-empty and bounded from below, it has a least element.

Now, there is a result from Cottle and Veinott (1972) stating exactly the converse:

Property 2 *If A is quasi-positive, a convex polyhedron of the form $P = \{\vec{x} \geqslant 0 \mid A\vec{x} \leqslant \vec{b}\}$ has a least element if it is non-empty.*

So, for linear systems we see here a one-to-one correspondence between the algebraic property (min-closed system), the geometric property (existence of a minimum) and the syntactic property (quasi-positive matrix).

Furthermore, we have a special case ensuring integrality (Chandrasekaran 1984):

Property 3 *Assuming that A and $\vec{b}$ have integer coefficients, if A is quasi-positive and if the negative entries of A are equal to -1, then the polyhedron $P = \{\vec{x} \geqslant 0 \mid A\vec{x} \leqslant \vec{b}\}$ has an integral least element if it is non-empty.*

Note this result holds for integer as well as real valued variables. It follows that propagation is complete for the numeric CSPs of that class since it involves no division. The polyhedron's least element is a corner-point for both the CP and LP relaxations.

A tension problem in a network (which is the dual of a flow problem) (Chvátal 1983) is characterized by difference bounded constraints of the form $a \leqslant x - y \leqslant b$. With 2 non-zero entries per row, equal to $+1$ and -1, its matrix is quasi-positive and quasi-negative. Such a linear system globally acts as a primitive constraint for interval solving — a feature which has direct applications to discrete- as well as continuous-time scheduling.

6.2 Application to rule-based systems

Property 4 *A conditional constraint of the form: $(x_2 \geqslant a_2) \wedge \ldots \wedge (x_k \geqslant a_k) \Rightarrow (x_1 \geqslant a_1)$ is min-closed.*

Again, the property holds for integer and real variables (regarding the expressivity of a knowledge engineering language, this point is worth noting). More generally, the right-hand side of the implication can be any min-closed constraint.

We proved elsewhere (2010) that a finite-domain CSP made of rules of the above form can be translated into a Horn propositional clausal system, i.e., into a well-known linearly satisfiable SAT problem. We coined the term *Horn-reencodable* for characterizing a rule base of such kind.

Now, according to Jeavons and Cooper (1995), every min-closed finite-domain constraint is logically equivalent to a conjunction a conditionals of that form. This means that a min-closed finite-domain CSP is indeed Horn-reencodable.

The ability to provide a solution by joining the minima of the reduced domains thence derives from the integer least element property of a Horn polytope (Chandru and Hooker).

6.3 Application to simulation systems

Convergence and numerical stability of the computation methods used are major concerns in Numerical analysis when solving large (often sparse) equation systems. Many come from the discretization of differential equations like the Laplacian equation for simulating heat transfer. A typical 2D pattern is : $-4x_{i,j} + x_{i,j+1} + x_{i,j-1} + x_{i-1,j} = 0$. It relates each cell to its north, est, south and west neighbours. We have one equation per cell, hence a system $A\vec{x} = \vec{b}$ with as many equations as unknowns (cell values).

We observe that the square matrix A is quasi-positive. It is also diagonally dominant, so that the solution we expect to be non-negative is also known to be unique when it exists (cf. cell assignments stating boundary conditions). The equation system is the conjunction of 2 systems of inequalities: $A\vec{x} \leqslant \vec{b}, \vec{x} \geqslant \vec{0}$ and $A\vec{x} \geqslant \vec{b}, \vec{x} \geqslant \vec{0}$. The former is min-closed. The latter is max-closed. Therefore, the value of the physical solution will lie between a least and a greatest element, both computed with a high quality of approximation. Moreover, if only one bound is needed, solving half of the system will do.

7 Conclusion

Compared to polyhedral studies for MIPs, there are not many theoretical results available for figuring out in advance the outcome of a CP computation. However, it is a well-known fact that two formulations that are logically equivalent are rarely equal in efficiency. For understanding a CSP behaviour, a human modeler has to master the idiosyncrasies of the type of engine he intends to use.

In this paper, we have shown with interval solving how to generalize the completeness property of propagation on min-closed systems, from finite domains to infinite domains. We have answered the question of what is the meaning of a good approximation for an interval solver. It is a box enclosure of the feasible region that cannot be refined by search. With min-closed systems, the CP relaxation delineates the best outward approximation of the infimum of the solution set. And this is obtained free of search. The same is true for max-closed systems (with respect to the supremum).

The polynomial class of min-closed problems has a theoretical significance since, among its instances, we find Horn satisfiability and Critical Path optimization problems. It also has practical applications, for instance, to machine scheduling (Purvis and Jeavons 1999) or product configuration problems (Narboni 2010).

From the point of view of complexity theory, Bodirsky and Kára (2010) have thoroughly investigated the generalization of min-closed CSPs to infinite domains for temporal constraint languages. Though restricted to logical combinations of variable comparisons, those languages can be of use for specifying continuous-time disjunctive scheduling problems, or modeling rule-based systems too. The problem being amenable to quantifier elimination, it is decidable. The authors prove a dichotomy result and provide an exact solution algorithm for the polynomial case.

Our analysis gives an intuitive insight into the building and interpretation of interval-based models. In the light of it, we may identify 'well-solved' sub-problems, occasionally predict the robustness of a solution design or assess the effectiveness of a search heuristics, prior to passing test experiments.

References

Apt, K. and Wallace, M. 2007. *Constraint Logic Programming using ECLiPSe*, Cambridge.

Benhamou, F. et al. 1995. *Le manuel de Prolog IV*, PrologIA. http:// www.prolog-heritage.org.

Behamou, F. and Goualard, F. and Granvilliers, L. and Puget, J-F. 1999. Revising hull and box consistency. In *Procs. ICLP'99*, 230-244.

Benhamou, F. and Granvilliers, L. 2006. Continuous and Interval Constraints. In *Handbook of Constraint Programming*, Elsevier.

Bodirsky, M. and Kára, J. 2010. The complexity of temporal constraint satisfaction problems. *J. ACM*, 57:2.

Chandrasekaran, R. 1984. Integer programming problems for which a simple rounding type of algorithm works. In *Progress in Combinatorial Optimization* (Pulleyblanck Ed.), 101-106.

Chandru, V. and Hooker, J. 1999. *Optimization methods for logical inference*, Wiley.

Chiu Wo Choi and Harvey, W. and Lee, J. and Stuckey, P. 2006. Finite Domain Bounds Consistency Revisited. In *Procs. Australian Joint Conference on Artificial Intelligence*, 49-58.

Chvátal, V. 1983. *Linear Programming*, Freeman.

Cohen, D. and Jeavons, P. 2003. Tractable Constraints Languages. In *Constraint Processing* by R. Dechter, 299-331, Morgan Kaufmann.

Collavizza, H. and Delobel, F. and Rueher, M. 1999. Comparing partial consistencies. *Reliable computing*, 5:3 213-228.

Colmerauer, A. 1994. *Spécifications de Prolog IV*. Rapport technique, Faculté des Sciences de Luminy.

Cottle, R. and Veinott, A. 1972. Polyhedral sets aving a least element. *Mathematical Programming*, 3 238-249.

Jeavons, P. and Cooper, M. 1995. Tractable Constraints on Ordered Domains. *AI Journal*, 79:2 327-339.

Lhomme, O. 1993. Consistency techniques for Numeric CSPs. In *Procs. of the 13th IJCAI*, 232-238.

Mackworth, A. 1977. Consistency in networks of relations. *Artificial Intelligence*, 8:1 99-118.

Narboni, G. 2010. A domain language for expressing engineering rules and a remarkable sublanguage. In *Procs. IKBET ECAI'10* (A. Felferning, F. Wotowa Eds.). Journal version: 2013. On rule systems whose consistency can be locally maintained. *AI Communications*, 26:1 67-77, IOS Press.

Older, W. and Vellino, A. 1993. Constraint Arithmetic on Real Intervals. In *Constraint Logic Programming: Selected Research*, MIT Press.

Purvis, L. and Jeavons, P. 1999. Constraint tractability theory and its application to the product development process for a constraint-based scheduler. *In Procs. of PACLP'99*, The Practical Application Company.

Van Hentenryck, P. et al. 1997. *Numerica*, MIT Press.

Towards an Efficient Prolog System by Code Introspection

George S. Oliveira and Anderson F. da Silva
Departament of Informatics
State University of Maringá, Brazil
(*e-mail:* geo.soliveira@gmail.com, anderson@din.uem.br)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Several Prolog interpreters are based on the Warren Abstract Machine (WAM), an elegant model to compile Prolog programs. In order to improve the performance several strategies have been proposed, such as: optimize the selection of clauses, specialize the unification, global analysis, native code generation and tabling. This paper proposes a different strategy to implement an efficient Prolog System, the creation of specialized emulators on the fly. The proposed strategy was implemented and evaluated at YAP Prolog System, and the experimental evaluation showed interesting results.

KEYWORDS: Prolog System, Introspection, Mutability

1 Introduction

Several programming languages provide introspection, a way for programs to reason about their own internal structure. Besides in programming languages, introspection has been employed in dynamic translation systems (Kiriansky et al. 2002), security and reliability (Reis et al. 2005) and deadlock detection (Wen et al. 2011).

Prolog provides a significant form of introspection by the *clause* predicate. It allows the user to find metadata related to a goal in the database. Besides, using this predicate it is possible to create metacircular interpreters and write evaluators that use nonstandard search orders.

The use of metadata provided by introspection should not be limited to outside tools. So that, this paper proposes the use of introspection (and metadata) to build an efficient Prolog System. Introspection can allow the system to reason about its internal structure and self specialize on the fly. This strategy was implemented and evaluated on YAP (Santos Costa et al. 2012), and the results indicated that is possible to reduce the emulator's overhead.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the proposed strategy to construct an efficient Prolog System. Section 4 presents experimental results and Section 5 ends with the concluding remarks.

2 Related Works

Several researches have been conducted with the purpose of improving the WAM performance. These include researches about optimization of selection of clauses (Costa et al. 2007), specialization of unification (Meier 1990), global analysis (Hermenegildo et al. 2012), generation of efficient code (Taylor 1996) and tabling (Santos Costa et al. 2012), which are, in part, already employed in current Prolog system.

These researches, however, have their drawbacks: some of them require a refinement of the WAM (or variants) instruction set, or are very difficult to implement, or require the use of another technique to be truly effective, or are effective only for programs that manipulate large amount of data, in particular those for deductive database.

The purpose of this paper is to add a new strategy in order to improve the Prolog system performance, the use of introspection to reason about the engine's internal structure and based on it to generate on the fly specialized emulators.

3 A Prolog System by Code Introspection

Prolog emulators have provided an attractive system with a simple and elegant language. Such systems have been used on several applications such as theorem proving (Indo 2007), answer set programming (Inclezan 2013), heap solver (Albert et al. 2013), and others. Therefore, it is important to provide a system that can be specialized to the current program in order to minimize the intrinsic overhead of emulation. It is the purpose of this paper, to describe a Prolog System that uses introspection in order to create specialized emulators on the fly.

The Prolog language, although not be fully introspective, provides a degree of introspection. In Prolog, it is possible to execute the following example:

```
degree(rio, 36)
degree(maringa, 34)
degree(curitiba, 24)

hotsummer(X) :- degree(X, Y), Y > 30.

?- clause(hotsummer(X),B)
B = degree(X, Y), Y > 30
```

This example demonstrated how Prolog provides a custom database perusal. A similar functionality can be used in the engine level. Through introspection, the engine can peruse its own metadata and use them to create mutable emulators.

Reasoning about its internal structure makes feasible not only return the control flow graph of a specific instruction, but also the basic blocks, the number of basic blocks, the size of each basic block, among other metadata. With this metadata, the system can build a new emulator, more precisely, a specialized emulator (hereafter called S.emulator).

It is important to note that such feature raises some questions, such as:

- How can the system provide introspection?

- How does the generation of a specialized emulator occur?
- What is the mutability and how does the system manage it?

3.1 Providing Introspection

In a context in which the emulator source code is available, it is possible to feed the system with metadata describing each instruction, besides the structure of the system. On the other hand, if the source code is not available it is possible to use some strategy that comes from dynamic binary translation (Smith and Nair 2005), to perform this task. Therefore, the ultimate goal is to capture during execution time this metadata, specialize them (if necessary), and thus generating a specialized emulator that be able to reduce the overhead of emulation.

A simple and effective strategy is to create specialized emulators in a conservative way to assure it will never reach an unsafe state. Such task has the potential to increase the interpretation's overhead. Therefore, it is not desirable that such task be enabled throughout the whole execution of the program. Ideally, it should only be enabled when the program reaches a state in which type of its structures is consistent. As a result, it is possible to generate a sYAAM emulator and ensure that an unsafe condition will not be met.

Adding metadata to the system implies on the addition of new features such as capacity of perusing and marking such metadata. At this point, a critical question arises: how does the system begin to reason about its metadata?

In the context of just-in-time compilation, the compilation system is achieved when a certain piece of code is invoked frequently, which is determined with the use of counters. In this case, a piece of code must be compiled when its amount of invocations exceeds a threshold.

Using the same approach, the proposed strategy instruments the predicates with counters and uses them to initialize the task of perusing metadata. Therefore, when a counter reaches a certain threshold, the system begins to reason about its internal structure and marks the basic blocks executed by the current emulator instruction. It is important to note that the use of counters incurs in a minimal overhead because only one increment is executed in the head of the current clause (or fact). However, the markup task has an overhead that can negatively impact the system. Due to the task of perusing the metadata of the current instruction in order to find the current basic block and then mark it. Therefore, the proposed strategy provides a mechanism to enable and disable the markup task.

Disabling the markup task occurs when the system is running a S.emulator. If the system requires a new specialization, which will cause a return to the default emulator, the system enables the markup task and begisns the process of creating a new S.emulator. It is important to note that the system handles several emulators, however only the default emulator is the one be able to handle all the instructions and exceptions. On the other hand, a S.emulator does not have this capacity in order to be a specialized emulator.

3.2 Creating a S.emulator on the Fly

At the implementation level, every emulator instruction contains basic blocks that will always be executed and others that will only be executed from conditional branches. Eliminating conditionals branches and build a S.emulator only with the basic blocks executed can ensure the new emulator is compact and efficient in terms of execution time. Basically, the markup of basic blocks starts from the first emulator instruction of a critical clause[1] and continues until the last statement of this clause. This process will create a trace that will form a S.emulator. There are two important moments in the system life: when some clause becomes critical, and when the same clause becomes hot. The first indicates the system needs to begin the creation of a S.emulator, and the second indicates that there is a complete trace. In the second moment, the system can finish to reason about its internal structure, compile the new trace and finally install the new S.emulator.

When a trace is initialized, the critical clause and its first basic block executed becomes the head of the trace. After that, each basic block executed is marked to form the S.emulator. The trace consists of the control flow of each emulator instruction, represented by an ordered sequence of basic blocks which begins with a label and finishes with a unconditional branch to the label of the next emulator instruction. It is important to note that a trace does not contain only basic blocks from only one clause. In fact, critical and hot clauses are used to initialize and finalize the construction of a trace. It means a trace does not have knowledge about clauses.

Ideally, an optimal trace should consist solely of basic blocks executed, without conditional statements or useless basic blocks. However, the construction of such trace is not always feasible because it can ensure the efficiency of a program but not the execution of another program. For this reason, each trace, in addition to being efficient in terms of execution time also must ensure the completion of the program and a correct output. These conditions are fulfilled according to the following criteria:

1. The trace must be appropriate for the data type it was built;
2. The trace must maintain the conditions on the occurrence of exceptions;
3. A dereferenciation must complete successfully; and
4. The indexing instructions must invoke correct clauses.

Handling Data Types In practice, all emulator instructions are generic for all data type. Therefore, the system should ensure an efficient trace eliminating conditional statements. However, the Prolog data type can change during runtime. Thus, the system adjusts the trace (more precisely, the S.emulator built previously) to the new data type. Thus, while no change occurs, the trace will have a linear characteristic.

However, if a previously successful conditional statement fails, it is necessary the trace be modified to contain the new basic blocks. This feature, mutability, indicates that during the execution of a program, a trace can be rebuilt to change a previously built S.emulator. The mutability ensures an accurate control flow for all data type.

Handling Exceptions In the YAP context, an exception is a control flow that does not belong to a running clause. In practice, exceptions occur during backtracking and garbage

[1] A critical clause is that which counter reached a threshold.

collection. Both exceptions are prioritized during the trace construction, which infer that the trace will not ignore the occurrence of such exceptions.

At the implementation level, conditional statements that throw exceptions for garbage collection are not preceded by instructions that evaluate data type and, therefore, constitute a single basic block in the trace. In this case, the basic block represents a conditional statement that, if successful, returns the control flow to the default emulator and invoke the garbage collector. After that, the system triggers the emulators; thus the execution continues from that instruction that throws the exception on the S.emulator.

Moreover, exceptions caused by backtracking are always preceded by conditional statements including instructions for evaluating data type. Unlike exceptions for garbage collection, backtracking are handled by the S.emulator and should not have the conditional statements ignored.

Handling Dereferenciation In YAP, the dereferenciation is implemented as a loop to transverse a chain of pointers built during the unification of two variables. This loop finishes when the value of the term is found or when it is determined that a term has not been unified with a value yet. Before this loop is executed, a conditional statement to verify if the term is variable is performed, and the dereferenciation occurs only if this statement conditional is true.

Considering the dereferenciation implemented in YAP, this operation is handled by the proposed strategy in the following ways:

- If the conditional statement is executed before the dereferenciation loop determines that the term is not a variable, the dereferenciation operation is not inserted in the trace.
- On the other hand, if the term is variable, a basic block containing the dereferenciation operation is inserted in the trace. This block contains two branches. One will be executed only if the dereferenciation process finishes with a variable, and, therefore, the flow jumps to the basic block which handles variables. The other will be executed if this process founds a non-variable value, in this case the target basic block handles a non-variable.

Handling Indexing Instructions In practice, the indexing instructions are not profiled as the others emulator instructions and are entire inserted in the trace, even if none of its basic blocks are executed. It is important to note that before a clause be effectively executed, the indexing instructions need to be executed first so that the system can identify the correct clause to invoke. As several stop conditions are related to changing some data type, this condition should be detected by the trace. Therefore, the strategy is to use a conservative construction and maintain the indexing instructions in order to detect these cases, even if such cases never occurred during the construction of the trace. Besides, it is necessary to insert an instruction that performs a return to the default emulator in case of invoked clause has not been inserted into the trace.

In YAP, the indexing instructions are generated on demand (Santos Costa 2009); thus it is difficult to determine all the clauses that will be invoked after them, which further reinforces the need to use a conservative strategy.

3.3 Handling the Mutability

The system recompiles a trace when the execution flow is modified. Normally, every trace has at least an elementary basic block that contains several branches. In these basic blocks is inserted an additional target to allow the control flow returns to the default emulator when it is identified the execution of some basic block taht was not inserted in the trace. Besides, the system sets the two new registers, namely: K and BADDR. The former indicates whether the return occurred by a elementary basic block or a garbage collection exception. The latter indicates the address from where the trace needs to be rebuilt.

In other words, when a return to the default emulator occurs by an elementary basic block, the system captures the metadata of the current emulator and enables the markup task; thus the new basic blocks executed are inserted in this trace. When the default emulator reaches the head of the trace, this is recompiled, installed and invoked instead of the previous one. The new trace replaces the previous one. It reduces the space cost and avoids the maintenance of a garbage collector to the S.emulator area.

It is necessary to ensure the return to the default emulator, in the case of changing in the behavior of the program. In a situation without mutability, the earlier built S.emulators can become invalid for the new behavior, causing a lot of return to the default emulator, hurting the performance. With mutability, it is possible to change a previously built trace and minimize the overhead of changing the current emulator.

3.4 Putting It all Together

The original YAP's architecture comprises four components, namely: Libraries, Engine, Compiler and Internal Database. The Libraries are collections of high and low level libraries responsible for initializing the Engine, supporting threads, native predicates and SWI emulation. The Engine is in charge of executing the Prolog program. The Compiler translates the Prolog code into YAAM instructions. Finally, the Internal database is the database of clauses. A detailed description about the original YAP's architecture can be found in the work of Santos Costa et al. (Santos Costa et al. 2012). In order to provide mutability, the Engine has a new component, called Mutability System that is responsible for creating and handling the S.emulators. Figure 1 provides a general overview of YAP's architecture with mutability.

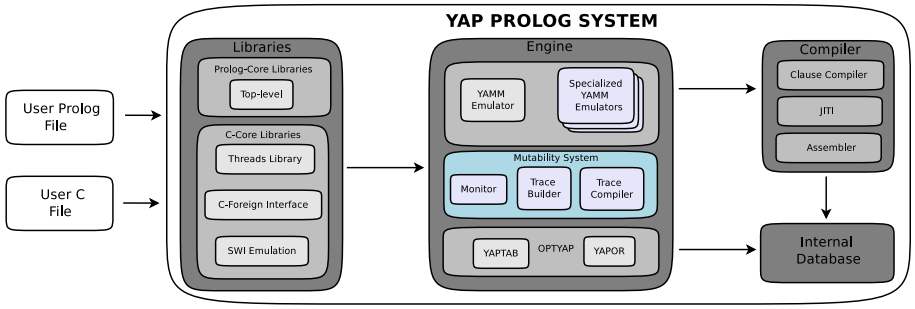


Fig. 1. The YAP's architecture with mutability.

When a clause becomes critical, the system enables the MONITOR that reasons about the internal system structure and marks the basic blocks that were executed. After that, when that clause becomes hot, the system captures the basic blocks, builds a control flow graph, compiles it and finally installs the compiled code (the new S.emulator). Thus, when the engine verifies that the current clause is a head of a S.emulator, it triggers from the default emulator to the correct S.emulator.

The TRACE COMPILER is a dynamic compiler invoked at runtime which generates a S.emulator from the control flow graph constructed by the TRACE BUILDER. The system uses the LLVM (Low Level Virtual Machine[2]) (Lattner and Adve 2004) to implement the dynamic compiler. The attractiveness of this framework is the fact it generating native code on memory, besides providing a library to tune the process of generating native code.

4 The Results

The Experimental Setup The experiments in this paper are conducted using YAP version 6.3 [3] and they are carried out on an Intel x86_64 based machine, supporting a Intel(R) Xeon(R) CPU E5504 processor running at 2.00GHz, and 24GB of RAM. The operating system on the machine was Ubuntu, running kernel 3.11.0. The Table 1 describes the programs used in the experiments.

Table 1. *The programs used in the experiments.*

Program	Preds.	Input Size	Program	Preds.	Input Size
append	1	63000000	hanoi	1	24
nreverse	2	52000	quicksort	1	52000
tak	1	57, 21, 36	binary trees	6	18
fannkuch	11	11	fasta	13	8000000
mandelbrot	5	2400	n-body	9	3000000
nsieve	9	5	nsieve bits	9	12
partial sum	3	30000000	pidigits	7	20000000
recursive	4	11	spectral norm	13	800

The validation of the results is based on the average of ten executions. Besides, in the experiments, the machine workload was minimum as possible, in other words, every instance was executed sequential, and the machine did not have external interference.

The improvement is calculated as follows.

$$Speedup = old\_runtime / new\_runtime$$

$$Improvement = (Speedup - 1) * 100$$

[2] http://www.llvm.org
[3] http://www.dcc.fc.up.pt/ vsc/Yap

The Performance In general, the performance improvement is proportional to the portion of time running on S.emulator. Mutability provided better results with performance improvement up to 10.99% considering all programs, and 23.57% considering only program with performance improvement. The Figure 2 shows the performance improvement of the proposed strategy.

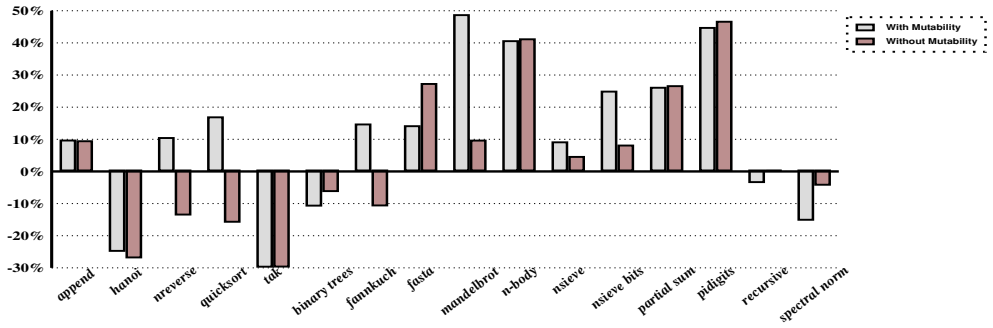


Fig. 2. The improvement of the proposed strategy.

The results indicates the proposed strategy achieves better results to programs that contains several predicates. kernels are small programs that have few predicates, besides short runtime. These characteristics impact the performance negatively. In some cases, the performance loss can be addressed by mutability (`nreverse`, and `quicksort`), but in general kernels can show slowdown.

The programs with several predicates obtained better performance, using mutability and without this feature. Only on three programs (`binary trees`, `recursive`, and `spectral norm`), the proposed strategy loses performance.

The results indicate that, in some cases, the mutability can degrade the performance (`fasta`, `pidigits`, `n-body`). In these cases, the return to the default emulator (due to the S.emulator throws an exception) and the decision of rebuilding the S.emulator did not bring the desired effect. It indicates it is difficult to predict the future based on past. However, in general the best choice is to use mutability.

Such results are not only consequences of constructing S.emulators on the fly, but the rules applied on control flow graph construction, which emphasizes the elimination of conditional statements. Building S.emulators without such statements benefits prefetching and branch predictions, and increase the scope of code transformations on a global level. Therefore, these benefits are the reason for achieving results.

To understand the sources of performance loss, it is necessary to evaluate the mutability system in details. So that, the evaluation described below emphasizes the rate of time spent by the mutability system. In the Figure 3 each bar is composed by several components, namely: default emulator, overflow, garbage collector, monitor e trace builder, trace compiler, and S.emulator. Besides, for each program is shown three bars. The first represents the default emulator, the second represents the proposed system using mutability, and the last the proposed system without mutability.

The cost of monitoring and building traces, as well as, the cost of compilation are minimal and do not degrade the performance. Without mutability, monitoring and building ranged from 0.002% (`mandelbrot`) to 0.12% (`recursive`) of elapsed time and compiling

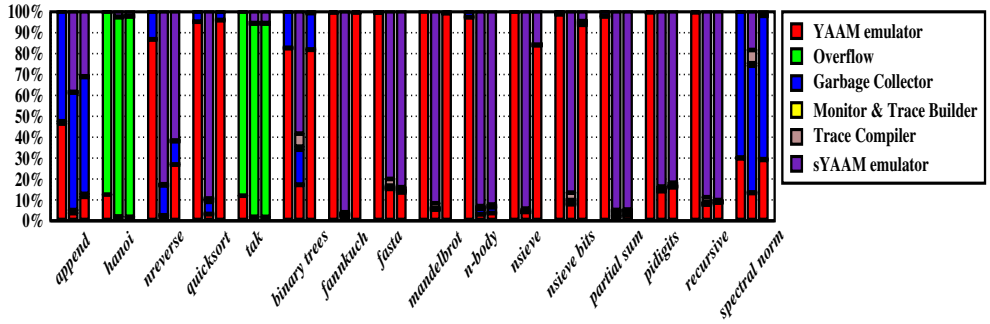


Fig. 3. The breakdown of mutability handling.

ranged from just 0.084% (`tak`) to 1.496% (`pidigits`). These results show that invoking the MONITOR only on frequent regions of the program is crucial to achievef performance. Now, with mutability, the cost ranged from 0.005% (`hannoi`) to 1.914% (`binary trees`) to monitor and build and ranged from 0.073% (`hanoi`) to 1.914% (`spectral norm`) to compile. This increase is evident by the need to keep the mutability system's modules active for long runtime, but it was not enough to degrade the performance due to the high runtime in S.emulator. The results, from Figure 3, indicate that garbage collection, overflow, and the time spend on YAAM emulator are sources of performance loss.

The results indicate it is possible to improve performance even in cases in which the native code is executed for a short time. Such cases include `mandelbrot` (gain of 9.59% with S.emulator active at 0.01% of the time), `nsieve` (gain of 4.51% with S.emulator active at 18.18% of the time), and `nsieve bits` (gain of 8.06% with S.emulator active at 4.82% of the time), but even so, the performance for these was lower than that achieved by mutability.

Discussion The results indicate that the performance gain is not only dependent on the implementation of useful traces, but it can be achieved if useless traces are avoided especially the traces with a high level of useless (`nreverse` and `quicksort`, and `fannkuch`). In this sense, the mutability was very important because, in addition to achieving better performance, in general it minimizes the overhead of useless traces.

Another important point to consider is that, with mutability, the construction and compilation time of traces did not impact the performance of programs that had been improved the performance without mutability, showing that the cost of rebuilding a S.emulator is minimal. The only exception to this rule is the program `Hanoi`, but for this, the case refers to the exceptions handled only on the default emulator. Additionally, the construction of optimal traces (only) is no guarantee of performance for programs with a high concentration on exception handling.

Finally, based on the results, an important future work is to modify the system so that it addresses the tasks of manipulating the memory areas also on S.emulator in order to ensure efficient implementation of all programs since all useful traces are built. Besides, the task of avoiding useless traces indicates be a good strategy in some cases, but it can be enhanced by the use of global analysis, for example, in order to detect the best starting clause of a trace that is the beginning of an invocation chain.

5 Concluding Remarks

This paper proposed the use of introspection in order to create a specialized emulator on the fly. The proposed strategy monitors the emulator execution and then generate a control flow graph in memory only formed by basic blocks executed, which are then compiled and executed with the highest priority.

The proposed strategy did not outperform the default emulator for all programs, due to the overhead of returning to default emulator on the presence of exceptions. Currently, a future work will investigate strategies to minimize this overhead.

References

Albert, E., de la Banda, M. J. G., Gmez-Zamalloa, M., Rojas, J. M., and Stuckey, P. J. 2013. A CLP Heap Solver for Test Case Generation. *Theory and Practice of Logic Programming 13,* 4-5, 721–735.

Costa, V. S., Sagonas, K., and Lopes, R. 2007. Demand-driven Indexing of Prolog Clauses. In *Proceedings of the International Conference on Logic Programming*. Springer-Verlag, Porto, Portugal, 395–409.

Hermenegildo, M. V., Bueno, F., Carro, M., Lpez-Garca, P., Mera, E., Morales, J. F., and Puebla, G. 2012. An Overview of CIAO and its Design Philosophy. *Theory and Practice of Logic Programming 12,* 1-2 (January), 219–252.

Inclezan, D. 2013. An Application of ASP to the Field of Second Language Acquisition. In *Logic Programming and Nonmonotonic Reasoning*, P. Cabalar and T. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer Berlin Heidelberg, 395–400.

Indo, K. 2007. Proving Arrows theorem by PROLOG. *Computational Economics 30,* 1, 57–63.

Kiriansky, V., Bruening, D., and Amarasinghe, S. P. 2002. Secure Execution via Program Shepherding. In *Proceedings of the USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 191–206.

Lattner, C. and Adve, V. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Palo Alto, California, 75–86.

Meier, M. 1990. Compilation of Compound Terms in Prolog. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, Austin, Texas, United States, 63–79.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. 2005. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 243–254.

Santos Costa, V. 2009. On Just in Time Indexing of Dynamic Predicates in Prolog. In *Progress in Artificial Intelligence*, L. Lopes, N. Lau, P. Mariano, and L. Rocha, Eds. Lecture Notes in Computer Science, vol. 5816. Springer Berlin Heidelberg, 126–137.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog System. *Theory and Practice of Logic Programming 12,* 1-2 (January), 5–34.

Smith, J. E. and Nair, R. 2005. *Virtual Machines: Versatile Platforms for Systems and Process*. Morgan Kaufmann Publishers.

Taylor, A. 1996. Parma – Bridging the Performance GAP Between Imperative and Logic Programming. *Journal of Logic Programming 29,* 1-3 (October), 5–16.

Wen, Y., Zhao, J., Huang, M., and Chen, H. 2011. Towards Detecting Thread Deadlock in Java Programs with JVM Introspection. In *Proceedings of the International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE Computer Society, Washington, DC, USA, 1600–1607.

Joint Tabling of Logic Program Abductions and Updates

ARI SAPTAWIJAYA* and LUÍS MONIZ PEREIRA

Centro de Inteligência Artificial (CENTRIA)
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
(*e-mail:* ar.saptawijaya@campus.fct.unl.pt, lmp@fct.unl.pt)

submitted n/a; revised n/a; accepted n/a

Abstract

Abductive logic programs offer a formalism to declaratively represent and reason about problems in a variety of areas: diagnosis, decision making, hypothetical reasoning, etc. On the other hand, logic program updates allow us to express knowledge changes, be they internal (or self) and external (or world) changes. Abductive logic programs and logic program updates thus naturally coexist in problems that are susceptible to hypothetical reasoning about change. Taking this as a motivation, in this paper we integrate abductive logic programs and logic program updates by jointly exploiting tabling features of logic programming. The integration is based on and benefits from the two implementation techniques we separately devised previously, viz., tabled abduction and incremental tabling for query-driven propagation of logic program updates. A prototype of the integrated system is implemented in XSB Prolog.

KEYWORDS: abduction, logic program updates, tabled abduction, incremental tabling.

1 Introduction

Abduction has been well studied in logic programming (Denecker and de Schreye 1992; Inoue and Sakama 1996; Fung and Kowalski 1997; Eiter et al. 1997; Kakas et al. 1998; Satoh and Iwayama 2000; Alferes et al. 2004), and it offers a formalism to declaratively represent and reason about problems in a variety of areas. Furthermore, the progress of logic programming promotes new techniques for implementing abduction in logic programs. For instance, we have shown recently in (Saptawijaya and Pereira 2013d), that abduction may benefit from tabling mechanisms; the latter mechanisms are now supported by a number of Prolog systems, to different extent. In that work, tabling is employed to reuse priorly obtained abductive solutions from one abductive context to another, thus avoiding potential unnecessary recomputation of those solutions.

Given the advances of tabling features, like incremental tabling (Saha 2006) and answer subsumption (Swift and Warren 2010), we have also explored these in addressing logic program updates. Our first attempt, reported in (Saptawijaya and Pereira 2013b), exploits incremental tabling of fluents in order to automatically maintain the consistency

* Affiliated with Fakultas Ilmu Komputer at Universitas Indonesia, Depok, Indonesia.

of program states, analogously to assumption based truth-maintenance system, due to assertion and retraction of fluents. Additionally, answer subsumption of fluents allows to address the frame problem by automatically keeping track, at low level, of their latest assertion or retraction, whether as a result of updated facts or concluded by rules. In (Saptawijaya and Pereira 2013a), the approach is improved, by fostering further incremental tabling. It leaves out the superfluous use of the answer subsumption feature, but nevertheless still allows direct access to the latest time a fluent is true, via system table inspection predicates. In the latter approach, incremental assertions of fluents automatically trigger *system level* incremental upwards propagation and tabling of fluent updates, on the initiative of top goal queries (i.e., by need only). The approach affords us a form of controlled (i.e., query-driven) but automatic truth-maintenance (i.e., automatic updates propagation via incremental tabling), up to actual query time.

When logic programs are used to represent agent's knowledge, then the issue of logic program updates pertains to expressing knowledge updates. Many applications of abduction, as in reasoning of rational agents and decision making, are typically susceptible to knowledge updates and changes, whether or not hypothetical. Thus, abductive logic programs and logic program updates naturally coexist in these applications. Taking such applications as a motivation, one of which we currently pursue (Saptawijaya and Pereira 2014), here we propose an implementation approach to integrate abductive logic programs and logic program updates by exploiting together tabling features of logic programming. The integration is strongly based on the reported approaches implemented in our two systems: Tabdual (Saptawijaya and Pereira 2013d) for tabled abduction, and Evolp/r (Saptawijaya and Pereira 2013a) for query-driven propagation of logic program updates with incremental tabling. In essence, we show how tabled abduction is jointly combined with incremental tabling of fluents in order to benefit from each feature, i.e., abductive solutions can be reused from one context to another, while also allowing query-driven, system level, incremental fluent update upwards propagation. The integration is achieved by a program transformation plus a library of reserved predicates. The different purposes of the dual program transformation, employed both in Tabdual and Evolp/r, are now consolidated in one integrated program transformation: on the one hand, it helps to efficiently deal with downwards by-need abduction under negated goals; on the other hand, it helps to incrementally propagate upwards the dual negation complement of a fluent.

The paper is organized as follows. Section 2 recaps tabled abduction and logic program updates with incremental tabling. We detail our approach to the integration in Section 3, and conclude, in Section 4, by mentioning related and future work.

2 Tabdual and Evolp/r

Tabled Abduction (Tabdual) We illustrate the idea of tabled abduction. Consider an abductive logic program P_0, with a and b abducibles:

$$q \leftarrow a. \qquad s \leftarrow b, q. \qquad t \leftarrow s, q.$$

Suppose three queries: q, s, and t, are individually launched, in that order. The first query, q, is satisfied simply by taking $[a]$ as the abductive solution for q, and tabling it. Executing the second query, s, amounts to satisfying the two subgoals in its body, i.e., abducing b followed by invoking q. Since q has previously been invoked, we can benefit from reusing

its solution, instead of recomputing, given that the solution was tabled. I.e., query s can be solved by extending the current ongoing abductive context $[b]$ of subgoal q with the already tabled abductive solution $[a]$ of q, yielding $[a,b]$. The final query t can be solved similarly. Invoking the first subgoal s results in the priorly registered abductive solution $[a,b]$, which becomes the current abductive context of the second subgoal q. Since $[a,b]$ subsumes the previously obtained (and tabled) abductive solution $[a]$ of q, we can then safely take $[a,b]$ as the abductive solution to query t. This example shows how $[a]$, the abductive solution of the first query q, can be reused from one abductive context of q (i.e., $[b]$ in the second query, s) to its other context (i.e., $[a,b]$ in the third query, t). In practice the body of rule q may contain a huge number of subgoals, causing potentially expensive recomputation of its abductive solutions, if they are not tabled.

Tabled abduction with its prototype TABDUAL, implemented in XSB Prolog (Swift and Warren 2012), consists of a program transformation from abductive normal logic programs into tabled logic programs; the latter are self-sufficient program transforms, which can be directly run to enact abduction by means of TABDUAL's library of reserved predicates. We recap the key points of the transformation. First, for every predicate p with arity n (p/n for short) defined in a program, two new predicates are introduced in the transform: $p_{ab}/(n+1)$ that tables one abductive solution for p in its single extra argument, and $p/(n+2)$ that reuses the tabled solution of p_{ab} to produce a solution from a given input abductive context into an output abductive context (both abductive contexts are the two extra arguments of p). The role of abductive contexts is important, e.g., in contextual abductive reasoning, cf. (Pereira et al. 2014). Second, for abducing under negative goals, the program transformation employs the *dual transformation* (Alferes et al. 2004), which makes negative goals 'positive' literals, thus permitting to avoid the computation of all abductive solutions of the positive goal argument, and then having to negate their disjunction. The dual transformation enables us to obtain one abductive solution at a time, just as when we treat abduction under positive goals. In essence, the dual transformation defines for each atom A and its set of rules R in a normal program P, a set of dual rules whose head not_A is true if and only if A is false by R in the considered semantics of P. Note that, instead of having a negative goal $not\ A$ as the rules' head, we use its corresponding 'positive' literal, not_A. The reader is referred to (Saptawijaya and Pereira 2013d) and publications cited thereof for detailed aspects of tabled abduction.

Logic Program Updates with Incremental Tabling (Evolp/r) EVOLP/R follows the paradigm of Evolving Logic Programs (EVOLP) (Alferes et al. 2002), by adapting its syntax and semantics, but simplifies it by restricting updates to fluents only. Syntactically, every fluent F is accompanied by its fluent complement $\sim F$. Program updates are enacted by having the reserved predicate $assert/1$ in the head of a rule, which updates the program by fluent F, whenever the assertion $assert(F)$ is true in a model; or retracts F in case $assert(\sim F)$ obtains in the model under consideration. Though updates in EVOLP/R are restricted to fluents only, it nevertheless still permits rule updates by introducing a rule name fluent that uniquely identifies the rule for which it is introduced. Such a rule name fluent is placed in the body of a rule to turn the rule on and off, cf. (Poole 1988); this being achieved by asserting or retracting that specific fluent. The reader is referred to (Saptawijaya and Pereira 2013a) for a more detailed theoretical basis of EVOLP/R.

Like TABDUAL, EVOLP/R is implemented by a compiled program transformation plus

a library of reserved predicates. The implementation makes use of incremental tabling (Saha 2006), a feature in XSB Prolog that ensures the consistency of answers in a table with all dynamic clauses on which the table depends by incrementally maintaining the table, rather than by recomputing answers in the table from scratch to keep it updated. The main idea of the implementation is described as follows. The input program is first transformed and then the initialization phase takes place. It sets a predefined upper global time limit in order to avoid potential iterative non-termination of updates propagation and it additionally creates and initializes the table for every fluent. When fluent updates are given, they are initially kept pending in the database, and only on the initiative of top-goal queries, i.e., by need, incremental assertions make these pending updates become active (if not already so), but only those with timestamps up to an actual query time. Such assertions automatically trigger system-implemented incremental upwards propagation of updates and tabling of fluents (thanks to the incremental tabling). Because fluents are tabled, a direct access to the latest time a fluent is true can be made possible by means of existing table inspection predicates, and thus recursion through the frame axiom can be avoided. Consequently, in order to establish whether a fluent F is true at an actual query time, it suffices to inspect in the table the latest time both F and its complement $\sim F$ are true, and to verify whether F is supervened by $\sim F$.

We recap the key points of the transformation. First, the transformation adds to each program clause of fluent f/n the timestamp information that figures as the only extra argument of fluents (i.e., heads of clauses) and denotes a point in time when a fluent is true (known as *holds-time*). Having this extra argument, both fluent $f/(n+1)$ and its complement $\sim f/(n+1)$ are declared as dynamic and incremental. Second, each fluent (goal) G in the body of a clause is called via a reserved *incrementally* tabled predicate $\mathit{fluent}(G, H_G)$ that non-deterministically returns holds-time H_G of fluent G. In essence, this reserved predicate simply calls G and obtains H_G from G's holds-time argument. Since every fluent and its complement are *incrementally* dynamic, the dependency of the incrementally tabled predicate $\mathit{fluent}/2$ on them can be correctly maintained. Third, the holds-time of fluent f in the head of a clause is determined by which *inertial* fluent in its body holds *latest*. Fourth, the dual transformation from TABDUAL is adapted for helping propagate the dual negation complement $\sim F$ of a fluent F incrementally, making the holds-time of $\sim F$ (and other fluents that depend on it) also available in the table.

3 Integrating Tabdual and Evolp/r

When logic programs are used to represent agent's knowledge with abduction for decision making, such applications are typically susceptible to knowledge updates and changes, e.g., because of incomplete and imprecise knowledge, hypothetical updates, and changes caused by agent's actions (side-effects). Driven by such applications, one of which we are currently pursuing (Saptawijaya and Pereira 2014), and given that TABDUAL and EVOLP/R have been conceptualized to deal with abduction and logic program updates independently, our subsequent challenge is how to seamlessly integrate both approaches. In Section 2 we observe that tabling is employed both in TABDUAL and EVOLP/R, despite its different purposes. Therefore, in addition to enable abduction and knowledge updates in a unified approach, the integration also aims at keeping the different purposes served by tabling in TABDUAL and EVOLP/R. That is, on the one hand the integration should

allow reusing an abductive solution entry from an abductive context to another. On the other hand, it should also support system level incremental upwards updates propagation. We now detail an approach to achieve these aims through a program transformation and library of reserved predicates.

Enabling Abducibles In abduction it is desirable to generate only abductive explanations relevant for the problem at hand. One stance for selectively enabling the assumption of abducibles in abductive logic programs is introducing rules encoding domain specific information about which particular assumptions are to be considered in a specific situation. We follow the approach proposed in (Pereira et al. 2013), i.e., the notion of expectation is employed to express preconditions for enabling the assumption of an abducible. An abducible A can be assumed only if there is an expectation for it, and there is no expectation to the contrary. We say then that the abducible is *considered*, expressed by the rule:

$$consider(A) \leftarrow expect(A),\ not\ expect\_not(A),\ A.$$

This method requires program clauses with abducibles to be preprocessed. That is, for every abducible A appearing in the the body of a rule, A is substituted with $consider(A)$. For instance, given abducible a, rule $p \leftarrow a$ is preprocessed into rule $p \leftarrow consider(a)$.

The Roles of Abductive Contexts and Holds-Time In scientific reasoning tasks, it is common that besides the need to abductively discover which hypotheses to assume in order to justify some observation, one may also want to know some of the side-effects of those assumptions. This is one important extension of abduction, viz., to verify whether some secondary observations are plausible in the presence of already obtained abductive explanations, i.e., in the abductive context of the primary one.

As in TABDUAL, our integration makes use of abductive contexts. They permit a mechanism for reusing already obtained abductive solutions, which are tabled, from one context to another. Technically, this is achieved by having two types of abductive context: *input* and *output*, where an abductive solution is in the output context and obtained from the input context plus a tabled abductive solution. In Section 2 we show that these two contexts figure as extra arguments of a predicate.

Updates due to new observations or changes caused by side-effects of abductions may naturally occur, and from the logic program updates viewpoint the time when such changes or updates take place needs to be properly recorded. In EVOLP/R, this is maintained via the timestamp information, known as holds-time, that figures as an extra argument in a fluent predicate. Like in EVOLP/R, this timestamp information plays an important role in the integration for propagating updates and tabling fluents affected by these propagations, as shown in subsequent sections.

Based on the need for abductive contexts and holds-time, every predicate p/n, i.e., $p(X_1,\ldots,X_n)$ is now transformed into $p(X_1,\ldots,X_n,I,O,H)$, where the three extra arguments refer to the input context I, the output context O, and the timestamp H.

We next show the mechanisms to compute abductive solutions and maintain holds-time through updates propagation using the ingredients discussed earlier.

Example 3.1

Consider P_1 with abducible a: $q \leftarrow a.$ $expect(a).$

After preprocessing abducible a in the body of rule $q \leftarrow a$, cf. "Enabling Abducibles", we have the program:

$$q \leftarrow consider(a). \qquad expect(a).$$

The preprocessed program is now ready to transform. We first follow the rule name fluent mechanism of EVOLP/R, i.e., a unique rule name fluent of the form $\#r(Head, Body)$ is assigned to each rule $Head \leftarrow Body$. For this example, we have only one rule, i.e., $q \leftarrow consider(a)$, which is assigned the rule name fluent $\#r(q, [consider(a)])$. Recall, the rule name fluent is used to turn the corresponding rule on and off by introducing it in the body of the rule. Thus, we have:

$$q \leftarrow \#r(q, [consider(a)]), consider(a). \qquad expect(a).$$

Next, we attach the three additional arguments described earlier. For clarity of explanation, we do that in two steps: first, we add abductive context arguments and discuss how abductive solutions are obtained from them; second, we include the timestamp argument for the purpose of maintaining holds-time in updates propagation.

Finding Abductive Solutions Adding abductive contexts brings us to the transform below (*cons* is shorthand for *consider*):

$$q(I, O) \leftarrow \#r(q, [cons(a)], I, R), cons(a, R, O). \qquad expect(a, I, I).$$

The abductive solution of q is obtained in its output abductive context O from its input context I, by relaying the *ongoing* abductive solution stored in context R from subgoal $\#r(q, [cons(a)], I, R)$ to subgoal $cons(a, R, O)$ in the body. For $expect(a)$, the content of the context I is simply relayed from the input to the output context. That is, having no body, the output context does not depend on the context of any other goals, but depends only on its corresponding input context.

Maintaining Holds-Time Now, the timestamp argument is added to the transform:

$$\begin{aligned} q(I, O, H) \quad &\leftarrow \quad \#r(q, [cons(a)], I, R, H_r), cons(a, R, O, H_a), \\ &\qquad latest([\#r(q, [cons(a)], I, R, H_r), cons(a, R, O, H_a)], H). \\ expect(a, I, I, 1). & \end{aligned}$$

The time when q is true (holds-time H of q) is derived from the holds-time H_r of its rule name fluent $\#r(q, [consider(a)])$ and H_a of $consider(a)$, via the *latest*/2 reserved predicate. Conceptually, H is determined by which *inertial* fluent in its body holds *latest*. Therefore, the predicate $latest(Body, H)$ does not merely find the maximum H of H_a and H_r, but also assures that no fluent in *Body* was subsequently supervened by its complement at some time up to H. The holds-time for $expect(a)$ is set to 1, by convention the initial time when the program is inserted.

Finally, recursion through frame axiom can be avoided by tabling fluents – in essence, tabling their holds-time – so it is enough to look-up the time these fluents are true in the table, and pick-up the most recent holds-time. For this purpose, incremental tabling is employed to ensure the consistency of answers in the table due to updates or changes on which the table depends, by incrementally maintaining the table through updates propagation. Similar to EVOLP/R, the incremental tabling of fluents is achieved via a reserved incrementally tabled predicate $fluent(F, I, O, H)$, defined as follows:

```
:- table fluent/4 as incremental.
```

$$\mathit{fluent}(F,I,O,H) \leftarrow \mathit{upper}(\mathit{Lim}), \mathit{extend}(F,[I,O,H],F'), \mathit{call}(F'), H \leqslant \mathit{Lim}.$$

where $\mathit{extend}(F,\mathit{Args},F')$ extends the arguments of fluent F with those in list Args to obtain F'. The definition requires a predefined upper time limit Lim, which is used to delimit updates propagation due to potential iterative non-termination propagation, cf. (Saptawijaya and Pereira 2013a) for details. Since $\mathit{fluent}(F,I,O,H)$ simply calls fluent F with a given list of context arguments I, O, and holds-time H, calls to fluents in the body of a rule can be recast into calls via reserved predicate $\mathit{fluent}/4$. The above transform finally becomes:

$$:\!- \ \mathit{dynamic}\ \#r/5, \mathit{expect}/4\ \mathit{as}\ \mathit{incremental}.$$

$$\begin{aligned} q(I,O,H) \quad &\leftarrow \quad \mathit{fluent}(\#r(q,[\mathit{cons}(a)]),I,R,H_r),\\ &\qquad\ \ \mathit{cons}(a,R,O,H_a),\\ &\qquad\ \ \mathit{latest}([\#r(q,[\mathit{cons}(a)],I,R,H_r),\mathit{cons}(a,R,O,H_a)],H).\\ \mathit{expect}(a,I,I,1).& \end{aligned}$$

along with the assertion of rule name fluent $\#r(q,[\mathit{cons}(a)])$ at the initial time 1,

$$\#r(q,[\mathit{cons}(a)],I,I,1).$$

Note that rule name predicate $\#r/5$ and predicate $\mathit{expect}/4$ may be subjected to incremental updates, hence their declaration as dynamic and incremental. On the other hand, predicate $\mathit{consider}/4$ (i.e., $\mathit{cons}/4$ in the example) is not so declared, though it depends (directly or indirectly) on dynamic incremental predicates $\mathit{expect}/4$ and $\mathit{expect\_not}/4$, as we further show in the subsequent section. Thus, there is no need to wrap its call in the body with the reserved predicate $\mathit{fluent}/4$.

Tabling of Abductive Solutions In the preprocessing, cf. "Enabling Abducibles", every abducible A appearing in the body of a rule is substituted with $\mathit{consider}(A)$. Recall the definition of $\mathit{consider}(A)$:

$$\mathit{consider}(A) \leftarrow \mathit{expect}(A),\ \mathit{not}\ \mathit{expect\_not}(A),\ A.$$

After preprocessing, the abducible A thus only appears in the definition of $\mathit{consider}(A)$. Consequently, the transformation that deals with tabling of abductive solutions takes place only in the definition of $\mathit{consider}/1$. Like in TABDUAL, we introduce two new predicates for $\mathit{consider}/1$, namely $\mathit{consider}_{ab}/3$ and $\mathit{consider}/4$, where predicate $\mathit{consider}_{ab}/3$ is used to table an abductive solution. We first define $\mathit{consider}_{ab}/3$ (exp is shorthand for expect):

$$:\!- \ \mathit{table}\ \mathit{consider}_{ab}/3\ \mathit{as}\ \mathit{incremental}.$$

$$\begin{aligned} \mathit{consider}_{ab}(A,E,T) \quad &\leftarrow \quad \mathit{timed}(A,A_T),\\ &\qquad\ \ \mathit{fluent}(\mathit{exp}(A),[A_T],R,H_1),\\ &\qquad\ \ \mathit{fluent}(\mathit{not\_exp\_not}(A),R,E,H_2),\\ &\qquad\ \ \mathit{latest}([\mathit{exp}(A,[A_T],R,H_1),\mathit{not\_exp\_not}(A,R,E,H_2)],T). \end{aligned}$$

Observe that the tabled abductive solution entry E is derived by relaying the ongoing abductive solution stored in context R from subgoal $\mathit{fluent}(\mathit{exp}(A),[A_T],R,H_1)$ to subgoal $\mathit{fluent}(\mathit{not\_exp\_not}(A),R,E,H_2)$ in the body, given $[A_T]$ as the input abductive context of $\mathit{exp}(A)$. This input context $[A_T]$ comes from the abducible A appearing in the body of $\mathit{consider}(A)$ after it is equipped with T, i.e., the time A is abduced; A_T is obtained using predicate $\mathit{timed}(A,A_T)$. Notice that time T is the same time that $\mathit{consider}_{ab}(A)$ is true, which is the latest time between the two fluents, $\mathit{exp}(A)$ and $\mathit{not\_exp\_not}(A)$. Notice

also that the subgoal call $not\ expect\_not(A)$ in the original definition becomes a predicate $not\_exp\_not(A)$ in the subgoal call $fluent/4$, in the transform. This predicate is the dual of exp_not and is obtained by the dual transformation, as explained in the next section. Like $expect/4$, it is subject to updating, and thus, declared as dynamic and incremental too.

Next, we define predicate $consider/4$, which reuses the tabled solution entry E from $consider_{ab}/3$, for a given input context I, to obtain a solution in its output context O. It is defined as (the holds-time H is just passed from the body to the head):

$$consider(A, I, O, H) \leftarrow consider_{ab}(A, E, H), produce(O, I, E).$$

The reserved predicate $produce(O, I, E)$ should guarantee that it produces a *consistent* output context O from I and E that encompasses both. For instance, $produce(O, [b_3], [a_1])$ and $produce(O, [a_1, b_3], [a_1])$ both succeed with $O = [a_1, b_3]$, but $produce(O, [not\ a_1], [a_1])$ fails because conjoining $E = [a_1]$ and $I = [not\ a_1]$ results in an inconsistent abductive context $O = [a_1, not\ a_1]$.

The Dual Program Transformation The different purposes of the dual program transformation in TABDUAL and EVOLP/R, cf. Section 2, are consolidated in the integration. First, the dual predicate not_G for the negation of goal G in TABDUAL and $\sim G$ for the negation complement of fluent G in EVOLP/R are now represented uniquely as not_G, declared dynamic and incremental. Second, the abductive context and holds-time arguments jointly figure in dual predicates, as for the positive transform.

The reader is referred to (Saptawijaya and Pereira 2013c) for a formal specification and refinement of the dual transformation. We illustrate the transformation for $q/0$ and $expect/1$ of Example 3.1. With regard to q, the transformation will create dual rules for q that falsify q with respect to its only rule,[1] expressed by predicate q^{*1}:

$$not\_q(I, O, H) \leftarrow q^{*1}(I, O, H).$$

Next, predicate q^{*1} is defined by falsifying the body of q's rule in the transform. That is, the rule of q is falsified by alternatively failing one subgoal in its body at a time, i.e. by negating $\#r(q, [cons(a)])$ or, instead, by negating $consider(a)$ and keeping $\#r(q, [cons(a)])$. Therefore, we have:

$$\begin{aligned} q^{*1}(I, O, H) &\leftarrow fluent(not\_\#r(q, [cons(a)]), I, O, H). \\ q^{*1}(I, O, H) &\leftarrow fluent(\#r(q, [cons(a)]), I, R, H_r), not\_consider(a, R, O, H), \\ &\quad\ verify\_pos([\#r(q, [cons(a)], I, R, H_r)], H). \end{aligned}$$

Observe that in both rules, the holds-time of q^{*1} is determined by the dualized goal in the body, i.e., $fluent(not\_\#r(q, [cons(a)]), I, O, H)$ in case of the first rule, and $not\_consider(a, R, O, H)$ in case of the second. Because the final solution in O is obtained from the intermediate contexts of the preceding positive goals, the reserved predicate $verify\_pos(Pos, H)$ ensures that none of the positive goals in Pos were subsequently supervened by their complements at some time up to H.

With regard to $expect/1$, we have the dual rules:

$$not\_expect(A, I, O, H) \leftarrow expect^{*1}(A, I, O, H). \qquad expect^{*1}(A, I, I, H) \leftarrow A \neq a.$$

[1] In general, if q is defined by n rules, then not_q is obtained by falsifying each of these n rules, i.e., it is defined as the conjunction of $q^{*1}, \ldots, q^{*n}$ and relays the ongoing abductive solution from q^{*i} to $q^{*(i+1)}$ via abductive contexts. The holds-time of not_q is obtained as in the positive transform, i.e., via reserved predicate $latest/2$ from each holds-time of inertial dualized literals in $q^{*1}, \ldots, q^{*n}$.

The uninstantiated holds-time H may get instantiated later, possibly in conjunction with other goals, or if it does not, eventually so by the actual query time. The input context I of $expect^{*1}$ is simply relayed to its output, since $A \neq a$ induces no abduction at all.

Finally, the dual of $consider(A)$ is defined as (*exp* is shorthand for *expect*):

$$\begin{array}{lcl} not\_consider(A,I,O,H) & \leftarrow & consider^{*1}(A,I,O,H). \\ consider^{*1}(A,I,O,H) & \leftarrow & not\_A(I,O,H). \\ consider^{*1}(A,I,O,H) & \leftarrow & fluent(not\_exp(A),I,O,H). \\ consider^{*1}(A,I,O,H) & \leftarrow & fluent(exp(A),I,R,H_e), fluent(exp\_not(A),R,O,H), \\ & & verify\_lits([exp(A,I,R,H_e)],H). \end{array}$$

In the first rule of $consider^{*1}$, the negation of A, i.e. *not* A, is abduced by invoking the subgoal $not\_A(I,O,H)$. This subgoal is defined via the transformation of abducibles below (say for not_a):

$$not\_a(I,O,H) \leftarrow insert(not\ a(H),I,O).$$

where $insert(A,I,O)$ is a reserved predicate that inserts abducible A into input context I, resulting in output context O, while also keeping the consistency of the context (like in *produce*/3). Again, the holds-time H may get instantiated later, like in the case of *not_expect*/4, above.

The Top-Goal Query As in EVOLP/R, updates propagation by incremental tabling is query-driven, i.e., the actual query time is used to control updates propagation by first keeping the sequence of updates pending, say in the database, and then only making active, through incremental assertions, those with timestamps up to the actual query time (if they have not yet been so made already by queries of a later timestamp). Given that an upper time limit has been set (cf. *fluent*/4 definition) and that some pending updates may be available, the system is ready for a top-goal query. The query $holds(G,I,O,Qt)$ determines the truth and the abductive solution O of goal G at query time Qt, given input context I. It is defined as:

$$\begin{array}{lcl} holds(G,I,O,Qt) & \leftarrow & activate\_pending(Qt),\ compl(G,G'), \\ & & compute(G,I,O,H,Qt,V),\ compute(G',I,O,H',Qt,V'), \\ & & verify\_holds(H,V,H',V'). \end{array}$$

where $activate\_pending(Qt)$ activates all pending updates up to Qt and $compl(G,G')$ obtains the dual complement G' from G. The reserved predicate $compute(G,I,O,H,Qt,V)$ returns the highest timestamp $H \leqslant Qt$ of goal G, and its abductive solution O, given input context I. It additionally returns the truth value V of G, obtained through the XSB predicate *call_tv*/2. This is achieved by $call\_tv(fluent(G,I,O,H),V)$, where V may be instantiated with *true* or *undefined*.[2] Finally, the predicate $verify\_holds(H,V,H',V')$ ensures that $H \geqslant H'$, and determines the truth value of G based on V and V'. Note that, when $compute(F,I,O,H,Qt,V)$ fails, by convention it returns $V = false$ with $H = 0$ (the output context O is ignored). This is merely for a technical reason, to prevent *compute*/6 failing prematurely before *verify*/4 is called.

[2] Fluents, that are not defined in the program by any rule or fact, have the truth value *undefined* at the initial time 1. In this case, the content of its input context is simply relayed to its output one. Such fluents inertially remain *undefined* at query time Qt, if they are never updated up to Qt.

4 Concluding Remarks

Related Work Abductive logic programming with destructive databases (Kowalski and Sadri 2011) is a distinct but somewhat similar and complementary to ours. It defines an agent language based on abductive logic programming and relies on the fundamental role of state transition systems in computing, realizing fluent updates by destructive assignment. Their approach differs from ours in that it defines a new language and an operational semantics, rather than taking an existing one. Moreover, it is implemented in LPA Prolog with no underlying tabling mechanisms, whereas in our work both abduction and fluent updates are managed by tabling mechanisms supported by XSB Prolog.

The connection of knowledge updates and abduction is also studied in (Sakama and Inoue 1999), where techniques for updating knowledge bases are introduced and formulated through abduction. On the other hand, the technique we propose pertains to the integration of abduction and logic program updates via tabling, with no focus on formulating updates by means of abduction. Our approach also makes use of abductive contexts, making it suitable for contextual abductive reasoning.

A dynamic abductive logic programming procedure, called LIFF, is introduced in (Sadri and Toni 2006). It allows reasoning in dynamic environments without the need to discard earlier reasoning when changes occur. Though in that work updates are assimilated into abductive logic programs, its emphasis is distinct from ours, as we do not propose a new proof procedure in that respect, but rather an implementation technique using a pre-existing theoretical basis.

Updates propagation has been well studied in the context of deductive databases, e.g., extending the SLDNF procedure for updating knowledge bases while maintaining their consistency, including integrity constraints maintenance (Teniente and Olivé 1995), using abduction for view updating (Decker 1996), as well as fixpoint approaches (Behrend 2011). Though these methods do not directly deal with tabling mechanisms for the integration of abduction and logic program updates, the approaches proposed in those works seem relevant to ours and some cross-fertilization may lead to gains.

Conclusion and Future work In this work we have proposed a novel logic programming implementation technique that aims at integrating abduction and logic program updates by means of innovative tabling mechanisms. We have based the present work on our two previously devised techniques, viz., tabled abduction (Tabdual) and query-driven updates propagation by incremental tabling (Evolp/r). The main idea of the integration is to fuse and to mutually benefit from tabling features already employed in each of our previous approaches, and is afforded by a new program transformation synthesis, and library of reserved predicates. The current implementation has simplified the transformation to some extent, e.g., using tries data structure to construct dual rules only as they are needed (like in Tabdual). Future work consists in perfecting the implementation and conducting experimental evaluation to validate the implementation. We aim at deploying it in an agent life cycle comprising hypothetical reasoning, counterfactual, and moral decision making, which we are currently pursuing.

Acknowledgements Ari Saptawijaya acknowledges the support of FCT/MEC Portugal, grant SFRH/BD/72795/2010.

References

Alferes, J. J., Brogi, A., Leite, J. A., and Pereira, L. M. 2002. Evolving logic programs. In *JELIA 2002*. LNCS, vol. 2424. Springer, 50–61.

Alferes, J. J., Pereira, L. M., and Swift, T. 2004. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming 4*, 4, 383–428.

Behrend, A. 2011. A uniform fixpoint approach to the implementation of inference methods for deductive databases. In *INAP 2011*.

Decker, H. 1996. An extension of sld by abduction and integrity maintenance for view updating in deductive databases. In *Procs. of the 1996 Joint International Conference and Symposium on Logic Programming*.

Denecker, M. and de Schreye, D. 1992. SLDNFA: An abductive procedure for normal abductive programs. In *Procs. of the Joint Intl. Conf. and Symp. on Logic Programming*. The MIT Press.

Eiter, T., Gottlob, G., and Leone, N. 1997. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science 189*, 1-2, 129–177.

Fung, T. H. and Kowalski, R. 1997. The IFF procedure for abductive logic programming. *Journal of Logic Programming 33*, 2, 151–165.

Inoue, K. and Sakama, C. 1996. A fixpoint characterization of abductive logic programs. *J. of Logic Programming 27*, 2, 107–136.

Kakas, A., Kowalski, R., and Toni, F. 1998. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. Gabbay, C. Hogger, and J. Robinson, Eds. Vol. 5. Oxford U. P.

Kowalski, R. and Sadri, F. 2011. Abductive logic programming agents with destructive databases. *Annals of Mathematics and Artificial Intelligence 62*, 1, 129–158.

Pereira, L. M., Dell'Acqua, P., Pinto, A. M., and Lopes, G. 2013. Inspecting and preferring abductive models. In *The Handbook on Reasoning-Based Intelligent Systems*, K. Nakamatsu and L. C. Jain, Eds. World Scientific Publishers, 243–274.

Pereira, L. M., Dietz, E.-A., and Hölldobler, S. 2014. Contextual abductive reasoning with side-effects. In *ICLP 2014*.

Poole, D. L. 1988. A logical framework for default reasoning. *Artificial Intelligence 36*, 1, 27–47.

Sadri, F. and Toni, F. 2006. Interleaving belief updating and reasoning in abductive logic programming. In *ECAI 2006*. Frontiers of Artificial Intelligence and Applications (FAIA), vol. 141. IOS Press, 442–446.

Saha, D. 2006. Incremental evaluation of tabled logic programs. Ph.D. thesis, SUNY Stony Brook.

Sakama, C. and Inoue, K. 1999. Updating extended logic programs through abduction. In *LPNMR 1999*. LNAI, vol. 1730. Springer, 147–161.

Saptawijaya, A. and Pereira, L. M. 2013a. Incremental tabling for query-driven propagation of logic program updates. In *LPAR-19*. LNCS ARCoSS, vol. 8312. Springer, 694–709.

Saptawijaya, A. and Pereira, L. M. 2013b. Program updating by incremental and answer subsumption tabling. In *LPNMR 2013*. LNCS, vol. 8148. Springer, 479–484.

Saptawijaya, A. and Pereira, L. M. 2013c. Tabled abduction in logic programs. Tech. rep., CENTRIA, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa. Available at `http://centria.di.fct.unl.pt/~lmp/publications/online-papers/tabdual_lp.pdf`.

Saptawijaya, A. and Pereira, L. M. 2013d. Tabled abduction in logic programs (Technical Communication of ICLP 2013). *Theory and Practice of Logic Programming, Online Supplement 13*, 4-5.

Saptawijaya, A. and Pereira, L. M. 2014. Towards modeling morality computationally with logic programming. In *PADL 2014*. LNCS, vol. 8324. Springer, 104–119.

Satoh, K. and Iwayama, N. 2000. Computing abduction by using TMS and top-down expectation. *Journal of Logic Programming 44*, 1-3, 179–206.

Swift, T. and Warren, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *JELIA 2010*. LNCS, vol. 6341. Springer, 300–312.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming 12,* 1-2, 157–187.

Teniente, E. and Olivé, A. 1995. Updating knowledge bases while maintaining their consistency. *The VLDB Journal 4,* 2, 193–241.

On Strong and Default Negation in Answer-Set Program Updates

MARTIN SLOTA, JOÃO LEITE
CENTRIA, Universidade Nova de Lisboa

MARTIN BALÁŽ
Faculty of Mathematics, Physics and Informatics, Comenius University

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Existing semantics for answer-set program updates fall into two categories: either they consider only *strong negation* in heads of rules, or they primarily rely on *default negation* in heads of rules and optionally provide support for strong negation by means of a syntactic transformation.

In this paper we pinpoint the limitations of both these approaches and argue that both types of negation should be first-class citizens in the context of updates. We identify principles that plausibly constrain their interaction but are not simultaneously satisfied by any existing rule update semantics. Then we extend one of the most advanced semantics with direct support for strong negation and show that it satisfies the outlined principles as well as a variety of other desirable properties.

KEYWORDS: answer-set programming, updates, strong negation, default negation, well-supported models

1 Introduction

The increasingly common use of rule-based knowledge representation languages in dynamic and information-rich contexts, such as the Semantic Web (Berners-Lee et al. 2001), requires standardised support for updates of knowledge represented by rules. Answer-set programming (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991) forms the natural basis for investigation of rule updates, and various approaches to answer-set program updates have been explored throughout the last 15 years (Leite and Pereira 1998; Alferes et al. 2000; Eiter et al. 2002; Leite 2003; Sakama and Inoue 2003; Alferes et al. 2005; Banti et al. 2005; Zhang 2006; Šefránek 2006; Delgrande et al. 2007; Osorio and Cuevas 2007; Šefránek 2011; Krümpelmann 2012).

The most straightforward kind of conflict arising between an original rule and its update occurs when the original conclusion logically contradicts the newer one. Though the technical realisation and final result may differ significantly, depending on the particular rule update semantics, this kind of conflict is resolved by letting the newer rule prevail over the older one. Actually, under most semantics, this is also the *only* type of conflict that is subject to automatic resolution (Leite and Pereira 1998; Alferes et al. 1998; Alferes et al. 2000; Eiter et al. 2002; Alferes et al. 2005; Banti et al. 2005; Delgrande et al. 2007; Osorio and Cuevas 2007).

From this perspective, allowing for both *strong* and *default negation* to appear in heads of rules is essential for an expressive and universal rule update framework (Leite 2003). While strong negation is the natural candidate here, used to express that an atom *becomes explicitly*

false, default negation allows for more fine-grained control: the atom only *ceases to be true*, but its truth value may not be known after the update. The latter also makes it possible to move between any pair of epistemic states by means of updates, as illustrated in the following example:

Example 1.1 (*Railway crossing (Leite 2003)*)
Suppose that we use the following logic program to choose an action at a railway crossing:

$$\mathsf{cross} \leftarrow \neg\mathsf{train}. \qquad \mathsf{wait} \leftarrow \mathsf{train}. \qquad \mathsf{listen} \leftarrow {\sim}\mathsf{train}, {\sim}\neg\mathsf{train}.$$

The intuitive meaning of these rules is as follows: one should cross if there is evidence that no train is approaching; wait if there is evidence that a train is approaching; listen if there is no such evidence. Consider a situation where a train is approaching, represented by the fact $(\mathsf{train}.)$. After this train has passed by, we want to update our knowledge to an epistemic state where we lack evidence with regard to the approach of a train. If this was accomplished by updating with the fact $(\neg\mathsf{train}.)$, we would cross the tracks at the subsequent state, risking being killed by another train that was approaching. Therefore, we need to express an update stating that all past evidence for an atom is to be removed, which can be accomplished by allowing default negation in heads of rules. In this scenario, the intended update can be expressed by the fact $({\sim}\mathsf{train}.)$.

Concerning the support of negation in rule heads, existing rule update semantics fall into two categories: those that only allow for strong negation, and those that primarily consider default negation. As illustrated above, the former are unsatisfactory as they render many belief states unreachable by updates. As for the latter, they optionally provide support for strong negation by means of a syntactic transformation. Two such transformations are known, both based on the principle of coherence: if an atom p is true, its strong negation $\neg p$ cannot be true simultaneously, so ${\sim}\neg p$ must be true, and also vice versa, if $\neg p$ is true, then so is ${\sim}p$. The first transformation, introduced in (Alferes and Pereira 1996), encodes this principle directly by adding, to both the original program and its update, the following two rules for every atom p: ${\sim}\neg p \leftarrow p.$ and ${\sim}p \leftarrow \neg p$. This way, every conflict between an atom p and its strong negation $\neg p$ directly translates into two conflicts between the objective literals p, $\neg p$ and their default negations. However, the added rules lead to undesired side effects that stand in direct opposition with basic principles underlying updates. Specifically, despite the fact that the empty program does not encode any change in the modelled world, the stable models assigned to a program may change after an update by the empty program. This undesired behaviour is addressed in an alternative transformation from (Leite 2003) that encodes the coherence principle more carefully. Nevertheless, this transformation also leads to undesired consequences, as demonstrated in the following example:

Example 1.2 (*Faulty sensor*)
Suppose that we collect data from sensors and multiple sensors are used to supply information about the critical fluent p. In case of a malfunction of one of the sensors, we may end up with an inconsistent logic program consisting of the following two facts: $p.$ and $\neg p.$ At this point, no stable model of the program exists. If a problem is found in the sensor that supplied the first fact $(p.)$, after the sensor is repaired, this information needs to be reset by updating the program with the fact $({\sim}p.)$. Following the common pattern in rule updates, where recovery from conflicting states is always possible, this update should be sufficient to assign a stable model to the updated program. However, the transformational semantics for strong negation of (Leite 2003) still does not provide any stable model – we remain without a valid epistemic state when one should exist.

In this paper we address the combination of strong and default negation in the context of rule updates. We formulate a generic desirable principle that is violated by the existing approaches.

Then we show how two distinct definitions of one of the most well-behaved rule update semantics (Alferes et al. 2005; Banti et al. 2005) can be equivalently extended with support for strong negation while satisfying the formulated principle and retaining the formal and computational properties of the original semantics. Our main contributions are as follows: a) based on Example 1.2, we introduce the *early recovery principle* that captures circumstances under which a stable model after a rule update should exist; b) we extend the *well-supported semantics for rule updates* (Banti et al. 2005) with direct support for strong negation; c) we define a fixpoint characterisation of the new semantics, based on the *refined dynamic stable model* semantics for rule updates (Alferes et al. 2005); d) we show that the defined semantics enjoy the early recovery principle as well as a range of desirable properties for rule updates known from the literature.

This paper is organised as follows: In Sect. 2 we present logic programs, generalise the well-supported semantics from the class of normal programs to extended ones and define the rule update semantics from (Alferes et al. 2005; Banti et al. 2005). In Sect. 3, we establish the early recovery principle, define the new rule update semantics for strong negation and show that it satisfies the principle. In Sect. 4 we introduce other established rule update principles and show that the proposed semantics satisfies them. We discuss our findings and conclude in Sect. 5. An extended version of this paper with all the proofs is available as (Slota et al. 2014).

2 Background

Logic Programs We assume that a countable set of propositional atoms $\mathcal{A}$ is given and fixed. An *objective literal* is an atom $p \in \mathcal{A}$ or its strong negation $\neg p$. We denote the set of all objective literals by $\mathcal{L}$. A *default literal* is an objective literal preceded by $\sim$ denoting default negation. A *literal* is either an objective or a default literal. We denote the set of all literals by $\mathcal{L}^*$. As a convention, double negation is absorbed, so that $\neg\neg p$ denotes the atom p and $\sim\sim l$ denotes the objective literal l. Given a set of literals S, we introduce the following notation: $S^+ = \{ l \in \mathcal{L} \mid l \in S \}$, $S^- = \{ l \in \mathcal{L} \mid \sim l \in S \}$, $\sim S = \{ \sim L \mid L \in S \}$. An *extended rule* is a pair $\pi = (\mathsf{H}_\pi, \mathsf{B}_\pi)$ where H_π is a literal, referred to as the *head of* π, and B_π is a finite set of literals, referred to as the *body of* π. Usually we write π as $(\mathsf{H}_\pi \leftarrow \mathsf{B}_\pi^+, \sim \mathsf{B}_\pi^-.)$. A *generalised rule* is an extended rule that contains no occurrence of $\neg$, i.e., its head and body consist only of atoms and their default negations. A *normal rule* is a generalised rule that has an atom in the head. A *fact* is an extended rule whose body is empty and a *tautology* is any extended rule π such that $\mathsf{H}_\pi \in \mathsf{B}_\pi$. An *extended (generalised, normal) program* is a set of extended (generalised, normal) rules. An *interpretation* is a consistent subset of the set of objective literals, i.e., a subset of $\mathcal{L}$ not containing both p and $\neg p$ for any atom p. The satisfaction of an objective literal l, default literal $\sim l$, set of literals S, extended rule π and extended program P in an interpretation J is defined as usual: $J \models l$ iff $l \in J$; $J \models \sim l$ iff $l \notin J$; $J \models S$ iff $J \models L$ for all $L \in S$; $J \models \pi$ iff $J \models \mathsf{B}_\pi$ implies $J \models \mathsf{H}_\pi$; $J \models P$ iff $J \models \pi$ for all $\pi \in P$. Also, J is a *model of* P if $J \models P$, and P is *consistent* if it has a model.

Definition 2.1 (*Stable model*)
Let P be an extended program. The set $[\![P]\!]_{\mathsf{SM}}$ of *stable models of* P consists of all interpretations J such that $J^* = \mathsf{least}(P \cup \mathsf{def}(J))$ where $\mathsf{def}(J) = \{ \sim l. \mid l \in \mathcal{L} \setminus J \}$, $J^* = J \cup \sim(\mathcal{L} \setminus J)$ and $\mathsf{least}(\cdot)$ denotes the least model of the argument program with all literals treated as propositional atoms.

A *level mapping* is a function that maps every atom to a natural number. Also, for any default literal $\sim p$, where $p \in \mathcal{A}$, and finite set of atoms and their default negations S, $\ell(\sim p) = \ell(p)$, $\ell^{\downarrow}(S) = \min \{ \ell(L) \mid L \in S \}$ and $\ell^{\uparrow}(S) = \max \{ \ell(L) \mid L \in S \}$.

Definition 2.2 (*Well-supported model of a normal program*)
Let P be a normal program and ℓ a level mapping. An interpretation $J \subseteq \mathcal{A}$ is a *well-supported model of P w.r.t. ℓ* if the following conditions are satisfied: 1. J is a model of P and 2. For every atom $p \in J$ there exists a rule $\pi \in P$ such that $\mathsf{H}_\pi = p \wedge J \models \mathsf{B}_\pi \wedge \ell(\mathsf{H}_\pi) > \ell^{\uparrow}(\mathsf{B}_\pi)$. The set $[\![P]\!]_{\mathsf{WS}}$ of *well-supported models of P* consists of all interpretations $J \subseteq \mathcal{A}$ such that J is a well-supported model of P w.r.t. some level mapping.

Proposition 2.3 (*(Fages 1991)*)
Let P be a normal program. Then, $[\![P]\!]_{\mathsf{WS}} = [\![P]\!]_{\mathsf{SM}}$.

Well-supported Models for Extended Programs The well-supported models for normal logic programs can be generalised in a straightforward manner to deal with strong negation while maintaining their tight relationship with stable models (c.f. Proposition 2.3). This will come useful when we discuss adding support for strong negation to semantics for rule updates. We extend level mappings from atoms and their default negations to all literals: An *(extended) level mapping* ℓ maps every objective literal to a natural number. Also, for any default literal $\sim l$ and finite set of literals S, $\ell(\sim l) = \ell(p)$, $\ell^{\downarrow}(S) = \min\{\ell(L) \mid L \in S\}$ and $\ell^{\uparrow}(S) = \max\{\ell(L) \mid L \in S\}$.

Definition 2.4 (*Well-supported model of an extended program*)
Let P be an extended program and ℓ a level mapping. An interpretation J is a *well-supported model of P w.r.t. ℓ* if the following conditions are satisfied: 1. J is a model of P and 2. For every objective literal $l \in J$ there exists a rule $\pi \in P$ such that $\mathsf{H}_\pi = l \wedge J \models \mathsf{B}_\pi \wedge \ell(\mathsf{H}_\pi) > \ell^{\uparrow}(\mathsf{B}_\pi)$. The set $[\![P]\!]_{\mathsf{WS}}$ of *well-supported models of P* consists of all interpretations J such that J is a well-supported model of P w.r.t. some level mapping.

Proposition 2.5
Let P be an extended program. Then, $[\![P]\!]_{\mathsf{WS}} = [\![P]\!]_{\mathsf{SM}}$.

Rule Updates Rule update semantics assign stable models to a pair or sequence of programs where each represents an update of the preceding ones. Formally, a *dynamic logic program* (DLP) is a finite sequence of extended programs and by $\mathsf{all}(\boldsymbol{P})$ we denote the multiset of all rules in the components of $\boldsymbol{P}$. A rule update semantics S assigns a *set of S-models*, denoted by $[\![\boldsymbol{P}]\!]_{\mathsf{S}}$, to $\boldsymbol{P}$.

We focus on semantics based on the causal rejection principle (Leite and Pereira 1998; Alferes et al. 2000; Eiter et al. 2002; Leite 2003; Alferes et al. 2005; Banti et al. 2005; Osorio and Cuevas 2007) which states that a rule is *rejected* if it is in a direct conflict with a more recent rule. The basic conflict between rules π and σ occurs when their heads are complementary, i.e. when $\mathsf{H}_\pi = \sim\mathsf{H}_\sigma$. Based on such conflicts and on a stable model candidate, a *set of rejected rules* can be determined and it can be verified that the candidate is indeed stable w.r.t. the remaining rules.

We define the most mature of these semantics, providing two equivalent definitions: the *refined dynamic stable models* (Alferes et al. 2005), or *RD-semantics*, defined using a fixpoint equation, and the *well-supported models* (Banti et al. 2005), or *WS-semantics*, based on level mappings.

Definition 2.6 (*RD-semantics (Alferes et al. 2005)*)
Let $\boldsymbol{P} = \langle P_i \rangle_{i<n}$ be a DLP without strong negation. Given an interpretation J, the multisets of rejected rules $\mathsf{rej}_{\geqslant}(\boldsymbol{P},J)$ and of default assumptions $\mathsf{def}(\boldsymbol{P},J)$ are defined as follows:

$$\mathsf{rej}_{\geqslant}(\boldsymbol{P},J) = \{\, \pi \in P_i \mid i < n \wedge \exists j \geqslant i\, \exists \sigma \in P_j : \mathsf{H}_\pi = \sim\mathsf{H}_\sigma \wedge J \models \mathsf{B}_\sigma \,\} \quad ,$$
$$\mathsf{def}(\boldsymbol{P},J) = \{\, (\sim l.) \mid l \in \mathcal{L} \wedge \neg(\exists \pi \in \mathsf{all}(\boldsymbol{P}) : \mathsf{H}_\pi = l \wedge J \models \mathsf{B}_\pi) \,\} \quad .$$

Let J^* and $\mathsf{least}(\cdot)$ be defined as before. The set $[\![\boldsymbol{P}]\!]_{\mathsf{RD}}$ of *RD-models of* $\boldsymbol{P}$ consists of all interpretations J such that $J^* = \mathsf{least}\big([\mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}_{\geqslant}(\boldsymbol{P},J)] \cup \mathsf{def}(\boldsymbol{P},J)\big)$.

Definition 2.7 (*WS-semantics (Banti et al. 2005)*)
Let $\boldsymbol{P} = \langle P_i \rangle_{i<n}$ be a DLP without strong negation. Given an interpretation J and a level mapping ℓ, the multiset of rejected rules $\mathsf{rej}_{\ell}(\boldsymbol{P},J)$ is defined as follows:

$$\mathsf{rej}_{\ell}(\boldsymbol{P},J) = \{\, \pi \in P_i \mid i < n \wedge \exists j > i\ \exists \sigma \in P_j : \mathsf{H}_{\pi} = {\sim}\mathsf{H}_{\sigma} \wedge J \models \mathsf{B}_{\sigma} \wedge \ell(\mathsf{H}_{\sigma}) > \ell^{\uparrow}(\mathsf{B}_{\sigma}) \,\} \quad .$$

The set $[\![\boldsymbol{P}]\!]_{\mathsf{WS}}$ of *WS-models of* $\boldsymbol{P}$ consists of all interpretations J such that for some level mapping ℓ, the following conditions are satisfied: 1. J is a model of $\mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}_{\ell}(\boldsymbol{P},J)$ and 2. For every $l \in J$ there exists some rule $\pi \in \mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}_{\ell}(\boldsymbol{P},J)$ such that $\mathsf{H}_{\pi} = l \wedge J \models \mathsf{B}_{\pi} \wedge \ell(\mathsf{H}_{\pi}) > \ell^{\uparrow}(\mathsf{B}_{\pi})$.

Unlike most other rule update semantics, these semantics can properly deal with tautological and other irrelevant updates, as illustrated in the following example:

Example 2.8 (*Irrelevant updates*)
Consider the DLP $\boldsymbol{P} = \langle P, U \rangle$ with $P = \{\mathsf{day} \leftarrow {\sim}\mathsf{night}.,\ \mathsf{night} \leftarrow {\sim}\mathsf{day}.,\ \mathsf{stars} \leftarrow \mathsf{night}, {\sim}\mathsf{cloudy}.,\ {\sim}\mathsf{stars}.\}$, and $U = \{\, \mathsf{stars} \leftarrow \mathsf{stars}.\,\}$. Program P has the single stable model $J_1 = \{\, \mathsf{day} \,\}$ and U contains a single tautological rule, i.e. it does not encode any change in the modelled domain. Thus, we expect that $\boldsymbol{P}$ also has the single stable model J_1. However, many rule update semantics, such as those introduced in (Leite and Pereira 1998; Alferes et al. 2000; Eiter et al. 2002; Leite 2003; Sakama and Inoue 2003; Zhang 2006; Osorio and Cuevas 2007; Delgrande et al. 2007; Krümpelmann 2012), are sensitive to this or other tautological updates, introducing or eliminating models of the original program.

In this case, the unwanted model candidate is $J_2 = \{\, \mathsf{night}, \mathsf{stars} \,\}$ and it is neither an RD- nor a WS-model of $\boldsymbol{P}$, though the reasons for this are technically different under these two semantics. It is not difficult to verify that, given an arbitrary level mapping ℓ, the set of default assumptions and the respective sets of rejected rules are as follows: $\mathsf{def}(\boldsymbol{P},J_2) = \{\, ({\sim}\mathsf{cloudy}.), ({\sim}\mathsf{day}.) \,\}$, $\mathsf{rej}_{\geqslant}(\boldsymbol{P},J_2) = \{\, (\mathsf{stars} \leftarrow \mathsf{night}, {\sim}\mathsf{cloudy}.), ({\sim}\mathsf{stars}.) \,\}$, and $\mathsf{rej}_{\ell}(\boldsymbol{P},J_2) = \emptyset$. Note that $\mathsf{rej}_{\ell}(\boldsymbol{P},J_2)$ is empty because, independently of ℓ, no rule π in U satisfies the condition $\ell(\mathsf{H}_{\pi}) > \ell^{\uparrow}(\mathsf{B}_{\pi})$, so there is no rule that could reject another rule. Thus, the atom stars belongs to J_2^* but does not belong to $\mathsf{least}([\mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}_{\geqslant}(\boldsymbol{P},J_2)] \cup \mathsf{def}(\boldsymbol{P},J_2))$, so J_2 is not an RD-model of $\boldsymbol{P}$. Furthermore, no model of $\mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}_{\ell}(\boldsymbol{P},J_2)$ contains stars, so J_2 cannot be a WS-model of $\boldsymbol{P}$.

Furthermore, the resilience of RD- and WS-semantics is not limited to empty and tautological updates, but extends to other irrelevant updates as well (Alferes et al. 2005; Banti et al. 2005). For example, consider the DLP $\boldsymbol{P}' = \langle P, U' \rangle$ where $U' = \{\, (\mathsf{stars} \leftarrow \mathsf{venus}.), (\mathsf{venus} \leftarrow \mathsf{stars}.) \,\}$. Though the updating program contains non-tautological rules, it does not provide a bottom-up justification of any model other than J_1 and, indeed, J_1 is the only RD- and WS-model of $\boldsymbol{P}'$.

We also note that the two presented semantics for DLPs without strong negation provide the same result regardless of the particular DLP to which they are applied.

Proposition 2.9 (*(Banti et al. 2005)*)
Let $\boldsymbol{P}$ be a DLP without strong negation. Then, $[\![\boldsymbol{P}]\!]_{\mathsf{WS}} = [\![\boldsymbol{P}]\!]_{\mathsf{RD}}$.

In case of the stable model semantics for a single program, strong negation can be reduced away by treating all objective literals as atoms and adding, for each atom p, the integrity constraint $(\leftarrow p, \neg p.)$ to the program (Gelfond and Lifschitz 1991). However, this transformation does not serve its purpose when adding support for strong negation to causal rejection semantics for DLPs because integrity constraints have empty heads, so according to these rule

update semantics, they cannot be used to reject any other rule. For example, a DLP such as $\langle \{ p., \neg p. \}, \{ p. \} \rangle$ would remain without a stable model even though the DLP $\langle \{ p., \sim p. \}, \{ p. \} \rangle$ does have a stable model. To capture the conflict between opposite objective literals l and $\neg l$ in a way that is compatible with causal rejection semantics, a slightly modified syntactic transformation can be performed, translating such conflicts into conflicts between objective literals and their default negations. Two such transformations have been suggested in the literature (Alferes and Pereira 1996; Leite 2003), both based on the principle of coherence. For any extended program P and DLP $\boldsymbol{P} = \langle P_i \rangle_{i<n}$ they are defined as follows:

$P^\dagger = P \cup \{ \sim\neg l \leftarrow l. \mid l \in \mathcal{L} \}$, $\boldsymbol{P}^\dagger = \langle P_i^\dagger \rangle_{i<n}$,

$P^\ddagger = P \cup \{ \sim\neg \mathsf{H}_\pi \leftarrow \mathsf{B}_\pi. \mid \pi \in P \wedge \mathsf{H}_\pi \in \mathcal{L} \}$, $\boldsymbol{P}^\ddagger = \langle P_i^\ddagger \rangle_{i<n}$.

These transformations lead to four possibilities for defining the semantics of an arbitrary DLP $\boldsymbol{P}$: $[\![\boldsymbol{P}^\dagger]\!]_{\mathsf{RD}}$, $[\![\boldsymbol{P}^\ddagger]\!]_{\mathsf{RD}}$, $[\![\boldsymbol{P}^\dagger]\!]_{\mathsf{WS}}$ and $[\![\boldsymbol{P}^\ddagger]\!]_{\mathsf{WS}}$. We discuss these in the following section.

3 Direct Support for Strong Negation in Rule Updates

The problem with existing semantics for strong negation in rule updates is that semantics based on the first transformation ($\boldsymbol{P}^\dagger$) assign too many models to some DLPs, while those based on the second transformation ($\boldsymbol{P}^\ddagger$) sometimes do not assign any model to a DLP that should have one.

Example 3.1 (*Undesired side effects of the first transformation*)
Consider the DLP $\boldsymbol{P}_1 = \langle P, U \rangle$ where $P = \{ p., \neg p. \}$ and $U = \emptyset$. Since P has no stable model and U does not encode any change in the represented domain, it should follow that $\boldsymbol{P}_1$ has no stable model either. However, $[\![\boldsymbol{P}_1^\dagger]\!]_{\mathsf{RD}} = [\![\boldsymbol{P}_1^\dagger]\!]_{\mathsf{WS}} = \{ \{ p \}, \{ \neg p \} \}$, i.e. two models are assigned to $\boldsymbol{P}_1$ when using the first transformation to add support for strong negation. To verify this, observe that $\boldsymbol{P}_1^\dagger = \langle P^\dagger, U^\dagger \rangle$ where $P^\dagger = \{ p., \neg p., \sim p \leftarrow \neg p., \sim\neg p \leftarrow p. \}$ and $U^\dagger = \{ \sim p \leftarrow \neg p., \sim\neg p \leftarrow p. \}$. Consider $J_1 = \{ p \}$. Then, we have $\mathsf{rej}_{\geqslant}(\boldsymbol{P}_1^\dagger, J_1) = \{ \neg p., \sim\neg p \leftarrow p. \}$ and $\mathsf{def}(\boldsymbol{P}_1^\dagger, J_1) = \emptyset$, so it follows that $\mathit{least}([\mathsf{all}(\boldsymbol{P}_1^\dagger) \setminus \mathsf{rej}_{\geqslant}(\boldsymbol{P}_1^\dagger, J_1)] \cup \mathsf{def}(\boldsymbol{P}_1^\dagger, J_1)) = \{ p, \sim\neg p \} = J_1^*$. In other words, J_1 belongs to $[\![\boldsymbol{P}_1^\dagger]\!]_{\mathsf{RD}}$ and in an analogous fashion it can be verified that $J_2 = \{ \neg p \}$ also belongs there. A similar situation occurs with $[\![\boldsymbol{P}_1^\dagger]\!]_{\mathsf{WS}}$ since the rules that were added to the more recent program can be used to reject facts in the older one.

Thus, the problem with the first transformation is that an update by an empty program, which does not express any change in the represented domain, may affect the original semantics. This behaviour goes against basic and intuitive principles underlying updates, grounded already in the classical belief update postulates (Keller and Winslett 1985; Katsuno and Mendelzon 1991) and satisfied by virtually all belief update operations (Herzig and Rifi 1999) as well as by the vast majority of existing rule update semantics, including the original RD- and WS-semantics.

This undesired behaviour can be corrected by using the second transformation instead. The more technical reason is that it does not add any rules to a program in the sequence unless that program already contains some original rules. However, its use leads to another problem: sometimes *no model* is assigned when in fact a model should exist.

Example 3.2 (*Undesired side effects of the second transformation*)
Consider again Example 1.2, formalised as the DLP $\boldsymbol{P}_2 = \langle P, V \rangle$ where $P = \{ p., \neg p. \}$ and $V = \{ \sim p. \}$. It is reasonable to expect that since V resolves the conflict present in P, a stable model should be assigned to $\boldsymbol{P}_2$. However, $[\![\boldsymbol{P}_2^\ddagger]\!]_{\mathsf{RD}} = [\![\boldsymbol{P}_2^\ddagger]\!]_{\mathsf{WS}} = \emptyset$. To verify this, observe that $\boldsymbol{P}_2^\ddagger = \langle P^\ddagger, V^\ddagger \rangle$ where $P^\ddagger = \{ p., \neg p., \sim p., \sim\neg p. \}$ and $V^\ddagger = \{ \sim p. \}$. Given an interpretation J and level

mapping ℓ, we conclude that $\mathsf{rej}_\ell(\boldsymbol{P}_2^\ddagger, J) = \{\, p. \,\}$, so the facts $(\neg p.)$ and $(\sim\neg p.)$ both belong to the program $\mathsf{all}(\boldsymbol{P}_2^\ddagger) \setminus \mathsf{rej}_\ell(\boldsymbol{P}_2^\ddagger, J)$. Consequently, this program has no model and it follows that J cannot belong to $[\![\boldsymbol{P}_2^\ddagger]\!]_{\mathsf{WS}}$. Similarly it can be shown that $[\![\boldsymbol{P}_2^\ddagger]\!]_{\mathsf{RD}} = \emptyset$.

Based on this example, in the following we formulate a generic *early recovery principle* that formally identifies conditions under which *some* stable model should be assigned to a DLP. For the sake of simplicity, we concentrate on DLPs of length 2 which are composed of facts. We discuss a generalisation of the principle to DLPs of arbitrary length and containing other rules than just facts in Sect. 5. After introducing the principle, we define a semantics for rule updates which directly supports both strong and default negation and satisfies the principle.

We begin by defining, for every objective literal l, the sets of literals $\overline{l} = \{\sim l, \neg l\}$ and $\overline{\sim l} = \{l\}$. Intuitively, for every literal L, $\overline{L}$ denotes the set of literals that are in conflict with L. Furthermore, given two sets of facts P and U, we say that U *solves all conflicts in* P if for each pair of rules $\pi, \sigma \in P$ such that $\mathsf{H}_\sigma \in \overline{\mathsf{H}_\pi}$ there is a fact $\rho \in U$ such that either $\mathsf{H}_\rho \in \overline{\mathsf{H}_\pi}$ or $\mathsf{H}_\rho \in \overline{\mathsf{H}_\sigma}$.

Considering a rule update semantics S, the new principle simply requires that when U solves all conflicts in P, S will assign *some model* to $\langle P, U\rangle$. Formally:

Early recovery principle: If P is a set of facts and U is a consistent set of facts that solves all conflicts in P, then $[\![\langle P, U\rangle]\!]_{\mathsf{S}} \neq \emptyset$.

We conjecture that rule update semantics should generally satisfy the above principle. In contrast with the usual behaviour of belief update operators, the nature of existing rule update semantics ensures that recovery from conflict is always possible, and this principle simply formalises and sharpens the sufficient conditions for such recovery.

Our next goal is to define a semantics for rule updates that not only satisfies the outlined principle, but also enjoys other established properties of rule updates that have been identified over the years. Similarly as for the original semantics for rule updates, we provide two equivalent definitions, one based on a fixed point equation and the other one on level mappings.

To directly accommodate strong negation in the RD-semantics, we first need to look more closely at the set of rejected rules $\mathsf{rej}_{\geqslant}(\boldsymbol{P}, J)$, particularly at the fact that it allows conflicting rules within the same component of $\boldsymbol{P}$ to reject one another. This behaviour, along with the constrained set of defaults $\mathsf{def}(\boldsymbol{P}, J)$, is used to prevent tautological and other irrelevant cyclic updates from affecting the semantics. However, in the presence of strong negation, rejecting conflicting rules within the same program has undesired side effects. For example, the early recovery principle requires that some model be assigned to the DLP $\langle \{\, p., \neg p. \,\}, \{\sim p\}\rangle$ from Example 3.2, but if the rules in the initial program reject each other, then the only possible stable model to assign is $\emptyset$. However, such a stable model would violate the causal rejection principle since it does not satisfy the initial rule $(\neg p.)$ and there is no rule in the updating program that overrides it.

To overcome the limitations of this approach to the prevention of tautological updates, we disentangle rule rejection per se from ensuring that rejection is done without cyclic justifications. We introduce the set of rejected rules $\mathsf{rej}^{\neg}_{>}(\boldsymbol{P}, S)$ which directly supports strong negation and does not allow for rejection within the same program. Prevention of cyclic rejections is done separately by using a customised immediate consequence operator $T_{\boldsymbol{P},J}$. Given a stable model candidate J, instead of verifying that J^* is the least fixed point of the usual consequence operator, as done in the RD-semantics using $\mathsf{least}(\cdot)$, we verify that J^* is the least fixed point of $T_{\boldsymbol{P},J}$.

Definition 3.3 (*Extended* RD*-semantics*)

Let $\boldsymbol{P} = \langle P_i \rangle_{i<n}$ be a DLP. For an interpretation J and set of literals S, the multiset of rejected rules $\mathsf{rej}^{\neg}_{>}(\boldsymbol{P},S)$, the remainder $\mathsf{rem}(\boldsymbol{P},S)$ and the consequence operator $T_{\boldsymbol{P},J}$ are defined as follows:

$$\mathsf{rej}^{\neg}_{>}(\boldsymbol{P},S) = \left\{ \pi \in P_i \;\middle|\; i<n \wedge \exists j>i\, \exists \sigma \in P_j : \mathsf{H}_\sigma \in \overline{\mathsf{H}_\pi} \wedge \mathsf{B}_\sigma \subseteq S \right\},$$

$$\mathsf{rem}(\boldsymbol{P},S) = \mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}^{\neg}_{>}(\boldsymbol{P},S),$$

$$T_{\boldsymbol{P},J}(S) = \left\{ \mathsf{H}_\pi \mid \pi \in (\mathsf{rem}(\boldsymbol{P},J^*) \cup \mathsf{def}(J)) \wedge \mathsf{B}_\pi \subseteq S \wedge \neg (\exists \sigma \in \mathsf{rem}(\boldsymbol{P},S) : \mathsf{H}_\sigma \in \overline{\mathsf{H}_\pi} \wedge \mathsf{B}_\sigma \subseteq J^*) \right\}.$$

Furthermore, $T^0_{\boldsymbol{P},J}(S) = S$ and for every $k \geqslant 0$, $T^{k+1}_{\boldsymbol{P},J}(S) = T_{\boldsymbol{P},J}(T^k_{\boldsymbol{P},J}(S))$. The set $[\![\boldsymbol{P}]\!]^{\neg}_{\mathsf{RD}}$ of *extended RD-models of* $\boldsymbol{P}$ consists of all interpretations J such that $J^* = \bigcup_{k\geqslant 0} T^k_{\boldsymbol{P},J}(\emptyset)$.

Adding support for strong negation to the WS-semantics is done by modifying the set of rejected rules $\mathsf{rej}_\ell(\boldsymbol{P},J)$ to account for the new type of conflict. Additionally, to ensure that rejection of a literal L cannot be based on the assumption that some conflicting literal $L' \in \overline{L}$ is true, a rejecting rule σ must satisfy the stronger condition $\ell^{\downarrow}(\overline{L}) > \ell^{\uparrow}(\mathsf{B}_\sigma)$. Finally, to prevent defeated rules from affecting the resulting models, we require that all supporting rules belong to $\mathsf{rem}(\boldsymbol{P},J^*)$.

Definition 3.4 (*Extended WS-semantics*)
Let $\boldsymbol{P} = \langle P_i \rangle_{i<n}$ be a DLP. Given an interpretation J and a level mapping ℓ, the multiset of rejected rules $\mathsf{rej}^{\neg}_{\ell}(\boldsymbol{P},J)$ is defined by:

$$\mathsf{rej}^{\neg}_{\ell}(\boldsymbol{P},J) = \left\{ \pi \in P_i \mid i<n \wedge \exists j>i\, \exists \sigma \in P_j : \mathsf{H}_\sigma \in \overline{\mathsf{H}_\pi} \wedge J \models \mathsf{B}_\sigma \wedge \ell^{\downarrow}(\overline{\mathsf{H}_\pi}) > \ell^{\uparrow}(\mathsf{B}_\sigma) \right\}.$$

The set $[\![\boldsymbol{P}]\!]^{\neg}_{\mathsf{WS}}$ of *extended WS-models of* $\boldsymbol{P}$ consists of all interpretations J such that for some level mapping ℓ, the following conditions are satisfied: 1. J is a model of $\mathsf{all}(\boldsymbol{P}) \setminus \mathsf{rej}^{\neg}_{\ell}(\boldsymbol{P},J)$ and 2. For every $l \in J$ there exists some rule $\pi \in \mathsf{rem}(\boldsymbol{P},J^*)$ such that $\mathsf{H}_\pi = l \wedge J \models \mathsf{B}_\pi \wedge \ell(\mathsf{H}_\pi) > \ell^{\uparrow}(\mathsf{B}_\pi)$.

The following theorems establishe that the two defined semantics are equivalent, that they coincide with the original on DLPs without strong negation, and, unlike the transformational semantics for strong negation, the new semantics satisfy the early recovery principle.

Theorem 3.5
Let $\boldsymbol{P}_1$ be a DLP and $\boldsymbol{P}_2$ be a DLP without strong negation. Then, $[\![\boldsymbol{P}_1]\!]^{\neg}_{\mathsf{WS}} = [\![\boldsymbol{P}_1]\!]^{\neg}_{\mathsf{RD}}$ and $[\![\boldsymbol{P}_2]\!]^{\neg}_{\mathsf{WS}} = [\![\boldsymbol{P}_2]\!]^{\neg}_{\mathsf{RD}} = [\![\boldsymbol{P}_2]\!]_{\mathsf{WS}} = [\![\boldsymbol{P}_2]\!]_{\mathsf{RD}}$.

Theorem 3.6
The extended RD-semantics and extended WS-semantics satisfy the early recovery principle.

4 Properties

The various approaches to rule updates (Leite and Pereira 1998; Alferes et al. 2000; Eiter et al. 2002; Leite 2003; Sakama and Inoue 2003; Alferes et al. 2005; Banti et al. 2005; Zhang 2006; Šefránek 2006; Osorio and Cuevas 2007; Delgrande et al. 2007; Šefránek 2011; Krümpelmann 2012) share a number of basic characteristics, significantly differing in their technical realisation and classes of supported inputs, and desirable properties such as immunity to tautologies are violated by many of them. Table 1 lists several generic properties proposed for rule updates that have been identified and formalised throughout the years (Leite and Pereira 1998; Eiter et al. 2002; Leite 2003; Alferes et al. 2005). The rule update semantics we defined in the previous section enjoys all of them, while retaining the same computational complexity as the stable models.

Theorem 4.1
The extended RD-semantics and extended WS-semantics satisfy all properties listed in Table 1.

Table 1. Desirable properties of rule update semantics

Generalisation of stable models	$[\![\langle P\rangle]\!]_{\mathsf{s}} = [\![P]\!]_{\mathsf{SM}}$.
Primacy of new information	If $J \in [\![\langle P_i\rangle_{i<n}]\!]_{\mathsf{s}}$, then $J \models P_{n-1}$.
Fact update	A sequence of consistent sets of facts $\langle P_i\rangle_{i<n}$ has the single model $\{\, l \in \mathfrak{L} \mid \exists i < n : (l.) \in P_i \wedge (\forall j > i : \{\neg l., \sim l.\} \cap P_j = \emptyset) \,\}$.
Support	If $J \in [\![\boldsymbol{P}]\!]_{\mathsf{s}}$ and $l \in J$, then there is some rule $\pi \in \mathsf{all}(\boldsymbol{P})$ such that $\mathsf{H}_\pi = l$ and $J \models \mathsf{B}_\pi$.
Idempotence	$[\![\langle P,P\rangle]\!]_{\mathsf{s}} = [\![\langle P\rangle]\!]_{\mathsf{s}}$.
Absorption	$[\![\langle P,U,U\rangle]\!]_{\mathsf{s}} = [\![\langle P,U\rangle]\!]_{\mathsf{s}}$.
Augmentation	If $U \subseteq V$, then $[\![\langle P,U,V\rangle]\!]_{\mathsf{s}} = [\![\langle P,V\rangle]\!]_{\mathsf{s}}$.
Non-interference	If alphabets of U and V are disjoint, then $[\![\langle P,U,V\rangle]\!]_{\mathsf{s}} = [\![\langle P,V,U\rangle]\!]_{\mathsf{s}}$.
Immunity to empty updates	If $P_j = \emptyset$, then $[\![\langle P_i\rangle_{i<n}]\!]_{\mathsf{s}} = \left[\!\!\left[\langle P_i\rangle_{i<n \wedge i\neq j}\right]\!\!\right]_{\mathsf{s}}$.
Immunity to tautologies	If $\langle Q_i\rangle_{i<n}$ is a sequence of sets of tautologies, then $[\![\langle P_i \cup Q_i\rangle_{i<n}]\!]_{\mathsf{s}} = [\![\langle P_i\rangle_{i<n}]\!]_{\mathsf{s}}$.
Causal rejection principle	For every $i < n$, $\pi \in P_i$ and $J \in [\![\langle P_i\rangle_{i<n}]\!]_{\mathsf{s}}$, if $J \not\models \pi$, then there exists some $\sigma \in P_j$ with $j > i$ such that $\mathsf{H}_\sigma \in \overline{\mathsf{H}_\pi}$ and $J \models \mathsf{B}_\sigma$.

Theorem 4.2
Let $\boldsymbol{P}$ be a DLP. The problem of deciding whether some $J \in [\![\boldsymbol{P}]\!]^{\neg}_{\mathsf{WS}}$ exists is NP-complete. Given a literal L, the problem of deciding whether for all $J \in [\![\boldsymbol{P}]\!]^{\neg}_{\mathsf{WS}}$ it holds that $J \models L$ is coNP-complete.

5 Concluding Remarks

In this paper we have identified shortcomings in the existing semantics for rule updates that fully support both strong and default negation, and proposed a generic *early recovery principle* that captures them formally. Subsequently, we provided two equivalent definitions of a new semantics for rule updates. We have shown that the newly introduced rule update semantics constitutes a strict improvement upon the state of the art in rule updates as it enjoys the following combination of characteristics, unmatched by any previously existing semantics: - It allows for both strong and default negation in heads of rules, making it possible to move between any pair of epistemic states by means of updates; - It satisfies the *early recovery principle* which guarantees the existence of a model whenever all conflicts in the original program are satisfied; - It enjoys all rule update principles and desirable properties reported in Table 1; - It does not increase the computational complexity of the stable model semantics upon which it is based.

However, the early recovery principle, as it is formulated in Sect. 3, only covers a single update of a set of facts by another set of facts. Can it be generalised further without rendering it too strong? Certain caution is appropriate here, since in general the absence of a stable model can be caused by odd cycles or simply by the fundamental differences between different approaches to rule update, and the purpose of this principle is not to choose which approach to take.

Nevertheless, one generalisation that should cause no harm is the generalisation to iterated updates, i.e. to sequences of sets of facts. Another generalisation that appears very reasonable is the generalisation to *acyclic DLPs*, i.e. DLPs such that $\mathsf{all}(\boldsymbol{P})$ is an acyclic program. An acyclic program has at most one stable model, and if we guarantee that all potential conflicts within it

certainly get resolved, we can safely conclude that the rule update semantics should assign some model to it. We formalise these ideas in what follows.

A program P is *acyclic* (Apt and Bezem 1991) if for some level mapping ℓ, such that for every $l \in \mathcal{L}$, $\ell(l) = \ell(\neg l)$, and every rule $\pi \in P$ it holds that $\ell(\mathsf{H}_\pi) > \ell^{\uparrow}(\mathsf{B}_\pi)$. Given a DLP $\boldsymbol{P} = \langle P_i \rangle_{i<n}$, we say that *all conflicts in* $\boldsymbol{P}$ *are solved* if for every $i < n$ and each pair of rules $\pi, \sigma \in P_i$ such that $\mathsf{H}_\sigma \in \overline{\mathsf{H}_\pi}$ there is some $j > i$ and a fact $\rho \in P_j$ such that either $\mathsf{H}_\rho \in \overline{\mathsf{H}_\pi}$ or $\mathsf{H}_\rho \in \overline{\mathsf{H}_\sigma}$.

Generalised early recovery principle: If $\mathsf{all}(\boldsymbol{P})$ is acyclic and all conflicts in $\boldsymbol{P}$ are solved, then $[\![\boldsymbol{P}]\!]_{\mathsf{s}} \neq \emptyset$.

Note that this generalisation of the early recovery principle applies to a much broader class of DLPs than the original one. We illustrate this in the following example:

Example 5.1 (*Recovery in a stratified program*)
Consider the following programs programs P, U and V: $P = \{\, p \leftarrow q, {\sim}r.\,, {\sim}p \leftarrow s.\,, q.\,, s \leftarrow q.\,\}$, $U = \{\, \neg p.\,, r \leftarrow q.\,, \neg r \leftarrow q, s.\,\}$, and $V = \{\, {\sim}r.\,\}$. Looking more closely at program P, we see that atoms q and s are derived by the latter two rules inside it while atom r is false by default since there is no rule that could be used to derive its truth. Consequently, the bodies of the first two rules are both satisfied and as their heads are conflicting, P has no stable model. The single conflict in P is solved after it is updated by U, but then another conflict is introduced due to the latter two rules in the updating program. This second conflict can be solved after another update by V. Consequently, we expect that some stable model be assigned to the DLP $\langle P, U, V \rangle$.

The original early recovery principle does not impose this because the DLP in question has more than two components and the rules within it are not only facts. However, the DLP is acyclic, as shown by any level mapping ℓ with $\ell(p) = 3$, $\ell(q) = 0$, $\ell(r) = 2$ and $\ell(s) = 1$, so the generalised early recovery principle does apply. Furthermore, we also find the single extended RD-model of $\langle P, U, V \rangle$ is $\{\, \neg p, q, \neg r, s \,\}$, i.e. the semantics respects the stronger principle in this case.

The stronger principle is generally satisfied by the semantics introduced in this paper.

Theorem 5.2
The extended RD-semantics and extended WS-semantics satisfy the generalised early recovery principle.

Both the original and the generalised early recovery principle can guide the future addition of full support for both kinds of negations in other approaches to rule updates, such as those proposed in (Sakama and Inoue 2003; Zhang 2006; Delgrande et al. 2007; Krümpelmann 2012), making it possible to reach any belief state by updating the current program. Furthermore, adding support for strong negation is also interesting in the context of recent results on program revision and updates that are performed on the *semantic level*, ensuring syntax-independence of the respective methods (Delgrande et al. 2013; Slota and Leite 2014; Slota and Leite 2012a; Slota and Leite 2010), in the context of finding suitable condensing operators (Slota and Leite 2013), and unifying with updates in classical logic (Slota and Leite 2012b).

Acknowledgments

J. Leite was partially supported by FCT under project PTDC/EIA-CCO/121823/2010 and M. Slota under project PTDC/EIA-CCO/110921/2009. The collaboration between the co-authors resulted from the Slovak–Portuguese bilateral project supported by APVV agency under SK-PT-0028-10 and by FCT under FCT/2487/3/6/2011/S.

References

ALFERES, J. J., BANTI, F., BROGI, A., AND LEITE, J. A. 2005. The refined extension principle for semantics of dynamic logic programming. *Studia Logica 79,* 1, 7–32.

ALFERES, J. J., LEITE, J. A., PEREIRA, L. M., PRZYMUSINSKA, H., AND PRZYMUSINSKI, T. C. 1998. Dynamic logic programming. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June 2-5, 1998*, A. G. Cohn, L. K. Schubert, and S. C. Shapiro, Eds. Morgan Kaufmann, 98–111.

ALFERES, J. J., LEITE, J. A., PEREIRA, L. M., PRZYMUSINSKA, H., AND PRZYMUSINSKI, T. C. 2000. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming 45,* 1-3 (September/October), 43–70.

ALFERES, J. J. AND PEREIRA, L. M. 1996. Update-programs can update programs. In *Non-Monotonic Extensions of Logic Programming (NMELP '96), Selected Papers*, J. Dix, L. M. Pereira, and T. C. Przymusinski, Eds. Lecture Notes in Computer Science, vol. 1216. Springer, Bad Honnef, Germany, 110–131.

APT, K. R. AND BEZEM, M. 1991. Acyclic programs. *New Generation Computing 9,* 3/4, 335–364.

BANTI, F., ALFERES, J. J., BROGI, A., AND HITZLER, P. 2005. The well supported semantics for multidimensional dynamic logic programs. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005)*, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Computer Science, vol. 3662. Springer, Diamante, Italy, 356–368.

BERNERS-LEE, T., HENDLER, J., AND LASSILA, O. 2001. The semantic web. *Scientific American 284,* 5, 28–37.

DELGRANDE, J., SCHAUB, T., TOMPITS, H., AND WOLTRAN, S. 2013. A model-theoretic approach to belief change in answer set programming. *ACM Transactions on Computational Logic (TOCL) 14,* 2 (June), 14:1–14:46.

DELGRANDE, J. P., SCHAUB, T., AND TOMPITS, H. 2007. A preference-based framework for updating logic programs. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, Tempe, AZ, USA, 71–83.

EITER, T., FINK, M., SABBATINI, G., AND TOMPITS, H. 2002. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming (TPLP) 2,* 6, 721–777.

FAGES, F. 1991. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *New Generation Computing 9,* 3/4, 425–444.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Seattle, Washington, 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3-4, 365–385.

HERZIG, A. AND RIFI, O. 1999. Propositional belief base update and minimal change. *Artificial Intelligence 115,* 1, 107–138.

KATSUNO, H. AND MENDELZON, A. O. 1991. On the difference between updating a knowledge base and revising it. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, J. F. Allen, R. Fikes, and E. Sandewall, Eds. Morgan Kaufmann Publishers, Cambridge, MA, USA, 387–394.

KELLER, A. M. AND WINSLETT, M. 1985. On the use of an extended relational model to handle changing incomplete information. *IEEE Transactions on Software Engineering 11,* 7, 620–633.

KRÜMPELMANN, P. 2012. Dependency semantics for sequences of extended logic programs. *Logic Journal of the IGPL 20,* 5, 943–966.

LEITE, J. A. 2003. *Evolving Knowledge Bases*. Frontiers of Artificial Intelligence and Applications, xviii + 307 p. Hardcover, vol. 81. IOS Press.

LEITE, J. A. AND PEREIRA, L. M. 1998. Generalizing updates: From models to programs. In *Proceedings of the 3rd International Workshop on Logic Programming and Knowledge Representation (LPKR '97),*

October 17, 1997, Port Jefferson, New York, USA, J. Dix, L. M. Pereira, and T. C. Przymusinski, Eds. Lecture Notes in Computer Science, vol. 1471. Springer, 224–246.

OSORIO, M. AND CUEVAS, V. 2007. Updates in answer set programming: An approach based on basic structural properties. *Theory and Practice of Logic Programming 7,* 4, 451–479.

SAKAMA, C. AND INOUE, K. 2003. An abductive framework for computing knowledge base updates. *Theory and Practice of Logic Programming (TPLP) 3,* 6, 671–713.

ŠEFRÁNEK, J. 2006. Irrelevant updates and nonmonotonic assumptions. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA 2006)*, M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, Eds. Lecture Notes in Computer Science, vol. 4160. Springer, Liverpool, UK, 426–438.

ŠEFRÁNEK, J. 2011. Static and dynamic semantics: Preliminary report. *Mexican International Conference on Artificial Intelligence*, 36–42.

SLOTA, M., BALÁŽ, M., AND LEITE, J. 2014. On strong and default negation in logic program updates (extended version). *CoRR abs/1404.6784*.

SLOTA, M. AND LEITE, J. 2010. On semantic update operators for answer-set programs. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, H. Coelho, R. Studer, and M. Wooldridge, Eds. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press, 957–962.

SLOTA, M. AND LEITE, J. 2012a. Robust equivalence models for semantic updates of answer-set programs. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press, Rome, Italy, 158–168.

SLOTA, M. AND LEITE, J. 2012b. A unifying perspective on knowledge updates. In *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, L. F. del Cerro, A. Herzig, and J. Mengin, Eds. Lecture Notes in Computer Science, vol. 7519. Springer, 372–384.

SLOTA, M. AND LEITE, J. 2013. On condensing a sequence of updates in answer-set programming. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, F. Rossi, Ed. IJCAI/AAAI.

SLOTA, M. AND LEITE, J. 2014. The rise and fall of semantic rule updates based on se-models. *Theory and Practice of Logic Programming FirstView*, 1–39.

ZHANG, Y. 2006. Logic program-based updates. *ACM Transactions on Computational Logic 7,* 3, 421–472.

Towards Assertion-based Debugging of Higher-Order (C)LP Programs *

NATALIIA STULOVA[1] JOSÉ F. MORALES[1] MANUEL V. HERMENEGILDO[1,2]

[1] *IMDEA Software Institute*
(*e-mail:* {nataliia.stulova, josef.morales, manuel.hermenegildo}@imdea.org)
[2] *School of Computer Science, Technical University of Madrid (UPM)*
(*e-mail:* manuel.hermenegildo@upm.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Extended Abstract

Higher-order programming adds flexibility to the software development process. Within the (Constraint) Logic Programming ((C)LP) paradigm, Prolog has included higher-order constructs since the early days, and there have been many other proposals for combining the first-order kernel of (C)LP with different higher-order constructs. Many of these proposals are currently in use in different (C)LP systems and have been found very useful in programming practice, inheriting the well-known benefits of code reuse (templates), elegance, clarity, and modularization.

A number of extensions have also been proposed for (C)LP in order to enhance the process of error detection and program validation. In addition to the use of classical strong typing, a number of other approaches have been proposed which are based on the dynamic and/or static checking of user-provided, optional *assertions*. Of these, the model of (Hermenegildo et al. 2005) has perhaps had the most impact in practice and different aspects of this model have been incorporated in a number of widely-used (C)LP systems, such as Ciao, SWI, and XSB. A similar evolution is represented by the soft/gradual typing-based approaches in functional programming and the contracts-based extensions in object-oriented programming.

These two aspects, assertions and higher-order, are not independent. When higher-order constructs are introduced in the language it becomes necessary to describe properties of arguments of predicates/procedures that are themselves also predicates/procedures. While the combination of contracts and higher-order has received some attention in functional programming, within (C)LP the combination of higher-order with the previously mentioned assertion-based approaches has received comparatively little attention to date. Current Prolog systems simply use basic atomic types (i.e., stating simply that the argument is a `pred`, `callable`, etc.) to describe predicate-bearing variables. Other approaches are oriented instead to meta programming, describing meta-types but there is no notion of directionality (modes), and only a single pattern is allowed per predicate.

* Research supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2008-05624 *DOVES*, and Comunidad de Madrid TIC/1465 *PROMETIDOS-CM*.

Our work (Stulova et al. 2014) contributes towards filling this gap between higher-order (C)LP programs and assertion-based extensions for error detection and program validation. Our starting point is the Ciao assertion model, since, as mentioned before, it has been adopted at least in part in a number of the most popular (C)LP systems.

We have proposed an extension of the traditional notion of programs and derivations in order to deal with higher-order calls and we have adapted the notions of first-order conditional literals, assertions, program correctness, and run-time checking to this type of derivations. This has allowed us to revisit the traditional model in this new, higher-order context, while introducing a different formalization than the original one, which is more compact and gathers all assertion violations as opposed to just the first one, among other differences. We have defined an extension of the properties used in assertions and of the assertions themselves to higher-order, and provided corresponding semantics and results.

We have defined a new class of properties, "predicate properties" (*predprops* in short), and proposed a syntax and semantics for them. These new properties can be used in assertions for higher-order predicates to describe the properties of the higher-order arguments. We have also proposed several operational semantics for performing run-time checking of programs including predprops and provided correctness results for them.

Our predprop properties specify conditions for predicates that are independent of the usage context. This corresponds in functional programming to the notion of *tight* contract satisfaction, and it contrasts with alternative approaches such as *loose* contract satisfaction. In the latter, contracts are attached to higher-order arguments by implicit function wrappers. The scope of checking is local to the function evaluation. Although this is a reasonable and pragmatic solution, we believe that our approach is more general and more amenable to combination with static verification techniques. For example, avoiding wrappers allows us to remove checks (e.g., by static analysis) without altering the program semantics. Moreover, our approach can easily support *loose* contract satisfaction, since it is straightforward in our framework to optionally include wrappers as special predprops.

We have included the proposed predprop extensions in an experimental branch of the Ciao assertion language implementation. This has the immediate advantage, in addition to the enhanced checking, that it allows us to document higher-order programs in a much more accurate way. We have also implemented several prototypes for operational semantics with dynamic predprop checking, which are being integrated into the already existing assertion checking mechanisms for first-order assertions. Finally, we are developing analyses for static verification of assertions containing predprops.

References

Hermenegildo, M., Puebla, G., Bueno, F., and García, P. L. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2.

Stulova, N., Morales, J. F., and Hermenegildo, M. V. 2014. An Approach to Assertion-based Debugging of Higher-Order (C)LP Programs. Tech. Rep. CLIP-1/2014.0, The CLIP Lab. January. CoRR abs/1404.4246 [cs.PL].

Interclausal Logic Variables

PAUL TARAU and FAHMIDA HAMID
Department of Computer Science and Engineering
(*e-mail:* tarau@cs.unt.edu) (*e-mail:* fahmidahamid@my.unt.edu)

submitted February 14, 2014; revised April 15, 2014; accepted March 25, 2014

Abstract

Unification of logic variables instantly connects present and future observations of their value, independently of their location in the data areas of the runtime system. The paper extends this property to "interclausal logic variables", an easy to implement Prolog extension that supports instant global information exchanges without dynamic database updates. We illustrate their usefulness with two of algorithms, *graph coloring* and *minimum spanning tree*. Implementations of interclausal variables as source-level transformations and as abstract machine adaptations are given. To address the need for globally visible chained transitions of logic variables we describe a DCG-based program transformation that extends the functionality of interclausal variables.

KEYWORDS: declarative programming language constructs, Prolog implementation, logic variables, definite clause grammars, continuation passing Prolog.

1 Introduction

Referred to by Einstein as "spooky action at a distance", quantum entanglement is the fact that observation of the values of a particle's physical attributes binds instantly entangled particles to identical values independently of their physical distance.

In the field of quantum computing, entanglement plays a crucial role in designing new algorithms and communication mechanisms, as well as in fine-tuning physical realizations of quantum computing machines (Panangaden 2011).

In logic programming languages, the prototypical instance of such an "entanglement pattern" is unification of logic variables (Robinson 1965). It instantly connects present and future observations of their value, independently of their location in the data areas of the runtime system.

While indulging into deviations from the strict entanglement analogy, thinking in terms of it can clarify some interesting algorithms that are part of the "folklore" of logic programming since the early years of Prolog. For instance, a simple and elegant graph coloring algorithm is derived by using logic variables to denote colors associated to a vertex. Avoiding cycles in graph visiting algorithms, solving a knight's tour puzzle or finding a Hamiltonian circuit in a graph have also simple declarative programs exhibiting the entanglement analogy centered on unique bindings to logic variables.

We will revisit a few of these algorithms while proposing some new language constructs. The fact that they are unusually easy to implement in a language like Prolog, gives us hope that they will lead to interesting uses in everyday programming.

The paper is organized as follows. Section 2 introduces interclausal variables. Section 4 describes their implementation in the Styla Prolog system and discusses some alternative source-level and WAM-level implementations. Section 3 describes the use of interclausal variables in algorithms like graph coloring and minimum spanning tree and their use to inject dynamic code in a program without using assert operations. Section 5 discusses a source-level implementation of backtrackable assumptions using Prolog's DCG transformation. Section 6 discusses related work and section 7 concludes the paper.

2 Interclausal logic variables

A natural extension of unification seen as an instance of the entanglement pattern is to apply it to variables shared among different clauses, that we will call here *interclausal variables*.

In a given logic program we could syntactically mark such a variable `X`, shared among clauses, as `~X`, for example

```
a(~X).
b(~X).
```

The execution algorithm will then be modified to share bindings between `~X` occurring in the two clauses as in

```
?- a(10),b(V).
a(10),b(V).
V = 10.
```

At the same time, it makes sense to trail such bindings, as one would do with ordinary logic variables. This means that a query like the following would also succeed, with a different binding

```
?- a(V),b(20).
a(V),b(20).
V = 20.
```

Therefore, the semantics of interclausal variables is the same as passing them along in a shared compound term containing them as arguments, or passing them directly as additional arguments to all predicates occurring in the program. Like in the case of ordinary logic variables, their behavior on backtracking provides a form of memory reuse. At the same time, indexing of Prolog clauses provides comparable access to the shared variables as if they would be passed along in a data structure or as extra arguments.

As a result of our intended semantics, one would also expect that the interclausal variables are trailed and reset to free after the query is answered.

3 Interclausal Variables at Work

Interclausal variables can be used in Prolog facts representing (possibly large) graphs as markers associated to vertices. This assertional representation can provide scalability and memory efficiency superior to equivalent representations as a data structure.

3.1 Graph Coloring with Interclausal Variables

We will start by illustrating a use of interclausal variables on a graph coloring program, derived from a classic example exhibiting the use of logic variables as colors to be assigned to vertices.

First we define our colors:

```
color(red). color(green). color(blue).
```

Next we define our vertices with interclausal variables `~C1`..`~C6` representing the colors associated to each vertex.

```
vertex(1,~C1). vertex(2,~C2). vertex(3,~C3).
vertex(4,~C4). vertex(5,~C5). vertex(6,~C6).
```

The graph will be described as a set of edges connecting our vertices.

```
edge(1,2). edge(2,3). edge(1,3). edge(3,4). edge(4,5).
edge(5,6). edge(4,6). edge(2,5). edge(1,6).
```

The coloring algorithm will iterate over all edges to color their endpoint vertices and then collect the facts describing the colorings.

```
coloring(Vs):-
  E=edge(_,_),findall(E,E,Es),
  color_all(Es),
  V=vertex(_,_),findall(V,V,Vs).
```

The iteration over all edges ensures at each step that adjacent vertices are colored differently

```
color_all([]).
color_all([edge(X,Y)|Es]):-
   vertex(X,C), color(C),
   vertex(Y,D), color(D),
   \+(C=D),
   color_all(Es).
```

The algorithm will return multiple possible colorings on backtracking, as if the colors were passed along as additional arguments to each clause.

```
?- coloring(Vs).
Vs = [vertex(1,red),vertex(2,green),vertex(3,blue),
      vertex(4,red),vertex(5,blue),vertex(6,green)];
...
Vs = [vertex(1,blue),vertex(2,green),vertex(3,red),
      vertex(4,blue),vertex(5,red),vertex(6,green)].
```

At the end, the interclausal variables are ready for being reused, back to an unbound state:

```
?- listing(vertex).
vertex(1,~C1).
...
vertex(6,~C6).
```

Note the mild deviation from our entanglement analogy, given that (sound) negation as failure is used to ensure that colors associated to neighboring vertices are distinct.

Note also that in ASP systems (Gebser et al. 2007) or SAT-based constraint solver extensions to Prolog (Zhou 2013) that rely on grounding, interclausal variables could be introduced with the same semantics, to control combinatorial explosion that depends on the total number of distinct variables.

3.2 A Minimum Spanning Tree Algorithm using Interclausal Variables

Our next example uses interclausal variables for a variant of Kruskal's minimum spanning tree algorithm with logic variables working as markers for connected sets of edges that grow progressively until they cover the graph (assuming it is connected). It has been derived from a Prolog program using a data structure passed along between clauses and posted on Usenet by the author in 1992[1].

The algorithm proceeds by first sorting by cost the set of edges.

```
mst(NbOfVertices,Edges,MinSpanTree):-
  sort(Edges,SortedEdges),
  mst0(NbOfVertices,SortedEdges,MinSpanTree).
```

Next the program explores the set of edges, given as the second argument of the predicate `mst0/3`. At a given step, it calls the predicate `mst1/7` which decides about unifying or not the components `C1` and `C2`.

```
mst0(1,_,[]). % no more vertices left
mst0(N,[E|Es],T):- N>1,
  E=edge(_Cost,V1,V2),
  vertex(V1,C1), % C1,C2 are the components of V1,V2
  vertex(V2,C2),
  mst1(C1,C2,E,T,NewT,N,NewN),
  mst0(NewN,Es,NewT).
```

The predicate `mst1/7` checks if both endpoints of an edge are already in an incrementally grown set of connected edges, in which case it skips the edge. Otherwise, if the sets represented by `C1` and `C2` are distinct, they will be merged by unifying the variables, adding the edge to the minimum spanning tree and counting the vertex as processed. Note that we are reusing here the vertex definitions of our graph coloring program, with colors interpreted as components.

```
mst1(C1,C2,_,T,T,N,N):-C1==C2.
mst1(C1,C2,E,T,NewT,N,NewN):-C1\==C2,C1=C2,
  % Put endpoints in the same component
  T=[E|NewT],   % Add the the edge to the MST
  NewN is N-1.  % Count a new vertex
```

Finally the predicate `test_mst` tries out the algorithm on a small graph.

```
test_mst(MinSpanTree):-
  Edges = [ edge(70,1,3),edge(80,3,4),edge(90,1,5),
            edge(60,2,3),edge(20,4,5),edge(30,1,4),
            edge(40,2,5),edge(50,3,5),edge(10,1,2)
          ],
  mst(5,Edges,MinSpanTree).
```

[1] A time when such uses of logic variables were still waiting to be uncovered.

Note that an answer is returned as a list of edges ordered by cost, such that each vertex is an endpoint of at least one edge.

```
?- test_mst(Mst).
Mst = [edge(10,1,2),edge(20,4,5),edge(30,1,4),edge(50,3,5)]
```

3.3 Injecting Dynamic Code without Asserts

When used in a metavariable position, an interclausal variable can provide a lightweight alternative to the assert/retract interface to dynamic code. In a clause like a_0`:-`$a_1, \ldots$`~V`$\ldots, a_n$ the metavariable `~V` can be bound to a Prolog terms that gets "injected" in the possibly statically compiled code of the clause. In particular, injecting `~V=fail` in a clause like a_0`:-~V`$\ldots, a_n$ would temporarily disable the clause without the need to use a `retract` operation. In a different branch of the computation, one could inject `~V=true` to enable the clause.

4 Implementing Interclausal Variables

We will describe here a few mechanisms for adding support for interclausal variables to Prolog systems.

4.1 Interclausal variables in Styla

We have implemented *interclausal variables* in our *Styla* Scala-based Prolog system (Tarau 2012a) by taking advantage of its object oriented term structure and its distributed unification and term copying algorithms, designed in such a way that various subterms contribute small steps depending on their type. We have also used the fact that inheritance enables "surgical" overriding of the small methods implementing these algorithms together.

First, we have created a new type `EVar` for interclausal variables as an extension of the class of ordinary logic variables `Var`.

```
package prolog.terms
class EVar() extends Var {
  override def tcopy(dict : Copier) : Term = this.ref
}
```

We made it inherit all properties of logic variables except one: behavior on copying. The method `tcopy`, instead of creating a fresh variable, simply returns the reference `ref` of our interclausal variable. As a result, bindings of interclausal variables are shared between calls, while their unification behavior, including trailing for undoing bindings on backtracking, is inherited unchanged.

Styla uses Scala's *combinator parsing* API where only two simple modifications were needed to process our new data type.

First, we specified the regular expression

```
val evarToken: Parser[String] = """~[A-Z_]\w*""".r
```

defining that interclausal variables start with the ~ symbol and have, otherwise, the same token specification as the usual ones.

Next, we have ensured that the parser knows about them, by adding a rule associated to their token type calling the method `mkEVar`

```
def mkEVar(x: String) = {
    vars.getOrElseUpdate(x, new EVar())
}
```

Finally, a small change to the `toString` method marks with a "~" the string representation of interclausal variables. Besides helping with debugging, this is also useful as Styla keeps track of variable names in the source code and uses them in predicates like `listing/1`, when the source code of a predicate is displayed.

4.2 Source-level Implementations

Given a set of interclausal variables, one can implement them at source level simply by adding them as extra arguments to each clause of a program. This would ensure that a Datalog program remains a Datalog program after the transformation. While linear, the resulting code explosion can be avoided by adding a single variable to each clause representing a compound term, together with an `arg/3` predicate call accessing the appropriate position in it, for each interclausal variable occurring in a given clause.

4.3 WAM-level Implementations

In a way similar to BinProlog's implementation of multiple DCG streams (Dahl et al. 1997), the argument registers (represented as an array in BinProlog (Tarau 2012b)) can be extended with as many positions as needed to accommodate all interclausal variables, to which the compiler would generate appropriate references in instructions like `unify_variable` and `unify_value` (Aït-Kaci 1991). Alternatively, a heap area could be reserved for them, say at a lower address range than that reserved for ordinary variables, and instructions would be generated to create them on the heap before execution begins.

4.4 Scoping constructs and interclausal logic variables

Limiting the scope of interclausal variables to smaller code units can be achieved easily in the case of a source-level implementation by limiting their addition as extra arguments to only the clauses of a given module.

On the other hand, in a Prolog systems that would support *local clauses*, with a semantics similar to Haskell's "`where`" construct (usable for local function definitions), one could implement variants of interclausal variables as logic variables one or more levels up from the point where they are used with or without copying on new clause calls.

5 Source-level backtrackable assumptions

We will overview here another, less "pure" instance of the entanglement pattern that provides, at source-level, a richer set of functionalities than interclausal or backtrackable global variables.

A limitation of interclausal variables is that they do not allow threading information that changes over multiple recursive calls, for which the prototypical example is Prolog's

Definite Clause Grammar (DCG) mechanism (Pereira and Warren 1980), which has been extended to support multiple independent chains of variables at source level (Van Roy 1989) or at WAM-level (Tarau et al. 1995).

As an application of the WAM-level implementation of (Tarau et al. 1995), specific to the BinProlog system, *Assumption Grammars* have been introduced in (Dahl et al. 1997) featuring backtrackable dynamic database updates and a mechanism allowing the programmer to chose between copying or sharing semantics for the assumed clauses.

We will describe here a source level implementation of the functionality of Assumption Grammars, by overloading the standard DCG mechanism. As a result, it is portable to virtually all Prolog systems.

Note that predicates defined here with arity 3 should be used within clauses defined with DCG arrow "`-->/2`" rather than the usual clause neck "`:-/2`".

The Assumption Grammar API is implemented as follows as source level program transformation.

5.1 Setting and getting the database and the DCG tokens

`'#<'(Xs)` sets the DCG token list to be Xs for processing by the assumption grammar.

```
'#<'(Xs,_,Db-Xs):-new_assumption_db(Db).
```

`'#>'(Xs)` unifies current assumption grammar token list with `Xs`.

```
'#>'(Xs,Db-Xs,Db-Xs).
```

`'#:'(X)` matches `X` against the current DCG token the assumption grammar is working on.

```
'#:'(X,Db-[X|Xs],Db-Xs).
```

5.2 Adding new assumptions

`'#+'(X)` adds "linear" assumption +(X) to be consumed at most once, by a `'#-'` operation.

```
'#+'(X,Db1-Xs,Db2-Xs):-add_assumption('+'(X),Db1,Db2).
```

Note that variables occurring in a clause assumed with the `'#+'`operation are "interclausal" and their bindings provide a long distance communication channel between the points where they are produced and consumed. `'#*'(X)` adds 'intuitionistic' assumption `'*'(X)` to be used indefinitely by `'#-'`operation.

```
'#*'(X,Db1-Xs,Db2-Xs):-add_assumption('*'(X),Db1,Db2).
```

The semantics of these clauses is essentially the same as Prolog's dynamic database with "immediate update", except that assumptions are backtrackable.

5.3 Querying the assumptions

`'#='(X)` unifies X with any matching existing or future `'+'(X)` linear assumptions.

```
'#='(X,Db1-Xs,Db2-Xs):-equate_assumption('+'(X),Db1,Db2).
```

`'#-'(X)` consumes a +(X) linear assumption or matches a `'*'(X)` intuitionistic assumption.

```
'#-'(X,Db1-Xs,Db2-Xs):-consume_assumption('+'(X),Db1,Db2).
'#-'(X,Db-Xs,Db-Xs):-match_assumption('*'(X),Db).
```

Note that this operation provides a mechanism to call either linear or intuitionistic assumptions, except that in the later case, matching assumptions are "consumed" i.e; removed from the database. `'#?'(X)` matches `'+'(X)` or `'*'(X)` assumptions without any binding.

```
'#?'(X,Db-Xs,Db-Xs):-match_assumption('+'(X),Db).
'#?'(X,Db-Xs,Db-Xs):-match_assumption('*'(X),Db).
```

5.4 Auxiliary predicates

A few auxiliary predicates implement internals of the API:

```
new_assumption_db(Xs/Xs).

add_assumption(X,Xs/[X|Ys],Xs/Ys).

consume_assumption(X,Xs/Ys,Zs/Ys):-nonvar_select(X,Xs,Zs).

match_assumption(X,Xs/_):-nonvar_member(X0,Xs),copy_term(X0,X).

equate_assumption(X,Xs/Ys,XsZs):- \+(nonvar_member(X,Xs)),!,
  add_assumption(X,Xs/Ys,XsZs).
equate_assumption(X,Xs/Ys,Xs/Ys):-nonvar_member(X,Xs).
```

Finally, `nonvar_member(X,XXs)` and `nonvar_select(X,XXs,Xs)` are variants of `member/2` and `select/3` working on open ended lists.

```
nonvar_member(X,XXs):-nonvar(XXs),XXs=[X|_].
nonvar_member(X,YXs):-nonvar(YXs),YXs=[_|Xs],nonvar_member(X,Xs).

nonvar_select(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
nonvar_select(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],nonvar_select(X,Xs,Ys).
```

5.5 Using Assumption Grammars

One can use phrase/3 to test out assumption grammar components, as follows:

```
?- phrase(('#<'([a,b,c]),'+'(t(99)),'#*'(p(88)),'#-'(t(A)),'#-'(p(B)),
           '#:'(X),'#>'(As)),Xs,Ys).
A = 99, B = 88, X = a, As = [b, c],
Ys = [*(p(88))|_G2344]/_G2344-[b, c] .

?- phrase(('#<'([a,b,c]),'#+'(t(99)),'#*'(p(88)),'#-'(t(A)),
           '#-'(p(B)),'#:'(X),'#>'(As)),Xs,Ys).
A = 99, B = 88, X = a, As = [b, c],
Ys = [*(p(88))|_G1161]/_G1161-[b, c] .
```

We refer to (Dahl et al. 1997) for various examples of their use both for expressing concisely some Prolog algorithms and for capturing long distance dependencies in natural language processing phenomena like anaphora resolution and agreement.

Note that one could also implement similar constructs by combining interclausal variables storing compound terms in which mutable backtrackable state is updated with built-ins like `setarg/3`.

6 Related Work

The first author must confess that about 25 years ago he has thought about and even wrote a short draft paper about interclausal logic variables that got forgotten and lost. Being quite sure that something similar might have popped-up over time and has made it into the logic programming folklore, we have not revisited the subject until now, except for a footnote in (Tarau and Majumdar 2009) where inter-clausal variables are mentioned as write-once global variables relating them to the semantics of term copying. Other than that, we have not found despite an extensive search, any reference to them or closely related concepts.

In (Tarau and Dahl 1994), after applying the binarization transformation (Tarau 1993), multi-headed clauses are introduced, which give direct access to continuations at source-level. The technique makes possible long distance communication between logic variables otherwise inaccessible.

Global variables (both backtrackable and persistent) have been present in BinProlog since the mid-1990s (De Bosschere and Tarau 1996) and are these days available in various Prolog systems. Among them, we mention `SWI-Prolog`'s implementation (Wielemaker et al. 2012) where their values live on the Prolog global stack. Like in the case of interclausal variables, this implies that lookup time is independent of the size of the term. As a result, they can efficiently store large data structures like parsed XML syntax trees or global constraint stores.

By contrast to non-backtrackable global variables, our interclausal variables are single assignment and behave similarly to ordinary logic variables. Backtrackable global variables are semantically similar to interclausal variables. However, like in BinProlog 2.0's original implementation (Tarau 1994), they are named with constants and used through an API like SWI-Prolog's `b_setval/2` and `b_getval/2`, requiring a hash-table look-up to find their values on the heap, while the interclausal variables in this proposal are implemented simply as a special case of logic variables resulting also in a more natural notation.

7 Conclusion

Interclausal variables extend natural properties of the usual logic variables to variables shared among clauses. Given the simplicity of their implementation, for which we have outlined a few alternative scenarios, we hope they can contribute to adding flexibility to logic programming languages while keeping intact their declarative flavor.

We have also described a source-level implementation of "assumption grammars" an extension to Prolog's DCGs that circumvents some limitations of interclausal variables.

We plan future work on implementing interclausal variables at WAM-level and experiments with their uses in probabilistic logic programming. We also plan to work on

mechanisms based on interclausal logic variables that optimize the grounding phase in ASP systems and SAT-based constraint solvers used by Prolog systems.

References

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press.

Dahl, V., Tarau, P., and Li, R. 1997. Assumption Grammars for Processing Natural Language. In *Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed. MIT press, 256–270.

De Bosschere, K. and Tarau, P. 1996. Blackboard-based Extensions in Prolog. *Software — Practice and Experience 26,* 1 (Jan.), 49–69.

Gebser, M., Schaub, T., and Thiele, S. 2007. GrinGo: A New Grounder for Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning*, C. Baral, G. Brewka, and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer Berlin Heidelberg, 266–271.

Panangaden, P. 2011. The Search for Structure in Quantum Computation. In *Foundations of Software Science and Computational Structures*, M. Hofmann, Ed. Lecture Notes in Computer Science, vol. 6604. Springer Berlin Heidelberg, 1–11.

Pereira, F. and Warren, D. 1980. Definite Clauses for Language Analysis. *Artificial Intelligence* 13, 231–278.

Robinson, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *JACM 12,* 1, 23–41.

Tarau, P. 1993. An Efficient Specialization of the WAM for Continuation Passing Binary programs. In *Proceedings of the 1993 ILPS Conference*. MIT Press, Vancouver, Canada. poster.

Tarau, P. 1994. BinProlog 2.20 User Guide. Tech. Rep. 94-1, Dept. d'Informatique, Université de Moncton. Feb. ftp://clement.info.umoncton.ca/BinProlog.

Tarau, P. 2012a. Styla: a Lightweight Scala-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy. `https://code.google.com/p/styla/`.

Tarau, P. 2012b. The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. *Theory and Practice of Logic Programming 12,* 1-2, 97–126.

Tarau, P. and Dahl, V. 1994. Logic Programming and Logic Grammars with First-order Continuations. In *Proceedings of LOPSTR'94, LNCS, Springer*. Pisa.

Tarau, P., Dahl, V., and Fall, A. 1995. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. In *Proceedings of ILPS'95*, J. Lloyd, Ed. Portland, Oregon, 642. poster abstract.

Tarau, P. and Majumdar, A. 2009. Interoperating Logic Engines. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009*. Springer, LNCS 5418, Savannah, Georgia, 137–151.

Van Roy, P. 1989. A useful extension to Prolog's Definite Clause Grammar notation. *SIGPLAN notices 24,* 11 (Nov.), 132–134.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. Swi-prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

Zhou, N.-F. 2013. Picat: A scalable logic-based language and system (invited talk). In *SLATE*, J. P. Leal, R. Rocha, and A. Simões, Eds. OASICS, vol. 29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5–6.

ESmodels: An Epistemic Specification Solver

ZHIZHENG ZHANG and KAIKAI ZHAO

School of Computer Science and Engineering
Southeast University, NanJing 211189, China
(*e-mail:* seu_zzz@seu.edu.cn)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

ESmodels is designed and implemented as an experiment platform to investigate the semantics, language, related reasoning algorithms, and possible applications of epistemic specifications. We first give the epistemic specification language of *ESmodels* and its semantics. The language employs only one modal operator K but we prove that it is able to represent luxuriant modal operators by presenting transformation rules. Then, we describe basic algorithms and optimization approaches used in *ESmodels*. After that, we discuss possible applications of *ESmodels* in conformant planning and constraint satisfaction. Finally, we conclude with perspectives.

KEYWORDS: logic programming, epistemic specification, knowledge representation

1 Introduction

The language of epistemic specification initially proposed in (Gelfond and Przymusinska 1991), (Gelfond and Przymusinska 1993), (Gelfond 1994), and (Gelfond 1991) is an extension of the language of answer set programs by modal operators K and M to represent beliefs of the agent capable of introspection in the presence of multiple belief sets. Intuitively, it use KF to denote an proposition F is believed to be true in each of the agent's belief sets, and MF to denote an proposition F is believed to be true in some of the agent's belief sets. This extension is believed to be useful by discussing its application to formalization of commonsense reasoning. Along its syntax and semantics in (Gelfond and Przymusinska 1991), a few efforts were made to establish reasoning algorithms in (Zhang 2006) and (Watson 1994), and theoretical foundation in (Zhang 2003), (Watson 2000), and (Wang and Zhang 2005). Recently, research on epistemic specifications increases again because introspective reasoning is becoming reality and forseeable as showed in (Faber and Woltran 2011), (Faber and Woltran 2009), and (Truszczyński 2011). To eliminate some unintended interpretations which exist under the original definition, a new semantics is defined in (Gelfond 2011) to arguably close to the intuitive meaning of modalities. Currently, efforts are still desired to made to establish and validate properties of epistemic specifications and the corresponding reasoning algorithms, and to investigate the use of the language. The design and implementation of an epistemic specification solver is hoped to facilitate those efforts.

This article introduces an epistemic specification solver *ESmodels* that is recently being designed and implemented as a flexible platform for experiment with epistemic specifications. The language of *ESmodels* has two types of subjective literals Kl and $\neg$Kl. To express other types of subjective literals, we propose a group of transformation rules rewriting epistemic specifications

with arbitrary types of subjective literals in *ESmodels*'s language. In *ESmodels*, a generate-test algorithm for computing world views of the epistemic specification is employed. It is worth noting that efficient ASP solver *Clasp* is coupled into *ESmodels* to help to generate candidate world views efficiently. Optimization approaches are preliminarily used to promoting the efficiency of the basic algorithm. Presently, we are applying *ESmodels* in solving security conditions in conformant planning, and encoding constraint satisfaction problems.

2 Language

2.1 Syntax and Semantics

An *ESmodels*'s epistemic specification is a collection of finite rules in the following form

$$l_0 \ or ... or \ l_k \ :- \ l_{k+1}, ..., l_j, \ \mathrm{Sl}_{j+1}, ..., \ \mathrm{Sl}_m, \ not \ l_{m+1}, ..., \ not \ l_n$$

where each l_i for $0 \leqslant i \leqslant n$ is an *objective literal*, ie. either an atom A or the negation $\neg A$ of A, and S is either K or $\neg$K, *not* is negation as failure. The set of all objective literals appears in an epistemic specification Π is denoted by Lit_Π. Given a rule r in the above form, let $head(r)$ denote its *head* $\{l_0, ..., l_k\}$, and $body(r)$ the *body* $\{l_{k+1}, ..., l_j, \ \mathrm{Sl}_{j+1}, ..., \ \mathrm{Sl}_m, \ not \ l_{m+1}, ..., \ not \ l_n\}$. Furthermore, let $body^P(r)$ be the *positive objective body* $\{l_{k+1}, ..., l_g\}$ and $body^N(r)$ *negative objective body* $\{l_{m+1}, ..., l_n\}$ of r, and $body^S(r)$ the *subjective body* $\{l_{j+1}, ..., l_m\}$. In addition, we use $body^K(r)$ to denote the set of objective literals in the body of r which appears in term K, and $body^{\neg K}(r)$ to denote the set of objective literals in the body of r which appears in term $\neg$K.

Epistemic specifications with variables are considered as shorthands for their ground instantiations. In the rest of this section, except special noted, we always consider the epistemic specification is grounded.

Let W be a non-empty collection of sets of objective literals, and l an objective literal.

- Kl is satisfied with regard to W, denoted by $W \models \mathrm{K}l$, iff $\forall \omega \in W : l \in \omega$.
- $\neg$Kl is satisfied with regard to W, denoted by $W \models \neg \mathrm{K}l$, iff $\exists \omega \in W : l \notin \omega$.

Definition 1
Let Π be an epistemic specification and W be a non-empty collection of sets of objective literals in Π. W is a world view of Π iff W is the collection of all answer sets of Π^W denoted by $AN(\Pi^W)$, where Π^W is an ASP program obtained from Π by the following reduct laws:

- RL1: removing all rules containing subjective literals not satisfied by W;
- RL2: removing any remaining subjective literals of the form $\neg$Kl;
- RL3: replacing any remaining subjective literals of the form Kl by l.

Example 1
Let an epistemic specification Π_1 consist of the following three rules:

$$p \text{ or } q. \quad p \ :- \neg \mathrm{K} \ q. \quad q \ :- \neg \mathrm{K} \ p.$$

With regard to $\{\{p\}\}$, $\neg$K q is satisfied while $\neg$K p is not satisfied. Hence, $\Pi_1^{\{\{p\}\}} = \{p \text{ or } q. \quad p \ :-.\}$ and then $AN(\Pi_1^{\{\{p\}\}}) = \{\{p\}\}$. So $\{\{p\}\}$ is a world view of Π_1. Similarly, $\{\{q\}\}$ is also a world view of Π_1.

2.2 Representation of Other Subjective Literals

To handle other subjective literals using *ESmodels*, namely K *not* l, $\neg$K *not* l, Ml, $\neg$Ml, M *not* l, and $\neg$M *not* l, we can convert an epistemic specification Π with arbitrary subjective literals in rules bodies into an epistemic specification Π^{ES} such that Π^{ES} has only subjective literals in the form Kl or $\neg$Kl by the following transformation procedure.

1 For each objective literal l, add a rule $l' :- not\ l$ to Π^{ES} if there exist a subjective occurrence of $\neg$K *not* l or Ml or $\neg$Ml or K *not* l in Π, where l' is a new created objective literal corresponding to l.
2 Add each rule of Π to Π^{ES} after performing the following operations on it.

- Replace $\neg$K *not* l by $\neg$Kl';
- Replace Ml by $\neg$Kl';
- Replace $\neg$Ml by Kl';
- Replace K *not* l by Kl';
- Replace M *not* l by $\neg$Kl;
- Replace $\neg$M *not* l by Kl.

Then, we define its *world view* based semantics as follows.

Definition 2
For an epistemic specification Π with arbitrary subjective literals, let *Lit* be a set of objective literals appearing in Π, and Π^{ES} its corresponding *ESmodels* epistemic specification, a collection of sets of objective literals W is a world view of Π iff there exists a world view W' of Π^{ES} such that $W = \{\omega \cap Lit | \omega \in W'\}$.

Example 2
Given an epistemic specification $\Pi_2 : \{p\ :- \neg\text{M}q.\quad q\ :- \neg\text{K}p.\}$ then we have $Lit_2 = \{p, q\}$ and $\Pi_2^{ES} : \{p :- \text{K}l.\quad l :- not\ q.\quad q\ :- \neg\text{K}p.\}$. Π_2^{ES} has two world views $\{\{q\}\}$ and $\{\{p, l\}\}$, hence, Π_2 has two world views $\{\{q\}\}$ and $\{\{p\}\}$.

Example 3
Given an epistemic specification $\Pi_3 : \{p\ :- not\ q, \text{M}q.\quad q\ :- not\ p, \text{M}q.\}$, then we have $\Pi_3^{ES} : \{p :- not\ q, \neg\text{K}l.\quad l :- not\ q.\quad q :- not\ p, \neg\text{K}i.\quad i :- not\ q.\}$. Π_3^{ES} has two world views $\{\{i, l\}\}$ and $\{\{i, p, l\}, \{q\}\}$, hence, Π_3 has two world views $\{\{\}\}$ and $\{\{p\}, \{q\}\}$.

2.3 Connection to Gelfond's New Epistemic Specification

In the syntactic aspect of the epistemic specification defined in (Gelfond 2011), it allows two more subjective literals of forms, K not l and $\neg$K not l, in the rule's body. The modality M is defined to be expressed in terms of K by M $l =_{def} \neg$K *not* l. Semantically, let W be a non-empty collection of sets of objective literals, and l an objective literal.

- Kl is satisfied with regard to W, denoted by $W \models$Kl , iff $\forall S \in W$: $l \in S$.
- $\neg$Kl is satisfied with regard to W, denoted by $W \models \neg$Kl , iff $\exists S \in W$: $l \notin S$.
- K not l is satisfied with regard to W, denoted by $W \models$ K not l iff for every $S \in W$, $l \notin S$, otherwise $S \models \neg$ K not l

The set W is called a world view of Π if W is the collection of all answer sets of Π^W, where Π^W is obtained by

- removing all rules containing subjective literals not satisfied by W;
- removing any remaining subjective literals of the form $\neg$Kl or $\neg$K*not* l;
- replacing any remaining subjective literals of the form Kl by l and any K*not* l by *not* l.

Theorem1 shows that *ESmodels* can compute the world view of any Gelfond's new epistemic specification.

Theorem 1
For any Gelfond's new epistemic specification Π, let Lit be a set of objective literals appearing in Π, a collection of sets of objective literals W is a world view of Π under Gelfond's new definition iff there exists a world view W' of Π^{ES} such that $W = \{S \cap Lit | S \in W'\}$.

Proof
The main idea of this proof is as follows. Let Lit^{ES} be objective literals appearing in Π^{ES},
$\leftarrow$ direction: if there is a world view W' of Π^{ES}, then for any $\omega \in W'$, ω is an answer set of $(\Pi^{ES})^{W'}$. Let $W = \{S \cap Lit | S \in W'\}$, then $\omega \cap Lit$ is an answer set of Π^W under Gelfond's new definition (because the Gelfond-Lifschitz reduction of Π^W wrt. $\omega \cap Lit$ just possibly has less facts $\{l : -. | l \in \omega - Lit$ and l does not appear in bodies of any rules$\}$ than the Gelfond-Lifschitz reduction of $(\Pi^{ES})^{W'}$ wrt. ω).
$\rightarrow$ direction: if W is a world view of Π, then we create W' as follows: for each $\omega \in W$, we have $\omega' = \omega \cup \{l \in Lit^{ES} - Lit | l : -not\ l' \in \Pi^{ES}, l' \notin \omega\}$ in W'. Then, ω' is an answer set of $(\Pi^{ES})^{W'}$ (because the Gelfond-Lifschitz reduction of $(\Pi^{ES})^{W'}$ wrt. ω' just possibly has more facts $\{l : -. | l \in \omega - Lit$ and l does not appear in bodies of any rules$\}$ than the Gelfond-Lifschitz reduction of Π^W wrt. ω). $\square$

Example 4
Given an epistemic specification $\Pi_4 : \{p : -$M$p.\}$, under Gelfond's definition Π_4 has two world views $\{\{\}\}$ and $\{\{p\}\}$. By the transformation defined in last subsection, we have $\Pi_4^{ES} : \{p : -\neg$K$l.\quad : -not\ p.\}$, and *ESmodels* can find Π_4^{ES} 's two world views: $\{\{l\}\}$ and $\{\{p\}\}$, that is, Π_4 also has two world views $\{\{\}\}$ and $\{\{p\}\}$ by *ESmodels*.

3 Computing World Views in *ESmodels*

A generate-test algorithm forms a basis of computing world views in *ESmodels*. Now, we are taking two preliminary steps to optimize the algorithm.

3.1 Basic Algorithm

Let Π be an epistemic specification, $EL(\Pi)$ be a set of objective literals such that $l \in EL(\Pi)$ iff Kl or -Kl occurring in Π. Then, we call a pair (S, S') an *assignment* of $EL(\Pi)$ iff

$$S \cup S' = EL(\Pi) \text{ and } S \cap S' = \emptyset$$

Then, we define an answer set program $\Pi^{(S,S')}$ obtained by:

- removing from Π all rules containing subjective literals Kl such that $l \in S'$, or subjective literal $\neg$Kl such that $l \in S$,

- removing from the rest rules in Π all other occurrences of subjective literals of the form $\neg \mathrm{K}l$,
- replacing remaining occurrences of literals of the form $\mathrm{K}l$ by l.

Theorem 2
Given an epistemic specification Π and a collection W of sets of objective literals. W is a world view of Π if an assignment (S, S') of $EL(\Pi)$ exists such that

- W is the collection of all answer sets of $\Pi^{(S,S')}$,
- W satisfies the assignment, that is, $S \cap \left(\bigcap_{A \in W}\right) == S$ and $S' \cap \left(\bigcap_{A \in W}\right) == \emptyset$.

Proof
If both $S \cap \left(\bigcap_{A \in W}\right) == S$ and $S' \cap \left(\bigcap_{A \in W}\right) == \emptyset$ are satisfied, we have $\Pi^{(S,S')} = \Pi^{W}$. Hence, if W is the collection of all answer sets of $\Pi^{(S,S')}$ then W is the collection of all answer sets of Π^{W}, that is, W is a world view of Π. $\square$

By Theorem 2, an immediate method of computing the world views of an epistemic specification includes three main stages: generating a possible assignment, reducing the epistemic specification into an answer set program, and testing if the collection of the answer sets of the answer set program satisfies the assignment. At a high level of abstraction, the method can be implemented as showed in the following algorithm.

Algorithm 1 ESMODELS.

Input:
Π: An epistemic specification;
Output:
All world views of Π;

```
1: for every possible assignment of EL(Π) (S, S') of Π do
2:    Π' = Π^(S,S') {reduces Π to an answer set program Π' by (S, S')}
3:    W = computerASs(Π') {computes all answer sets of Π'}
4:    if S ∩ (⋂_{A∈W}) == S and S' ∩ (⋂_{A∈W}) == ∅ then
5:       output W
6:    end if
7: end for
```

ESMODELS firstly gets all subjective literals $EL(\Pi)$ and generates all possible assignments of $EL(\Pi)$. For each assignment(S, S'), the algorithm reduces Π to an answer set program Π', i.e., $\Pi' = \Pi^{(S,S')}$. Next, it calls exiting ASP solver like Smodels, Clasp to compute all answer sets W of Π'. Finally, it verifies the W. W is a world view of Π, if W satisfies $S \cap \left(\bigcap_{A \in W}\right) == S$ and $S' \cap \left(\bigcap_{A \in W}\right) == \emptyset$. ESMODELS stops, when all possible assignments are tested.

3.2 Optimization Approaches

3.2.1 Reducing Subjective Literals

However, ESMODELS has a high computational cost, especially with a large number of subjective literals. Therefore, we introduce a new preprocessing function to reduce reduce $EL(\Pi)$ before generating all possible assignments of $EL(\Pi)$. We first give several propositions.

Let Π be an epistemic specification and a pair (S, S') of objective literals of Π, T_Π be an *lower bound* operator on (S, S') defined as follows:

$$T_\Pi(S, S') = \big(\{head(r) || head(r)| = 1, body^+(r) \subseteq S, body^-(r) \subseteq S'\},$$
$$\{l | \neg\exists r \in \Pi(l \in head(r)), \text{ or } \forall r \in \Pi, l \in head(r) \Rightarrow (body^+(r) \cap S' \neq \emptyset \text{ or } body^-(r) \cap S \neq \emptyset)\}\big)$$

where $body^+(r) = body^P(r) \cup body^K(r)$, $body^-(r) = body^N(r) \cup body^{-K}(r)$. Intuitively, $T_\Pi(S, S')$ computes the objective literals that must be true and that not true with regard to S and S' which are sets of literals known true and known not true respectively. Clearly, we can use this operation to reduce the searching space of subjective literals. This idea is guaranteed by the following definitions and propositions.

Definition 3
A pair (S, S') of sets of objective literals is a partial model of an epistemic specification Π if, for any world view W of Π, $S \cap \big(\bigcap_{A \in W}\big) == S$ and $S' \cap \big(\bigcap_{A \in W}\big) == \emptyset$.

Theorem 3
$T_\Pi(S, S')$ is a partial model if (S, S') is a partial model of an epistemic specification Π, .

Proof
Let $(A, B)|_1$ to denote A of a pair (A, B), and $(A, B)|_2$ to denote B. The main idea of this proof is as follows. For any world view W of Π, $S \cap \big(\bigcap_{A \in W}\big) == S$ and $S' \cap \big(\bigcap_{A \in W}\big) == \emptyset$, by the definition of T_Π, the Gelfond-Lifschitz reduction of Π^W wrt. any $\omega \in W$ must have $l :- | l \in T_\Pi(S, S')|_1$ and must not have any rule with head in $T_\Pi(S, S')|_2$, hence, we have $T_\Pi(S, S')|_1 \cap \big(\bigcap_{A \in W}\big) == T_\Pi(S, S')|_1$ and $T_\Pi(S, S')|_2 \cap \big(\bigcap_{A \in W}\big) == \emptyset$. □

Corollary 1
Let, $T^i_\Pi(S, S') = T_\Pi(T^{i-1}_\Pi(S, S'))$, then $T^k_\Pi(\emptyset, \emptyset)$ is a partial model of Π.

Proof
Because $(\emptyset, \emptyset)$ is a partial model, $T_\Pi(\emptyset, \emptyset)$ is a partial model, and so on, $T^2_\Pi(\emptyset, \emptyset)$ ... $T^k_\Pi(\emptyset, \emptyset)$ are partial models of Π □

An epistemic specification rule r is *defeated* by (S, S') if $body^+(r) \cap S' \neq \emptyset$ or $body^-(r) \cap S' \neq \emptyset$. Let (S, S') be a partial model of an epistemic specification Π, $\Pi|_{(S,S')}$ is obtained by

- removing from Π all rules defeated by (S, S'),
- removing from the rest rules in Π all other occurrences of literals of the form not l or $\neg$Kl such that $l \in S'$,
- removing remaining occurrences of literals of the form l or Kl such that $l \in S$.
- adding $l \leftarrow .$ if $l \in S$
- adding $\leftarrow l.$ if $l \in S'$

Theorem 4
If (S, S') is a partial model of an epistemic specification Π, $\Pi|_{(S,S')}$ and Π have the same world views.

Proof
The main idea in this proof is as follows. For any world view W of Π, if $S \cap \left(\bigcap_{A \in W}\right) == S$ and $S' \cap \left(\bigcap_{A \in W}\right) == \emptyset$, then Π^W and $(\Pi|_{(S,S')})^W$ have the same answer sets. And, for any world view W of $\Pi|_{(S,S')}$, we have that W is a world view of Π. □

By theorem 3 and 4, we can design PreProcess showed in algorithm 2. Firstly, it sets the pair $(S, S^{'})$ as $(\emptyset, \emptyset)$. Then it expands the partial model of Π and reducts the $\Pi^{'}$ according to $(S, S^{'})$. Next, we updates the partial model by the new program. Finally, it compares the new partial model with the previous one. If the partial model is stable, it stops and returns $\Pi^{'}$; Otherwise, it repeats this procedure.

Algorithm 2 PreProcess.

Input:
Π: An epistemic specification;
Output:
$\Pi^{'}$: A reduction of Π;
1: $(S, S') = (\emptyset, \emptyset)$,
2: **repeat**
3: $(S, S') = T_{\Pi'}(S, S')$
4: $\Pi' = \Pi'|_{(S,S')}$
5: **until** S, S' are fixed
6: return Π'

Obviously, PreProcess and partial model are very helpful for reducing search space. We thus provide an EFFICIENT ESMODELS as follows:

Algorithm 3 EFFICIENT ESMODELS.

Input:
Π: An epistemic specification;
Output:
All world views of Π;
1: Π'=PreProcess(Π)
2: **for** every possible assignment of $EL(\Pi')$ **do**
3: $\Pi' = \Pi'^{(S,S')}$
4: Π'=PreProcess(Π')
5: W = computerASs(Π')
6: **if** $S \cap \left(\bigcap_{A \in W}\right) == S$ and $S' \cap \left(\bigcap_{A \in W}\right) == \emptyset$ **then**
7: output W
8: **end if**
9: **end for**

3.2.2 Using Multicore Technology

In *ESmodels*, another way of improving efficiency is the use of multicore technology. Based on Algorithm 3, by parallel generation of possible assignments and parallel calling of ASP solver, the efficiency of *ESmodels* can be improved greatly.

4 Applications

4.1 Conformant Planning

Consider the planning problem with multiple possible initial states, what makes it become much harder is to find a so called *secure* plan that enforces the goal from any initial state. (Eiter et al. 2003) gives three security conditions to check whether a plan is secure:

1. the actions of the plan are executable in the respective stages of the execution;
2. at any stage, executing the respective actions of the plan always leads to some legal successor state; and
3. the goal is true in every possible state reached if all steps of the plan are successfully executed.

Here, we consider a track of effects of executing an action sequence as a belief set, thus can intuitively encode those security conditions in epistemic specification constraints. We use *nonexecutable* to denote the actions are not executable, *inconsistent* to denote that a state is illegal, *success* to sign a state satisfies the goal, and $goal(m)$ to denote the state reached after a given steps number m satisfies the goal, and $o(A, T)$ to denote an action A happens in the step T:

- for security condition 1: $\leftarrow M$ *nonexecutable*.
- for security condition 2: $\leftarrow M$ *inconsistent*.
- for security condition 3: *success* $\leftarrow goal(m)$. and $\leftarrow \neg K$ *success*.

Moreover, to guarantee the above security testing is put on tracks caused by the same action sequence, we write a new constraint.

$$\leftarrow \neg K o(A, T), o(A, T). \tag{1}$$

Intuitively, rule (1) says that *if one action A happened in stage T of one track, it happened in stage T of all tracks*. Thus, we can easily get a **Conformant Planning Module** consisting of the above five constraints and the following action generation rules:

- Set a planning horizon m: $\#const\ x = m.\ step(0..x)$.
- Generating one action for each step: $1\{o(A, T) : action(A)\}1 \leftarrow step(T), T < m$.

Combine the conformant planning module with a planning domain (including action axioms e.g., inertial law) encoded in an answer set program, the result epistemic specification represents a conformant planning problem, and its world view(s) corresponds to the secure plan(s) of the problem. Here, we use a case provided in (Palacios and Geffner 2006) to demonstrate the conformant planning approach using epistemic specification. Given a conformant planning problem P with an initial state $I = p \vee q$ (i.e., nothing else is known; there is no CWA), and action a and b with effects a causes q if r, a causes $\neg s$ if r, and b causes s if q, the planning goal is q, s. Then, we describe the planning domain as follows.

- Signatures: $action(a).\ action(b).$
  $fluent(in, p).\ fluent(in, q).\ fluent(in, r).\ fluent(in, s).$
- Causal Laws: $h(pos(q), T+1) : -o(a, T), h(pos(p), T), step(T).$
  $h(neg(s), T+1) : -o(a, T), h(pos(r), T), step(T).$
  $h(pos(s), T+1) : -o(b, T), h(pos(q), T), step(T).$
- Inertial Laws:
  $h(pos(X), T+1) : -fluent(in, X), h(pos(X), T), step(T), not\ h(neg(X), T+1).$
  $h(neg(X), T+1) : -fluent(in, X), h(neg(X), T), step(T), not\ h(pos(X), T+1).$

- Initial: $1\{h(pos(p),0),h(pos(q),0)\}2.$
  $1\{h(pos(F),0),h(neg(F),0)\}1 :- fluent(in,F).$
- Goal: $goal(T) :- h(pos(q),T),h(pos(s),T),step(T).$

When we set $m = 2$, *ESmodels* can find the unique world view including twelve literal sets, and each of them includes $o(a,0)$ and $o(b,1)$ that means the program has a conformant plan *a b*.

4.2 Constraints Satisfaction

In some situations, constraints on the variable are with epistemic features, that is, a variable's value is not only affected by the values of other variables, but also determined by all possible values of other variables. Here, we demonstrate the use of *ESmodels* in solving such constraint satisfaction problems using a *dinner* problem:Jim, Bones, Checkov, Mike, Jack, Uhura, and Scotty, and Tommy received a dinner invitation, and the constraints on their decisions and the constraints description in epistemic specification rules are as follows:

- if Checkov may not participate, then Jim will participate: $jim :- not\ checkove.$
- if Jim may not participate, then bones will participate: $bones :- not\ jim.$
- if only one of Jack and Mike will participate: $jack :- not\ mike.\quad mike :- not\ jack.$
- if Jack must participate, then Uhura will participate: $uhura :- \mathrm{K}jack.$
- if Uhura may not participate, then Scotty will participate: $scotty :- not\ uhura.$
- if Scotty must participate, then Tommy will participate: $tommy :- \mathrm{K}scotty.$
- Checkov will participate. $checkov.$

ESmodels can find the unique world view$\{\{checkov, tommy, scotty, jim, mike\}$ $\{checkov, tommy, scotty, jim, jack\}\}$ that means Jim, Checkov, Scotty, and tommy must participate, Bones and Uhura must not participate, Jack and Mike may or may not participate.

5 Conclusion

ESmodels is an epistemic specification solver designed and implemented as an experiment platform to investigate the semantics, language, related reasoning algorithms, and possible applications of epistemic specifications. A significant feature of this solver is that its language is more compact than that defined in literatures, but capable of representing many subjective literals via a group of transformation rules. Besides, this solver can compute world views under Gelfond's new definition, while that presented by Zhang in (Zhang 2007) and Watson in (Watson 1994) are based on the early definition of epistemic specifications. In addition, we find the compact encoding of conformant planning problems and constraint satisfaction problems in the epistemic specification language, which primarily shows *ESmodels*'s potential in applications[1].

The work presented here is primary. Now, we are designing and exploring more efficient algorithm for *ESmodels* and evaluate it using those benchmarks in the conformant planning field.

[1] In the early related work, Gelfond investigated the value of epistemic specifications in formalizing commonsense reasoning

Acknowledgment

We acknowledge the support from Project 60803061 and 61272378 by National Natural Science Foundation of China, and Project BK2008293 by Natural Science Foundation of Jiangsu.

References

EITER, T., FABER, W., LEONE, N., PFEIFER, G., AND POLLERES, A. 2003. A logic programming approach to knowledge-state planning, ii: The dlvk system. *Artificial Intelligence 144,* 1, 157–211.

FABER, W. AND WOLTRAN, S. 2009. Manifold answer-set programs for meta-reasoning. In *Logic Programming and Nonmonotonic Reasoning*. Springer, 115–128.

FABER, W. AND WOLTRAN, S. 2011. Manifold answer-set programs and their applications. In *Logic programming, knowledge representation, and nonmonotonic reasoning*. Springer, 44–63.

GELFOND, M. 1991. Strong Introspection. In *National Conference on Artificial Intelligence*. 386–391.

GELFOND, M. 1994. Logic programming and reasoning with incomplete information. *Annals of mathematics and artificial intelligence 12,* 1-2, 89–116.

GELFOND, M. 2011. New semantics for epistemic specifications. In *Logic Programming and Nonmonotonic Reasoning*. Springer, 260–265.

GELFOND, M. AND PRZYMUSINSKA, H. 1991. Definitions in epistemic specifications. In *LPNMR* (2002-01-03). 245–259.

GELFOND, M. AND PRZYMUSINSKA, H. 1993. Reasoning on open domains. In *LPNMR*. Vol. 1993. 397–413.

PALACIOS, H. AND GEFFNER, H. 2006. Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes). In *AAAI*. AAAI Press, 900–905.

TRUSZCZYŃSKI, M. 2011. Revisiting epistemic specifications. In *Logic programming, knowledge representation, and nonmonotonic reasoning*. Springer, 315–333.

WANG, K. AND ZHANG, Y. 2005. Nested epistemic logic programs. In *Logic Programming and Nonmonotonic Reasoning*. Springer, 279–290.

WATSON, R. 1994. An inference engine for epistemic specifications. *1994.M.S. Thesis, Department of Computer Science, University of Texas at El Paso.*.

WATSON, R. 2000. A splitting set theorem for epistemic specifications. *In Proceedings of the 8th International Workshop on Non-MonotonicReasoning (NMR-2000)*.

ZHANG, Y. 2003. Minimal change and maximal coherence for epistemic logic program updates. In *IJCAI*. 112–120.

ZHANG, Y. 2006. Computational properties of epistemic logic programs. In *KR*. 308–317.

ZHANG, Y. 2007. Epistemic reasoning in logic programs. In *IJCAI*. 647–653.