BENELOG 1994

Proceedings of the Sixth Benelux Workshop on Logic Programming

Amsterdam, September 2, 1994

Elena Marchiori (Ed.)
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands

BENELOG 1994

Proceedings of the Sixth Benelux Workshop on Logic Programming

Amsterdam, September 2, 1994

Acknowledgments Thanks to Mieke Bruné and Frank Teusink for their help in the local organization of this workshop.

CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands

BENELOG 1994

Sixth Benelux Workshop on Logic Programming
Friday September 2, 1994
CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Room Z009

CONTENTS

Reasoning about Run-Time Properties of Logic Programs

(abstract)

Włodzimierz Drabent*

August 20, 1994

In this presentation we discuss methods of proving run-time properties of Prolog programs. More precisely we consider LD-resolution as an abstraction of Prolog. The methods are also applicable for (side effect free) Prolog builtins, like *var*. We are interested in partial correctness, proving termination is outside of the scope of this presentation. We study run-time properties, i.e. properties of LD-derivations. They are a generalization of the declarative properties which are, roughly speaking, properties of computed answers.

In the context of partial correctness, a generic form of a run-time property is that a certain assertion is satisfied whenever the control reaches a certain point. A typical example is a mode — an assertion describing groundness of predicate arguments, e.g. "at every call of $p/3$ its first argument is ground and the third is non-ground". Another example may be: "At every call of $q/2$ its first argument is a list of distinct variables which do not occur in the second argument; at every success of $q/2$ its first argument is non-ground" Clearly such properties are inexpressible in terms of the declarative semantics. Other typical examples are related to types, sharing / non sharing of variables etc.

So we want to reason about the actual form of terms appearing in the computation. Properties of interest may be not monotonic, in other words not closed under substitution. An assertion may be true for some term t but false for some its instance $t\theta$. Non monotonicity of assertions complicates the proofs, indeed some proof methods listed below are restricted to monotonic assertions.

There are two natural ways of specifying run-time properties, i.e. of assigning assertions to programs. One can assign a pre- and a postcondition to each predicate. Alternatively one can assign assertions to program points in the clauses (assuming a program point after each atom in the clause). In the first case assertions refer to predicate arguments, in the second case they refer to the variables of the clause. The second case requires more assertions, this in turn may lead to simpler proofs.

There exist proof methods for both specification styles. For the first case (assertions for predicates) we have the methods of [DM88, Dra88], [BC89], Section 3 of [BLS92] and for the second (assertions in the clauses) [CM91], [Dra94]. In the

*IPI PAN, Polish Academy of Sciences; IDA, Linköping University, and IIUW, Warsaw University; e-mail: wdr@ida.liu.se.

first case a partial correctness proof boils down to checking a collection of verification conditions for each program clause. In the second case one has to prove two conditions for every (unifiable) pair of a body atom and a clause head from the program. The method of [DM88] is representative for the first case, as the others can be seen as its specializations. (The methods of [BC89] and [BLS92] are restricted to monotonic assertions). Presentation of proof methods will be the main part of my talk.

Assertions make it possible to provide formal "axiomatic" semantics for many Prolog built-ins (whose meaning cannot be expressed by means of the declarative semantics).

The proof methods for run-time properties are closely related to static analysis methods like abstract interpretation. Roughly speaking, an abstract domain may be seen as a supply of assertions, usually quite restricted. The role of abstract interpretation is to assign assertions to the program in a way that correctly describes its run-time behaviour. The fixpoint equation of abstract interpretation corresponds to verification conditions of proof methods. An example of proving correctness of a static analysis algorithm using the method of [DM88] is given in [RNP92]. A study of relations between proof methods and abstract interpretations is a subject of future work.

The discussed proof methods are valid for Prolog selection rule and any search strategy (including the cut and OR-parallelism). A subject for future work is to provide a proof method for other selection rules, like Prolog with delays, ANDORRA or AND-parallelism.

This research is partly supported by Swedish Research Council for Engineering Sciences (dnr 221-93-942), Polish Academy of Sciences and by ESPRIT BRA project Compulog 2, contract no. 6810.

References

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89, vol. 2*, pages 96–110. Springer-Verlag, 1989. Lecture Notes in Computer Science.

[BLS92] F. Bronsard, T. K. Lakshman, and Reddy U. S. A framework of directionality for proving termination of logic programs. In K. Apt, editor, *Logic Programming. Proc. of the Joint Int. Conference and Symposium*, pages 321–335. MIT Press, 1992.

[CM91] L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 629–642. MIT Press, 1991.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, June 1988. Special issue with selected papers from TAPSOFT'87, Pisa.

[Dra88] W. Drabent. On completeness of the inductive assertion method for logic programs. Unpublished note, May 1988.

[Dra94] W. Drabent. A Floyd-Hoare method for Prolog. Forthcoming, 1994.

[RNP92] Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In K. Apt, editor, *Logic Programming. Proc. of the Joint Int. Conference and Symposium*, pages 85–97. MIT Press, 1992.

Concurrent Constraint Programming with Information Removal

Frank S. de Boer[1], Catuscia Palamidessi[2], Eike Best[3]

[1] Department of Computer Science, Free University of Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
email: frankb@cs.vu.nl

[2] DISI,Università di Genova,
Viale Benedetto XV, 3, 16132 Genova, Italy
e.mail: catuscia@dipisa.di.unipi.it

[3] Institut für informatik, Universität Hildesheim,
Marienburger Platz 22, D-31141 Hildesheim, Germany
email: E.Best@informatik.uni.-hildesheim.de

Abstract

In this paper we introduce an extension of the concurrent constraint paradigm which allows to remove information. The operation of information removal is defined logically in terms of a new concept of a constraint system, which we call *linear* constraint system. The entailment relation of a linear constraint system is based on the main underlying idea of linear logic: Hypotheses in a logical derivation represent physical resources which are *consumed* once used in the entailment relation.

We give a semantical analysis of the proposed non-monotonic extension of ccp in terms of the *causal* relations among occurrences of basic actions (*events*). Our semantical model, which is a true concurrent model, allows us to compare various sublanguages of the proposed extension of ccp (including ccp itself) from the point of view of the degree of parallelism.

1 Introduction

In the concurrent constraint paradigm [12, 14, 15] (ccp, for short) parallel processes interact via a common store, represented by a constraint, which expresses some partial information on the values of the variables involved in the computation. One of the most characteristic features of the ccp paradigm is a formalization of the basic operations which allow to update and to query the common store, in terms of the logical notions of consistency, conjunction and entailment supported by a given underlying constraint system. An update of the common store consists of adding (consistently) a constraint, whereas a query consists of checking whether the current store entails some constraint. Thus the computational model of ccp gives rise to a monotonic evolution of the store: more and more information is added in the course of the computation. Moreover, since the definition of the entailment relation of a constraint system is based

on classical logic, it is difficult to envisage extensions of ccp which allow, for example, to *remove* constraints from the common store. Such extensions would greatly enhance the expressive power of ccp and would allow very compact solutions to programming problems which in a purely monotonic setting require considerable elaborate programming techniques.

One of the main contributions of this paper is the introduction of an operator in ccp which consists of removing some constraint from the common store, and which thus gives rise to non-monotonic computations. The concept of store becomes thus similar to the notion of *blackboard* used in Linda [7, 4] and in Shared Prolog [2].

We describe logically the mechanism of information removal in terms of a new concept of a constraint system, which we call *linear* constraint system. The entailment relation of a linear constraint system is based on the main underlying idea of linear logic: Hypotheses in a logical derivation represent physical resources which are *consumed* once used in the entailment relation. The availability of various copies of the same hypothesis is of relevance in a derivation. Therefore it is natural to describe the entailment relation of a constraint system in terms of *multisets* of tokens instead of sets. More precisely, the entailment relation of a linear constraint system is defined from multisets of tokens to tokens. A derivation of a token from a multiset of tokens according to the entailment relation then amounts to the consumption of all the hypotheses and the production of the conclusion.

Parallel agents operate on a common store by adding and removing tokens. Adding tokens is modeled basically as in ccp by tell actions. Tokens can be removed by the operation of a *get* action of the form $get(c)$, where c is a token of the underlying linear constraint system. The execution of a $get(c)$ with respect to a given store consists of removing those tokens from which c can be derived. Thus in a sense we identify an occurrence of c with a multiset of tokens from which c can be derived. If the current store does not contain a multiset of tokens which entails c, then the get action suspends waiting for parallel processes to add the necessary information. On the other hand in case the store contains different multisets entailing c one is nondeterministically selected and subsequently removed.

Other approaches to combine ccp with linear logic concepts have been proposed in [1, 13, 6]. In general, the above cited works are directly based on the framework of linear logic, i.e. the operators of the language are interpreted as operators of linear logic, and its operational semantics is expressed in terms of the proof theory of linear logic. The approach of this paper, on the contrary, is based on a separation among *the data* (the constraints), whose theory has some linear logic flavour, and *the programming language*, whose operational semantics is defined in terms of a transition system in the usual SOS style. The advantage of our approach is that it gives more freedom in the definition of the language operators. For instance, it is not clear what would be the correspondent in linear logic of the typical ccp actions *ask* and *atomic tell*.

Another difference with [1] is that [1] defines a special kind of clauses and linear inference rules to model synchronous message-passing, whereas our approach is based on (the blackboard kind of) asynchronous communication.

The following example illustrates the kind of inference mechanism and the kind of computations we want to model.

Example 1.1 Let $available(x)$ denote the situation in which the theatre place number x is available, and let $mpc(y)$ denote the fact that Mr. y makes a phone call to reserve a theatre place. Then we want to have an inference of the form

$$available(x), mpc(y) \vdash booked(x, y).$$

Now, assume that the price of a place is 50\$. If a person has booked a place, and he has the money to pay for it, than he can take the place:

$$booked(x, y), has(y, 50\$) \vdash takes(y, x).$$

If we have, say, two places available with numbers 1 and 2, then the initial situation (store) is

$$available(1), available(2).$$

Now, suppose that Mr. A and Mr. B want to go to the theatre, make phone calls, and have 50\$ to pay the place. The store becomes

$$available(1), available(2), mpc(A), mpc(B), has(A, 50\$), has(B, 50\$).$$

The two possible possible final stores, nondeterministically generated by the inference system, are

$$takes(A, 1), takes(B, 2)$$

or

$$takes(A, 2), takes(B, 1).$$

If a third person Mr. C makes now a phone call, he cannot reserve a place, because there aren't places available anymore. The correct booking and selling of the places, in this case, is completely controlled by the inference system.

The role of the language and the user-defined agents becomes apparent in more interesting situations, like in the case that one of the two people who have booked, say Mr. B, has no money to pay the ticket. At a certain point the store could be

$$takes(A, 1), booked(2, B), mpc(C), has(C, 50\$).$$

Some time before the performance starts, the director of the theatre could decide, for instance, to activate an agent which makes available again all places which are not taken yet, thus giving to Mr. C the possibility to get the place. This kind of dynamical decisions are treated at the language level; they could not be easily described by an inference system, which is statical by nature.

We give a semantical analysis of the proposed non-monotonic extension of ccp in terms of the *causal* relations among occurrences of basic actions (*events*). Such a semantical description is of interest because it allows to treat parallelism as a primitive concept, whereas most semantical models describe parallelism in terms of interleaving and thus reduce it to a form of non-determinism. Our semantical model, which is a true concurrent model, allows us to compare various sublanguages of the proposed extension of ccp (including ccp itself) from the point of view of the degree of parallelism.

The construction of our model is based on the basic distinction between two kinds of causal relations among actions: dependencies based on the 'flow of control' and dependencies based on the 'flow of data'. The control dependencies can be derived in a straightforward manner from the structure of the program (they derive mainly from the sequencing operator). The data dependencies are obtained from a computation, 'at run-time', and are defined in terms of the entailment relation of the underlying linear constraint system. In order to evaluate the adequacy (correctness and completeness) of the description of the causal dependencies we introduce an operational semantics which allows us to observe temporal order and simultaneity of actions. This is the main

difference with respect to other approaches to the true concurrent semantics of ccp [8, 9, 11]. In those works in fact the adequacy of a partial order model is established with respect to an *interleaving* operational semantics. We will argue that our approach allows a stronger formulation of the completeness result.

The concurrent execution of get actions can be understood at the physical level by imagining an architecture in which the common store is implemented as a kind of 'ring'. Agents are connected to this ring. Tokens are added by the agents and 'flow' along the ring, 'passing' tokens can be taken by the agents, used in the entailment relation and consumed.

2 Linear constraint systems

In this section we define the notion of *linear constraint system*. As announced in the introduction, our aim is to model a constraint as a physical resource, which can be produced in several copies and each copy of which disappears once it is consumed, i.e. used as a hypothesis in the entailment relation. We mimic the definition of *constraint system* by Saraswat, Rinard and Panangaden [15], which is based on the notion of *information systems* by Scott [16], and introduce only the necessary modifications.

The construction of Saraswat, Rinard and Panangaden starts from the notion of *simple constraint system*, namely a domain of "tokens" together with a "simple entailment relation" which tells us whether a token can be derived from other tokens. The *constraint system* induced by it is a structure whose elements are sets of tokens, and the entailment relation is the pointwise extension of the simple one.

In order to represent the idea that the same resource can have multiple copies, it is natural to consider constraints as multisets of tokens instead of sets. Furthermore the underlying logic must be modified. The derivation of a token by the entailment relation *consumes the hypothesis*, hence, if the derivation of a token uses twice the same information, then this information must be represented twice in the hypothesis. Therefore the entailment relation must be defined from multisets of tokens to tokens.

We first recall some basic notions about multisets. Given a set $(c, d, \ldots \in)S$, a multiset of elements of S is a mapping

$$M : S \to \mathbb{N}$$

where $\mathbb{N}$ is the set of natural numbers. In the following we will use sometimes the extensional notation, for instance the multiset M such that $M(c) = 2$, $M(d) = 1$ and $M(e) = 0$ for $e \neq c, d$, will be represented by $M = \{c, c, d\}$. We denote by $\mathcal{M}(S)$ the set of the multisets on S, and by $\mathcal{M}_{\mathrm{F}}(S)$ the set of the *finite* multisets on S.

The *multiset sum* is a (total) function $\oplus : \mathcal{M}(S) \times \mathcal{M}(S) \to \mathcal{M}(S)$ defined as

$$(M \oplus M')(c) = M(c) + M'(c).$$

The *multiset difference* is a partial function $\ominus : \mathcal{M}(S) \times \mathcal{M}(S) \to \mathcal{M}(S)$ such that $M \ominus M'$ is undefined if, for some $c \in S$, $M(c) < M'(c)$. Otherwise, for each $c \in S$

$$(M \ominus M')(c) = M(c) - M'(c).$$

Another notion which will be useful is the *multiset inclusion*, denoted by $\subseteq$, defined as

$$M \subseteq M' \text{ iff } \forall c \in S.\, M(c) \leq M'(c).$$

Note that $M \ominus M'$ is defined if and only if $M' \subseteq M$.

Definition 2.1 A *simple linear constraint system* is a pair $\langle S, \vdash \rangle$ where S is a set of tokens and the *entailment relation* $\vdash \subseteq \mathcal{M}_{\mathrm{F}}(S) \times S$ satisfies:

(i) $\{c\} \vdash c$,

(ii) if $M_1 \vdash c_1, \ldots M_n \vdash c_n$ and $\{c_1, \ldots, c_n\} \vdash u$ then $M_1 \oplus \ldots \oplus M_n \vdash u$.

Furthermore we assume as elements of S the booleans *true* and *false* such that $\emptyset \vdash \mathit{true}$, and $\{\mathit{false}\} \vdash c$, for any $c \in S$. A multiset M of elements of S is called consistent if $M' \not\vdash \mathit{false}$ for any finite submultiset M' of M.

Definition 2.2 The *linear constraint system* induced by $\langle S, \vdash \rangle$ is a structure $\langle \mathcal{M}(S), \vdash \rangle$ where $\vdash \subseteq \mathcal{M}(S) \times \mathcal{M}(S)$ is the least relation such that

$$\text{if } \forall i \in I.\, M_i \vdash c_i \text{ then } \bigoplus_{i \in I} M_i \vdash \bigoplus_{i \in I} \{c_i\}.$$

It is easy to show that $\vdash$ is a preorder on $\mathcal{M}(S) \times \mathcal{M}(S)$. Conditions (i) and (ii) express in fact a sort of reflexivity and transitivity property respectively. We can obtain an ordering by considering the quotient set of $\mathcal{M}(S)$ w.r.t. the equivalence relation associated to $\vdash$:

$$M \dashv\vdash M' \text{ iff } M \vdash M' \text{ and } M' \vdash M.$$

In the rest of the paper it is not relevant whether we consider the preorder or the order on the quotient set. We will use the notation $M, M' \ldots$ to indicate constraints. The reader can choose to interpret them as multisets or as their equivalence classes.

Let us compare the above definition with the one given in [15], where the reflexivity and transitivity properties are obtained by the following requirements on the constraint system:

(i′) if $c \in M$ then $M \vdash c$,

(ii′) if $M \vdash c_1, \ldots M \vdash c_n$ and $\{c_1, \ldots, c_n\} \vdash c$ then $M \vdash c$.

(Here M represents a set. In the following discussion M represents either a set or a multiset, depending on the context.)

Our condition (i) expresses just reflexivity, and induces a relation less coarse than (i′). Using (i) instead of (i′) allows us to avoid to *consume* unnecessary hypotheses. Remember in fact that in our setting the intended meaning of $M \vdash c$ is that there exists a derivation of c using as resources all the hypotheses of M. Condition (ii′), reformulated for multiset, would be

$$\text{if } M \vdash c_1, \ldots M \vdash c_n \text{ and } \{c_1, \ldots, c_n\} \vdash c \text{ then } M \oplus \ldots \oplus M \vdash c.$$

Our condition (ii) induces a coarser relation, which is necessary in our setting in order to obtain transitivity. For instance, if we have $\{c_1\} \vdash d_1$, $\{c_2\} \vdash d_2$, and $\{d_1, d_2\} \vdash d$, then the system should derive also $\{c_1, c_2\} \vdash d$. In the framework of [15] this is done by deriving $\{c_1, c_2\} \vdash c_1$ and $\{c_1, c_2\} \vdash c_2$ from (i′), and then applying (ii′). In our framework $\{c_1, c_2\} \vdash c_1$ cannot (in general) be derived (because in general c_1 can be derived from c_1 without the use of the hypothesis c_2), but from (ii) we can derive directly $\{c_1, c_2\} \vdash d$.

Another important difference is the following: according to [15], a constraint is the *deductive closure* of a set M of tokens:

$$\bar{M} = \{c \mid \text{there exists a finite } M' \subseteq M \text{ s.t. } M' \vdash c\},$$

and the entailment relation among constraints is defined as superset inclusion. In our framework, a set of tokens and its deductive closure are both constraints, and they cannot be identified because they represent different set of resources. Note that in our case M and $\bar{M}$ are not equivalent: in general $M \not\vdash \bar{M}$ and $\bar{M} \not\vdash M$.

In order to formalize the parameter mechanism of procedure calls we assume a (denumerable) set of variables Var, with typical elements $x, y, z, \ldots$, and for every pair $x, y \in Var$ a renaming operator $rename_{xy} : S \to S$. The constraint $rename_{xy}(c)$ represents the result of replacing x by y in c, and it will be denoted by $c[y/x]$. We assume that the entailment relation models appropriately the concept of renaming. For example, $\{(x = a)[y/x]\} \dashv\vdash \{y = a\}$.

3 The language

In order to compare our language with previous ccp paradigms, we consider a sort of "superlanguage" $\mathcal{L}$ embodying the main mechanisms for communication and synchronization which have been proposed: the *eventual tell* (*tell*), the *atomic tell* (*atell*), the *ask*, and also the *removing ask* (*get*) which is our present proposal. A sublanguage $\mathcal{L}_G$ of $\mathcal{L}$ will be characterized by the set G of basic actions which are available in it. For instance, *atomic ccp* [14] corresponds to $\mathcal{L}_{\{atell,ask\}}$, and *eventual ccp* (or *ccp*, [14, 15]) corresponds to $\mathcal{L}_{\{tell,ask\}}$. In the last section we will compare these two languages and the language we propose, $\mathcal{L}_{\{tell,ask,get\}}$, from the point of view of potential parallelism and expressiveness.

We assume given a cylindrical linear constraint system $\mathbf{L}$, with tokens $c, d, \ldots$ and elements $M, M' \ldots$. The description of the language is parametric with respect to it.

The actions listed above have an argument which is a token. We could be more general and admit a finite multiset as argument, but this would give rise to some technical complications when building the partial order model of the language, possibly deflecting the attention from what we consider to be the main ideas and results of that construction.

The basic actions operate on a common *store*, which ranges on $\mathbf{L}$. The effect of $tell(c)$ consists of simply adding the token c to the current store. The action $atell(c)$ has the same effect, but it can be executed only if the resulting store is consistent. Otherwise, we say that $atell(c)$ is *not enabled*. The action $ask(c)$ is a test on the current store and its execution does not modify the store: $ask(c)$ is *enabled* in the store M iff $M' \vdash c$, for some $M' \subseteq M$. The action $get(c)$ differs from $ask(c)$ because, when enabled, it removes information from the current store. More precisely, it removes from the current store a multiset of tokens $d_1, \ldots, d_m$ that entail c.

The following grammar defines the syntax of the language $\mathcal{L}$; g stands for one of the above actions.

$$\textit{Declarations} \quad D ::= \epsilon \mid p(x) \text{ :- } A \mid D, D$$

$$\textit{Agents} \qquad A ::= stop \mid g \to A \mid A + A \mid A \parallel A \mid p(x)$$

The agent *stop* represents successful termination. The agent $g \to A$ executes g, if enabled, and then it behaves like A. If g is not enabled, then the agent *suspends*, waiting for other (parallel) agents to add information to the store. The choice operator in $A + B$ selects nondeterministically one of the agents A and B which is enabled, that is, the current store allows it to perform one of its initial actions. If neither the agent A nor B is enabled then the composite

agent $A + B$ suspends. Parallel composition is represented by $||$. The situation in which all components of a system of parallel agents suspend is called *global suspension* or *deadlock*. Finally, the agent $p(x)$ is a procedure call, where p is the name of the procedure and x is the actual parameter. The meaning of $p(x)$ is given by a procedure declaration of the form $p(y)$:- A, where y is the formal parameter.

The parameter passing is modeled by renaming the formal parameter y in the body A by the actual parameter x. This is formally expressed by substituting every token c occurring in A by $c[x/y]$. The result of this substitution we denote as $A[x/y]$. Concerning the local variables of the body, we assume a mechanism which takes care of renaming them into fresh ones, thus avoiding possible clashes with the global variables of the store.

In the following, we assume the set of declarations to be fixed. We end this section with an example, illustrating the concept of a get action.

Example 3.1 Assume that $x = y, y = a \vdash x = a$. Then, the result of a get action $get(x = a)$ on the store $\{x = y, y = a\}$ will be the empty multiset. If, on the other hand, we consider the store $\{x = a, y = a\}$, then the result of $get(x = a)$ will be $\{y = a\}$, because $x = a \vdash x = a$. Note that in our setting $\{x = y, y = a\}$ and $\{x = a, y = a\}$ are not necessarily equivalent.

4 The operational model

In this section we introduce a transition system which allows the construction of a partial order model describing the *causal dependencies* among (occurrences) of actions. In order to describe the *data dependencies* among actions, we represent a store as a multiset of occurrences of those actions which have been executed. Furthermore with each action is associated the set of its "causes", namely, those actions in the store that enabled it. Actions in the store are supposed to be labeled in order to distinguish between different occurrences of the same action[1]. In the transition system given below however we suppress for technical convenience the explicit use of a labeling mechanism and identify an action with its occurrence.

Sets of elements $\langle D, a\rangle$, where a is an action and D is a set of actions, we denote by $M, M', \ldots$. The store represented by M can be obtained as follows:

$$Store(M) = \{tell(c), atell(c) \mid \text{there exists } \langle D, tell(c)\rangle \in M, \langle D, atell(c)\rangle \in M \text{ such that } tell(c) \notin D,\ atell(c) \notin D, \text{ for any } \langle D, get(d)\rangle \in M\}$$

So $Store(M)$ consists of all the tell actions of M which do not occur as a cause of a get action. We will identify $Store(M)$ with its corresponding multiset of constraints.

A configuration is a pair $\langle A, M\rangle$ where A is an agent and M consists of all the actions with their corresponding causes, which have occurred till then. Since the intended model is *true concurrent*, we allow also transitions which represent concurrent execution of more than one action, provided they are independent. Hence in general a transition will be labeled by sets of actions. Summarizing, a transition has the following format:

$$\langle A, M\rangle \xrightarrow{\alpha} \langle A', M'\rangle$$

where A and A' are agents, M and M' are multisets of actions togetther with their causes, and α is a set of acions.

The basic intuition underlying this transition can be described as follows:

[1] All actions in a store represent different occurrences, hence, strictly speaking, the store is a set, not a multiset.

A1	$\langle tell(c) \to A, M\rangle \xrightarrow{tell(c)} \langle A, M \oplus \{\langle \emptyset, tell(c)\rangle\}\rangle$	
A2	$\langle atell(c) \to A, M\rangle \xrightarrow{atell(c)} \langle A, M \oplus \{\langle Store(M), atell(c)\rangle\}\rangle$	if $Store(M) \oplus \{atell(c)\}$ is consist
A3	$\langle ask(c) \to A, M\rangle \xrightarrow{ask(c)} \langle A, M \oplus \{\langle D, ask(c)\rangle\}\rangle$	if $D \subseteq Store(M)$ and $D \vdash c$
A4	$\langle get(c) \to A, M\rangle \xrightarrow{get(c)} \langle A, M \oplus \{\langle D, get(c)\rangle\}\rangle$	if $D \subseteq Store(M)$ and $D \vdash c$

Table 1: Basic Actions.

R1	$\dfrac{\langle A, M\rangle \xrightarrow{\alpha} \langle A', M'\rangle}{\langle A + B, M\rangle \xrightarrow{\alpha} \langle A', M'\rangle}$	
R2	$\dfrac{\langle A, M\rangle \xrightarrow{\alpha} \langle A', M'\rangle}{\begin{array}{l}\langle A \parallel B, M\rangle \xrightarrow{\alpha} \langle A' \parallel B, M'\rangle \\ \langle B \parallel A, M\rangle \xrightarrow{\alpha} \langle\langle B \parallel A', M'\rangle\end{array}}$	$atell(\alpha) \neq \emptyset$
R3	$\dfrac{\langle A, M_1\rangle \xrightarrow{\alpha} \langle A', M_1'\rangle \quad \langle B, M_2\rangle \xrightarrow{\beta} \langle B', M_1'\rangle}{\langle A \parallel B, M_1 \oplus M_2\rangle \xrightarrow{\alpha \oplus \beta} \langle A' \parallel B', M_1' \oplus M_2'\rangle}$	$atell(\alpha) = \emptyset$ and $atell(\beta) = \emptyset$
R4	$\dfrac{\langle A[y/x], M\rangle \xrightarrow{\alpha} \langle A', M'\rangle}{\langle p(y), M\rangle \xrightarrow{\alpha} \langle A', M'\rangle}$	$p(x)$:- A is the declaration for p

Table 2: The transition system L. By $atell(\alpha)$ we denote the set of atomic tell actions occurring in α.

> Given a store M the agent A evolves into the agent A' by the concurrent execution of the actions of α, with a resulting store M'.

Table 1 describes the transition rules for the basic actions. The rules for the choice and parallel operator and recursion are given in Table 2.

Rule **R2** rules out the concurrent execution of an atomic tell action with other tell actions. Note that otherwise we would have to check whether the resulting store is consistent; an inconsistency then would require an 'undoing' of the tell actions which can be considered as a highly unrealistic implementation of the consistency check. Rule **R3** applies in all the other cases. The concurrent execution of get actions is controlled by restricting access to disjoint parts of the common store. We implicitly assume that M is partitioned in M_1 and M_2 in such a way that if $\langle D, get(c)\rangle \in M_i$ then $D \subseteq Store(M_i)$, for $i = 1, 2$.

Rule **R3** forces all parallel agents to make a transition step at the same time (synchronous cooperation). In order to allow each component to proceed "at its own speed", we allow an agent to perform an *idling step*, which is described by the axiom of Table 3. Intuitively, an idling step represents the fact that the agent "skips a turn".

I	$\langle A, M\rangle \xrightarrow{\emptyset} \langle A, M\rangle$

Table 3: The idling step.

The rules for parallel composition can be best understood in terms of the following architecture: to optimalize the degree of parallelism the store is modeled by a 'ring of tokens', that is, a data-flow like structure from which each process can select some tokens and to which tokens can be added. Note that in this way get actions can be executed in parallel since disjoint access to the common store is automatically guaranteed. On the other hand, the consistency check forces the atomic tell actions to interleave. Ask actions can be implemented by creating private copies of the selected tokens.

5 Partial order semantics

In this section we develop a partial order semantics for our language which describes the dependencies among (occurrences of) actions in possible runs. Our method is as follows. First we derive the data dependencies from the transition system, then the control dependencies, which can be obtained statically. Finally, the intended model is constructed by combining those informations. Formally the dependencies are expressed by (strict) orderings, hence the combination just corresponds to take the union of the two relations. For technical convenience we restrict the construction of a partial order semantics to finite agents.

In order to obtain the data dependencies, we first define a semantics $\mathcal{O}$ which associates to each agent all its final results.

Definition 5.1 The operational semantics of an agent A, $\mathcal{O}(A)$, is defined as follows:

$$\mathcal{O}(A) = \{M \mid \langle A, \emptyset\rangle \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \langle B, M\rangle \not\longrightarrow\}$$

(Here $\langle B, M\rangle \not\longrightarrow$ indicates that the agent B, given M, can perform only idling steps.)

Note that $M \in \mathcal{O}(A)$ implicitly defines a partial order semantics describing the data dependencies: let the relation $<_M$ be the least transitive relation on the actions of M such that $a <_M b$ iff a occurs in the dependency set of b in M. Note that $<_M$ is irreflexive and antisymmetric by definition. In fact no (occurrence of an) action can depend upon itself or upon actions which have still to be performed. In the sequel we will identify M with its associated partial order as given by $<_M$.

Let us now consider the control dependencies among actions of an agent. As announced before, they can be derived statically from the structure of the agent.

Definition 5.2 Let A be a finite agent (with all actions occurring in it uniquely labeled). Define $\mathcal{C}(A)$ as the collection of partial (strict) orders given by:

$$\begin{array}{lcl}
\mathcal{C}(stop) & = & \{(\{stop\}, \emptyset)\} \\
\mathcal{C}(g \rightarrow A) & = & \{g \rightarrow (P, <) \mid (P, <) \in \mathcal{C}(A)\} \\
\mathcal{C}(A + B) & = & \mathcal{C}(A) \cup \mathcal{C}(B) \\
\mathcal{C}(A \parallel B) & = & \{(P, <) \cup (P', <') \mid (P, <) \in \mathcal{C}(A),\ (P', <') \in \mathcal{C}(B)\}
\end{array}$$

Here $g \rightarrow (P, <)$ stands for $(P \cup \{g\}, \mathcal{T}(< \cup \{\langle g, g' \rangle \mid \ g' \in P\}))$, where $\mathcal{T}(\rho)$ indicates the transitive closure of the relation ρ. Furthermore, $(P, <) \cup (P', <')$ stands for $(P \cup P', \mathcal{T}(< \cup <'))$.

Note that the above definition is based on the implicit assumption of a mechanism which allows to distinguish among different occurrences of an action. Thus, for example, the two $tell(c)$ actions occurring in $(tell(c) \rightarrow A) \parallel (tell(c) \rightarrow B)$ are assumed to be different and recognizable.

The following definition shows how to combine the data and the control dependencies.

Definition 5.3 Let A be a (finite) agent. Given $M \in \mathcal{O}(A)$, the partial order associated to M, $\mathcal{P}(M)$, is the structure $(P, \mathcal{T}(<_d \cup <_c))$, where $(P, <_d) = M$, and $<_c$ denotes the restriction to P of a relation $<$ such that $(P', <) \in \mathcal{C}(A)$, for some P' which contains all the actions of M.

In the previous definition, P' represents a computation of which the computation leading to M is prefix. In general P is shorter because the computation of an agent suspends when no actions are enabled, whereas P' is obtained simply by unfolding the program without taking into account possible suspensions. Note that there might be more than one such P' because of the possible presence of choice points in the continuation of the computation leading to M. However, the possibility of many such P''s is not relevant in the previous definition: all the corresponding relations will be the same when restricted to P.

We can now define the partial order semantics of an agent A:

Definition 5.4 The partial order semantics of A is

$$\mathcal{P}(A) = \{\mathcal{P}(M) \mid \ M \in \mathcal{O}(A)\}$$

The obvious question now arises whether the above definition of the set of partial orders associated with an agent adequately captures the degree of parallelism. More precisely, whether the degree of parallelism given by the partial order model is neither too restrictive nor too liberal. The adequacy of the partial order model is measured with respect to the possible *observations*.

Intuitively, an observation is the sequence of sets of (labeled) actions which are performed during a possible run. We assume that the observer is capable to observe actions which are performed at the same time (represented as a set).

Definition 5.5 The operational semantics of an agent A, $Obs(A)$, is defined as follows:

$$Obs(A) = \{\alpha_1, \ldots, \alpha_n \mid \ \langle A, \emptyset \rangle \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \langle B, M \rangle \not\longrightarrow\}$$

(Here $\langle B, M \rangle \not\longrightarrow$ indicates that the agent B, given M, can perform only idling steps.)

Correctness of the partial order model, i.e. the guarantee that it is not too liberal, is ensured by the fact that all linearizations of a partial order associated to an agent represent valid observations. Intuitively, the set of linearizations of a partial order $(P, <)$ consists of all those sequences σ of sets of actions of P such that whenever $a < b$ then the set containing a precedes the set containing b in σ. Formally:

$$Lin(P, <) = \{X_1, \ldots, X_n \mid \ \bigcup_1^n X_i = P \text{ and } a < b \text{ implies } a \in X_i, b \in X_j \text{ for some } i < j \ \}$$

The correctness of the partial order model is then given by the following theorem.

Theorem 5.6 (Correctness) *For any agent A we have*

$$\bigcup\{Lin(P,<) \mid (P,<) \in \mathcal{P}(A)\} \subseteq Obs(A).$$

The reverse inclusion which we could call *completeness*, guarantees that our partial order model is not too restrictive.

Theorem 5.7 (Completeness) *For any agent A we have*

$$Obs(A) \subseteq \bigcup\{Lin(P,<) \mid (P,<) \in \mathcal{P}(A)\}.$$

A similar result is stated in [8], but with an essential difference: in [8] the observations of an agent consist of sequences of basic actions generated by an *interleaving* operational semantics (and, correspondingly, linearizations are sequences of actions instead of sequences of sets of actions).

We argue that, having an operational semantics which allows us to observe simultaneity of actions, our result is stronger. In fact, consider the agent $A = a \parallel b$, where a and b are some basic actions. If we would consider an *interleaving* operational semantics, then it would follow that a partial order semantics $\mathcal{P}'(A)$ containing only the two posets with $a < b$, and $b < a$, is correct and complete (according to the formulation of completeness in [8]). On the other hand, having the possibility to observe the simultaneity of actions, we have a way to rule out that semantics as *incomplete*.

However, also our formulation of completeness does not identify a *best* semantics. More precisely, consider the following definition:

Definition 5.8 Let $\mathcal{P}'$ be a function which maps agents into sets of partial orders. We say that $\mathcal{P}'$ is a *correct interpretation* iff for any agent A

$$\bigcup\{Lin(P,<) \mid (P,<) \in \mathcal{P}'(A)\} \subseteq Obs(A).$$

Now, if we take the interpretation

$$\bar{\mathcal{P}}(A) = \bigcup\{\mathcal{P}'(A) \mid \mathcal{P}' \text{ is a correct interpretation}\}$$

we have that $\bar{\mathcal{P}}$ is correct and complete in the above sense, but it does not coincide (in general) with the model $\mathcal{P}$. For instance, in the case of $A = a \parallel b$, the model $\mathcal{P}$ would contain only the poset where a and b are not related, whereas $\bar{\mathcal{P}}$ contains also the other two posets with with $a < b$ and $b < a$.

However the model $\mathcal{P}$ should be "more general". In what sense precisely is a topic of future research.

6 Language comparison

The model we have presented in the previous section allows us to compare the various sublanguages of $\mathcal{L}$ from the point of view of potential parallelism. In particular we are interested in comparing ccp and atomic ccp with $\mathcal{L}_{\{tell,ask,get\}}$, which we call ccp_{get}.

We can see that ccp_{get} presents an intermediate degree of parallelism: less than ccp, but more than atomic ccp. This is formally justified by the following proposition:

Proposition 6.1 *Let A be an agent, let $\sigma \in \mathcal{O}(A)$, and let $(P, <) \in \mathcal{D}(\sigma)$. Then*

- *If $g, g' \in P$ are atomic tell actions, then either $g < g'$ or $g' < g$.*
- *If $g, g' \in P$ are tell actions, then neither $g < g'$ nor $g' < g$.*
- *If $g, g' \in P$ are ask actions, then neither $g < g'$ nor $g' < g$.*

This proposition states that two atomic tell actions are always executed in interleaving, and two tell or two ask actions are always independent, hence they can be performed simultaneously. Concerning two get actions, they can be always independent (which happens when the resources they need are disjoint), or not. In the second case, they can still be performed in parallel if the store is "rich enough" to entail both of them.

Concerning other dependencies, ask and get actions depend in general on some tell or atomic tell actions. Atomic tell actions can depend on some get actions. In fact the removal of information might eliminate some item which is inconsistent with the constraint to be added. Note that a get action is also capable to restore consistency.

The presence of the ask actions in ccp_{get} might seem redundant, since $ask(c)$ would be equivalent, concerning the results in all possible contexts, to two consecutive actions $get(c)$ and $tell(c)$. However, on the basis of the above discussion, we see that from the point of view of the distributed efficiency it is more convenient to use $ask(c)$.

Let us now consider also one of the main important features of a programming language: its expressiveness. With no doubts, atomic ccp is more expressive than ccp (for a formal proof see [3]), but the necessary sequentialization of atomic tell actions is a serious drawback for distributed implementation. On the other hand, ccp is very suitable for distributed implementation, but many distributed algorithms, which can be easily expressed in atomic ccp, require complicated adaptations in ccp, and it is often necessary to use an arbiter or some kind of centralized control. We argue that ccp_{get} could be a reasonable compromise also in terms of expressiveness.

Let us consider the problem of the dining philosophers, which, according to Dijkstra, can be regarded as a benchmark of expressiveness. This problem has a very simple distributed solution in atomic ccp, which can be obtained by translating the Concurrent Prolog program proposed by Shapiro [17]. In ccp, a solution can be obtained by translating the PARLOG program proposed by Ringwood [10]. This solution, which is rather complicated (70 lines of code), uses a series of agents to implement each state a philosopher can occupy–thinking, preparing to eat, and eating. Philosophers circulate along data streams among these process-states. The agents representing the states can be regarded as a sort of central entity which control the overall situation.

In ccp_{get} a simple, natural and distributed solution can be obtained as follows. Assuming that there are n philosophers, we initialize the store with $n-1$ tokens, representing available tickets, and n tokens representing the forks. When a philosopher wants to eat, he must first get a ticket. If he succeeds (i.e. if there are tickets still available), then he can get the forks, when available. When he has eaten, he releases (by tell actions) forks and ticket. This algorithm, which is inspired by [4], is proved to be deadlock-free.

References

[1] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with Built-in inheritance. *New Generation Computing* 9, 1991.

[2] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM TOPLAS* 13(1):99–123,1991.

[3] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison. *Information and Computation.* To appear.

[4] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM* 32(4):445–458, 1989.

[5] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I).* North-Holland, 1971.

[6] N. Kobayashi and A. Yonezawa. ACL–A Concurrent Linear Logic Programming Paradigm. In *Proc. of the International Logic Programming Symposium*, The MIT Press, 1992.

[7] D. Gelernter. Generative Communication in Linda. *ACM TOPLAS* 7(1):80–112, 1985.

[8] U. Montanari and F. Rossi. True concurrency in concurrent constraint programming. In *Proc. of the International Conference on Logic Programming*, The MIT Press, 1991.

[9] U. Montanari and F. Rossi. Graph rewriting for a partial order semantics of concurrent constraint programming. *Theoretical Computer Science*, Special issue on graph grammars, 1992.

[10] G.A. Ringwood. Parlog86 and the dining logicians. *Comm. ACM* 31(1):10–25, 1988.

[11] F. Rossi. *Constraints and Concurrency.* PhD thesis, University of Pisa, March 1993.

[12] V.A. Saraswat. *Concurrent Constraint Programming.* PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, USA, 1992.

[13] V.A. Saraswat and P. Lincoln. Higher-order, linear concurrent constraint programming. Technical report, Xerox PARC, 1992.

[14] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.

[15] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. of the eighteenth ACM Symposium on Principles of Programming Languages.* ACM, New York, 1991.

[16] D. Scott. Domains for denotational semantics. In *Proc. of ICALP*, 1982.

[17] E. Shapiro. Embedding Linda and other joys of concurrent logic programming. Technical Report CS89–07, The Weizmann Institute of Science, Rehovot, Israel, 1989.

Modular Transformations of CLP Programs.

Sandro Etalle[1,2], Maurizio Gabbrielli[1]

[1] CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

[2] Dipartimento di Matematica Pura ed Applicata,
Università di Padova,
Via Belzoni 7, 35131 Padova, Italy.

email: {etalle, gabbri}@cwi.nl

Abstract

In this paper we propose an unfold/fold transformation system for Constraint Logic Programs. The framework is inspired by the one of Tamaki and Sato for pure logic programs [18]. The use of CLP permits a more concise definition for the folding operation. We provide conditions for applying the system in a modular way. Under these conditions the system is correct wrt a semantics (Ω-semantics) which is compositional wrt the union of programs. As corollaries we also prove the correctness of the non-modular system wrt the answer constraint semantics and the least model semantics. Finally, we show how these results can be applied to the logic programming case.

1 Introduction

As shown by a number of applications, program transformation is a powerful methodology for program development. In this field, the unfold/fold transformation rules were first introduced by Burstall and Darlington [5] for transforming clear, simple functional programs into equivalent, more efficient ones, and then adapted to logic programs both for program synthesis and for program specialization and optimization. Soon later, Tamaki and Sato [18] proposed an elegant framework for the transformation of logic programs based on unfold/fold rules. Their system was proved to be correct wrt the least Herbrand model semantics [18] and the computed answer substitution semantics [14]. The system was then extended to logic programs with negation and serious research effort has been devoted to proving its correctness wrt the various semantics available for normal programs.

The aim of this paper is to present an unfold/fold transformation system for CLP based on [18]. The full use of CLP allows us to give new applicability conditions for the folding operation which are more concise and simple than the ones in the literature, in particular the use of substitutions is avoided.

We also introduce a definition of *modular* transformation sequence, which is obtained by adding to the standard definition some new simple applicability conditions. This allows us to transform separate parts of programs independently, and then to combine the results of transformation while preserving the original meaning of the program. This kind of modularity is particularly relevant from a practical point of view. Even if the program is not completely specified in all its components, we would often like to perform some transformations on it without affecting its original meaning. Such a possibility supports an incremental development of programs, according to a well established software-engineering technique. Each time a new part of program is added, instead of transforming the whole program from scratch, we can compose the transformed version of the new piece of program with the transformed version of the old one.

Our system is proved to be correct wrt a generalization of the Ω-semantics ([4]) for constraint logic programs. There are two main reasons that lead us to the choice of such a semantics.

First, it is a declarative semantics which generalizes both the least model semantics [12] and the answer constraint semantics [9]. As a corollary, we can then prove the correctness of the system wrt these semantics as well, and this applies also to the standard (non-modular) system.

Secondly, the Ω-semantics is *compositional* wrt the union of programs, that is, the semantics of a program can be obtained from the semantics its subprograms, or modules[1]. As such, the Ω-semantics provides a natural reference semantics for a modular transformation system: the correctness of the system wrt this semantics ensure us that we can safely recompose the results of separate transformations of modules.

Recently, an extension of the Tamaki-Sato method to CLP programs has also been proposed by Bensaou and Guessarian [2], in Section 6 we compare it to the one defined here. Another related work is the one of Maher [17], which considers transformations of deductive databases (with constraints, and allowing negation in the bodies of the clauses), and refers to the perfect model semantics. The main difference between this paper and [17] lies in the presence of negation (which is not allowed here) and the fact that the definition of folding used in [17] is rather restrictive, in particular it lacks the possibility of introducing recursion. This kind of folding has also been investigated in [11, 3].

The paper is organized as follows: Section 2 contains the preliminaries on CLP programs and the definition of the program composition we consider. Section 3 provides the definition of (modular) transformation sequences for CLP. In Section 4 we give the definition of Ω-semantics and we prove the correctness of the transformation. The system is then compared with Tamaki-Sato's for pure logic program in Section 5. Finally, Section 6 concludes by comparing our results to those contained in [2]. All the proofs can be found in [7].

2 Preliminaries: CLP programs

The *Constraint Logic Programming* paradigm CLP(X) (CLP for short) has been proposed by Jaffar and Lassez [12] in order to integrate a generic computational mechanism based on constraints in the logic programming framework. Such an integration results in a framework which preserves the existence of equivalent operational, model-theoretic and fixpoint semantics. Indeed, as discussed in [17], most of the results which hold for pure logic programs can be lifted to CLP in a quite straightforward way.

The reader is assumed to be familiar with the terminology and the main results on the semantics of (constraint) logic programs. In this subsection we introduce some notations we will use in the following. Lloyd's book and the survey by Apt [16, 1] provide the necessary background material for logic programming theory. For constraint logic programs we refer to the original paper [12] by Jaffar and Lassez and to the recent survey [13] by Jaffar and Maher.

The CLP framework was originally defined using a many-sorted first order language. In this paper, to keep the notation simple, we consider a one sorted language (the extension of our results to the the many sorted case is immediate).

We assume programs defined on a signature with predicates where the set of predicate symbols, denoted by Π, is partitioned into two disjoint sets: Π_c (containing predicate symbols used for constraints) which contains also the equality symbol "=", and Π_b (containing symbols for user definable predicates).

We find convenient to use the notation $\exists_{-\tilde{X}}\ \phi$ from [13] to denote the existential closure of the formula ϕ *except* for the variables $\tilde{X}$ which remain unquantified. The notations $\tilde{t}$ and $\tilde{X}$ will denote a tuple of terms and of distinct variables respectively, while $\tilde{B}$ will denote a (finite, possibly empty) conjunction of atoms. The connectives "," and $\Box$ will often be used instead of "$\wedge$"to denote conjunction.

A *primitive constraint* is an atomic formula $p(t_1, \ldots, t_n)$ where $p \in \Pi_c$. A *constraint* is a first order formula built using primitive constraints[2]. A CLP rule is denoted by $H \leftarrow c \Box B_1, \ldots, B_n.$ where c is

[1] This kind of compositionality is also called OR-compositionality or $\cup$-compositionality.

[2] In the original formulation ([12]) a constraint was defined as a conjunction of atoms. We follow here the more general definition given in [13], since our results do not need such a restriction.

a constraint, H (the head) and $B_1, \ldots, B_n$ (the body) are atomic formulas which use predicate symbols from Π_b only. Analogously a *goal* (or query) is denoted by $c \square B_1, \ldots, B_n$.

The semantics of CLP programs is based on the notion of *structure* $\mathcal{D}$ which gives an interpretation for the constraints.

Given a structure $\mathcal{D}$ and a constraints c, $\mathcal{D} \models c$ denotes that ϕ is true under the interpretation provided by $\mathcal{D}$. Moreover if ϑ is a valuation (i.e. a mapping of variables on the domain D), and $\mathcal{D} \models c\vartheta$ holds, then ϑ is called a $\mathcal{D}$-*solution* of c ($c\vartheta$ denotes the application of ϑ to the variables in c).

We refer to the mentioned papers for the basic notions and results concerning the semantics of CLP. Here we just recall that there exists ([12]) the least $\mathcal{D}$-model of a program P and this is considered the standard semantics of P. As for the operational model of CLP, it is obtained from SLD resolution by simply substituting $\mathcal{D}$-solvability for unifiability. More precisely, a derivation step for a goal G : $c_0 \square B_1, \ldots, B_n$ in the program P results in a goal of the form $c_1 \square B_1, \ldots, B_{i-1}, \tilde{B}, B_{i+1}, \ldots, B_n$ if B_i is the atom selected by the selection rule and there exists a clause in P renamed apart (i.e. with no variables in common with G) $H : -c \square \tilde{B}$ such that $c_1 : (c_0 \wedge (B_i = H) \wedge c)$ is $\mathcal{D}$-satisfiable, that is, $\mathcal{D} \models \exists\, c_1$. Here and in the following, given the atoms A, H, we write $A = H$ as a shorthand for:

- $a_1 = t_1 \wedge \ldots \wedge a_n = t_n$, if, for some predicate symbol p and natural n, $A = p(a_1, \ldots, a_n)$ and $H = p(t_1, \ldots, t_n)$.
- *false*, otherwise.

This notation readily extends to conjunctions of literals.

The notion of derivation (in the program P) of a goal G_i from a goal G is the usual one and in the following is will be denoted by $G \overset{P}{\rightsquigarrow} G_i$. A derivation is *successful* if it is finite and its last element is a goal of the form $(c \square)$. In this case, $\exists_{-Var(G)}\, c$ is called the *answer constraint* [3]. Finally, we need a definition.

Definition 2.1 Let $cl_1 : A_1 \leftarrow c_1 \square \tilde{B}_1$ and $cl_2 : A_2 \leftarrow c_2 \square \tilde{B}_2$ be two clauses. We write

$$cl_1 \simeq cl_2$$

iff for any $i, j \in [1, 2]$ and for any $\mathcal{D}$-solution ϑ of c_i there exists an $\mathcal{D}$-solution γ of c_j such that $A_i\vartheta = A_j\gamma$ and $\tilde{B}_i\vartheta$ and $\tilde{B}_j\gamma$ are equal as multisets. □

For the sake of simplicity, we will denote the $\simeq$ equivalence class of a clause c by c itself.

2.1 Modular CLP Programs

Since we are interested in a modular transformation system, first of all we have to define precisely the notions of module and composition. A module, called Ω-open program for consistency with the notation used elsewhere, is a program P together with a set Ω containing the symbols of those predicate which are only partially specified in P.

Definition 2.2 (Ω-open program, [4]) An Ω-open program (Ω-program for short) is a program P together with a set Ω of predicate symbols. □

An Ω-program P is then a module which can be be composed with other modules which may further specify the predicates in Ω. A typical practical example can be a logic database whose intensional part (i.e. the rules) is completely known while the extensional one (i.e. the facts) is partially specified and could be incrementally added. The composition that we consider here is simply union, modified in order to take into account the "interface" described by the set Ω.

In the following, $Pred(E)$ denotes the set of predicate symbols which appear in the expression E and we say that a predicate p is defined in a program P, if P contains a clause whose head has predicate symbol p.

[3] We follow here the more recent terminology used in [13]. In the original paper ([12]) a derivation step was defined by rewriting in parallel all the atoms of the goal. As far as successful derivation are concerned the two formulations are equivalent. Moreover in [12] the answer constraint was considered c (without quantification).

Definition 2.3 (Ω-union, [4]) Let P and Q be Ω-programs. If $(Pred(P) \cap Pred(Q)) \subseteq \Omega$ then $P \cup_\Omega Q$ is the Ω-program $P \cup Q$. Otherwise $P \cup_\Omega Q$ is not defined. □

Another notion that we have to establish is the kind of "observational" equivalences among programs that we want to maintain. We consider here the answer constraint notion of observable. This is a natural choice since, as previously mentioned, answers constraints are the standard results of CLP computations. Moreover we take into account also the contexts given by $\cup_\Omega$ composition, since these formalize our notion of module. Therefore the following.

Definition 2.4 (Observational equivalences) Let P_1, P_2 be Ω-open programs, we define

- $P_1 \approx_{ac} P_2$
  iff, for any goal G and for any $i, j \in [1,2]$, if there exists a derivation $G \overset{P_i}{\rightsquigarrow} c_i \square$, then there exists a derivation $G \overset{P_j}{\rightsquigarrow} c_j \square$, such that $D \models \exists_{-Var(G)} c_i \leftrightarrow \exists_{-Var(G)} c_j$
- $P_1 \approx_\Omega P_2$
  iff for every Ω-program Q such that $P_i \cup_\Omega Q$, $i \in [1,2]$, is defined, we have that $P_1 \cup_\Omega Q \approx_{ac} P_2 \cup_\Omega Q$. □

So $P_1 \approx_{ac} P_2$ if P_1 and P_2 have the same answer constraints (up to logical equivalence in the structure D), while $P_1 \approx_\Omega P_2$ iff their answer constraints are the same in any $\cup_\Omega$-context. By taking Q as the empty program we immediately see that if $P_1 \approx_\Omega P_2$ then $P_1 \approx_{ac} P_2$. In the following, we will call $\approx_\Omega$ also *compositional equivalence*.

3 Unfold/fold transformations for CLP

In this section we define an unfold/fold system for CLP programs. The system is inspired by the one proposed by Tamaki and Sato [18] for pure logic programs. However, the extension to the context of constraint logic programs requires a generalization of the conditions for the folding operation in [18] which is not straightforward. We believe that the result is worth the effort, as now those applicability conditions can be expressed in a more concise way.

First, it is worth noticing that all the observable properties we refer to are invariant under $\simeq$; this implies that we can always replace any clause cl in a program P with a clause cl', provided that $cl' \simeq cl$. This operation is often useful to *clean up* the constraints, and, in general, to present the clause in a more readable form.

We start from some requirements on the original (i.e. initial) program.

Definition 3.1 (Initial program) We call a CLP program P_0 an *initial program* if the following three conditions are satisfied:

(I1) P_0 is divided into two disjoint sets $P_0 = P_{new} \cup P_{old}$;

(I2) All the predicates which are defined in P_{new} occur neither in P_{old} nor in the bodies of the clauses in P_{new}. □

The following Example is kept simple for the sake of clarity. Programs are written in an unspecified $CLP(X)$ language.

Example 3.2 Let P_0 be the following Ω-program

```
member(Element, List) ←
    Element is an element of the list List.

member(El, List) ← List = [ El | _ ] □ .
member(El, List) ← List = [ _ | List2 ] □ member(El, List2).

meet(Path1, Path2) ←
```

`Path1` and `Path2` intersect in a place which satisfy `good`.

`meet(Path1, Path2)` $\leftarrow$ *true* $\square$
`member(El, Path1), member(El, Path2), good(El).`

where $\Omega = \{\texttt{good}\}$. So `good` is only partially specified in P_0 and its definition, which could involve some arithmetic constraints, can be added later. Moreover we assume that `meet` is the only *new* predicate. So P_{new} consists of the clause defining `meet`. $\square$

The unfolding operation is basic to all the transformation systems, and it consists essentially in applying a resolution step to the unfolded atom in all possible ways. Here its definition is given modulo reordering of the bodies of the clauses, moreover, it is assumed that all the clauses are renamed apart.

Definition 3.3 (Unfolding) Let $cl : A \leftarrow c \square H, \tilde{K}.$ be a clause of a program P, and $\{H_1 \leftarrow c_1 \square \tilde{B}_1, \ldots, H_n \leftarrow c_n \square \tilde{B}_n\}$ be the set of clauses of P for which $c \wedge c_i \wedge (H = H_i)$ is $\mathcal{D}$-satisfiable. For $i \in [1, n]$, let cl'_i be the clause

$$A \leftarrow c \wedge c_i \wedge (H = H_i) \square \tilde{B}_i, \tilde{K}$$

Then *unfolding H in cl in P* consists of substituting cl by $\{cl'_1, \ldots, cl'_n\}$ in P. $\square$

In this situation we say that $cl : A \leftarrow c \square H, \tilde{K}.$ is the *unfolded* clause, while $\{H_1 \leftarrow c_1 \square \tilde{B}_1, \ldots, H_n \leftarrow c_n \square \tilde{B}_n\}$ are the *unfolding* ones.

Example 3.2 (part 2) By unfolding `member(El, Path1)` in the body of the clause defining `meet`, we obtain P_1, which, after cleaning up the constraints is:

```
meet(PathA, PathB) ← PathA = [ Place | _ ]  □
   member(Place, PathB), good(Place).
meet(PathA, PathB) ← PathA = [ _ | PathA' ]  □
   member(Place, PathA'), member(Place, PathB), good(Place).
```

together with the clauses defining predicate `member`. $\square$

In the above example we have already renamed the variables in order to "prepare" the clauses for the next operation: the folding. This operation is often used in order to introduce recursion in the new definitions. Unlike unfolding, the folding operation requires some conditions which ensure its correctness also when modularity is not taken into account. Following [18], we define the transformation sequence and the folding operation in terms of each other.

Definition 3.4 (Transformation sequence) A *transformation sequence* is a sequence of programs $P_0, \ldots, P_n$, $n \geq 0$, such that P_0 is an initial program, and each P_{i+1}, $0 \leq i < n$, is obtained from P_i by unfolding or folding a clause of P_i. $\square$

Here, we also assume the folding and the folded clause to be renamed apart, and, as a notational convenience, that the body of the folded clause had been reordered so that the atoms that are going to be folded are found on its left hand side.

Definition 3.5 (Folding) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence. Let also
$cl : A \leftarrow c_A \square \tilde{K}, \tilde{J}.$ be a clause in P_i,
$d : D \leftarrow c_D \square \tilde{H}.$ be a clause in P_{new}.
If $\tilde{K}$ is an instance of $\tilde{H}$ and $e = \{x_1 = t_1, \ldots, x_m = t_m\}$ is a constraint where $\{x_1, \ldots, x_m\} \subseteq Var(D)$ and $Var(t_1, \ldots, t_m) \subseteq Var(cl)$, then *folding D in cl via e* consists of substituting cl with

$$cl' : \; A \leftarrow c_A \wedge e \square D, \tilde{J}.$$

provided that the following three conditions hold:

(F1) *"If we unfold D in cl' using d as unfolding clause, then we obtain cl back"*:

$$P \models \exists_{-Var(A,J,\tilde{H})}\ c_A \wedge e \wedge c_D \leftrightarrow \exists_{-Var(A,J,\tilde{H})}\ c_A \wedge (\tilde{H} = \tilde{K})$$

(F2) *"d is the only clause of P_{new} that can be used to unfold D in cl'"*:
there is no clause $b: \quad B \leftarrow c_B \square \tilde{L}$ in P_{new} such that $b \neq d$ and $c_A \wedge e \wedge (D = B) \wedge c_B$ is $\mathcal{D}$-satisfiable"

(F3) *"No self-folding is allowed"*:

(a) either the predicate in A is an old predicate;

(b) or cl is the result of at least one unfolding in the sequence $P_0, \ldots, P_i$; □

Here, the constraint e acts as a bridge between the variables of d and cl.

Conditions **F1** and **F2** ensure that the folding operation behaves, to some extent, as the inverse of the unfolding one: the underlying idea is that if we unfolded the atom D in cl' using only clauses from P_{new} as unfolding clauses, then we would obtain cl back. In this context condition **F2** ensures that in P_{new} there exists no clause other than d that can be used as *unfolding clause*. Consequently, the result of such an operation would be the clause:

$$A \leftarrow c_A \wedge e \wedge (D = D') \wedge c'_D \square \tilde{H}', \tilde{J}$$

Where $d' : D' \leftarrow c'_D \square \tilde{H}'$ is an appropriate renaming of d. Now, by the standardization apart, the variables of $c_D, \tilde{H}, c'_D, \tilde{H}'$ which do not occur in D, D', do not occur anywhere else in this clause, so, by explicitating $(D = D')$, we can identify c'_D with c_D and $\tilde{H}'$ with $\tilde{H}$. The previous clause becomes then: $A \leftarrow c_A \wedge e \wedge c_D \square \tilde{H}, \tilde{J}$. Now, by **F1**, this can be rewritten as: $A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \square \tilde{H}, \tilde{J}$, and, because of the constrain $(\tilde{H} = \tilde{K})$, this is equivalent (modulo $\simeq$) to $A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \square \tilde{K}, \tilde{J}$.
Now recall that the folding and the folded clause are assumed to be standardized apart, therefore $\tilde{H}$ has no variables in common with the rest of this clause (that is, with A, c_A, $\tilde{K}$ and J). Since we also assume that $\tilde{K}$ is an instance of $\tilde{H}$, it follows that the constraint $\tilde{H} = \tilde{K}$ can be eliminated, and we have the original clause cl back.

Of course, during a transformation sequence, such an "undo by unfolding" is often not possible; this is due to the fact that the folding clause is usually not found in the "current" program.

Finally, we should mention that the purpose of **F3** is to avoid the introduction of loops which can occur if a clause is folded by itself. This condition is the same one that is found in Tamaki-Sato's definition of folding for logic programs.

Example 3.2 (part 3) We can now fold `member(Place, PathA')`, `member(Place, PathB)`, `good(Place)` in the second clause defining `meet` in program P_1. In this case, the constraint e has to be

$$\texttt{Path1} = \texttt{PathA}' \ \wedge\ \texttt{Path2} = \texttt{PathB}.$$

Checking **F1** is a trivial task: its left hand side is:

```
∃Path1, Path2, El   PathA = [ _ | PathA' ] ∧ Path1 = PathA' ∧ Path2 = PathB'
```

while its right hand side is

```
∃Path1, Path2, El   PathA = [ _ |PathA'] ∧ El = Place
```

And it is immediate to see how one side reduces to the other one. The resulting program, P_3 after cleaning up the constraints, is:

```
meet(Path1, Path2)  ← Path1 = [ El | _ ]  □
    member(El, Path2), good(El).
meet(Path1, Path2)  ← Path1 = [ _ | Path1' ]  □ meet(Path1', Path2').
```

together with the clauses defining predicate `member`

Notice that, because of this last operation, the definition of `meet` is now recursive. □

3.1 A Modular Transformation system

We are interested here in a *modular* transformation system. Previous conditions on the folding operation are sufficient to ensure the correctness of the system wrt the answer constraint and the least $\mathcal{D}$-model semantics (as we prove in the next section). However, as shown by the following example, if we want to obtain compositionally equivalent programs, we need to specify some further applicability conditions. Here and in the sequel, we call an atom Ω-*atom* if its predicate symbol is in Ω.

Example 3.6
(i) First, we cannot allow the unfolding of Ω-atoms. In fact let P_0 be the following program

```
p ← q.
q ← r.
```

where $\Omega = \{\mathtt{q}\}$. If we unfold q in the first clause, we obtain the program P_1:

```
p ← r.
q ← r.
```

Now the two programs are not compositionally equivalent. In fact, if we add the program $Q = \{\mathtt{q}\}$ then the query $\leftarrow \mathtt{p}$ succeeds in $P_0 \cup Q$ and fails in $P_1 \cup Q$ and hence $P_0 \not\approx_\Omega P_1$.

(ii) Secondly, we cannot let a *new* predicate to be also an Ω-predicate. Let P_0 be the following program

```
p ← q.
r ← q.
```

Where $\Omega = \{\mathtt{p}\}$ and $P_{new} = \{\mathtt{p} \leftarrow \mathtt{q}\}$. Since r is an old atom, we can fold q in the second clause of P_0; the resulting program P_1 is

```
p ← q.
r ← p.
```

Again $P_0 \not\approx_\Omega P_1$. In fact, if we add the program $Q = \{\mathtt{p}\}$ we have that the query $\leftarrow \mathtt{r}$ succeeds in $P_1 \cup Q$, but fails in $P_0 \cup Q$. □

Therefore the following.

Definition 3.7 (Modular transformation sequence) Let P_0 be an Ω-program and $P_0, \ldots, P_i$ be a transformation sequence. If

(O1) no Ω-atom is unfolded in $P_0, \ldots, P_i$ and

(O2) no *new* predicate belongs to Ω,

then we call $P_0, \ldots, P_i$ a *modular* transformation sequence. □

So, Ω-atoms can only be folded. Notice that this is what happens in the transformation sequence of Example 3.2, which is indeed a modular transformation sequence.

For $\Omega = \emptyset$, **O1** and **O2** are trivially satisfied and we have a standard (non modular) unfold/fold system. In fact when $\Omega = \emptyset$, composition is allowed only among programs which do not share predicate symbols, and, from a semantic point of view, this is equivalent to consider no composition at all.

4 Correctness of the unfold/fold system

The aim of this section is to show that a modular transformation sequence preserves the compositional equivalence. To this end, first we introduce a semantics ([10]) for CLP which models answer constraints and which is compositional wrt union of programs, then we show that a modular transformation sequence preserves this semantics. The desired result follows from the correctness of the semantics wrt the equivalence $\approx_\Omega$.

4.1 A compositional semantics for CLP

The semantics we introduce now is basically a straightforward lifting to the CLP case of that one defined in [4] for logic programs, where compositionality wrt union of programs is obtained by choosing a semantic domain based on clauses. This semantics can also be seen as a generalization of the answer constraint semantics of [9].

Using $\simeq$ to abstract from purely syntactic details, we define denotations as follows.

Definition 4.1 (Ω-Denotations) Let Ω be a set of predicate symbols and let $\mathcal{C}$ be the set of the $\simeq$-equivalence classes of the clauses in the given language. The *interpretation base* $\mathcal{C}_\Omega$ is the set $\{A \leftarrow c \Box B_1, \ldots, B_n \in \mathcal{C} \mid Pred(B_1, \ldots, B_n) \subseteq \Omega\}$. An Ω-*denotation* is any subset of $\mathcal{C}_\Omega$. □

The following is an operational definition of the Ω-semantics for CLP. An equivalent fixpoint definition can be obtained by using a suitable operator ([10]).

Definition 4.2 ($\mathcal{O}_\Omega(P)$ semantics, [10]) Let P be an Ω-program. Then we define

$$\mathcal{O}_\Omega(P) = \{p(\bar{X}) \leftarrow c \Box B_1, \ldots, B_n \in \mathcal{C}_\Omega \mid \text{ there exists a derivation } true \Box p(\bar{X}) \overset{P}{\rightsquigarrow} c \Box B_1, \ldots, B_n. \}$$

□

The compositionality of previous semantics wrt $\cup_\Omega$ is proved in [10]. From such a result it follows the correctness of the Ω-semantics wrt the equivalence $\approx_\Omega$, as shown by the following Corollary of [10].

Corollary 4.3 Let P, Q be Ω-open programs. If $\mathcal{O}_\Omega(P) = \mathcal{O}_\Omega(Q)$ then $P \approx_\Omega Q$. □

In other words, $\mathcal{O}_\Omega(P)$ contains all the information necessary to model the behaviour of P, in terms of answer constraints, compositionally wrt union of programs. In the particular case $\Omega = \emptyset$, i.e. when all the predicates are completely defined, $\mathcal{O}_\Omega$ coincides with the answer constraint semantics defined in [9]. In this case we have that the semantics is not only correct, but also fully abstract wrt $\approx_{ac}$. In fact in [9] it is proven that $\mathcal{O}_\emptyset(P) = \mathcal{O}_\emptyset(Q)$ iff $P \approx_{ac} Q$.

4.2 Correctness result

We can now state the the main result of the paper: the unfold/fold modular transformation preserves the $\mathcal{O}_\Omega$ semantics.

Theorem 4.4 (Correctness) If P_0 is an Ω-program and $P_0, \ldots, P_n$ is a modular transformation sequence then for any $i, j \in [0, n]$,

- $\mathcal{O}_\Omega(P_i) = \mathcal{O}_\Omega(P_j)$

□

Notice also that, as shown in Example 3.6, conditions **O1** and **O2** are necessary to guarantee the correctness of a transformation sequence wrt the Ω-semantics.

From the correctness of the Ω-semantics wrt $\approx_\Omega$ we can derive the compositional equivalence of all the programs given by a modular transformation sequence.

Corollary 4.5 Let P_0, Q be Ω-open programs and $P_0, \ldots, P_n$ be a modular transformation sequence. If $P_0 \cup_\Omega Q$ is defined, then for any $i, j \in [0, n]$,

(i) $P_i \cup_\Omega Q$ is defined,

(ii) $P_i \approx_\Omega P_j$

□

In particular, we have that, for any program Q, the answer constraint semantics and the least $\mathcal{D}$-models [12] of $P_i \cup_\Omega Q$ and $P_j \cup_\Omega Q$ coincide.

Since both the least $\mathcal{D}$-model and the answer constraint semantics can be seen as abstractions of the Ω-semantics with $\Omega = \emptyset$, Theorem 4.4 implies that these semantics are preserved by any transformational sequence (conditions **O1, O2** are then trivially satisfied). Hence the following.

Corollary 4.6 Let $P_0, \ldots, P_n$ be a (non-modular) transformation sequence. Then, for any $i, j \in [0, n]$,

(i) $\mathcal{O}_\emptyset(P_i) = \mathcal{O}_\emptyset(P_j)$ (the answer constraint semantics of P_i and P_j coincide),

(ii) The least $\mathcal{D}$-models of P_i and P_j are equal. □

5 From CLP to LP

Pure logic programming (LP for short) can be seen as a particular instance of the constraint logic programming scheme. This is obtained by taking as structure the Herbrand universe where "=" is the only predicate symbol for constraints, and it is interpreted as identity.

In the following first we introduce the unfold/fold transformation system for logic programs proposed by Tamaki and Sato [18], then, by using a "canonical mapping" from logic to (pure) CLP programs[4], we show that it can be embedded in the one we presented in the previous section. This will allow us to show that, under the hypothesis expressed by conditions **O1** and **O2**, the unfold/fold system preserves the Ω-semantics for pure logic programs.

5.1 Unfold/fold transformations for LP

First, let us consider the unfold operation. Again, we assume that the clause are standardized apart.

Definition 5.1 (Unfolding, in LP) Let $cl : A \leftarrow H, \tilde{K}$. be a clause of a logic program P, and let $\{H_1 \leftarrow \tilde{B}_1, \ldots, H_n \leftarrow \tilde{B}_n\}$ be the set of clauses of P whose heads unify with H, by mgu's $\{\theta_1, \ldots, \theta_n\}$. For $i \in [1, n]$ let cl'_i be the clause

$$(A \leftarrow \tilde{B}_i, \tilde{K})\theta_i$$

Then *unfolding H in cl in P* consists of substituting cl by $\{cl'_1, \ldots, cl'_n\}$ in P. □

We can now introduce the folding operation. In this context, we adopt the same definitions of *initial program* and of *transformation sequence* given for CLP. Again, we assume the folding and the folded clause to be renamed apart and (for notational convenience) that the body of the folded clause has been reordered (as in Definition 3.5).

Definition 5.2 (Folding, in LP, [18]) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence. Let also
$cl : A \leftarrow \tilde{K}, \tilde{J}$. be a clause in P_i,
$d : D \leftarrow \tilde{H}$. be a clause in P_{new}.
Let also $V = Var(\tilde{H}) \backslash Var(D)$ be the set of local variables of d. If there exists a substitution τ such that $Dom(\tau) = Var(d)$ and the following conditions hold:

(LP1) $\tilde{H}\tau = \tilde{K}$;

(LP2) For any $x, y \in V$

- $x\tau$ is a variable;
- $x\tau$ does not appear in $A, \tilde{J}, D\tau$;
- if $x \neq y$ then $x\tau \neq y\tau$;

(LP3) d is the only clause in P_{new} whose head is unifiable with $D\tau$;

(LP4) one of the following two conditions holds

1. the predicate in A is an old predicate;
2. cl is the result of at least one unfolding in the sequence $P_0, \ldots, P_i$;

then *folding D in cl (via τ)* consists of substituting cl with $cl' : A \leftarrow D\tau, \tilde{J}$. □

[4] *Pure* CLP programs are CLP programs in which the atoms in the clauses, apart from constraints, are always of the form $p(\tilde{X})$, where $\tilde{X}$ is a tuple of distinct variables.

5.2 LP vs CLP

While for the unfold operation it is clear that Definition 5.1 is the counterpart of Definition 3.3, for the fold operation the similarities are less obvious. In order to compare Definitions 3.5 and 5.2 we first need to define formally the "canonical" mapping μ from logic program to pure constraint logic programs.

Definition 5.3 Let $cl : \; p_0(\tilde{t}_0) \leftarrow p_1(\tilde{t}_1), \ldots, p_n(\tilde{t}_n)$ be a clause in LP. Then $\mu(cl)$ is the CLP clause $p_0(\tilde{x}_0) \leftarrow \tilde{x}_0 = \tilde{t}_0 \wedge \tilde{x}_1 = \tilde{t}_1 \wedge \ldots \wedge \tilde{x}_n = \tilde{t}_n \;\square\; p_1(\tilde{x}_1), \ldots, p_n(\tilde{x}_n)$, where $\tilde{x}_0, \ldots, \tilde{x}_n$ are tuple of new and distinct variables. □

By using the correspondence μ, we can compare Definition 3.5 with Definition 5.2. The following theorem shows that if cl' is the result of folding cl with d in the logic program P_i then we can obtain $\mu(cl')$ by folding $\mu(cl)$ with $\mu(d)$ in $\mu(P_i)$. In other words, if d and cl are such that the conditions **LP1, LP2, LP3** and **LP4** are satisfied, and cl' is the result of the folding operation, then by translating the whole program into CLP we end up in a situation in which conditions **F1**, **F2** and **F3** are satisfied. Moreover, by appropriately choosing the constraint c, we obtain $\mu(cl')$ by folding $\mu(cl)$ with $\mu(d)$.

Theorem 5.4 If $P_0, \ldots, P_n$ is a transformation sequence of logic programs, then $\mu(P_0), \ldots, \mu(P_n)$ is a transformation sequence in CLP. □

Clearly, even though a logic program is itself a CLP program, a transformation sequence of logic programs cannot be regarded as a transformation sequence of CLP programs. This is due to the fact that the unfold and fold operations for LP use substitutions whereas those for CLP use constraints. This is the reason why we use the mapping μ for proving the correspondence between the two kind of sequences.

5.3 Semantic Consequences for Logic Programs

Theorem 5.4 allows us to prove a counterpart of Theorem 4.4 for the LP case.

First we need the definition of the original Ω-semantics for pure logic programs [4]. It can be obtained from Definition 4.2 by simply replacing CLP derivations by SLD derivations: $\mathcal{O}_\Omega(P)$ is now the set

$$\{(p(\tilde{X})\vartheta \leftarrow B_1, \ldots, B_n)/\simeq \mid \begin{array}{l}\text{there exists an SLD-derivation } p(\tilde{X}) \overset{\vartheta}{\leadsto}_P B_1, \ldots, B_n. \\ \text{and } Pred(B_1, \ldots, B_n) \subseteq \Omega \;\}.\end{array}$$

where $\simeq$ denotes variance and ϑ is the compositions of the mgu's used in the derivation $p(\tilde{X}) \overset{\vartheta}{\leadsto}_P B_1, \ldots, B_n$. Unsurprisingly, this semantics enjoys the same correctness properties stated for the CLP case stated in and Corollary 4.3 (see [4]).

The semantic equivalence between the original pure logic program P and its CLP version $\mu(P)$ is due essentially to the isomorphism existing between the (lattice structures on) idempotent substitutions and equations [15]. Because of this isomorphism we can then use equivalently idempotent mgu's in SLD derivations or equations in CLP derivations. In the first case, the result of the computation is the composition of the mgu's used, restricted to the variables in the goal. In the second the (equivalent) result is given by the answer constraint. This is formalized in [19] and (by extending the definition of μ to the $\simeq$-equivalence classes) brings to the following conclusion:

$$\mu(\mathcal{O}_\Omega(P)) = \mathcal{O}_\Omega(\mu(P))$$

We can now prove the correctness of the transformation sequence for logic programs wrt the Ω-semantics.

Corollary 5.5 Let $P_0, \ldots, P_n$ be a transformation sequence of pure logic programs and let Ω be a set of predicate symbols. If conditions **O1** and **O2** are satisfied then

- $\mathcal{O}_\Omega(P_0) = \mathcal{O}_\Omega(P_n)$. □

Analogously to the CLP case, the results summarized in Corollary 4.5 apply also to pure logic programs. Their proof follows directly from the previous Corollary. In particular we have that for any P_i, P_j in the transformation sequence and for any program Q, if $P_0 \cup_\Omega Q$ is defined, then a generic goal G has the same computed answer substitutions in $P_i \cup_\Omega Q$ and in $P_j \cup_\Omega Q$, and the least Herbrand models of $P_i \cup_\Omega Q$ and $P_j \cup_\Omega Q$ coincide.

6 Conclusions

A definition of unfold/fold transformation system for CLP based on [18] has also been given by Bensaou and Guessarian in [2], yet there are some substantial differences between [2] and our proposal:

First, the semantics they refer to is an extension to the CLP case of the C-semantics ([6, 8]). Such a semantics characterizes the logical consequences of the program on $\mathcal{D}$-models, but does not allow to model answer constraints. For example, the C-semantics identifies the programs $\{p(X) : -X = a \ \Box.,\ p(X) : -X = Y \ \Box.\}$ and $\{p(X) : -X = Y \ \Box.\}$ which have different answer constraint for the goal $p(X)$, and consequently are not identified by the answer constraint semantics in [9]. We believe that the answer constraints semantics provides a better reference semantics for unfold/fold systems, since answer constraints are the most natural properties that one would like to preserve while transforming programs. Moreover, the C-semantics can be obtained as an abstraction (upward closure) of the answer constraint semantics. As a consequence our correctness result are more general.

A second relevant difference is due to the fact that modularity is not take into account in [2].

Finally, a third point where our approaches depart from each other is that in [2] the folding conditions are obtained by a straightforward extension to the CLP case of **LP1**...**LP4**. In this respect, CLP programs are there treated as "extended" logic programs. Of course, [2] allows a more uniform treatment of LP and CLP, and, for those who know [18], it provides a more familiar definition of folding. On the other hand, our choice of departing from the notation of [18] leads to a definition of folding which is more compact and does not need the use of substitutions.

To conclude, the contributions of this paper can be summarized as follows.

We have defined an unfold/fold transformation system for CLP based on the framework of Tamaki and Sato for logic programs [18]. The use of CLP allows us to express the applicability conditions for the folding operation in a more concise and elegant way.

A definition of a modular transformation sequence is given by adding to the usual definition some further simple applicability conditions. These conditions are shown to be necessary and sufficient to guarantee the correctness of the system wrt the Ω-semantics. Since this semantics is compositional wrt the union of programs, this provides a natural theoretical framework for the modular transformation of CLP programs. To the best of our knowledge, this is the first study of the issue of modularity in the context of transformation systems. Moreover, since the Ω-semantics is a generalization of the answer constraint semantics [9] and of the least model semantics [12], the correctness of the (normal, non-modular) system wrt those semantics follows as a corollary.

Finally, the relations between transformation sequences for CLP and LP are discussed. By "canonically" mapping into CLP the transformation system for LP proposed by Tamaki and Sato [18], we prove that, under the "modular" conditions **O1** and **O2**, the Ω-semantics is preserved by the transformation system also in the LP case.

Acknowledgements

The authors want to thank Annalisa Bossi and the referees for their useful suggestions.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.

[3] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

[4] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[5] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*. 24(1):44–67, January 1977.

[6] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[7] S. Etalle and M. Gabbrielli. Modular Transformations of CLP Programs. Technical report, CWI, Amsterdam, 1994. to appear.

[8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[9] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238–252. The MIT Press, Cambridge, Mass., 1991.

[10] M. Gabbrielli and G.M. Dore G. Levi. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 1994.

[11] P.A. Gardner and J.C. Shepherdson. Unfold/fold transformations of logic programs. In J-L Lassez and editor G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. 1991.

[12] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[13] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 1994. To appear.

[14] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 413–422. Institute for New Generation Computer Technology, Tokyo, 1988.

[15] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[16] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[17] M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[18] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[19] D. A. Wolfram, M.J. Maher, and J-L. Lassez. A Unified Treatment of Resolution Strategies for Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 263–276, 1984.

11-07-94

BinProlog Implementation of an AC-5 Based Finite Domains Solver

R.Bisdorff, S. Laurent & L. Marcellin
STADE
Centre de Recherche Public - Centre Universitaire
13, rue de Bragance
L-1255 Luxembourg

Keywords: arc-consistency algorithm, blackboard, BinProlog

Introduction

In this abstract, we describe the implementation of a finite domains solver in BinProlog [5] using its inbuilt blackboard facilities [6] as global data stores for finite domain variables and their associated constraints graph. Constraint programming over finite domains uses, for main operations, local node- and arc-consistency propagation over the constraints graph. To do so, we use the generic arc-consistency algorithm AC-5 [3]. In our implementation test we consider basic constraints like linear equations, disequations and inequalities over natural number terms and constants.

First, we present the basic ideas of our finite domain solver using global prolog structures. Next, we describe the blackboard facilities in BinProlog and their use in our program. Then we discuss a general implementation of a finite domain solver, built on the AC-5 algorithm, for basic binary constraints. Some experimental results are also given.

1. Blackboard for a finite domain solver

Intuitively, one could see the domain and constraints related to a variable as global data to the corresponding finite system. Indeed, defining and modifying a variable domain could correspond to accessing this object, changing its content and put it back in its initial place.

At the ICLP'93, a special workshop dealt with blackboard-based logic programming. Particularly, the presented BinProlog non-associative blackboard [6] seemed to be an appropriate tool to implement our concept. Actually, instead of «assert and retract», this prolog proposes an efficient access to terms on blackboard through a 2-key hashing table.

Our system is based on the AC-5 algorithm which is an improved version of the one used by CHIP called AC-3 [1]. Through the implementation of this constraints propagation method, the basic mechanisms and also the complexity of a complete finite domain solver will be analysed.

2. Global data structure

BinProlog is proposing a permanent data store called blackboard, in which Prolog terms may be saved. Data access is achieved through a double key hashing mechanism with the help of init, get and a modify functions [6]. Two implementations of AC-5 were elaborated, one with global variables resisting backtracking and the other using backtrackable variables.

2.1. Non backtrackable variables

Non backtrackable variables may be defined with the `bb_def(key1, key2, value)` BinProlog primitive which resists to the backtracks of the Prolog resolution tree. Moreover, `bb_val(key1,key2,Value)` and `bb_set(key1,key2,value_new)` allow respectively to read and modify the object identified by `key1` and `key2`.

In our first implementation, the list of domain variables, their actual domains, the binary constraints between them and the constraints graph are considered as global non backtrackable data. Apart from the binary constraints, all the preceding global data need only one key to be accessed. For example, the solution generator gets the list of variables to be instantiated through `bb_val(variables, Variables_List)`.

A binary constraint relating two domain variables may be seen as a predicate giving the success or failure of the constraint for particular values. Arithmetic constraints are represented through a Prolog term of the following shape: `(v1, Arithmetic_operator, v2, Relational_Operator, Val)`, where `Arithmetic_Opreator` $\in$ {=, -} and `Relational_Operator` $\in$ {<, >, =, >=, <=, ==, =\=}. Therefore, for instance, `bb_val(v1,v2,[(v1,+,v2,==,2),(v2,-,v1,=\=,0),(v2,+,v1,<,7)])` gives the constraints linking the variables `v1` and `v2`.

2.2. Backtrackable variables

Backtrackable variables may be stored and accessed on the blackboard through the following predicates: `Key1#Key2 => Value` or `lval(Key1,Key2,Value)`. Modifying the value is achieved by copying the new value to a new address and changing the corresponding references, i.e. `Key1_new # Key2 => New_Value, setref(Key1,Key1_new)`. At least one of the two keys has to be a Prolog variable and, also, the value cannot be used on the identifier side, i.e. as a key.

During AC-5 execution it is necessary to compare the variables. However, it is impossible to test if two Prolog variables are equal or not. To overcome this problem, in the second implementation, a static symbol is associated to each variable and is placed on the blackboard as, for instance, `symbol # X => sx`. Then the comparison between variables is done on the related symbols. Furthermore, as a key cannot be put on the blackboard, the constraints list for 2 variables is placed on the blackboard through non backtrackable entry with their symbols identifier, i.e. `bb_def(sx, sy, List_Constraints)`.

3. AC-5 Implementation

3.1. Arc-consistency algorithm

The AC-5 algorithm uses a special queue containing elements of the form *((I,J),V)*, where *(I,J)* is an arc of the constraints graph and *V* a value withdrawn from the domain of variable *J*. The operations needed on this queue are *init*, *append*, *remove* and *emptiness queue checking* predicates which are immediate in Prolog. AC-5 needs furthermore the following operations on domains: deleting values and list of values, accessing successor and predecessor of values, checking existence of values, and intersecting domains with one another. They all use the preceding defined blackboard primitives.

The `general_arc_cons(Constraint,I,J,Delta)` predicate reads the domains of variables *I*

and *J* from the blackboard, and for each value of*I* checks the constraints against all values of domain *J*. The list *Delta* gives all value of domain *I* for which there is no valid value in the domain *J*.

The `general_local_arc_cons(Constaint,I,J,V,Delta)` predicate reads in the domains of *I* and *J* and returns in *Delta* all the values *V* verifying the predicate `c(Constraint,I,J,V,W)` and not verifying this constraint for all other values *W'* of domain *J*.

AC-5 runs in $O(ed^2)$ where *e* is the number of edges in the constraints graph and *d* the size of the largest domain. But, for important classes of constraints such as functional, anti-functional, element and monotonic constraints, the local- and arc-consistency functions can be specialized to produce an *O(ed)* time complexity. There is a different implementation for each of these particular algorithm and a general predicate chooses, according to the type of the constraint, the one that has to be called.

3.2. Experimental results

3.2.1. Using global variables in Prolog by BIM

In order to test the specific performances of the BinProlog implementation, we built a non backtrackable version of our solver in Prolog by BIM using the record primitive showing a comparable behaviour as the blackboard primitives of BinProlog. Prolog by BIM does not support backtrable blackboard variables as proposed by BinProlog.

The performances of the blackboard operations (experimented on a Sun Sparc10 machine under Sun OS 4.1.2) are quite comparable in time, even if the Prolog by BIM is as a standard Prolog engine much faster then the BinProlog engine. Comparison on a single loop with 100.000 iterations and with 200.000 iterations showed respectively for BinProlog, 167 and 667 milliseconds, whereas Prolog by BIM made it in 99 respectively 200 milliseconds. Testing our solver on two simple test problems showed that the Prolog by BIM version is approximately twice as fast as the corresponding BinProlog's one.

3.2.2. Specialized versus non-specialized algorithm

We have made some test runs with a non-specialized and a specialized version of the AC-5 propagation with a four variables {`x,y,z,w`} example related by the following constraints[1]:`x #>= z, X + Y #= 4, X #< Y, W - Y #= 5, Y- Z #\= 1.`

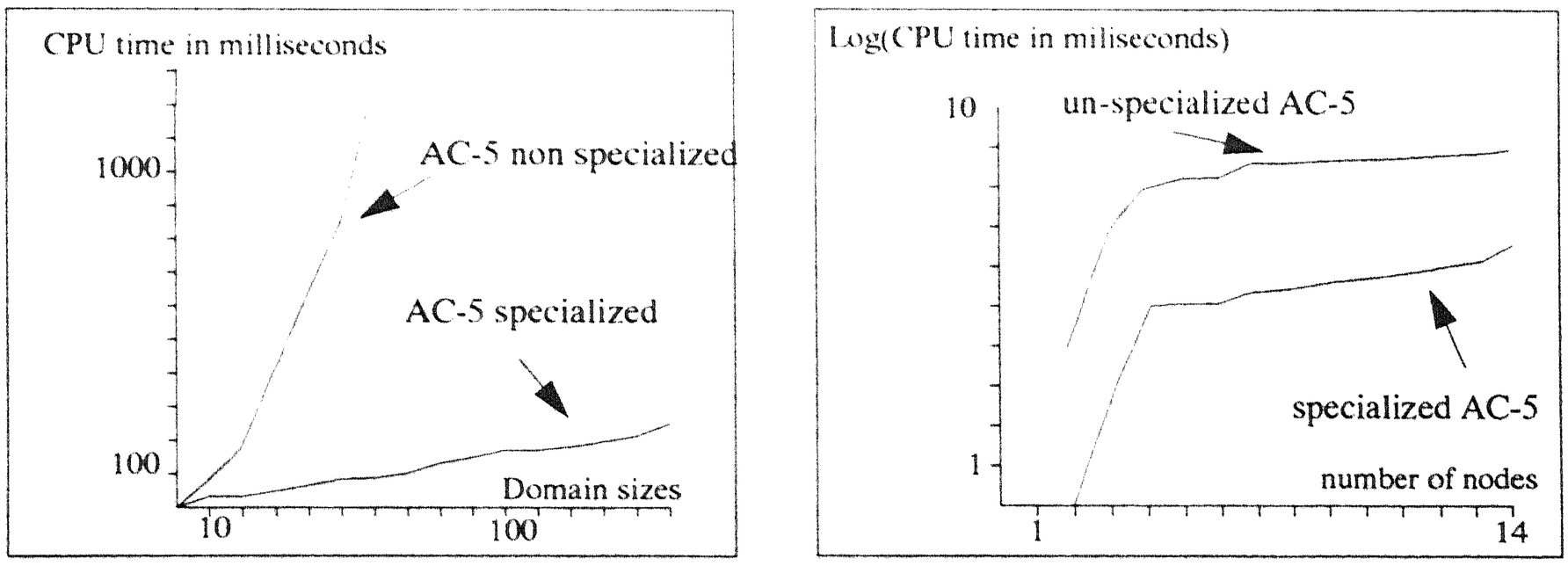


Figure 1: Comparing the non specialized and the specialized version of the AC-5 based propagation:
a. for domain size growing, b. for nodes number growing

As illustrated in figure 1a, the specialization reduces substantially execution time for monotonic, functional and anti-functional constraints.

1. We use the CHIP notation for finite domains constraints

In order to test the specialization advantage in relation with the growing number of nodes in the constraints graph, we constructed the following sample run. A set of 14 domain variables {X, Y, Z, T, U, V, W, A, B, C, D, E, F, G} is used with a fixed domain range of (1,50). An initial constraints graph with two variables X and Y and the following constraint: X + Y #> 23 is used. With each successive variable a set of constraints is added to the graph and the final constraints list is the following:

```
X + Y #> 23, Y + Z #> 12, X + Z #< 40, Y + T #< 60, V +
X #= 45, V + W #> 30, V + W #< 40, Z + T #= 65, A + B #=
50, B + C #> 30, E + X #< 20, F + E #> 30, F + Z #< 50,
A + G #= 10, H + W #= 50, I + T #> 30, K + C #> 60, K +
X #< 30.
```

For each variable (node) added, the execution time of the non specialized versus the specialized AC-5 propagation is noted in the figure 1b. After 4 nodes, the adding of new variables does not increasingly influence the growing of CPU time for the non specialized as well as for the specialized version. However, the overall level of CPU time is significantly lower for the last one.

3.3. Solver implementation

In order to get a complete solver, the AC-5 algorithm should be completed by delay mechanisms, ternary constraints propagation and also labeling process.

3.3.1. «Delay» declaration

With each delayed predicate is associated the list of arguments which have to be ground before execution. As the delayed objects should be woken up as soon as possible, execution attempts are tried during the values generation but also during the arc-consistence search.

3.3.2. Propagating ternary constraints

In order to propagate ternary arithmetic constraints, we introduce three virtual nodes in the constraints graph representing the possible arithmetic combinations of the three variables. Then, the associated virtual domains are calculated from the original nodes. Finally, the actual ternary constraint is approximated by three binary constraints linking effective and virtual nodes. Now the AC-5 is able to propagate these constraints over the hole system.

3.3.3. Variable instanciation

Generating a valid instanciation for finite domain variables is achieved by a searching run through the constraints graph with dynamic propagation by repeated application of the AC-5 algorithm. A specific running order on the nodes of the graph is required. For the non backtrackable variables, the searching procedure needs restoration of actual domains in the case of failure of AC-5 or in the case of searching for further solutions. Therefore, a `restore(Variable_List,Domain_List)` predicate restores the `Variable_List` on the blackboard with the corresponding `Domain_List`. A predicate `select(Nodes_Ordering)` searches for a solution through the constraints graph following the ordering of the graph nodes given by `Nodes_Ordering`.

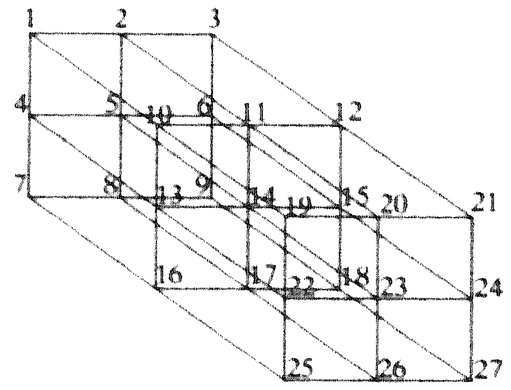
3.3.4. Experimental results

As the order of instanciation during the solution generating step is of crucial importance, we studied on a sample problem the incidence of different instanciation orders on execution time for exhibiting the three first solutions. We used the constraints graph with 27 nodes and 52 monotonic constraints shown below, where an arc between I and J means that I #< J.

First the subgraph with 9 nodes and 12 constraints is used, then the subgraph with the first 18 nodes and 32 constraints and finally the whole constraints graph is used. Test runs were made with

four different orderings of the nodes:

- from the less constrained to the most constrained variable,
- from the most constrained to the less constrained variable,
- in increasing # of the variables,
- in decreasing # of the variables.



Monotonic constraints graph

The associated domains, for each test, are chosen as (1, number of nodes). Domains are considered in increasing values. The initial AC-5 propagation times for the three tests are respectively 50, 150 and 400 milliseconds. CPU time for calculating the first three solutions is reported in the tables below:

Nodes ordering	Sol. 1	Sol. 2	Sol. 3
- contr.to +	117	133	67
+ contr. to -	83	33	0
decreasing #	133	150	50
increasing #	100	0	33

Nodes ordering	Sol. 1	Sol. 2	Sol. 3
- contr.to +	1050	1050	234
+ contr. to -	700	50	50
decreasing #	1000	1033	217
increasing #	434	33	83

Nodes ordering	Sol. 1	Sol. 2	Sol. 3
- contr.to +	3316	3234	616
+ contr. to -	2383	67	67
decreasing #	3233	3283	684
increasing #	1016	67	50

Figure 2: Comparing CPU time in milliseconds for different instanciation strategies

These empirical results show that the instanciation order of the variables during the solution generating step is of great importance. In our example, the increasing # order of instanciation is the most appropriated.

But on a monotonic constraints graph, there exists a specific better result. One can easily show that if on such a constraints graph, the AC-5 propagation is successful, then the lowest remaining values for the variables represent the smallest immediate valid instanciation. In fact these values represent the height of the node in the order structure defined by the monotonic constraints on the variables. Symmetrically, the highest remaining values for the variables after AC-5 propagation, if different from the first ones, represent an immediate second solution. These values represent the depth of the node (distance from the highest variable) in the order structure defined by the monotonic constraints. Moreover this result even allows to enumerate and immediately generate all completely different solutions that exist.

Conclusion

This paper presents the elaboration of a simple finite domain solver implemented in BinProlog. This system considers the domain variables and their related constraints as global objects stored on a blackboard. Based on the AC_5 arc-consistency algorithm, this system has been completed with delay declarations, labeling processes and also propagating of ternary constraints. Experimental results showed some interesting property of such a simplified solver. However, one missing point should have been added to get better performances. Indeed, the constraints graph is only changing its nodes but never its edges, even though some constraints can be simplified or eliminated. Finally, the use of blackboard variables for the implementation still remains to be discussed in order to conclude if this intuitive idea was really applicable.

References

[1] Dincbas, M., Van Henteryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, *F.*, "The constraint logic programming language CHIP", *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokio, 1988, 693-702.

[2] Lee, J. H. M., and van Emden, M. H., «Interval computation as deduction in CHIP», *J. Logic Programming*, New York, vol. 16, n°1 & 2, July/August, 1993.

[3] Mackworth, A. K., «The logic of constraint satisfaction», *Artificial intelligence* , vol. 58, 3-20, December, 1992.

[4] le Provost, T., and Wallace, M., «Generalized constraint propagation over the CLP scheme», *J. Logic Programming*, New York, vol. 16, n°1 & 2, July/August, 1993.

[5] Tarau, P., «BinProlog 2.20 User Guide, Université de Moncton, Canada, January 7, 1994.

[6] Tarau, P., De Bosschere, K., «Non-associative Blackboard Programming», *Proceedings of the ICLP'93 Post-Conference Workshop on Blackboard-Based Logic Programming*, Ed. De Bosschere, Jacquet, Tarau, Budapest, June 24, 1993, pp77-89.

On Backtracking in Finite Domain Problems

Henk Vandecasteele

Department of Computer Science
K. U. Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium.
henk.vandecasteele@cs.kuleuven.ac.be

Keywords: Constraint Logic Programming, Finite Domains, Backtracking.

1 Introduction

Solving a finite domain problem always consist of finding one or more assignments for a set of variables such that none of the constraints of the problem are violated. A shared constraint in a finite domain problem is that each variable has a domain. When an assignment has been achieved every value assigned to a variable must be a member of the domain. A straightforward technique then is, trying all the different assignments possible with the values in the domains and check the remaining constraints from the problem. If they hold we have found a solution. Of course for real problems this results in an huge search space, which cannot be searched in a lifetime! A usual optimisation of this technique is *pruning a priori* together with *backtracking*. Pruning a priori consist of removing values from the domain of a variable which are inconsistent with the values still left in the domain of the other variables. Examples of such techniques are forward checking, lookahead, partial lookahead [Hen89], k-consistency [Mac77] or the pruning techniques presented in [VDS94]. In most cases these techniques will not find a solution without trying to assign values to the variables. This is the place where backtracking comes in. The method consists of alternately assigning a variable and applying the pruning a priori rules. If a domain of a variable becomes empty during the checking of the constraints, the set of previous assignments cannot be extended to obtain a solution and we have to *backtrack*. The usual approach here is to assign another value to the same variable. Although in some applications assigning another variable is more useful e.g. the perfect square problem in [AB92]. In this paper we present a more general approach to this topic.

2 The three faces of enumeration.

Actually enumeration (assigning values to a variable) consists of three aspects. First we have to choose which variable we will assign. Next we choose which value (or values) we want to assign to this variable. And at last there are different alternatives backtrack behaviours: do we assign the same variable with another value, or do we take another variable. And how does the removal of the old value of the domain and the assignment of the new value interact with the pruning techniques used.

2.1 Choosing a variable.

Although pruning a priori usually reduces the search-space several orders of magnitude, still the remaining part of this search-space can be quite large. It has been noticed that certain choices in the order of assigning variables can reduce the search-space more quickly than others. Firstfail [Hen89] for example selects first the variables that are more constrained than the others. Wrong assignment of such a variable fails more quickly, and reduces the search-space this way. On the other hand the user may have a clue on which part of the search space has more chances to contain a solution than the rest of the search-space. For addressing that part of the search space a specific ordering of the variables may be required.
With these guidelines in mind for selecting a variable for assignment, there can be no fixed selection method which is best for any problem. Therefore a finite domain language must provide a general selection method parameterised with heuristics.

2.2 Selecting a value

After a variable has been selected for assignment a value must be chosen for assignment. Also the way of choosing this value can affect the efficiency of the search. Here the best heuristics are based on selecting a *good* branch in the search tree. Although similar techniques like firstfail exist here also. For example in permutation-problems one can reverse the problem of assigning a value to a variable to assigning a variable to each value. In this way selection-criteria for selecting a variable can be transferred to selection-criteria for selecting a value[Gee91].
Remark that not all methods select one single value for assigning the finite domain variable. It can be interesting to restrict the domain of the finite domain variables to a subset of the current domain. This variable must be further instantiated at a later stage of the enumeration process. Such a technique is used in interval arithmetic where intervals are split in two parts instead of assigning values [BO92].
Here the same principle holds as in the previous subsection, a strategy for selecting a value can be different for every finite domain problem and therefore needs to be a parameter of the solver.

2.3 The backtrack-strategy.

Suppose that after selecting a variable, and choosing a value (or a subset of the values left in the domain) the pruning-algorithm finds that some constraints are not satisfied. Standard backtracking selects another value in the domain of the selected variable assigns this value (or subset of values) to the variable. A first variant of this scheme is removing the value (or set of values) from the domain and apply the pruning strategy, with this restricted domain. If there is no inconsistency a new value can be selected for assignment. The idea behind this technique is that in some cases a lot of inconsistent values can be removed from the domains of other variables after removing the failing value from the domain we are working with. Work which otherwise would have been done all over again for each value in the domain assigned to the variable. A next change we can propose is not to assign the same variable, but

after removing the failing value from the domain, we can again apply the selection-criteria for selecting a variable. Because after removing the value from the domain the selection-criteria can indicate another variable for assignment! This results in four different backtrack-strategies:

standard This is the usual way of backtracking. After assignment of a variable fails another value is selected, if available, and the pruning algorithm is applied.

standard_ex Before selecting another value in the domain, the failing value is removed from the domain. After applying the pruning a priori, in case of success, a new value is selected for assignment.

non_standard First the failing value is removed from the domain of the corresponding variable. Then the selection-criterion for selecting the variable for assignment is applied again. For this variable a value for assignment is chosen.

non_standard_ex In this case an additional step to the strategy *non_standard* is added. After removing the value from the domain, and before selecting a new variable pruning a priori is applied.

3 A parameterised algorithm

In this section we give a parameterised algorithm which further explains the behaviour of the notions described above. The algorithm takes as input: a set of variables V = $\{\forall i \in 1..n : v_i\}$, for each variable v_i there is a corresponding domain $D_i = \{\forall j \in 1..m_i : a_{ij}\}$ and C is the set of constraints $\{\forall k \in 1..p : C_k\}$. A constraint is a function of $(D_1, D_2, ...D_n)$ to $\{true, false\}$. The algorithm is looking for the set of solutions X = $\{(x_1, x_2, ...x_n) | \forall i \in 1..n, x_i \in D_i, \forall C_k \in C, C_k(x_1, x_2, ...x_n) = true\}$. The algorithm is parameterised with: a heuristic specifying the strategy to select a variable in a list of variables (StratVar), a method for selecting a value (or a set of values) given the domain of a finite domain variable (StratVal) and a parameter specifying the backtrack behaviour (Mode).

The following functions are assumed:

- *selectVariable(SetOfVariables, StratVar, Domains).* This functions returns a finite domain variable from the set of variables using the given strategy for selecting the variable.
- *selectValues(i, Domains, StratVal).* Selects a value (or a set of values) from the Domain of variable i given the strategy for selecting values in a domain.
- *pruneDomains(i, Domains, C).* This functions prunes the domains of the other variables, with the constraints, given that the variable i has changed, and returns true if all domains are non-empty after the pruning, false otherwise.
- *solutionFound(Domains)* adds the solutions found to the set of solutions found so far.

```
procedure search(C, (D_1, D_2, ...D_n), StratVar, StratVal, Mode);
begin
    D^(0) ← (D_1, D_2, ..., D_n);
    V^(0) = {v_i|∀i ∈ 1..n};
    assign(0, C, StratVar, StratVal, Mode);
end;

procedure assign(Level, C, StratVar, StratVal, Mode);
var consistent: boolean;
var i: variable;
var S: set of value;
begin
    consistent ← true;
    i ← selectVariable(V^(Level), StratVar, D^(Level));
    while consistent do begin
        D^(Level+1) ← D^(Level)
        S ← selectValues(i, D^(Level+1), StratVal);
        D^(Level+1)[i] ← S;
        if singleton(S) then V^(Level+1) ← V^(Level) \ {i}
        else V^(Level+1) ← V^(Level);
        consistent ← pruneDomains(i, D^(Level+1), C);
        if consistent then begin
            if emptySet(V^(Level+1)) then solutionFound(D^(Level+1))
            else assign(Level+1, C, StratVar, StratVal, Mode);
            D^(Level)[i] ← D^(Level)[i] \ S;
            if emptySet(D^(Level)[i]) then consistent ← false;
        end;
        if consistent and (Mode=standard_ex or Mode=non_standard_ex)
        then consistent ← pruneDomains(i, D^(Level), C);
        if consistent and (Mode=non_standard or Mode = non_standard_ex)
        then i ← selectVariable(V^(Level), StratVar, D^(Level));
    end;
end;
```

4 Applications

4.1 The nurse scheduling problem

The nurse scheduling problem consist of allocating different nurses with different preferences and constraints to the different shifts (day and night) in a hospital. Algorithms, with classical backtrack-behaviour, for solving this problem almost always end up with a solution which is very good for certain nurses but some nurses get a very bad schedule. Although in general the solution might be optimal, such a solution is not satisfactory. Using the non_standard enumeration this behaviour can be changed by changing the variable enumerated on.
One of the reasons for a bad allocation for a nurse is that while enumerating on

one nurse values can be pruned from the domain of another nurse. Continuous enumeration on the first nurse could result in a very bad schedule for the last one. In our backtrack scheme it is possible to allocate a nurse as soon as the quality of her schedule deteriorates by changing the enumeration variable to the finite domain variable that represents that nurse.

4.2 The perfect square

In this example it is quite clear that the non_standard enumeration method is helpful. The problem consists of fitting a number of squares with known size into a big square with know size. The little squares fill up the large square completely. Given this information it is straightforward to first select a free corner and then try to fit the available squares into this corner. The non_standard backtrack technique gives the power to do this: After the assignment of one square into the corner failed, another square can be selected to fit the corner. This method was also used (in a more ad hoc way) in [AB92].

4.3 Task-scheduling in time

In some applications, different tasks have to scheduled one after the other on a time-axes, satisfying some resource constraints (bridge problem [Hen89], ship loading problem [AB92]) and minimising the total duration. In most cases, after each task there will be another task. Similar to the perfect square example, one wants to try the different candidates which fit immediately after the previous task on the time-axes instead of trying to fix a task on the time-axes leaving open space for other tasks. This again can be solved using the non_standard backtracking strategy which selects another task for enumerating after a task has failed to fit after the current task on the time-axes.

5 Further work.

Some further investigation is needed for validating the significance of these backtrack schemes. The method together with some special enumeration heuristics has already been used for solving an allocation problem concerning the assignment of master-thesis subjects to students. The solutions already seems quite promising, but still fails in some cases.

We even think of developing more controversial backtrack schemes which backtrack before all different alternatives have been tried. The search is then resumed at an earlier stage and the information gathered in the previous search is then used for guiding the new search. The main problems with such algorithms is completeness and avoiding redundant work.

References

[AB92] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *JFPL*, pages 51–66, 1992.

[BO92] Frédéric Benhamou and William Older. Applying Interval Arithmetic to Integer and Boolean Constraints. Technical report, Bell Northern Research, 1992.

[Gee91] P.A Geelen. Een duale methodologie voor binaire constraint satisfaction problemen. In Prof. Dr. J. Treur, editor, *Proceedings of NAIC'91*, pages 169–182, 1991.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT press, 1989.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[VDS94] Henk Vandecasteele and Danny De Schreye. Implementing a finite-domain CLP-language on top of prolog : a transformational approach. To appear in the proceedings of LPAR94, 1994.

Towards Resource Handling in Logic Programming: the *PPL* Framework and its Semantics

Jean-Marie Jacquet[1] and Luís Monteiro[2]

Abstract

The *PPL* framework is proposed as a simple extension to logic programming aiming at handling resources. It is argued that the separation between logical treatments and resource handling is desirable and, to that end, resources are proposed to be manipulated by means of pre- and post-conditions associated with usual Horn clauses. The expressiveness of the resulting framework is evidenced through the coding of several applications involving objects, databases, actions and changes. Operational and declarative semantics are presented as well. The operational semantics rests on a derivation relation stating how goals and conditions are evaluated. The declarative semantics extends the classical model and fixed-point theories to take into account the evaluation of pre- and post-conditions, and in particular the non-monotonic behavior of the world of resources they induce in general. As suggested, an effort has been made to keep the work close to the classical logic programming setting. In particular, the semantics are in the main streams of logic programming semantics. However, the *PPL* framework raises new problems for which fresh solutions are proposed.

1 Introduction

The execution of many programs can be depicted as the progressive access to and update of a world of resources. For instance, the execution of a simple imperative program can be schematized as the successive executions of assignments, each one being caracterized by the three following actions: access to a state associating a value to each variable, computation of a value for the right-hand side of the assignment, update of the state in order to associate the value just computed to the left-hand side variable of the assignment. As a second example, constraint logic languages (see e.g. [27]) have a so-called store of constraints as world of resources. It is progressively accessed by means of the *tell* primitive, which adds the told constraint to the store provided consistency is preserved, and by the *ask* primitive, which tests whether a given constraint is entailed by the current contents of the store. Primitives for non-monotonically modifying the store are proposed in addition in [7]. As a third example, in a new approach to parallelism in logic programming ([6, 4]), a blackboard is used as means of communication between concurrent logic processes and is accessed by means of Linda-like primitives putting, testing and removing both passive data structures and active processes from it. Finally, to conclude this non-exhaustive list of examples, the synchronous communication mechanism introduced in [5, 1, 15, 16] can also be viewed as another instance of this general resource-based scheme. There, processes are simultaneously reduced to others and thus act as resources for their synchronizing partners.

Although resources thus appear to be central, most of the above mentioned logic languages have handled them by means of ad hoc primitives inserted into logical formulae and consequently have separated the handling of resources and the derivation of logical formulae only at the conceptual level and not at the programming one. Examples of this trend include the tell and ask primitives of [27] and the blackboard primitives of [4] which may be inserted at any place inside clause bodies and queries. From a software engineering point of view, this is quite regrettable since one better separate different issues clearly, and, consequently here, separate resource handling and logical treatments clearly. This paper proposes an alternative approach achieving this goal: on the one hand, "logical atoms" only compose Horn clauses and queries, and, on the other hand, resources

[1]Department of Computer Science, University of Namur, 5000 Namur, Belgium. Supported by the Belgian National Fund for Scientific Research as a Senior Research Assistant.

[2]Departamento de Informática, Universidade Nova de Lisboa, 2825 Monte da Caparica, Portugal.

are tackled in pre- and post-conditions associated with each Horn clause. Restated in programming terms, program clauses now take the form

$$H \leftarrow G \quad < C : D > \tag{1}$$

where

i) H is an (usual) atom,
ii) G is a goal, formed of (usual) atoms combined with the operators " ; " and " || ", for sequential and parallel compositions, respectively,
iii) C and D are lists of (resource-oriented) atoms separated by the symbol " , ".

Such clauses are subsequently called *pp-clauses*. They thus consist essentially of Horn clauses decorated by a pair of conditions $< C : D >$. This slight extension induces the following modificiation to the usual SLD-derivation. The condition C lists atoms (possibly duplicated) that should be present in the considered world of resources at the time of the reduction of a considered goal to the clause. If so, the atoms are removed from the resources and the (induced instance) of the body of the clause is evaluated. Then, atoms of the postconditions are added as new resources. Conditions C and D thus act as pre- and post-conditions, respectively.

The actual framework, called *PPL*, is slightly more general in two ways. On the one hand, resources may be aggregated by making use of atoms defined by means of Horn clauses. Such clauses take the usual form with body goals formed from atoms separated by the symbol ",". In particular, they do not include pre- and post-conditions. They are denoted as follows to be distinguished from the pp-clauses:

$$H \hookleftarrow G$$

The usual SLD-resolution is then used to verify the pre- and post-conditions.

On the other hand, pre-conditions are allowed to be non-destructive in the sense that their evaluation does not remove the resources they involve. Such clauses are subsequently denoted as

$$H \leftarrow G \quad < C \mid D > \tag{2}$$

if the whole pre-condition is non-destructive or, as

$$H \leftarrow G \quad < C_1 \mid C_2 : D >$$

if the C_1 part of the pre-condition is non-destructive and its C_2 part is destructive. On the point of the notation, although the above form is general and will be used for our semantic study, we shall stick, in the examples, to the notation proposed in clause (1) when the C_1 part of the pre-condition is empty and its C_2 part is not, and to the notation proposed in clause (2) when C_2 is empty. Note that with this convention, the form (2) is used when both C_1 and C_2 are empty to emphasize that no change has been operated to the resources.

Although simple, the *PPL* framework is quite expressive. This is suggested in section 2 through the coding of several applications. However, the purpose of this paper is more to identify a framework, to study semantics for it, and, on the way, to argue its declarativeness, than to create a new language. Hence, we shall not pay attention to implementation issues in the following but rather will concentratre on a semantic study. An operational semantics is first presented on the basis of a derivation relation. Then, the classical model and fixed-point theories are extended to our context. Finally, classical results of soundness and completeness are established. As will be appreciated by the reader, both the framework and the semantics have been conceived as simple extensions of logic programming remaining in its classical mainstreams.

Related work can be compared both at the language and the semantic points of views.

At the language level, it is worth stressing again that the language *PPL* clearifies the resource-handling made in other concurrent logic languages. It also shares with languages like Shared Prolog ([6]) and Multi-Prolog ([4]) the advantages induced by a communication realized at a global level and not by means of the sharing of variables: synchronization and mutual exclusion are achieved

implicitly via the world of resources and do not require the coding of auxiliary merge processes and the use of commitment operators. Moreover, thanks to the decoupling of logical reductions and resource handling, one could think of combining programs developed and tested in isolation under the assumption that the suitable information will eventually be available.

The idea of pre- and post-conditions has already been exploited in Shared Prolog ([6]). The two main differences are that, on the one hand, pre- and post-conditions are introduced at the finer level of clauses instead of the larger level of theories and, on the other hand, that aggregates of resources can be manipulated as such by means of predicates defined by Horn clauses.

Resource handling is, of course, tackled by constraint languages. For simplicity, this paper has been kept in the classical lines of logic programming but a generalization to constraints is appealing and is planned for future work. However, it should be noted that such a generalization leads to higher-order constructs (as in [26]) since the state of the *PPL* computations to be generalized is composed of a substitution, describing the values computed for the variables, but also of a world of resources composed of atoms namely of first-order constructs. One of the interests of the PPL framework is that it precisely hides the higher-order constructions.

As already said, a further difference with constraint languages lies in the clear separation, both at the conceptual and programming levels, of resources handling and logical derivations. In particular, aggregating resources by means of resource-dedicated clauses is support in *PPL* but is not supported in such a clear way in [7, 27].

Another difference with [27] is the non-monotonic behavior of the world of resources. Compared with [7], which also proposes non-monotonic stores, the main difference with our work rests in the handling of unconsumed resources and in the communication of bindings. The primitive $update_x(\theta)$ is proposed there to transform a given (current) store σ into the new one $\exists_x(\sigma) \sqcup \theta$ where the $\exists_x$ operator makes x a local variable in σ, thereby removing *all* the constraints on it in σ. However, in general, only part of these constraints need to be removed. Using this technique, it is thus necessary to collect all the constraints — which does not seem to be a trivial task using the proposed primitives — and to copy those constraints that should remain. In contrast, such a selected deletion is performed easily in our framework by citing the constraints that need to be present and the constraints that need to be removed in two kinds of non-destructive and destructive preconditions. In fact, these preconditions embody basic operations on mutlisets implicitly. Moreover, while checking these constraints, some unification may take place and be propagated. Again, it is not obvious to achieve this effect by using the primitives proposed in [7].

It remains that the language $\mathcal{L}$ proposed in [7] is so general that the semantics proposed for it can be used to design semantics for the *PPL* framework. However, taking profit of its specificity, we have been able to design semantics that, although they borrow from imperative techniques, still constitute reasonable extensions to the classical logic programming framework.

Linear logic provides an alternative way of handling resources (see e.g. [1, 18, 26]). Although it is certainly promising, we have preferred to follow the lines of an extension of classical logic programming. This paper proves that this is possible and does not lead to intricacies. However, the connections with linear logic will be explored in future work.

The concept of resource handling is closely related to the notions ofhe concept of resource handling is closely related to the notions of action and change, which have recently been the subjects of many researches: see e.g. [20, 22, 21, 19, 13, 11, 14, 23, 10, 8, 25]. Our work differs from them as follows.

The language AbstrAct [25] specifies the activities of systems by action rules characterizing agents capable of performing actions that operate on a global shared data space. A feature of AbstrAct is that it distinguishes between actions to induce state transformations and deductions that can be performed in each state. Our work differs from this work in that we do not have an explicit notion of agent, relying rather on the more logic-based notion of goal, and use an aggregation technique of resources based on Horn clauses.

The works [20, 22] have introduced the so-called situation calculus. Essentially, it consists of using an argument to state that a particular resource is available in a particular situation and of using so-called frame axioms, one for each action and each resource, to solve the so-called frame problem, i.e. to express that a resource remains invariant after an action. As noted in [14], the

essential problem with this solution is that the number of frame axioms rapidly increases with the number of actions and resources. This number has been reduced in [19] and has become linear with respect to the number of actions. Our proposal does not suffer from these problems since the non-consumption of resources is expressed either by default or by means of non-destructive pre-conditions.

The article [21] has proposed to use nonmonotonic inference rules and a default law of inertia to tackle the frame problem. The paper [23] has employed linear logic to describe actions and changes. The paper [10] has used extended logic programs with both classical negation and negation-as-failure for that purpose. The paper [8] has used normal logic programs with abduction. Our work differs from all of them by using a slight extension of Horn clause programs involving no negation and remaining in the (classical) mainstreams of logic programming.

The work reported in [13, 11, 14] is the closest to ours. Rephrased in our terms, it proposes to describe each action by incorporating our pre- and post-conditions as predicate arguments and the non-consumption problem by extending normal unification to cope with multisets. As an example (taken from [14]), the clause

$$action(Pre, load, Post) \quad \leftarrow \quad Pre =_{AC1} unloaded \wedge Post =_{AC1} loaded \tag{3}$$

states that the action of loading a gun assumes that the gun is unloaded and, if so, it moves the gun in a loaded status. The effect of the sequences of actions is expressed by means of the ternary predicate *causes*, defined by the following two clauses

$$\begin{array}{l} causes(Pre, [], Post) \leftarrow Pre =_{AC1} Post \\ causes(Pre, [A|As], Post) \leftarrow action(Pre\_A, A, Post\_A) \wedge Pre\_A \; o \; Pre\_rem =_{AC1} Pre \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge causes(Post\_A \; o \; Pre\_rem, As, Post). \end{array}$$

The frame problem is solved by using a multiset structure, defined in a way similar to lists but with the binary function symbol o, and by using a new unification, denoted $=_{AC1}$ which actually defines the operator o as associative, commutative and admitting the constant $\emptyset$ as unit element.

Our proposal first contrasts by liberating the *action* predicate from the pre- and post-conditions arguments and by avoiding extended unification. As an example, clause (3) rewrites in our framework as

$$action(load) \leftarrow \triangle \quad < unloaded : loaded >$$

or, even, in a simpler way, as

$$load \leftarrow \triangle \quad < unloaded : loaded >$$

where $\triangle$ denotes the empty goal. Besides the syntactic difference, we believe that separating logical derivations from resource handling has interesting consequences from a software engineering point of view. For instance, since in our framework resources are only treated when need be by means of pre- and post-conditions and without explicit (extended) unification, the *causes* predicate can be rewritten more naturally as

$$\begin{array}{l} causes([]) \leftarrow \triangle \quad < \square \mid \square > \\ causes([A|As]) \leftarrow action(A) \,;\, causes(As) \quad < \square \mid \square > \end{array}$$

where $\square$ denotes the empty (pre- and post-) condition. Another difference with our work is that we care here for the execution of the "$\wedge$" connector and provide semantics for both its sequential and parallel versions. As noted in [8], finding a way to represent parallel and non-deterministic actions is a real challenge, to which we believe to have given a solution.

Finally, this paper grew up as a continuation of our previous work [15, 16] and its related work [9, 5]. As already suggested, one difference with this work is the clear separation between resource handling and logical treatment. As an interesting consequence, resource need not be considered as active data when they are not actually. Compare for instance the stack example of section 2 and that of [15]. Moreover, the semantics are quite different from that presented in previously

cited work. On the one hand, with respect to [9] and [5], one should note that, in addition to the richer context of pre- and post-conditions handled here, arbitrary mixing of sequential and parallel compositions inside goals are treated here as well as an unrestricted form of variable sharing. On the other hand, pre- and post-conditions handling and the discard for synchronous communication makes the semantics reported in this paper quite different from that of [15] and of [16].

The remainder of this paper is organized as follows. Section 2 suggests the interest of pp-clauses through the coding of examples of actions, database manipulations, and programs integrating the object and logic programming styles. Section 3 describes the basic constructs of the language and explains our terminology. Section 4 defines auxiliary concepts. Section 5 presents the operational semantics O. Section 6 discusses the declarative semantics Decl_m and Decl_f, based on model and fixed point theories, respectively. Section 7 establishes the soundness and completeness properties and consequently connects the three semantics. Finally, section 8 gives our conclusions.

2 Examples

2.1 Objects

The behavior of objects is classically represented in logic programming by the evaluation of a call to a procedure defined recursively, the successive values of the arguments representing the successive states of the object. Following this line, the treatment of a message `mess(M)` by an object `obj(S)` by means of a method `method(M)` can be schematized by one of the following pp-clauses depending upon whether the message is consumed or not:

$$obj(S) \leftarrow method(M, S, NewS)\,;\, obj(NewS) \quad < mess(M) : \Box >$$
$$obj(S) \leftarrow method(M, S, NewS)\,;\, obj(NewS) \quad < mess(M) \mid \Box >$$

Indeed, in the *PPL* framework, any call to $obj(S)$ results successively

i) in the evaluation of the precondition $mess(M)$, which identifies the message from the world of resources,
ii) in the reduction of the call $method(M, S, NewS)$, which computes the new state $NewS$, and in the reduction of the recursive call to $obj(NewS)$, which describes the object with its new state
iii) in the evaluation of the empty post-condition $\Box$.

Note that, depending upon whether the message has to be consumed or not, the message $mess(M)$ is consumed or not from the world of resources by the evaluation of the pre-condition.

An instance of this scheme is given by the following description of the class of stacks:

$$stack(Id, S) \leftarrow stack(Id, [X|S]) \quad < push(Id, X) : \Box >$$
$$stack(Id, [X|S]) \leftarrow stack(Id, S) \quad < pop(Id) : \Box >$$

Stacks are identified there by the Id argument of the *stack* predicate and their state, implemented as a list, moves, respectively from S, $[X|S]$ to $[X|S]$, S according to the treatment of a *push* or *pop* message.

The classical airline reservation system provides another interesting instance of the above scheme. The task consists here of simulating an airline reservation system composed of n agencies communicating with a global database about m flights. It is achieved by evaluating the *PPL* query

$$agency(Id_1) \parallel \cdots \parallel agency(Id_n) \parallel air\_syst(DB\_init)$$

where $agency(Id_j)$ represents the j^{th} agency, identified by Id_j, and where DB_init represents the initial information about the m flights. The complete description of the agencies is out of interest for our illustrative purposes. It is here sufficient to assume that some internal actions successively generate queries for the database and behave correctly according to the answers. We

will just consider the message $reserve(Ag\_id, Flight\_id, Nb\_seats)$, other ones being treated in a similar way. This message consists of asking the reservation of Nb_seats in the flight $Flight\_id$ for the agency identified by Ag_id. A message $reserve\_ans(Ag\_id, Ans)$ reporting the issue of the reservation is expected as a result. According to the above scheme and using the auxiliary predicates $remaining\_seats$ and $update$, with obvious meanings, the treatment of the message can be coded as follows (as easily checked by the reader):

$$
\begin{array}{l}
air\_syst(DB) \leftarrow \\
\qquad make\_reservation(Ag\_id, Flight\_id, Nb\_seats, DB, New\_DB)\ ; \\
\qquad air\_syst(New\_DB) \\
\qquad\qquad < reserve(Ag\_id, Flight\_id, Nb\_seats) : \Box > \\
\\
make\_reservation(Ag\_id, Flight\_id, Requested\_seats, DB, New\_DB) \leftarrow \\
\qquad remaining\_seats(Flight\_id, Seats)\ ; \\
\qquad Seats \geq Requested\_seats\ ; \\
\qquad update(DB, Requested\_seats, New\_DB) \\
\qquad\qquad < \Box \mid reserve\_ans(Ag\_id, successful\_reservation) > \\
make\_reservation(Ag\_id, Flight\_id, Requested\_seats, DB, DB) \leftarrow \\
\qquad remaining\_seats(Flight\_id, Seats)\ ; \\
\qquad Seats < Requested\_seats \\
\qquad\qquad < \Box \mid reserve\_ans(Ag\_id, failed\_reservation) >
\end{array}
$$

The following points are worth noting from this example. First, accessing the database is achieved without handling lists of messages explicitly and without using merge processes, as usual in concurrent logic programming languages. Second, assuming the atomicity of the evaluation of the pre- and post-conditions, mutual access to the database is ensured without using commitment, as usual in concurrent logic programming as well.

2.2 Databases

The airline reservation system has already given an example of database handling. There, the database has been manipulated as an argument of the air_syst predicate. However, databases can also be tackled as worlds of resources. As an illustration, let us consider the education database of [11]. Three relations are specified about students, courses, and grades:

i) the relation $enr(St, C)$ states that the student St is enrolled in course C.
ii) the relation $grd(St, C, G)$ states that the grade of the student St for course C is G
iii) the relation $pre(L, C)$ states that the courses of the list L are prerequisite courses for the course C.

The aim of the program is to code the action of registering a student St for a course C given the constraint that a grade of at least 50 should have been obtained by the student for all prerequisite courses of C. This is achieved by the following pp-clause:

$$regis(St, C) \leftarrow satisfiable(L) \quad < list\_grd\_prec(St, C, L) \mid enr(St, C) >$$

The $regis(St, C)$ performs the registration transaction. Its definition uses the precondition $list\_grd\_prec(St, C, L)$ which builds the list L of grades corresponding to the prerequisite courses for C. This list L is then tested for the fulfillment of the above constraint rule by the $satisfiable(L)$ goal. Finally, in case this rule holds, the item $enr(St, C)$ is added to the database.

It is here worth noting that resource handling and logical treatment are well separated. Indeed, on the one hand, information retrieving and update on the database, is isolated in the pre- and post-conditions of the rule. This reflects the status of the database as a resource. On the other hand, the logical treatment on these data (i.e. $satisfiable(L)$) is performed in the body of the clause.

From a programming point of view, the reader should also note the ease of programming given, on the one hand, by distinguishing destructive and non-destructive preconditions, and, on the other hand, by allowing predicates to be defined on primitive resource-atoms.

The auxiliary predicate $satisfiable(L)$ is defined by the following self-explaining clauses.

$$satisfiable([]) \leftarrow \triangle \quad <\Box \mid \Box>$$
$$satisfiable([G|Gs]) \leftarrow G \geq 50 \parallel satisfiable(Gs) \quad <\Box \mid \Box>$$

The auxiliary predicate $list\_grd\_prec(St, C, L)$ is defined by first identifying the list of prerequisite courses for the course C (reduction of $pre(Lc, L)$) and then by collecting all the grades got by the student St for all these courses (reduction of $list\_grd(St, Lc, L)$).

$$list\_grd\_prec(St, C, L) \hookleftarrow pre(Lc, C), list\_grd(St, Lc, L)$$

$$list\_grd(St, [], []) \hookleftarrow \Box$$
$$list\_grd(St, [C|Cs], [G|Gs]) \hookleftarrow grd(St, C, G), list\_grd(St, Cs, Gs)$$

2.3 Actions and change

Actions and change can also be coded in the PPL framework. As a support to this claim, we now show how to code the classical Yale Shooting problem of [12]. It has as actors a gun, which is either unloaded or loaded, and a turkey, which is either alive or dead. Three actions are possible:

i) loading, which results in moving the status of the gun to loaded both if the gun was unloaded or loaded
ii) shooting, which results in unloading the gun and in killing the turkey if the gun was loaded
iii) waiting, which is a passive action.

The problem can be coded in PPL by representing the states of the turkey and of the gun by the following tokens and by taking them as resources:

i) loaded: the gun is loaded
ii) unloaded: the gun is unloaded
iii) alive: the turkey is alive
iv) dead: the turkey is dead

The loading, shooting and waiting actions, can then be described by the following pp-clauses:

$$wait \leftarrow \triangle \quad <\Box \mid \Box>$$
$$load \leftarrow \triangle \quad <unloaded : loaded>$$
$$load \leftarrow \triangle \quad <loaded \mid \Box>$$
$$shoot \leftarrow \triangle \quad <unloaded \mid \Box>$$
$$shoot \leftarrow \triangle \quad <loaded, alive : unloaded, dead>$$
$$shoot \leftarrow \triangle \quad <loaded, dead \mid \Box>$$

Other classical problems such as the fragile object problem ([28]), the murder mystery problem ([2]), and the stolen car problem ([17]) can be coded in a similar way.

3 The language

We now turn to the formal definition of the language *PPL*. As usual in logic programming, it comprises denumerably infinite sets of *variables*, *functions* and *predicates*, subsequently referred to as $Svar$, $Sfunct$ and $Spred$, respectively. They are assumed to be pairwise disjoint. The notions of term, atom, substitution, ... are defined therefrom as usual. Their sets are subsequently referred to as $Sterm$, $Satom$, $Ssubst$, ..., respectively. In contrast, two kinds of goals are subsequently distinguished. On the one hand, so-called pp-goals are formed from the atoms by combining them with the operators " ; " and " || ", for sequential and parallel compositions, respectively. They are

typically denoted by the letter G. Their set is subsequent referred to by *Sppgoal*. On the other hand, so-called l-goals consist of the usual lists of atoms, separated by the symbol " , ". They are typically denoted by the letters C and D, respectively. Their set is subsequently referred to as *Slgoal*.

Clauses are extended by pre- and post-conditions and take the form:

$$H \leftarrow G \quad < C_1 \,|\, C_2 : D >$$

where H is an atom, G is a pp-goal, C_1, C_2, and D are l-goals. They are called *pp-clauses*. Our semantic study requires that resources are ground only. To fulfill this property, we shall subsequently restrict pp-clauses in such a way that all the variables of the postcondition D occur in the preconditions C_1 and C_2. Note that, although restrictive at first sight, this constraint is satisfied by the programs of section 2. Such clauses are subsequently called pp-clauses.

Besides pp-clauses, usual Horn clauses are also considered. They take the form

$$H \hookleftarrow B$$

with H an atom and B a l-goal. They are called *r-clauses*. Again for our semantic purposes, we shall assume subsequently that r-clauses defining atoms appearing in postconditions obey the following property: all their variables appear both in their heads and in their bodies.

Programs are then composed of two parts: a set of pp-clauses and a set of r-clauses. They are typically denoted as P or as (P_p, P_r) with P_p the set of pp-clauses and P_r the set of r-clauses. Their set is subsequently denoted by *Sprog*.

4 Auxiliary concepts

It turns out that it is possible to treat the sequential and parallel composition operators in a very similar way by introducing the auxiliary notion of context. Basically, a context consists of a partially ordered structure where the place holder $\diamond$ has been inserted at some top-level places i.e. places not constrained by the previous execution of other atoms. Viewing pp-goals as partially ordered structures too, the atoms to be reduced are those that can be substituted by a place holder $\diamond$ in a context. Furthermore, the pp-goals resulting from the reductions are basically obtained by substituting the place holder by the corresponding clause bodies. The evaluation of the postconditions of the clauses under consideration need simply to be processed in addition. This is achieved by decorating the clause bodies with constructs of the form $\uparrow D$, where D stands for a postcondition. The intention is that the postcondition is evaluated as soon as its decorating goal is reduced to the empty goal. As it is necessary to decorate already decorated goals in such a way, we are naturaly lead to the following notion of gpp-goals.

Definition 1 *The gpp-goals are defined inductively by the following rules:*

i) any pp-goal is a gpp-goal
ii) if G is a gpp-goal and if D is a l-goal then the construct $G \uparrow D$ is a gpp-goal.

The set of gpp-goals is subsequently referred to as Sgppgoal. ■

The precise definition of the contexts is as follows.

Definition 2 *The* contexts *are the functions from Sgppgoal to Sgppgoal inductively defined by the following rules.*

i) $\diamond$ is a context that maps any gpp-goal to itself. For any gpp-goal G, this application is subsequently referred to as $\diamond[\![G]\!]$.
ii) If c is a context and if G is a gpp-goal, then $(c\,;G)$, $(c \parallel G)$, $(G \parallel c)$ are contexts. Their applications are defined as follows: for any gpp-goal G',

$$\begin{aligned} (c\,;G)[\![G']\!] &= (c[\![G']\!]\,;G) \\ (c \parallel G)[\![G']\!] &= (c[\![G']\!] \parallel G) \\ (G \parallel c)[\![G']\!] &= G \parallel (c[\![G']\!]) \end{aligned}$$

iii) If c is a context and if D is a l-goal, then $(c \uparrow D)$ is a context. Its application is defined as follows : for any gpp-goal G',

$$(c \uparrow D)[\![G']\!] = (c[\![G']\!] \uparrow D).$$

In the above rules, we further state that the structure (Sgppgoal, "; ", "$\|$ ", "$\triangle$") is a bimonoid. Moreover, in the following, we will simplify the gpp-goals resulting from the application of contexts accordingly. ■

5 Operational Semantics

The operational semantics O of the language PPL is defined in terms of derivation relations, themselves defined by means of rules of the form

$$\frac{Assumptions}{Conclusion} \quad if \quad Conditions,$$

asserting the *Conclusion* whenever the *Assumptions* and *Conditions* hold. Note that *Assumptions* and *Conditions* may be absent from some rules, in which case they are considered to be trivially satisfied.

Two auxiliary derivation relations need first to be defined. They formalize the evaluation of pre- and post-conditions. They are attached the following meaning. The relation $rr \Downarrow\!\vdash C\ [rr^*]\ [\theta]$ expresses the property that, assuming a given program and given a world of resources rr, the precondition C has a successful derivation producing the substitution θ and using the resources of rr^*. Similarly, the relation $\Uparrow\!\vdash D\ [rr^*]\ [\theta]$ expresses the property that, assuming a given program, the post-condition D has a successful derivation producing the substitution θ and the resources of rr^*.

The definitions are as follows. As usual, the above notations are used instead of the relational ones with the aim of clarity. Moreover, the notation $\mathcal{M}(E)$ is used to denote the set of multisets with elements from E. Usual set operations are also extended to multisets.

Definition 3 (Precondition) *Define $\Downarrow\!\vdash$ as the smallest relation of $\mathcal{M}(Satom) \times Slgoal \times \mathcal{M}(Satom) \times Ssubst$ that satisfies the following rules (C_1) to (C_3).*

$$(C_1) \qquad \frac{}{rr \Downarrow\!\vdash \Box\ [\emptyset]\ [\epsilon]}$$

$$(C_2) \qquad \frac{rr \Downarrow\!\vdash G\theta\ [rr^*]\ [\sigma]}{rr \Downarrow\!\vdash c, G\ [rr^* \cup \{c\theta\}]\ [\theta\sigma]}$$

$$if \left\{ \begin{array}{l} c\theta \in rr \\ c \text{ not defined in } P_r \end{array} \right\}$$

$$(C_3) \qquad \frac{rr \Downarrow\!\vdash (B, G)\theta\ [rr^*]\ [\sigma]}{rr \Downarrow\!\vdash c, G\ [rr^*]\ [\theta\sigma]}$$

$$if \left\{ \begin{array}{l} (H \hookleftarrow B) \in P_r \\ H \text{ and } c \text{ unify with mgu } \theta \end{array} \right\}$$

■

Definition 4 (Postcondition) *Define $\Uparrow\!\vdash$ as the smallest relation of $Slgoal \times \mathcal{M}(Satom) \times Ssubst$ that satisfies the following rules (D_1) to (D_3).*

(D_1) $$\frac{}{\uparrow\!\vdash\ \Box\ [\emptyset]\ [\epsilon]}$$

(D_2) $$\frac{\uparrow\!\vdash\ G\ [rr^*]\ [\sigma]}{\uparrow\!\vdash\ d, G\ [rr^* \cup \{d\}]\ [\sigma]}$$

$$\textit{if}\ \{\ \ d \textit{ is not defined in } P_r\ \}$$

(D_3) $$\frac{\uparrow\!\vdash\ (B, G)\theta\ [rr^*]\ [\sigma]}{\uparrow\!\vdash\ d, G\ [rr^*]\ [\theta\sigma]}$$

$$\textit{if}\ \left\{\begin{array}{l}(H \hookleftarrow B) \in P_r \\ H \textit{ and } d \textit{ unify with mgu } \theta\end{array}\right\}$$

■

A word on the above rules is in order. Rules (C_1) and (D_1) tackle the empty pre- and post-conditions. For any world of resources, they state that the empty pre- and post-conditions are derivable with the empty substitution ϵ as computed answer substitution and with no resources used or to be added. The remaining rules tackle the reduction of a resource-atom. If it is defined by a clause of P_r, i.e. if there is a clause of P_r whose head has the same predicate name and arity, then rules (C_3) and (D_3) state that a unifiable clause should be employed and that the induced instance should be reduced. Otherwise, on the one hand, with respect to the pre-conditions, an instance of the resource-atom should be in the world of resources and, if so, should be considered as used (rule (C_2)). On the other hand, as far as the post-conditions are concerned, the resource-atom is to be added to the world of resources (rule (D_2)).

The following properties illustrate the usefulness of the restrictions on pp-clauses and r-clauses.

Proposition 5

1) For any l-goal C and any multiset rr composed only of ground atoms, if the relation rr $\downarrow\!\vdash$ C $[rr^*]$ $[\theta]$ holds, then rr^* is composed only of ground atoms and $C\theta$ is ground. In particular, for any pp-clause $H \leftarrow B \quad < C_1 \,|\, C_2 : D >$, if the above relation holds both for C_1 and C_2 and for such a multiset rr, then $D\theta$ is ground as well.

2) For any ground l-goal D, if the relation $\uparrow\!\vdash$ D $[rr]$ $[\theta]$ holds, then rr is composed only of ground atoms.

■

We are now in a position to define the main derivation relation. It is written as $rr \vdash G\ [\theta]$ and expresses the property that, assuming a program and given the world of resources rr, the gpp-goal G has a successful derivation with θ as computed answer substitution. It is defined as follows.

Definition 6 *Define $\vdash$ as the smallest relation of $\mathcal{M}(Satom) \times Sgppgoal \times Ssubst$ that satisfies the following rules (E_1), (E_2), and (A).*

(E_1) $$\frac{}{rr\ \vdash\ \triangle\ [\epsilon]}$$

(E_2) $$\frac{rr \cup rr^*\ \vdash\ (c[\![\triangle]\!])\theta\ [\sigma]}{rr\ \vdash\ c[\![\triangle \uparrow D]\!]\ [\theta\sigma]}$$

$$\textit{if}\ \{\ \uparrow\!\vdash\ D\ [rr^*]\ [\theta]\ \}$$

$$(A) \qquad \frac{rr \backslash rr_2 \ \vdash \ (c[\![B \uparrow D]\!])\theta\mu\nu \ \ [\sigma]}{rr \ \vdash \ c[\![A]\!] \ \ [\theta\mu\nu\sigma]}$$

$$if \left\{ \begin{array}{l} (H \leftarrow B \ \ < C_1 \mid C_2 : D >) \in P_p \\ H \ unifies \ with \ A \ with \ mgu \ \theta \\ rr \ \Downarrow\vdash \ C_1\theta \ \ [rr_1] \, [\mu] \\ rr \ \Downarrow\vdash \ C_2\theta\mu \ \ [rr_2] \, [\nu] \\ (rr_1 \cup rr_2) \subseteq rr \end{array} \right\}$$

■

The above rules essentially rephrase the intuitive explanation already given. Rule (E_1) states that the empty goal is derivable for any world of resources with the empty substitution as answer substitution. Rule (E_2) expresses the reduction of a postcondition. Finally, rule (A) explains the reduction of an atom: it consists of finding a unifiable clause, of evaluating the (induced instance of the) pre-conditions, of reducing the (induced instances of the) clause body and finally the post-condition.

The conciseness and expressiveness of the contexts are worth stressing. Thanks to them, it is not necessary to specify rules for the sequential and parallel composition of atoms inside gpp-goals. Moreover, pp-goals as well as their extensions incorporating postconditions, the gpp-goals, are selected for reduction in a uniform manner.

An operational semantics can be derived from the derivation relation, as follows.

Definition 7 *Define the operational semantics* $O : Sprog \rightarrow Sppgoal \rightarrow Ssubst$ *as the following function: for any* $P \in Sprog$, *any* $G \in Sppgoal$, $O(P)(G) = \{\theta : \emptyset \ \vdash \ G \ \ [\theta]\}$. ■

6 Declarative semantics

One of the features of a logic programming language is that its semantics can be understood in two complementary ways, inherited from logic. On the one hand, the operational semantics, based on proof theory, describes the method of executing programs. On the other hand, the declarative semantics, based on model theory, explains the meaning of programs in terms of the set of their logical consequences. In our opinion, any claim that a given language is a logic programming language must be substantiated by providing suitable logic-based semantic characterizations. The operational semantics of the language *PPL* has been described in the previous section. We now turn to the discussion of the declarative semantics.

6.1 Model theory

Our first task is to find an appropriate notion of interpretation for *PPL*. Classically, an interpretation of Horn clause programs consists of a subset of the Herbrand base, intended to record all the facts that are considered to be true under the interpretation. In *PPL* which facts are true actually depend on the considered world of resources. Moreover, because of the pre- and post-conditions, this world may have a non-monotonic behavior and, because of the parallel composition inside pp-goals it may be influenced by concurrent evaluations. Hence, following previous work [24, 3], an interpretation should define the truth of an atom not in absolute terms nor with respect to a given world of resources but with respect to traces reporting the successive states of this world. We are thus naturally lead to the following definitions.

Convention 8 *For any set* S, *the notation* $ground(S)$ *is used to denote the set of ground instances of elements of* S. ■

Definition 9

1) *The set* Strace *is defined as the set of finite sequences of pairs of the form* (rr, rr^*) *with* rr, rr^* *multisets of ground atoms. Such sequences are subsequently called* traces *and are typically denoted as* $RR_1.RR_2.\cdots.RR_n$, *with* $RR_1, RR_2, \ldots, RR_N \in \mathcal{M}(ground(Satom)) \times$

$\mathcal{M}(ground(Satom))$. *They are said to start in* rr *if* rr *is the initial multiset* RR_1 *of their first pair. The empty trace is denoted by* λ.

2) *A trace* $t = (rr_1, rr_2).(rr_3, rr_4).\cdots.(rr_n, rr_{n+1})$ *is* continuous *iff the output resources of every pair is the input resources of the following pair in* t*, if any, i.e. iff the following property holds: for any even* $i \in 1, \ldots, n-1$, $rr_i = rr_{i+1}$. *The set of continuous traces is subsequently denoted as* Sctrace.

3) *The Herbrand base* B_H *is defined as the set* $Strace \times ground(Satom)$. *An Herbrand interpretation is defined as any subset of* B_H .

■

Our next task is to define the notion of truth. To that end, we first need an auxiliary notion capturing the evaluation of pre- and post-conditions. Note that, as specified in definitions 3 and 4, they essentially consist of using clauses of P_r until non-defined atoms are reached. This is formalized in the following relation $c \lhd C$, with c a ground lgoal and C the set of non-defined ground atoms reached by the reduction.

Definition 10 *For any program* (P_p, P_r)*, we define* $\lhd$ *as the smallest relation of* $ground(Slgoal) \times \mathcal{P}(ground(Satom))$ *that satisfies the following properties: for any* $c, c_1, \ldots, c_n \in ground(Satom)$*, any* $\bar{c} \in ground(Slgoal)$*, any* $C, C_1, \ldots, C_n \in \mathcal{M}(ground(Satom))$*,*

i) $c \lhd \{c\}$ *if* c *is not defined in* P_r
ii) $c \lhd C$ *if there is a ground instance* $c \hookleftarrow \bar{c}$ *of a clause of* P_r *such that* $\bar{c} \lhd C$
iii) $(c_1, \ldots, c_n) \lhd \cup_{i=1}^{n} C_i$ *if* $c_i \lhd C_i$ *, for every* $i = 1, \ldots, n$.

■

The satisfaction relation is defined at the level of ground constructs as follows.

Definition 11 *Given two traces* t_1 *and* t_2*, let* $t_1 \ominus t_2$ *and* $t_1 \otimes t_2$ *denote the concatenation and merge of the traces. Then* $\models$ *is defined as the samllest relation of* $Strace \times \mathcal{P}(B_H) \times (ground(Sggoal) \cup ground(Sppclause))$ *that verifies the following properties: for any interpretation* I*, any trace* t*, any ground atom* a, h*, any ground pp-goals* b, g_1, g_2*, any ground l-goals* c_1, c_2, d*,*

i) ground atom: $t \models_I a$ *iff* $(t, a) \in I$
ii) ground empty pp-goal: $\lambda \models_I \triangle$
iii) ground sequential pp-goal: $t \models_I g_1 ; g_2$ *iff there are* $t_1, t_2 \in Strace$ *such that* $t_1 \models_I g_1$, $t_2 \models_I g_2$ *, and* $t = t_1 \ominus t_2$
iv) ground parallel pp-goal: $t \models_I g_1 \parallel g_2$ *iff there are* $t_1, t_2 \in Strace$ *such that* $t_1 \models_I g_1$, $t_2 \models_I g_2$ *, and* $t \in t_1 \otimes t_2$
v) ground pp-clause: $t \models_I h \leftarrow b \ \ < c_1 \,|\, c_2 : d >$ *iff whenever there are* $rr_1, rr_2, rr_3, rr_4, C_1, C_2, D \in \mathcal{M}(ground(Satom))$*,* $u \in Strace$ *such that the following properties hold:*

1. $t = (rr_1, rr_2) \ominus u \ominus (rr_3, rr_4)$,
2. $c_1 \lhd C_1$
3. $c_2 \lhd C_2$
4. $(C_1 \cup C_2) \subseteq rr_1$
5. $rr_2 = rr_1 \backslash C_2$
6. $u \models_I b$
7. $d \lhd D$
8. $rr_4 = rr_3 \cup D$

then $t \models_I h$ *holds as well*

■

The notion of model, central for the declarative semantics, can now be defined.

Definition 12 *An interpretation I is a* model *of a* PPL *program P iff, for any trace t and any ground instance $h \leftarrow b \;\; <c_1 \,|\, c_2 : d>$ of a clause of P, the property $t \models_I h \leftarrow b \;\; <c_1 \,|\, c_2 : d>$ holds.* ■

It is easy to verify that the interesection of a family of models of any program is again a model of the program. Moreover, the Herbrand base B_H is also a model of any program. It follows that any program as a least model, which is identical to the intersection of all its models.

Proposition 13 *For any program P, the intersection $\cap_{I \in \mathcal{I}} I$ of all its models is again a model. It is the least model M_P of P.* ■

The notion of model is defined in a slightly different way for pp-goals than for programs. Although surprising at first sight, this difference is justified by the quite different nature of the two objects. On the one hand, clauses essentially consist of implications and any implication is considered to hold iff its consequent part holds whenever its antecedent part does. This is reflected in rule 5 of definition 11 and in the universal quantification over traces of definition 12. On the other hand, calling the operational semantics as support for our intuition, pp-goals should not manifestly hold for any trace but for particular ones. Moreover, these ones should be continuous and should start with the empty world of resources. This distinction of quantification is in the essence of the following definition.

Definition 14 *An interpretation I is a model of the pp-goal G iff there is a continuous trace t starting in $\emptyset$ and such that for any ground instance G_0 of G, the relation $t \models_I G_0$ is satisfied. This is subsequently denoted by $\models_I G$.* ■

We are now in a position to define the notion of logical consequence.

Definition 15 *The pp-goal G is a* logical consequence *of the program P iff any model of P is also a model of G. This is subsequently denoted by $P \models G$.* ■

The next notion to introduce in the classical study of logic program is that of success set. For a given program and a given pp-goal, it can be defined as the set of substitutions that make the pp-goal a logical consequence of the program. However, this involves checking all the models of P. Fortunately, just the least model need actually to be checked. This reduction is an easy consequence of the property that if I and J are interpretations such that $I \subseteq J$ and if $\models_I G$ holds then so does $\models_J G$.

We can then define the declarative semantics as follows.

Definition 16 *Define the model declarative semantics $Decl_m : Sprog \rightarrow Sppgoal \rightarrow Ssubst$ as the following function: for any $P \in Sprog$, any $G \in Sppgoal$, $Decl_m(P)(G) = \{\theta : \; P \models G\theta \} = \{\theta : \; \models_{M_P} G\theta \}$.* ■

6.2 Fixed-point theory

It turns out that the least model M_P can also be characterized as the least fixed point of a continuous transformation $T_P : \mathcal{P}(B_H) \rightarrow \mathcal{P}(B_H)$ called, as usual, the immediate consequence operator.

Definition 17 *Given a program $P = (P_p, P_r)$, the immediate consequence operator $T_P : \mathcal{P}(B_H) \rightarrow \mathcal{P}(B_H)$ is defined as the following function: for any interpretation $I \subseteq B_H$,*

$$
\begin{array}{rcll}
T_P(I) & = & \{(t,a) \in B_H : & a \leftarrow b \;\; <c_1 \,|\, c_2 : d> \in ground(P_p), \\
 & & & t = (rr_1, rr_2) \ominus u \oplus (rr_3, rr_4), \\
 & & & c_1 \triangleleft C_1 \,,\; c_2 \triangleleft C_2 \,,\; u \models_I b \,,\; d \triangleleft D \,, \\
 & & & (C_1 \cup C_2) \subseteq rr_1, rr_2 = rr_1 \backslash C_2, rr_4 = rr_3 \cup D \\
 & & & rr_1, rr_2, rr_3, rr_4, C_1, C_2 \in \mathcal{M}(ground(Satom)) u \in Strace \quad \}
\end{array}
$$

■

It is easy to check that the mapping T_P is well-defined, that is that for any interpretation I it returns an interpretation. Moreover, by endowing B_H by the set inclusion as order, B_H is turned into a complete lattice with the empty set and B_H as bottom and top elements, respectively. The mapping T_P can then be shown to be continuous and it can be proved that an interpretation I is a model of the program P iff it is a prefixed-point of T_P i.e. is such that $T_P(I) \leq I$. It follows from Tarki's lemma that T_P has a least fixed point which therefore is also the least model of P. Furthermore, this model can be computed by the standard iterative procedure. All these observations constitute the contents of the next proposition.

Proposition 18 *For any program P, the mapping T_P has a least fixed point, noted $lfp(T_P)$, which verifies the following equalities: $lfp(T_P) = T_P \uparrow \omega = M_P$* ■

The fixed-point semantics $Decl_f$, defined in terms of the least fixed point $lfp(T_P)$ can then be defined as follows.

Definition 19 *Define the fixed-point declarative semantics $Decl_f : Sprog \rightarrow Sppgoal \rightarrow Ssubst$ as the following function: for any $P \in Sprog$, any $G \in Sppgoal$, $Decl_f(P)(G) = \{\theta : \models_{lfp(T_P)} G\theta\}$.* ■

The equivalence between the declarative semantics $Decl_m$ and $Decl_f$ follows directly from proposition 18.

Proposition 20 *The semantics $Decl_m$ and $Decl_f$ are identical.* ■

7 Soundness and completeness properties

Soundness and completeness relating the operational and declarative semantics can also be proved. They are claimed in the following proposition.

Proposition 21 *For any program P and any pp-goal G,*

i) if $\emptyset \vdash G\ [\theta]$ holds for some substitution θ, then so does $P \models G\theta$;
ii) if $P \models G\gamma$ holds for some substitution γ, then so does $\emptyset \vdash G\ [\theta]$ for a substitution θ such that $G\theta \geq G\sigma$.

In particular, if $\alpha : \mathcal{P}(Ssubst) \rightarrow \mathcal{P}(Ssubst)$ is defined by $\alpha(\Theta) = \{\theta\gamma_{|S} : \theta \in \Theta, \gamma \in Subst, dom(\theta) \subseteq S\}$, for any $\Theta \in \mathcal{P}(SSubst)$, the equalities

$$Decl_m(P)(G) = Decl_f(P)(G) = \alpha(O(P)(G))$$

hold, for any program P and pp-goal G. ■

8 Conclusions

The paper has presented an extension of logic programming aiming at tackling resource handling, as well as operational and declarative semantics for it. From the language point of view, the extension mainly consists of adding pre- and post-conditions acting on resource-atom to Horn clauses. Roughly speaking, these conditions state which resource should be present to let any call be reduced by the considered clause and which atoms should be added at the end of this reduction, if successful. The extended framework has been shown to be well-suited for coding applications involving objects, databases, changes and actions.

The operational and declarative semantics have extended the classical notions of success set, model and immediate consequence operator to the new framework. An effort has been made to design these semantics as simple as possible as well as in the classical lines of logic programming semantics. However, the pre- and post-conditions have raised new problems for which fresh solutions have been proposed. Among these problems are the non-monotonic behavior of resources

and the context dependency of reduction. They have been handled by means of traces. Also of interest is the notion of context which allows to treat in a uniform manner both sequential and parallel composition operators. All these semantics have been related in the paper, as established by propositions 20 and 21.

Acknowledgments

The first author likes to thank the members of the C.W.I. concurrency group, headed by J.W. de Bakker, for their weekly intensive discussions, as well as B. Le Charlier, for his interest in his work. He also likes to thank the Belgian National Fund for support through a Senior Research Assistantship. The second authors wishes to thank Junta Nacional de Investigação Científica e Tecnológica for financial support.

References

[1] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conf. on Logic Programming*, pages 495–510, Jerusalem, Israel, 1990. The MIT Press.

[2] A. Baker. Nonmonotonic Reasoning in the Framework of the Situation Calculus. *Artificial Intelligence*, 49:5–23, 1991.

[3] K. De Bosschere and J.-M. Jacquet. Comparative Semantics of *μlog*. In D. Etiemble and J.-C. Syre, editors, *Proceedings of the PARLE'92 Conference*, volume 605 of *Lecture Notes in Computer Science*, pages 911–926, Paris, 1992. Springer-Verlag.

[4] K. De Bosschere and J.-M. Jacquet. Multi-Prolog: Definition, Operational Semantics, and Implementation. In D.S. Warren, editor, *Proceedings of the ICLP'93 Conference*, pages 299–314, Budapest, Hungary, 1993. The MIT Press.

[5] A. Brogi. And-Parallelism without Shared Variables. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conf. on Logic Programming*, pages 306–321, Jerusalem, Israel, 1990. The MIT Press.

[6] A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

[7] F.S. de Boer, J. Kok, C. Palamidessi, and J.J.M.M. Rutten. Non-Monotonic Concurrent Constraint Programming. In D. Miller, editor, *Proc. Int. Symp. on Logic Programming*, pages 315–334, Vancouver, Canada, 1993.

[8] P.M. Dung. Representing Actions in Logic Programming and its Applications in Database Updates. In D.S. Warren, editor, *Proc. 10th Int. Conf. on Logic Programming*, pages 222–238, Budapest, Hungary, June 1993. The MIT Press.

[9] M. Falaschi, G. Levi, and C. Palamidessi. A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses. *Information and Control*, 60:36–69, 1984.

[10] M. Gelfond and V. Lifschitz. Representing Actions in Extended Logic Programming. In K.R. Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, pages 559–573, Washington, USA, November 1992. The MIT Press.

[11] G. Groβe, S. Hölldobler, J. Schneeberger, U. Sigmund, and M. Tielscher. Equational Logic Programming, Actions, and Change. In K.R. Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, pages 177–191, Washington, USA, November 1992. The MIT Press.

[12] S. Hanks and D. Mac Dermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[13] S. Hölldobler and J. Schneeberger. A New Deductive Approach to Planning. *New Generation Computing*, 8:225–244, 1990.

[14] S. Hölldobler and M. Thielscher. Actions and Specificity. In D. Miller, editor, *Proc. Int. Symp. on Logic Programming*, pages 164–180, Vancouver, Canada, October 1993. The MIT Press.

[15] J.-M. Jacquet and L. Monteiro. Extended Horn Clauses: the Framework and its Semantics. In J.C.M. Baeten and J.F. Groote, editors, *Proc. 2nd Int. Conf. on Concurrency Theory (Concur'91)*, volume 527 of *Lecture Notes in Computer Science*, pages 281–297, Amsterdam, The Netherlands, 1991. Springer-Verlag.

[16] J.-M. Jacquet and L. Monteiro. Communicating Clauses: the Framework and its Semantics. In K.R. Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, Series in Logic Programming, pages 98–112, Washington, USA, November 1992. The MIT Press.

[17] H. Kautz. The Logic of Persistence. In *Proc. AAAI*, pages 401–405, 1986.

[18] N. Kobayashi and A. Yonezawa. ACL — A Concurrent Linear Logic Programming Paradigm. In D. Miller, editor, *Proc. Int. Symp. on Logic Programming*, pages 295–314, Vancouver, Canada, 1993.

[19] R. Kowalski. *Logic for Problem Solving.* North Holland, New York, 1979.

[20] J. Mac Carthy. Situations and Actions and Causal Laws. Standford Artificial Intelligence Project Memo 2, Stanford University, Palo Alto, CA, USA, 1963.

[21] J. Mac Carthy. Applications of Circumscription to Formlizing Commonsense Knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[22] J. Mac Carthy and P.J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[23] M. Masseron, C. Tollu, and J. Vauzeilles. Generating Plans in Linear Logic. In *Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Lecture Notes in Computer Science*, pages 63–75. Springer-Verlag, 1990.

[24] L. Monteiro. Distributed Logic, A Theory of Distributed Programming in Logic. Research report, Departamento de Informática, Universidade de Lisboa, 2885 Monte da Caparica, Lisbon, Portugal, 1986.

[25] A. Porto and P. Rosado. The AbstrAct Scheme for Concurrent Programming. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming*, volume 660 of *Lecture Notes in Artificial Intelligence*, pages 216–241, Berlin, 1993. Springer-Verlag.

[26] V. Saraswat and P. Lincoln. Higher-Order, Linear, Concurrent Constraint Programming. Research report, Xerox Palo Research Center, Palo Alto, CA, USA, 1992.

[27] V.A. Saraswat. *Concurrent Constraint Programming Languages.* The MIT Press, 1993.

[28] L. Schubert. Monotonic Solution for the Frame Problem in the Situation Calculus: an Efficient Method for Worlds with Fully Specified Actions. In H.E. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, 1990.

Heuristics for proving termination using rewriting

(extended abstract)

Thomas Arts and Hans Zantema
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.
e-mail: {thomas,hansz}@cs.ruu.nl

July 5, 1994

Abstract

In this paper we present an approach to prove left-termination of well-moded logic programs automatically. Although we do not yet succeed in defining heuristics such that termination of these programs is proved completely automatically, we are able to reduce the termination problem to an other termination problem that might be easier to solve. We present the approach by an example: the well-moded logic program `bubblesort`. Several steps are performed to reduce the termination problem. All but the last step can be performed automatically on the `bubblesort` program. The last step is to prove termination of a term rewrite system that represents the key-argument of termination.

Keywords and Phrases: logic programs, termination, left-termination of well-moded programs, term rewrite systems, automatic termination prover.

Introduction

One of the methods to prove left-termination of well-moded logic programs is to transform the logic program into a term rewrite system (TRS for short) and prove termination of this TRS (see [KKS91], [GW92], [AM93], [AZ94]). This method can be very useful, since there exist a lot of techniques to prove termination of TRSs. Some of these techniques have been implemented in automatic termination provers. Note that the question whether a TRS terminates is undecidable in general (like it is for logic programs).

In this paper we present an approach to prove left-termination of well-moded logic programs. We study well-moded logic programs that can not be proved terminating by the automatic termination provers that we know of. In our approach the termination proof is still not performed automatically, but we think it is an advancement, since the remaining termination problem seems more basic.

We present the heuristics by a typical example, the program `bubblesort`.

Bubblesort

Consider the well-moded logic program `bubblesort` to obtain a sorted list of numbers from an arbitrary list of (pairwise unequal) numbers.

$$\begin{array}{lcl} bubblesort(xs, xs) & \leftarrow & sorted(xs) \\ bubblesort(xs, zs) & \leftarrow & swap(xs, ys), bubblesort(ys, zs) \end{array}$$

augmented with programs for **sorted** and **swap**

$$
\begin{array}{lcl}
more(s(x), 0) & & \\
more(s(x), s(y)) & \leftarrow & more(x, y) \\
 & & \\
sorted(nil) & & \\
sorted(cons(x, nil)) & & \\
sorted(cons(x, cons(y, ys))) & \leftarrow & sorted(cons(y, ys)), more(y, x) \\
 & & \\
swap(cons(x, xs), cons(x, ys)) & \leftarrow & swap(xs, ys) \\
swap(cons(x, cons(y, ys)), cons(y, cons(x, ys))) & \leftarrow & more(x, y)
\end{array}
$$

with modings $bubblesort(in, out)$, $sorted(in)$, $swap(in, out)$, $more(in, in)$.

We describe in eight steps, how we prove termination of this logic program. The first seven steps can be performed automatically. In these steps the termination problem is reduced to another termination problem, of which we hope that it is in general easier to solve. The eight step is to solve that reduced problem, i.e., proving termination of a specific TRS. This step can not yet be performed automatically.

- First, we make copies of all procedure definitions of a predicate p that is called in different procedure definitions q_1 and q_2 (under the assumption that none of the predicates is mutual recursive). In case of **bubblesort**, we copy the defining procedure definition of the predicate *more*, since *more* is called by both *swap* and *sorted*. We do this to obtain maximal modularity. This results in the logic program:

$$
\begin{array}{lcl}
bubblesort(xs, xs) & \leftarrow & sorted(xs) \\
bubblesort(xs, zs) & \leftarrow & swap(xs, ys), bubblesort(ys, zs)
\end{array}
$$

$$
\begin{array}{lcl}
more_1(s(x), 0) & & \\
more_1(s(x), s(y)) & \leftarrow & more_1(x, y) \\
sorted(nil) & & \\
sorted(cons(x, nil)) & & \\
sorted(cons(x, cons(y, ys))) & \leftarrow & sorted(cons(y, ys)), more_1(y, x)
\end{array}
$$

$$
\begin{array}{lcl}
more_2(s(x), 0) & & \\
more_2(s(x), s(y)) & \leftarrow & more_2(x, y) \\
swap(cons(x, xs), cons(x, ys)) & \leftarrow & swap(xs, ys) \\
swap(cons(x, cons(y, ys)), cons(y, cons(x, ys))) & \leftarrow & more_2(x, y)
\end{array}
$$

- Second, we transform the well-moded logic program into a term rewrite system. This transformation can be performed completely automatically (see [AZ94] for a transformation algorithm). The logic program **bubblesort** transforms into the following TRS $R_{bubblesort}$

$$
\begin{array}{lcl}
bubblesort(xs) & \rightarrow & k_1(xs, sorted(xs)) \\
k_1(xs, sortedout) & \rightarrow & bubblesortout(xs) \\
bubblesort(xs) & \rightarrow & k_2(xs, swap(xs)) \\
k_2(xs, swapout(ys)) & \rightarrow & k_3(xs, bubblesort(ys)) \\
k_3(xs, bubblesortout(zs)) & \rightarrow & bubblesortout(zs)
\end{array}
$$

and the augmented *sorted* and *swap* into TRSs R_{sorted}

$$
\begin{array}{lcl}
more_1(s(x),0) & \rightarrow & out \\
more_1(s(x),s(y)) & \rightarrow & k_6(x,y,more_1(x,y)) \\
k_6(x,y,out) & \rightarrow & out \\
 & & \\
sorted(nil) & \rightarrow & out \\
sorted(cons(x,nil)) & \rightarrow & out \\
sorted(cons(x,cons(y,ys))) & \rightarrow & k_4(x,y,ys,sorted(cons(y,ys))) \\
k_4(x,y,ys,out) & \rightarrow & k_5(x,y,ys,more_1(y,x)) \\
k_5(x,y,ys,out) & \rightarrow & out
\end{array}
$$

and R_{swap}

$$
\begin{array}{lcl}
more_2(s(x),0) & \rightarrow & out \\
more_2(s(x),s(y)) & \rightarrow & k_9(x,y,more_2(x,y)) \\
k_9(x,y,out) & \rightarrow & out \\
 & & \\
swap(cons(x,xs)) & \rightarrow & k_7(x,swap(xs)) \\
k_7(x,swapout(ys)) & \rightarrow & swapout(cons(x,ys)) \\
swap(cons(x,cons(y,ys))) & \rightarrow & k_8(x,y,ys,more_2(x,y)) \\
k_8(x,y,ys,out) & \rightarrow & swapout(cons(y,cons(x,ys)))
\end{array}
$$

One of the main theorems of [AZ94] states that the logic program is left-terminating if the TRS $R_{bubblesort} \cup R_{sorted} \cup R_{swap}$ is innermost terminating. Therefore, we try to prove innermost termination of this TRS. Automatic termination provers normally aim at proving termination and not innermost termination. However, termination of a TRS implies innermost termination of that TRS, but not vice versa. Therefore, we can use automatic termination provers to obtain the desired result.

- Third, we use a termination prover for TRSs to check whether the obtained TRSs are terminating. If the union of all TRSs is terminating, then we are done. In case of `bubblesort` automatic termination provers fail on $R_{bubblesort} \cup R_{sorted} \cup R_{swap}$, whereas $R_{sorted} \cup R_{swap}$ can automatically be proved terminating by RPO.

- Fourth, we check for modularity. Modularity means that if TRSs R_0 and R_1 fulfill some demands and R_0 and R_1 are both innermost terminating, then $R_0 \cup R_1$ is innermost terminating. The modularity results that we need hold for innermost termination, but not for termination.

  We split the TRS in several parts and check if modularity conditions as stated in [Gra92] and [Kri93] are fulfilled. We again use a termination prover for TRSs to check whether some of the parts can be proved terminating automatically. Parts that can be proved terminating, are innermost terminating and hence can be removed from the rewrite system by the modularity results.

  If the `bubblesort` program is split in the two parts R_{sorted} and $R_{bubblesort} \cup R_{swap}$, it fulfills the demand that $R_{bubblesort} \cup R_{swap}$ is a *nice extension* of R_{sorted} (see [Kri93]). This is exactly the reason why we duplicated the procedure definition of the predicate *more*. The TRS R_{sorted} can automatically be proved terminating by a technique called RPO. By the modularity results, we only have to prove innermost termination of $R_{bubblesort} \cup R_{swap}$.

- Fifth, we check if the union of TRSs obtained from the augmented procedure definitions (or what is left over of this after the fourth step) can be proved terminating automatically.

  As we already saw in the third step, in the case of `bubblesort` R_{swap} can automatically be proved terminating by RPO.

- Sixth, we apply a technique, a special version of semantic labelling, on the TRS that has to be proved terminating after the fourth step. By applying the technique we make use of the termination of a part of the TRS, as obtained in the fifth step. Without explaining more about semantic labelling (see [Zan93] for details), we state here that this results for $R_{bubblesort} \cup R_{swap}$ in the following condition:

  The TRS $R_{bubblesort} \cup R_{swap}$ is terminating if there exists no set of terms $t_0, t_1, t_2, \ldots$ such that there is an infinite sequence

$$\begin{array}{lll} swap(t_0) & \rightarrow^+ & swapout(t_1) \\ swap(t_1) & \rightarrow^+ & swapout(t_2) \\ swap(t_2) & \rightarrow^+ & swapout(t_3) \\ & \vdots & \end{array}$$

  (where all reductions are in the terminating TRS R_{swap}). $(*)$

For the logic program **bubblesort** we also could have reduced the termination problem directly to the problem whether there exist terms $t_0, t_1, t_2, \ldots$ such that there is an infinite sequence

$$swap(t_0, t_1), swap(t_1, t_2), swap(t_2, t_3), \ldots$$

However, using TRSs and semantic labelling is a very general technique that is easy extendable to other programs. Moreover, we will see further on that formulating the problem with TRSs results in a practical method to prove that condition $(*)$ holds.

The question whether there is such a sequence of terms is in general undecidable. A major advantage of this method is that in a certain sense we reduced the problem to a problem in which the procedure definition of *bubblesort* does not play a role any more. This new problem is more basic, more to the point. For a well-moded logic program in general, the condition $(*)$ may be a condition about several terms.

- Seventh, we transform the condition $(*)$ in a termination problem for TRSs.

  We derive that if we add the rewrite rule $swapout(x) \rightarrow swap(x)$ to the TRS R_{swap} and this new TRS terminates, then condition $(*)$ is fulfilled. (Assume this new TRS is terminating and the condition $(*)$ does not hold, then there is an infinite reduction in $R_{swap} \cup \{swapout(x) \rightarrow swap(x)\}$, which contradicts that this TRS is terminating). Thus we have to prove termination of the TRS

$$\begin{array}{lll} more_2(s(x), 0) & \rightarrow & out \\ more_2(s(x), s(y)) & \rightarrow & k_9(x, y, more_2(x, y)) \\ k_9(x, y, out) & \rightarrow & out \\ & & \\ swap(cons(x, xs)) & \rightarrow & k_7(x, swap(xs)) \\ k_7(x, swapout(ys)) & \rightarrow & swapout(cons(x, ys)) \\ swap(cons(x, cons(y, ys))) & \rightarrow & k_8(x, y, ys, more_2(x, y)) \\ k_8(x, y, ys, out) & \rightarrow & swapout(cons(y, cons(x, ys))) \\ swapout(x) & \rightarrow & swap(x) \end{array}$$

- Eight and last, we prove termination of the term rewrite system obtained in the previous step. This is where automatic approaches fail so far. Note that from the original termination problem both *bubblesort* and *sorted* are completely eliminated. The resulting TRS is very short and we can prove termination of this TRS, although not automatically, rather straightforward with semantic labelling.

Conclusion

We presented an approach to prove left-termination of well-moded logic programs. The major advantage of this approach is that in an automatic way the termination problem is reduced to a more basic problem. In case of the logic program `bubblesort` we reduced the problem of termination to a termination problem that basically concerns the defining procedure of the *swap* predicate, which strokes with the intuition that the key-argument for termination is hidden in this defining procedure.

References

[AM93] G. Aguzzi and U. Modigliani. Proving termination of logic programs by transforming them into equivalent term rewriting systems. *Proceedings of FST&TCS 13*, Lecture Notes in Computer Science(761), 12 1993.

[AZ94] Thomas Arts and Hans Zantema. Termination of logic programs via labelled term rewrite systems. Technical Report UU-CS-1994-20, Utrecht University, PO box 80.089, 3508 TB Utrecht, May 1994.

[Gra92] Bernhard Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. *Proceedings of LPAR'92*, Lecture Notes in Artificial Intelligence(624):285 296, 1992. Subseries of Lecture Notes in Computer Science.

[GW92] Harald Ganzinger and Uwe Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. *Proceedings of CTRS*, Lecture Notes in Computer Science(656):430 437, July 1992.

[KKS91] M.R.K. Krishna Rao, Deepak Kapur, and R.K. Shyamasundar. A transformational methodology for proving termination of logic programs. *Proceedings of CSL*, Lecture Notes in Computer Science(626):213 226, 1991.

[Kri93] M.R.K. Krishna Rao. *Termination characteristics of logic programs.* PhD thesis, University of Bombay, Tata Institute of Fundamental Research, Homi Bhabha Road, Colaba, Bombay - 400 005, 1993.

[Zan93] Hans Zantema. Termination of term rewriting by semantic labelling. Technical Report RUU-CS-93-24, Utrecht University, July 1993. Accepted for special issue on term rewriting of Fundamenta Informaticae.

Functional Rippling on Relational Structures (abstract)

V. Lombart* Y. Deville
Département d'Ingéniérie Informatique
Université catholique de Louvain
Place Ste-Barbe, 2
B–1348 Louvain-la-Neuve
vl@info.ucl.ac.be yde@info.ucl.ac.be

July 10, 1994

Keywords: logic program synthesis, proof planning.

1 Introduction

The *rippling* heuristic [Bundy *et al* 93] has been rather successfully used in the automatic theorem proving domain, mainly in a functional framework. The use of rippling in a more relational framework, namely a proofs-as-programs approach to logic program synthesis, has shown its weak ability to deal with relational (flat) expressions.

The rippling method relies on the syntactic structure of an expression, and this is not adequate for a relational framework. We show here how to build an external representation of a relational expression, how to use the rippling heuristic on that representation, and how to adapt the induction schemes and rewriting rules to cope with that form of rippling.

While our method does not currently provide us with a truly relational rippling, its keeps the power (in terms of provable theorems) of the original rippling, adds the possibility to prove the same theorems when expressed in a relational form (on which the original rippling nearly always fails), and gives the possibility to deal with relational expressions, which is essential for logic program synthesis.

2 Rippling on Nested Structures

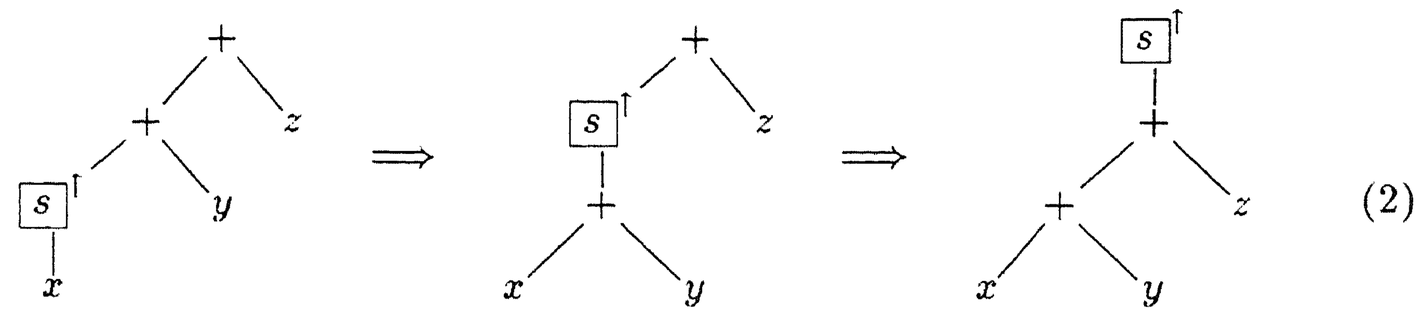
Rippling is a heuristic used for guiding inductive proofs. It tries to remove the "difference" between the induction hypothesis and the induction conclusion. For instance, in the step case of an induction on naturals, a successor function is introduced around the induction variable:

$$(x+y)+z = x+(y+z) \quad \vdash \quad (\boxed{s(\underline{x})}^{\uparrow} + y) + z = \boxed{s(\underline{x})}^{\uparrow} + (y+z) \qquad (1)$$

(the "difference" is what is boxed and not underlined, *i.e.* the s function). Successive rewritings will ripple that difference (also called a *wave front*) out. We can illustrate

*supported by a grant from the Belgian FONDS NATIONAL DE LA RECHERCHE SCIENTIFIQUE

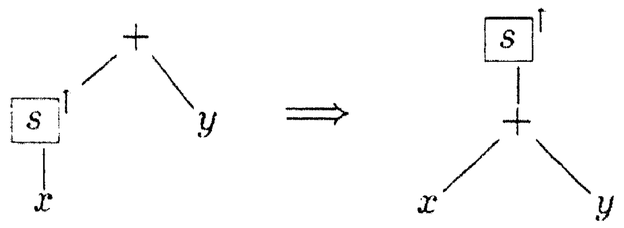
the rewriting process of $(\boxed{s(\underline{x})}^{\uparrow} + y) + z$ by drawing the syntactic tree of the successive expressions:



$$(\boxed{s(\underline{x})}^{\uparrow} + y) + z \implies \boxed{s(\underline{x + y})}^{\uparrow} + z \implies \boxed{s(\underline{(x + y) + z})}^{\uparrow} \tag{2}$$

The rewriting rules (also called *wave rules*) are also annotated with boxes:

$$\boxed{s(\underline{x})}^{\uparrow} + y \implies \boxed{s(\underline{x + y})}^{\uparrow} \tag{3}$$



(4)

A rewriting rule can be used only if there is a match both at the object level (bare expression) and at the meta level (annotations). That gives the rippling good properties: reasonable choice of a rewriting rule, guarantee of termination.

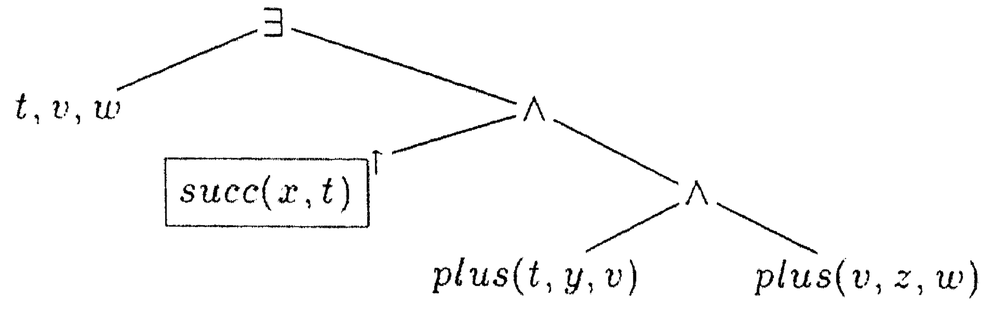
That heuristic relies heavily on the syntactic structure of the expressions. A good tree syntactic structure (high level of function nesting) is essential to obtain good performances.

3 Rippling on Flat Structures

In logic programming or logic program synthesis, it is common to use flatter structures. For instance, in the previous theorem (1), $(\boxed{s(\underline{x})}^{\uparrow} + y) + z$ could be expressed as:

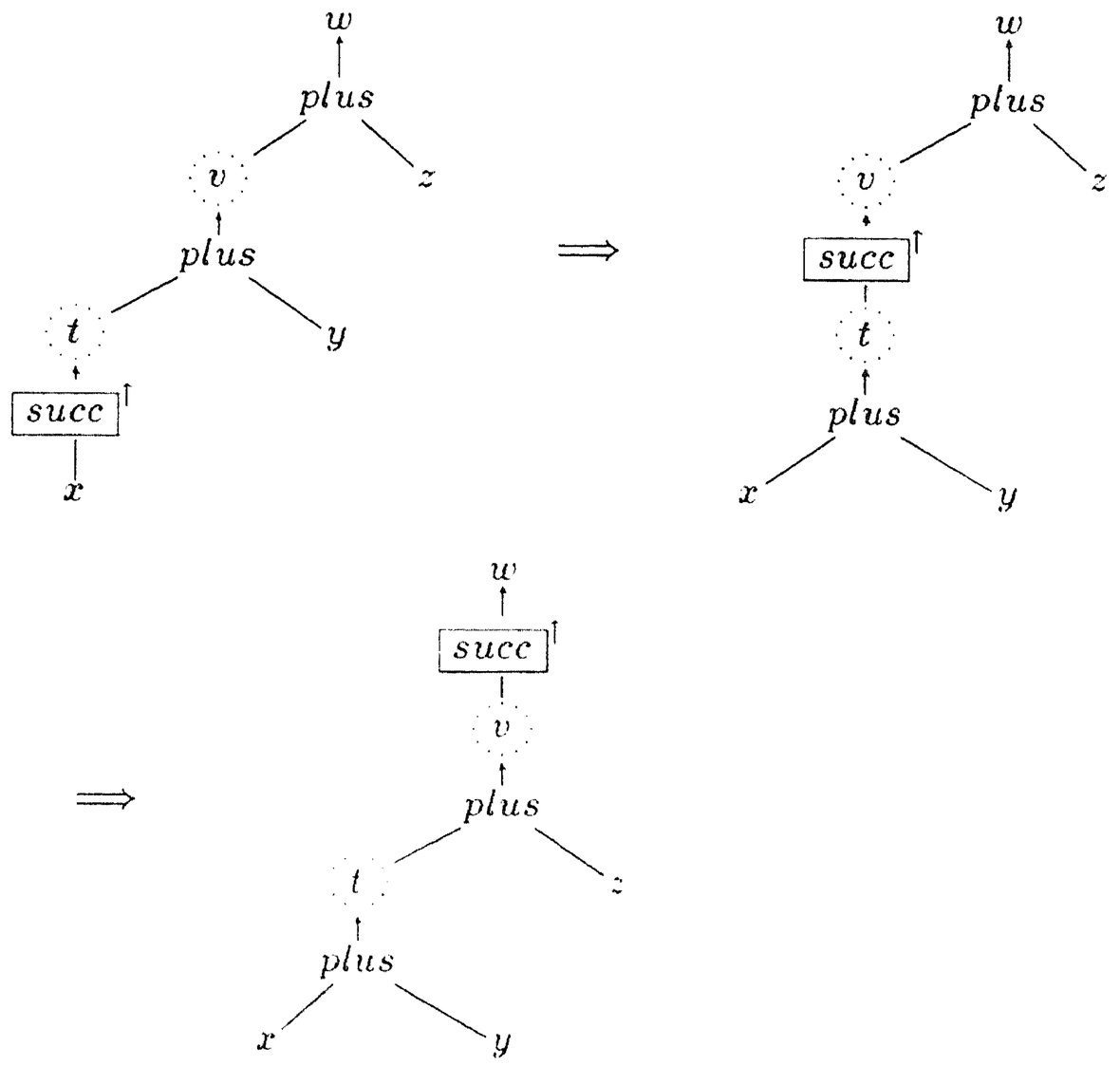
$$\exists t, v, w \,.\, \boxed{succ(x, t)}^{\uparrow} \wedge plus(t, y, v) \wedge plus(v, z, w) \tag{5}$$

But the syntactic tree is no longer useful:



(6)

To solve that problem, we considered an external representation that mimics the functional syntactic structure. The following transformations can be compared with (2), and it should be obvious that basic mechanism is the same:



(7)

How to get and keep such a representation and how to express the rippling heuristic for that structure are the heart of the results presented here. Those methods are presented in the next section, and a more detailed description can be found in [Lombart & Deville 94].

4 Technical Details

4.1 External Representation Construction

The (sub)expressions we deal with have the following form, which we call ε-form:

$$\exists \overline{v} \,.\, \bigwedge_i p_i(\overline{v_i}) \tag{8}$$

with $\overline{v_i}$ a subset of the bound variables ($\overline{v}$) and free variables ($\overline{x}$).

Why have those expressions been chosen? Because any equality between functional arguments can be transformed into an equivalent ε-expression, in the sense that

$$f(g(x)) = y \quad \Leftrightarrow \quad \exists v \,.\, p_g(x, v) \wedge p_f(v, y) \tag{9}$$

if we assume

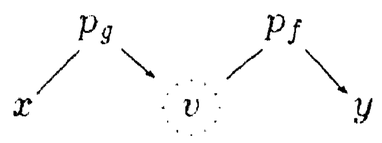
$$p_f(x, z) \Leftrightarrow f(x) = z \qquad \text{and} \qquad p_g(x, z) \Leftrightarrow g(x) = z \tag{10}$$

and that allows us to keep the power of the original rippling.

The external representation of an ε-expression is a graph, whose nodes are the predicates p_i and the components of $\overline{x}$ and $\overline{v}$. The vertices bind each p_i to its arguments v_{i_j}. The vertices are normally undirected, except if we know that an argument v_{i_j} is functionally determined from the others, *e.g.* it is known that in

$plus(x, y, z)$, z is a function of x and y. In that case, the vertex is directed from p_i to v_{i_j}. That functionality information can be given in the form of axioms, or can be deduced from a translation from a functional to a relational form.

Example: the external representation of $\exists v \,.\, p_g(x, v) \wedge p_f(v, y)$ (9) is (the existentially quantified variable is encircled):



(11)

In general, the external representation is a graph and not a tree. But functional expressions, when transformed into relational expressions, give us a tree. The functionality information (directed vertices) allows us to mimic the original syntactic tree, as we did in (7).

4.2 Rippling Control

The rippling heuristic moves a wave-front up (rippling-out), down (rippling-in) or sideways (rippling-sideways) in the syntactic tree. Here, the "up" direction is identified by a directed vertex going "out" of a predicate. This is sufficient for the rippling to proceed.

If we compare our method to the original rippling, it can be noticed that what we require (functionality information) is actually implicit in functional expressions (nested structure). This allows us to claim that with similar information, we can get the same performance as the original rippling.

4.3 Induction Schemes

The step case, in a functional context, of the inductive proof of $\forall x : nat \,.\, p(x)$ is:

$$p(x) \vdash p(s(x)) \tag{12}$$

and in a relational context:

$$p(x), succ(x, t) \vdash p(t) \tag{13}$$

In the functional case, the successor function is automatically introduced in the conclusion, but in the relational case, the successor predicate is not. Further rewriting is therefore not immediately possible. Fortunately, it can be proven that for common induction schemes (based on the successor function for naturals, or on the list construction (from head and tail) function for lists), we could rewrite the inductive step (13) to obtain an ε-expression:

$$p(x) \vdash \exists t \,.\, succ(x, t) \wedge p(t) \tag{14}$$

which solves the above mentioned problem.

4.4 Rewriting Rules

The rewriting rules also need some modifications. For instance,

$$\forall x, y \,.\, \boxed{s(\underline{x})}^{\uparrow} + y = \boxed{s(\underline{x + y})}^{\uparrow} \tag{15}$$

can be transformed into a rewriting rule, but it will be unusable for an ε-expression. A naive translation into a relational form gives:

$$\begin{aligned} \forall x, y \,.\, \exists v_1, v_2, v_3, v_4 \,.\, & succ(x, v_1) \wedge plus(v_1, y, v_2) \wedge plus(x, y, v_3) \\ & \wedge succ(v_3, v_4) \wedge equal(v_2, v_4) \end{aligned} \tag{16}$$

which cannot be transformed into a rewriting rule.

We propose a better translation method: the expressions produced can be transformed into rewriting rules, directly usable on ε-expressions. The expressions obtained have been proved equivalent to those obtained by the naive translation method. Our method gives the following translation of (15):

$$\forall x, y, z \;.\; (\exists v \;.\; \boxed{succ(x, v)}^{\uparrow} \wedge plus(v, y, z)) \\ \Leftrightarrow (\exists v \;.\; plus(x, y, v) \wedge \boxed{succ(v, z)}^{\uparrow}) \tag{17}$$

5 Application

Let us illustrate our methods on an example, the step part of an induction presented in (1). The relational form of (1) is:

$$succ(x, t), \tag{18}$$

$$(\exists v \;.\; plus(x, y, v) \wedge plus(v, z, w)) \\ \Leftrightarrow (\exists v \;.\; plus(x, v, w) \wedge plus(y, z, v)) \tag{19}$$

$$\vdash (\exists v \;.\; plus(t, y, v) \wedge plus(v, z, w)) \\ (\exists v \;.\; plus(t, v, w) \wedge plus(y, z, v)) \tag{20}$$

As we mentioned before (section 4.3), we have to introduce $succ(x, t)$ in (20) to allow further rewriting:

$$\vdash (\exists t, v \;.\; \boxed{succ(x, t)}^{\uparrow} \wedge plus(t, y, v) \wedge plus(v, z, w)) \\ \Leftrightarrow (\exists t, v \;.\; \boxed{succ(x, t)}^{\uparrow} \wedge plus(t, v, w) \wedge plus(y, z, w)) \tag{21}$$

The rewriting rule (3) has to be transformed (see section 4.4) into:

$$(\exists v \;.\; \boxed{succ(x, v)}^{\uparrow} \wedge plus(v, y, z)) \implies (\exists v \;.\; plus(x, y, v) \wedge \boxed{succ(v, z)}^{\uparrow}) \tag{22}$$

And in the left hand side of (21), the wave-front can be rippled-out as illustrated in (7):

$$\exists t, v \;.\; \overline{\boxed{succ(x, t)}}^{\uparrow} \wedge plus(t, y, v) \wedge plus(v, z, w) \\ \implies \\ \exists t, v \;.\; plus(x, y, t) \wedge \boxed{succ(t, v)}^{\uparrow} \wedge plus(v, z, w) \tag{23} \\ \implies \\ \exists t, v \;.\; plus(x, y, t) \wedge plus(t, z, v) \wedge \boxed{succ(v, w)}^{\uparrow} \tag{24}$$

by choosing the "out" direction as explained in section 4.2.

6 Conclusion

We have shown that it is possible to reproduce the functional rippling behaviour in a relational context, and to get the same efficiency if the same information is available (including the functionality information). Moreover, our method deals with relational expressions, which is essential in the logic program synthesis domain.

References

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Lombart & Deville 94] V. Lombart and Y. Deville. Functional rippling on relational structures. Technical Report 94-16, Department of Computing Science and Engineering, Université catholique de Louvain, 1994.

Simple Extended Logic Programming

C. M. Jonker
Institut für Informatik und angewandte Mathematik
Länggassstrasse 51
CH-3012 Bern, Switserland
jonker@iam.unibe.ch

August 24, 1994

Keywords: Extended Logic Programming, Explicit Negation, Supported Semantics.

Abstract

In the late eighties Normal Logic Programming was enriched with an explicit negation. In Normal Logic Programming something can only be false by default also known as negation as failure. The default negation is denoted by the default negation symbol $\sim$. The explicit negation, denoted by the symbol "$\neg$", can be used to express explicitly that some information is false. In Extended Logic Programming both the standard default negation and the new explicit negation are allowed. The form of the Extended Programming rules is:

$$C \leftarrow E_1, \ldots, E_n$$

where C is either an atom A or an explicitly negated atom $\neg A$ and every E_i $(1 \leq i \leq n)$ is either an atom A, a default negated atom $\sim A$, an explicitly negated atom $\neg A$ or an explicitly negated atom that is also negated by default, $\sim\neg A$. In other words, the explicit negation symbol is allowed on both sides of the arrow, whereas the default negation may only appear on the right-hand side. In this paper we show that restricting the explicit negation symbol to the left-hand side does not reduce the expressive power of formalism by much. Simple Extended Logic Programming is the form of Logic Programming in which the rules are of the form:

$$C \leftarrow D_1, \ldots, D_n$$

where C is either an atom A or an explicitly negated atom $\neg A$ and every D_i $(1 \leq i \leq n)$ is either an atom A or a default negated atom $\sim A$. We show that for every Extended Logic Program P there is a Simple Extended Logic Program $S(P)$ that is semantically equivalent to P under the condition that every explicitly negated atom of $S(P)$ is supported.

1 Introduction

We expect the reader to be familiar with Normal Logic Programming, its three-valued semantics and SLDNF-resolution. The form of Normal Logic Programming rules we use in this paper is:

$$A \leftarrow D_1, \ldots, D_n$$

where A is an atom and every D_i $(1 \leq i \leq n)$ is either an atom A' or a default negated atom $\sim A'$. The symbol "$\sim$" denotes the usual negation as failure. The symbol "$\leftarrow$" denotes a unidirectional

implication and is not the same as the usual logical implication. The informal meaning of such a Normal Logic Programming rule is: "*If each* D_i $(1 \leq i \leq n)$ *is true, then* A *is true.*" For more information on Normal Logic Programming we refer to [1, 4].

In Normal Logic Programming an atom can only be false by negation as failure. However, in database applications the need is often felt to make some atoms explicitly false. Therefore, the language of Normal Logic Programming has been extended with an explicit negation. Having an explicit negation can have two effects. The first and undesirable effect is that it is possible to write inconsistent programs. The program can contain the statement that atom A must be true and that A must be false. The second, and more desirable, possible effect is the following. In the three-valued semantics of Normal Logic Programming an atom corresponds to "undefined" if the SLDNF-derivation of the atom neither fails nor succeeds. In such cases the explicit statement that the atom is false can force the issue. In the following we will motivate the use of an explicit negation from the grey area between the natural language interpretations of opposite words like "hatred" and "love". Consider for example the sentences

Mary does not hate John (1)

Mary loves John (2)

People tend to interpret Sentence (1) a little bit different from Sentence (2). Sentence (2) leaves no room for doubt, it expresses a positive attitude towards John. On the hearing of Sentence (1) we realise that it does not necessarily mean that Mary loves John. On the other hand, on the bases of Sentence (2) we know that Sentence (1) is true. So "no hatred" does not imply "love", but "love" implies "no hatred".

We call this last relation, i.e., the positive statement implies the negation of its opposite, *Opposition*. Unfortunately, Opposition is not expressible in Normal Logic Programming. That "love" and "hatred" are opposite words is reflected in the observation that "no love" does not imply "hatred", whereas "hatred" does imply "no love". Note that the negation used in these implications is not the negation as failure of Normal Logic Programming. Whether or not an SLDNF-resolution for "hatred" fails, if "love" succeeds, we want "hatred" to be false. To indicate the difference between negation as failure and this new "negation as success", the symbol "$\neg$" for "negation as success" is used next to the symbol "$\sim$" for negation as failure. With the new symbol we express the opposition relation between "love" and "hatred" as follows:

$$\neg hatred \leftarrow love \quad (3)$$

$$\neg love \leftarrow hatred \quad (4)$$

Note that the opposition relation consists of two implications instead of one. This is because the symbol"$\leftarrow$" does not represent the usual logical implication. In Section 3 an example will be given that shows that rules (3) and (4) are not equivalent. Simple Extended Logic Programs are Normal Logic Programs enriched with the explicit negation symbol on the left-hand side of programming rules. Hence, Simple Extended Logic Programming rules are of the form

$$C \leftarrow D_1, \ldots, D_n$$

where C is either an atom A or an explicitly negated atom $\neg A$ and every D_i $(1 \leq i \leq n)$ is either an atom A or a default negated atom $\sim A$. Thus, the Opposition relation is expressible in Simple

Extended Logic Programming.

Once it is decided to allow the new negation symbol on the left-hand side of rules, it is not a great modification to allow it on the right-hand side as well. Thus we have Extended Logic Programs. It seems as if the formalism gains a lot of expressive power from this modification. However, we can easily transform any Extended Logic Program into a Simple Extended Logic Program by substituting every $\neg A$ that appears somewhere on the right-hand side of an extended rule by a new atomic symbol non_A and then express syntactically that non_A and A are opposites.

Example 1.1 Consider the Extended Logic Program P:

$$\begin{aligned} Hates(Mary,John) &\leftarrow Nasty\_Person(John), \neg Nice\_to(John,Mary) \\ \neg Nice\_to(John,Mary) &\leftarrow \\ Nasty\_Person(John) &\leftarrow \end{aligned}$$

We transform this program to the Simple Extended Logic Program P':

$$\begin{aligned} Hates(Mary,John) &\leftarrow Nasty\_Person(John), non_{\text{Nice\_to(John,Mary)}} \\ non_{\text{Nice\_to(John,Mary)}} &\leftarrow \\ Nasty\_Person(John) &\leftarrow \\ \neg non_{\text{Nice\_to(John,Mary)}} &\leftarrow Nice\_to(John,Mary) \\ \neg Nice\_to(John,Mary) &\leftarrow non_{\text{Nice\_to(John,Mary)}} \end{aligned}$$

△

As will be made clear in this paper, the increase of the number of atoms caused by the transformation of the Extended Logic Program leads to an increase in the number of interpretations of the program. Thus, it is impossible to have a one-to-one correspondence between the interpretations of P and P'. However, by placing a small constraint on the interpretations with respect to the explicitly negated atoms of P', the number of interpretations of P is the same as the number of interpretations of P', and a one-to-one correspondence is shown to exist. The constraint on the semantics is called $\neg$-*supportedness*. We show that for the $\neg$-supported semantics Extended Logic Programming is semantically equivalent to an extension of Normal Logic Programming with the ability to express the opposition relation between atoms, as is possible in Simple Extended Logic Programming.

In Section 2 the syntactical setting of this paper is presented. The first order language is characterized, program rules and programs are defined, and the various classes of logic programs are introduced. In Section 3 we present a four-valued semantics for extended logic programs. We define the notions *interpretation*, *Coherence*, *model*, and *interpretation-base*. Section 4 describes the supported semantics. The supported semantics was originally developped for Normal Logic Programming, but here we extend its definition to Extended Logic Programming. Furthermore, we define the $\neg$-supported semantics as a semantics that only demands supportedness for explicitly negated atoms. Section 5 contains a sketch proof of the main results of this paper, i.e., the semantical equivalence of the classes of Simple Extended and of Extended Logic Programming with respect to the $\neg$ - supported semantics.

2 Syntax

In this paper we consider a countable first order language $\mathcal{L}$ which consists of the following basic symbols:
- Countably many free and bound variables (x, x_1, x_2, ...).
- A countable number of function symbols of finite arities with at least one of arity 0. Function symbols of arity 0 stand for constants.
- Countably many relation symbols of finite arities greater than 0.
- The propositional constants C, T, U and F.
- The connectives $\wedge$, $\vee$, $\leftarrow$, $\sim$ and $\neg$, and the quantifiers $\exists$ and $\forall$.

The function and relation symbols are determinate for the language, as the basic vocabulary of all languages under consideration is the same. We assume the reader to be familiar with the notions "term" and "ground term". The terms are denoted by a, a_1, a_2, ... The *atoms* (A, A_1, A_2, ...) of $\mathcal{L}$ are all expressions $R(a_1, \ldots, a_n)$ where R is an n-ary relation symbol of $\mathcal{L}$. An atom $R(a_1, \ldots, a_n)$ is a "ground" atom iff a_1, ..., a_n are ground terms. The Herbrand Base $\mathcal{B}_\mathcal{L}$ is the set of all ground atoms of language $\mathcal{L}$. The *basic formulas* of $\mathcal{L}$ are the atoms plus the propositional constants C, T, U and F. The *formulas* (F, G, F_1, G_1, F_2, G_2, ...) of $\mathcal{L}$ are generated as follows:

1. If F is a basic formula of $\mathcal{L}$, then F, $\sim F$, $\neg F$ and $\sim\neg F$ are formulas.
2. If F and G are formulas, then $F \wedge G$, $F \vee G$ and $F \leftarrow G$ are formulas.
3. If $F(u)$ is a formula, then $\exists x F(x)$ and $\forall x F(x)$ are formulas.

We define $FORM$ to be the set of ground formulas of $\mathcal{L}$. The symbol "$\sim$" stands for the usual negation of Normal Logic Programming. The symbol "$\neg$" is the symbol for explicit negation used in Extended Logic Programming. The *classical literals* (C, C_1, C_2, ...) of $\mathcal{L}$ are all expressions of the form A or $\neg A$, where A is an atom of $\mathcal{L}$. The *default literals* (D, D_1, D_2, ...) of $\mathcal{L}$ are all expressions of the form A or $\sim A$, where A is an atom of $\mathcal{L}$. The *extended literals* (E, E_1, E_2, ...) of $\mathcal{L}$ are all expressions of the form C or $\sim C$, where C is a classical literal of $\mathcal{L}$. The *literals* (L, L_1, L_2, ...) of $\mathcal{L}$ are all expressions of the form F, $\sim F$, $\neg F$ and $\sim\neg F$, where F is a basic formula. The symbol "$\leftarrow$" is an implication, but not the usual logical implication. Its interpretation is given in Section 3.

Definition 2.1 (Rule) A rule r is of the form $L \leftarrow body$, where $body$ is a finite, possibly empty, set of literals. Literal L is called the *head* of r, also denoted by $hd(r)$. The body of rule r is denoted by $body(r)$.

A rule $L \leftarrow body$ is the abbreviated notion of $\forall x_1 \ldots \forall x_s (L \leftarrow \bigwedge body)$, where x_1, ..., x_s are all the variables occurring in $L \leftarrow body$. This implies that every rule is a formula. For convenience, a rule $L_0 \leftarrow \{L_1, \ldots, L_n\}$ will be abbreviated by $L_0 \leftarrow L_1, \ldots, L_n$. Informally, a rule $L_0 \leftarrow L_1, \ldots, L_n$ means that literal L_0 is true if each of the literals L_1, ..., L_n is true.

Formally, a Logic Program is a finite set of rules. However, we assume, without loss of generality, that a Logic Program is a possibly infinite set of ground rules. In [2] Gelfond and Lifschitz made the same simplifying assumption. $\mathcal{L}_P$ is the language determined by the function and relation symbols used in the program P. We abbreviate the Herbrand base of $\mathcal{L}_P$ with $\mathcal{B}_P$.

Based on the form of the rules we define three classes of programs: NLP, SELP and ELP.

Class	*Name*	*form of rules*
NLP	Normal	$A \leftarrow D_1, \ldots, D_n$
SELP	Simple Extended	$C \leftarrow D_1, \ldots, D_n$
ELP	Extended	$C \leftarrow E_1, \ldots, E_n$

The set $Lit(\mathcal{L})$ is the set of all literals of $\mathcal{L}$. If V is a set of formulas, then $\sim V$ denotes the set $\{\sim F \mid F \in V\}$.

3 Semantics

In this section we define the notions interpretation, valuation, and model. We define two orderings on the set of truth-values $\mathcal{FOUR}$ and extend these to orderings of interpretations. The truth-values used in this paper are **f** for *false*, **u** for *undefined*, **t** for *true*, and **c** for *contradictory*. The set of propositional constants $\{\mathrm{F}, \mathrm{U}, \mathrm{T}, \mathrm{C}\}$ is closely related to $\mathcal{FOUR}$: F corresponds to **f**, U to **u**, T to **t**, and C to **c**. We use two different transitive orderings of the truth values as a lattice: the lattice 4_t defined by $\mathbf{f} <_t \mathbf{u} <_t \mathbf{t}$ and $\mathbf{f} <_t \mathbf{c} <_t \mathbf{t}$ (truth-ordering) and the semi-lattice 4_k defined by $\mathbf{u} <_k \mathbf{f} <_k \mathbf{c}$ and $\mathbf{u} <_k \mathbf{t} <_k \mathbf{c}$ (knowledge-ordering). Let $V \subseteq \mathcal{FOUR}$, we define $glb_t(V)$ ($lub_t(V)$) to be the greatest lower (least upper) bound of set V with respect to the ordering $<_t$. The reason we use four truth-values instead of three is that inconsistencies can occur if an atom is made explicitly true and explicitly false:

Example 3.1 *Consider the Extended Program P:*

$$A \leftarrow$$
$$\neg A \leftarrow$$

Informally, the rules of P express that both A and $\neg A$ are true, since both rules have an empty body. The idea of having an explicit negation is that the truth of $\neg A$ should normally imply the falsity of A[1], so that A is both true and false. In the same way the truth of A implies the falsity of $\neg A$, so that $\neg A$ is also both true and false. Therefore, A is contradictory and the value **c** *will be assigned to it.* △

In the following we find it convenient to work with the set of literals that are true in an interpretation. We show that there exists a one-to-one correspondence between this set of literals and the interpretation itself. Since we assume that P is a possibly infinite set of ground rules it is sufficient to define interpretations of a language $\mathcal{L}$ as functions from the ground formulas of $\mathcal{L}$ to $\mathcal{FOUR}$. We assume the law of double negation to be an axiom for $\sim$ and $\neg$ seperately. So $\sim\sim F$ denotes F and $\neg\neg F$ denotes F. However, $\sim\neg F$ does in general not denote F.

Definition 3.2 (Interpretation) An interpretation I of a language $\mathcal{L}$ is a function from the set of literals of $\mathcal{L}$ to the set $\mathcal{FOUR}$ such that for every literal L of $\mathcal{L}$:

1. $I(L) = \mathbf{t}$ iff $I(\sim L) = \mathbf{f}$.
2. $I(L) = \mathbf{u}$ iff $I(\sim L) = \mathbf{u}$.
3. $I(L) = \mathbf{c}$ iff $I(\sim L) = \mathbf{c}$.

[1] See Definition 3.3.

4. If $F \in \{\mathrm{F}, \mathrm{U}, \mathrm{T}, \mathrm{C}\} \subseteq FORM$ and $y \in \mathcal{FOUR}$: $I(F) = y$ iff F corresponds to y.

The definition of an interpretation can be extended to the set $FORM$ of all ground form as follows:

1. If the formula F has the form $F_1 \wedge F_2$, then $I(F) = glb_t(\{I(F_1), I(F_2)\})$.
2. If the formula F has the form $F_1 \vee F_2$, then $I(F) = lub_t(\{I(F_1), I(F_2)\})$.
3. If the formula F has the form $F_1 \leftarrow F_2$, then

$$I(F_1 \leftarrow F_2) = \begin{cases} \mathbf{t} & \text{iff } I(F_2) \leq_t I(F_1) \text{ or } I(F_1) \in \{\mathbf{t}, \mathbf{c}\} \\ \mathbf{f} & \text{otherwise} \end{cases}$$

We say I is consistent iff there is no literal L of $\mathcal{L}$ such that $I(L) = \mathbf{c}$. We say I is two-v I is consistent and for every literal L of $\mathcal{L}$ we have $I(L) \neq \mathbf{u}$.
In the definition of interpretation the negation symbol "$\neg$" is treated differently from the n symbol "$\sim$". In fact, A and $\neg A$ are totally unrelated. For every atom A, we can even c $\neg A$ to be a new atomic symbol. In other words, if A and $\neg A$ are unrelated, then Extende Programming is exactly the same as Normal Logic Programming. Although this plays nc the proof of the main theorem of this paper, the least we want is that the truth of $\neg A$ the falsity of A. Therefore, Pereira et al. [5] suggested the following *Coherence Principle* be satisfied for all classical literals C:

$$C \text{ implies } \sim\neg C.$$

Note that the Coherence Principle is closely related to the Opposition relation. For every literal C, C and $\neg C$ are opposites, which are related by the Coherence Principle by us default negation symbol instead of using the explicit negation symbol of the Opposite r The Coherence Principle is reflected in the following definition.

Definition 3.3 (Coherence) [5] Let I be an interpretation of $\mathcal{L}$, then I is a *coherent* i tation of $\mathcal{L}$ if for all classical literals C of $\mathcal{L}$: If $I(C) \in \{\mathbf{t}, \mathbf{c}\}$, then $I(\neg C) \in \{\mathbf{f}, \mathbf{c}\}$.

Lemma 3.4 *If I is a coherent interpretation of $\mathcal{L}$, then $I(\sim\neg C \leftarrow C) = \mathbf{t}$ for all classica C of $\mathcal{L}$.*

Proof Let C be a classical literal. Consider the four possible truth-values of C:

$I(C) = \mathbf{c}$. Since I is coherent, we have $I(C) = \mathbf{c}$ implies that $I(\neg C) \in \{\mathbf{f}, \mathbf{c}\}$. By defi interpretation we have $I(\neg C) \in \{\mathbf{f}, \mathbf{c}\}$ implies that $I(\sim\neg C) \in \{\mathbf{t}, \mathbf{c}\}$ and thus $I($ $C) = \mathbf{t}$.

$I(C) = \mathbf{t}$. Since I is coherent, we have $I(C) = \mathbf{t}$ implies that $I(\neg C) \in \{\mathbf{f}, \mathbf{c}\}$. By defi interpretation we have $I(\neg C) \in \{\mathbf{f}, \mathbf{c}\}$ implies that $I(\sim\neg C) \in \{\mathbf{t}, \mathbf{c}\}$ and thus $I($ $C) = \mathbf{t}$.

$I(C) = \mathbf{u}$. Suppose $I(\sim\neg C \leftarrow C) \neq \mathbf{t}$. By definition of interpretation this means that $I($ $\{\mathbf{f}, \mathbf{u}\}$ and thus that $I(\neg C) \in \{\mathbf{t}, \mathbf{u}\}$. If $I(\neg C) = \mathbf{u}$, then $I(\sim\neg C) = \mathbf{u}$ and thus I $C) = \mathbf{t}$. If $I(\neg C) = \mathbf{t}$, then, since I is coherent, $I(\neg\neg C) \in \{\mathbf{f}, \mathbf{c}\}$ and thus $I(C) \in \{$ definition of interpretation $I(C) \in \{\mathbf{f}, \mathbf{c}\}$ contradicts $I(C) = \mathbf{u}$. Hence, $I(\sim\neg C \leftarrow$

$I(C) = \mathbf{f}$. For all y such that $I(\sim\neg C) = y$, we have $\mathbf{f} \leq_t y$ and thus $I(\sim\neg C \leftarrow C) = \mathbf{t}$. □

Definition 3.5 (Model) Let I be a consistent interpretation of a ground language $\mathcal{L}$, let F be a ground formula of $\mathcal{L}$, and let S be a set of ground formulas of $\mathcal{L}$. Then

a. I is a model for F if $I(F) = \mathbf{t}$.

b. I is a model for S if I is a model for each formula of S.

I is a *coherent model* for S iff I is a model for S, and I is a coherent and consistent interpretation of $\mathcal{L}$.

Example 3.6 The influence of Coherence can be shown by considering the Extended Logic Program P consisting of the formulas:

$$\begin{aligned} A_1 &\leftarrow A_2 \\ \neg A_2 &\leftarrow \sim\neg A_2 \end{aligned}$$

There exists a model I of P such that $I(A_2) = \mathbf{t}$ and $I(\neg A_2) = \mathbf{t}$, which makes I not coherent. In model I atom A_1 is true as well. In a coherent model A_2 and $\neg A_2$ cannot both be true. Although the coherence principle relates C and $\neg C$, the coherence principle does not imply that for every coherent interpretation I': $I'(C) = \mathbf{t}$ iff $I'(\neg C) = \mathbf{f}$ for every C. There exists, for example, a model I_1 for P such that $I_1(A_2) = \mathbf{f}$ and $I_1(\neg A_2) = \mathbf{u}$. △

In the introduction we defined the Opposition relation as consisting of two implications for any two opposite literals. The following example shows that these two implications are not equivalent.

Example 3.7 Consider the opposites A_1 and A_2. The Opposition relation for A_1 and A_2 is:

$$\neg A_1 \leftarrow A_2 \tag{5}$$
$$\neg A_2 \leftarrow A_1 \tag{6}$$

There exists a model I of these two rules such that $I(A_1) = \mathbf{f}$, $I(\neg A_1) = \mathbf{t}$, $I(A_2) = \mathbf{u}$ and $I(\neg A_2) = \mathbf{f}$. Model I is even a coherent model of these two rules. Essentially, the two rules are not equivalent, because even for coherent interpretations the truth-value of $\sim A$ need not be equivalent to the truth-value of $\neg A$, for any atom A. In this example, $I(\sim A_2) = \mathbf{u}$, while $I(\neg A_2) = \mathbf{f}$. △

The ordering $\leq_k$ between truth-values is extended to a knowledge ordering $\leq_k$ between interpretations of a language $\mathcal{L}$ by defining: If I and I' are interpretations of $\mathcal{L}$, then $I \leq_k I'$ iff $I(C) \leq_k I'(C)$ for every classical literal C of $\mathcal{L}$. The truth ordering $\leq_t$ for interpretations is defined as: If I and I' are two interpretations of $\mathcal{L}$, then $I \leq_t I'$ iff $I(C) \leq_t I'(C)$, for all classical literals C of $\mathcal{L}$.

Instead of working with interpretations we find it convenient to work with so called *interpretation-bases*. An interpretation-base is the set of literals that are true or inconsistent in an interpretation.

Definition 3.8 (Interpretation-base) Let I be an interpretation of $\mathcal{L}$. We define $B_I = \{L \in Lit(\mathcal{L}) \mid I(L) \in \{\mathbf{t}, \mathbf{c}\}\}$ to be the interpretation base of I.

Lemma 3.9 *For every set of literals B there is a unique interpretation I such that $B_I = B$.*

Proof We first prove that there is an interpretation I such that $B_I = B$: Define I to be the function from the literals of $\mathcal{L}$ to $\mathcal{FOUR}$ such that

a. $I(L) = \mathbf{t}$ iff $L \in B$ and $\sim L \notin B$.

b. $I(L) = \mathbf{c}$ iff $L \in B$ and $\sim L \in B$.

c. $I(L) = \mathbf{f}$ iff $L \notin B$ and $\sim L \in B$.

d. $I(L) = \mathbf{u}$ iff $L \notin B$ and $\sim L \notin B$.

e. If $F \in \{\mathrm{F}, \mathrm{U}, \mathrm{T}, \mathrm{C}\} \subseteq FORM$ and $y \in \mathcal{FOUR}$: $I(F) = y$ iff F corresponds to y.

It is easy to check that I is an interpretation and that $B_I = B$.
Suppose there are two interpretations I and I' such that $B_I = B = B_{I'}$. Since the interpretation of arbitrary formulas is determined by the interpretation of literals, we only have to prove that for every literal L: $I(L) = I'(L)$. We consider the case $I(L) = \mathbf{t}$. By definition of L this implies that $L \in B$ and $\sim L \notin B$. Assume that $I(L) \neq I'(L)$. We consider the three cases $I'(L) =$

c: then $I'(\sim L) = \mathbf{c}$ as well. By Definition 3.8 $\sim L \in B_{I'}$. Since $B_{I'} = B$, we have $\sim L \in B$, contradiction.

f: then $I'(\sim L) = \mathbf{t}$. By Definition 3.8 $\sim L \in B_{I'}$. Since $B_{I'} = B$, we have $\sim L \in B$, contradiction.

u: By Definition 3.8 $L \notin B_{I'}$. Since $B_{I'} = B$, we have $L \notin B$, contradiction.

The cases $I(L) = \mathbf{c}$, $I(L) = \mathbf{f}$ and $I(L) = \mathbf{u}$, can be dealt with in the same way. □

Let I be an interpretation with interpretation-base B_I. We define

$$\begin{aligned} I_T &= \{C \mid C \in B_I\} \\ I_F &= \{C \mid \sim C \in B_I\} \end{aligned}$$

In other words, I_T is the set of classical literals true or contradictory in I and I_F is the set of classical literals that are false or contradictory in I. On the basis of Lemma 3.9 we use interpretation-bases to stand for interpretations and vice versa. We define a *semantics* of a program P to be a specific set of interpretations. For example, the *stable semantics* [3, 5] is defined as the set of *stable interpretations* of the program in question. Or the *coherent semantics* is defined as the set of all coherent interpretations of the program.

4 Supported Semantics

In the definition of the supported semantics we use the notion of the defining formula of a literal which, informally, is the set of arguments within the program for making that literal true.

Definition 4.1 (Defining formula) Suppose P is a program, and that there are m rules in P with literal L as head, so that the i-th rule is of the form

$$L \leftarrow L_{i,1}, \ldots L_{i,k(i)}$$

and has $k(i)$ literals in its body. The *defining formula* $\Gamma_P(L)$ of literal L with respect to P is defined to be the formula

$$\Gamma_P(L) := \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k(i)} L_{i,j}$$

Empty disjunctions are equivalent with the special atom F. For ground programs it may be the case that there is an infinite number of rules with literal L as head. This means that the defining formula $\Gamma_P(L)$ might be an infinite disjunction of rule bodies. We therefore have to extend definition 3.2 by defining the valuation $I(H)$ of an infinite disjunction $H = \bigvee_{i \geq 0} H_i$ to be the least upper bound of the valuations $I(H_i)$ of the disjuncts H_i of H. So $I(H) = lub_t\{I(H_i) \mid i \geq 0\}$.

The supported semantics was originally a two-valued semantics for Normal Logic Programming, but in the remainder of this paper we need a more general definition that suits $\mathcal{FOUR}$ and Extended Logic Programming.

Definition 4.2 (Supported Semantics) Let P be a Logic Program and let I be an interpretation of P. We say that classical literal C is *supported* by interpretation I if $I(C) = I(\Gamma_P(C))$, where $\Gamma_P(C)$ is the defining formula of C with respect to P. I is a supported interpretation of P iff every classical literal C is supported by I.

We define a semantics to be supported if it is a subset of the set of supported models[2]. The supported semantics is the set of all supported models of the program. Informally, the value of a ground atom in a supported model must coincide with the value of the corresponding instance of the defining formula of the predicate symbol occurring in the ground atom. For example, circular dependencies, like the support of A_1 in example 4.3, are permitted, but assignment of the value **t** to an atom while assigning the value **f** to its defining formula is not. In other words, for every truth-value assigned by the model, a support must exist with respect to the model and the program.

Example 4.3 Consider the program $P = \{A_1 \leftarrow A_1,\ A_2 \leftarrow A_3\}$. Let M be the model $\{A_1, A_2, \sim A_3\}$ of P, i.e., the model that makes A_1 and A_2 true and A_3 false. The defining formulas of A_1, A_2 and A_3 are $\Gamma_P(A_1) = A_1$, $\Gamma_P(A_2) = A_3$ and $\Gamma_P(A_3) =$ F. Then M is not a supported model of P, since $M(A_2) = \mathbf{t} \neq M(\Gamma_P(A_2)) = \mathbf{f}$. Note that A_1 is supported by M, since $M(A_1) = M(\Gamma_P(A_1)) = M(A_1)$. △

Definition 4.4 (¬-Supported Semantics) Let P be an Extended Logic Program and I an interpretation of P. I is a ¬-supported interpretation of P iff every explicitly negated atom $\neg A$ is supported by I.

5 SELP and ELP

In this section we sketch the proof of the partial semantical equivalence between the classes SELP and ELP. We first define what it means that an Extended P and a Simple Extended Logic Program P' are semantically equivalent. Let *call* $(\mathcal{L}')$ be the language of P (respectively P'). Let I' be an interpretation of P'. Then $I'\dagger_{\mathcal{L}}$ means the interpretation I' restricted to the language $\mathcal{L}$, i.e., $I'\dagger_{\mathcal{L}} = (I'_T \cap (\mathcal{B}_P \cup \neg\mathcal{B}_P)) \cup \sim(I'_F \cap (\mathcal{B}_P \cup \neg\mathcal{B}_P))$. In the following we will see that $\mathcal{L}'$ extends $\mathcal{L}$, so

[2]Note that a supported interpretation is a supported model.

that this definition makes sense. Let $Sem(P)$ ($Sem(P')$) be the set of interpretations of a specific semantics SEM. For example, SEM could be the coherent semantics. We define $Sem(P')\dagger_{\mathcal{L}}$ to be the set $\{I' \dagger_{\mathcal{L}} \mid I' \in Sem(P')\}$.

Definition 5.1 (Equivalence) Extended Logic Program P and Simple Extended Logic Program P' are semantically equivalent with respect to semantics SEM iff $Sem(P) = Sem(P')\dagger_{\mathcal{L}}$. We define ELP, the class of Extended Logic Programs, to be semantically equivalent to SELP, the class of Simple Extended Logic Programs, with respect to semantics SEM iff for every Extended Logic Program P there is a Simple Extended Logic Program P' such that P and P' are semantically equivalent with respect to semantics SEM.

Theorem 5.2 [3] *ELP and SELP are semantically equivalent with respect to any semantics that for the class SELP satisfies $\neg$-supportedness.*

We prove this theorem by first transforming Extended Logic Programs into Simple Extended Logic Programs.

Definition 5.3 Let P be an Extended Logic Program, then
$S : ELP \mapsto SELP$ is the operator that transforms P into the Simple Extended Logic Program $S(P)$, by means of the following inductive definition.

$$
\begin{aligned}
S(L) &= \begin{cases} non_A & \text{if } L = \neg A \\ \sim non_A & \text{if } L = \sim\neg A \\ L & \text{otherwise} \end{cases} \\
S(body) &= \{S(L) \mid L \in body\} \\
S(r) &= S(hd(r)) \leftarrow S(body(r)) \\
S(P) &= \bigcup_{r \in P} S(r) \cup \{\neg A \leftarrow non_A, \neg non_A \leftarrow A \mid \text{there is a rule } r \in P\text{: } \neg A \in body(r)\}
\end{aligned}
$$

Note that the new atomic symbols non_A are related to the atoms A by way of the Opposition relation. Next we transform interpretations of Extended Logic Programs P into interpretations of the corresponding Simple Extended Logic Programs $S(P)$. We overload S to transform interpretations of P into interpretations of $S(P)$.

Definition 5.4 Let I be an interpretation of Extended Logic Program P and let I' be an interpretation of $S(P)$. Let $\mathcal{L}$ be the language of P. Then $S(I)$ and its inverse $S^{inv}(I')$ are the interpretations defined by:

$S(I)(non_A) = I(\neg A)$ where non_A is the new atom for $\neg A$ created in $S(P)$.

$S(I)(\neg non_A) = I(A)$ where non_A is the new atom for $\neg A$ created in $S(P)$.

$S(I)(C) = I(C)$ if $C \in \mathcal{B}_P \cup \neg\mathcal{B}_P$.

$S^{inv}(I') = I'\dagger_{\mathcal{L}}$.

In words, $S(I)$ is I extended with an interpretation for the new atomic symbols. And $S^{inv}(I')$ is I' restricted to the language $\mathcal{L}$ of P. Note that $S(I)$ is always a $\neg$-supported interpretation even if I itself is not.

Example 5.5 The program P' of Example 1.1 is the result of applying S to the program P of the same example. Furthermore,

$$M = \{Hates(Mary,John),\ Nasty\_Person(John),\ \neg Nice\_to(John,Mary),\ \sim\neg Hates(Mary,John),\\ \sim\neg Nasty\_Person(John),\ \sim Nice\_to(John,Mary)\}$$

is a coherent model of P. Operator S applied to M is

$$S(M) = M \cup \{non_{\text{Nice\_to(John,Mary)}},\ \sim\neg non_{\text{Nice\_to(John,Mary)}}\},$$

which is a coherent $\neg$-supported model of P'. $\triangle$

The exact semantical equivalence between the classes SELP and ELP requires a one-to-one correspondence between the sets of interpretations of these classes. In the following we prove that S and S^{inv} respect the $\leq_t$- and the $\leq_k$-ordering between interpretations. Let $\mathcal{L}'$ be the language $\mathcal{L}$ extended with the necessary new atomic symbols non_A of the operator S for programs.

Proposition 5.6 *For any two interpretations I and I' of $\mathcal{L}$:*

$$I \leq_t I' \ \Rightarrow\ S(I) \leq_t S(I')$$
$$I \leq_k I' \ \Rightarrow\ S(I) \leq_k S(I')$$

Proof Let $\leq$ be one of the orderings $\leq_t$ and $\leq_k$. Then for interpretations M and M' of language $\mathcal{L}'$ we have: $M \leq M'$ iff $M(C) \leq M'(C)$ for every classical literal C of language $\mathcal{L}'$. Suppose $I \leq I'$. We prove $S(I) \leq S(I')$ by considering the literals of $\mathcal{L}'$. For every classical literal of $\mathcal{L}$ we have: $S(I)(C) = I(C) \leq I'(C) = S(I')(C)$. For the new atomic symbols non_A we have: $S(I)(non_A) = I(\neg A) \leq I'(\neg A) = S(I')(non_A)$ and $S(I)(\neg non_A) = I(A) \leq I'(A) = S(I')(\neg non_A)$. $\square$

Proposition 5.7 *For any two interpretations I and I' of $\mathcal{L}'$:*

$$I \leq_t I' \ \Rightarrow\ S^{inv}(I) \leq_t S^{inv}(I')$$
$$I \leq_k I' \ \Rightarrow\ S^{inv}(I) \leq_k S^{inv}(I')$$

Proof Let $\leq$ be one of the orderings $\leq_t$ and $\leq_k$. Then for interpretations I and I' of language $\mathcal{L}$ we have: $I \leq I'$ iff $I(C) \leq I'(C)$ for every classical literal C of language $\mathcal{L}$. Suppose $I \leq I'$. We prove $S^{inv}(I) \leq S^{inv}(I')$ by considering the literals of $\mathcal{L}$. For every classical literal C of $\mathcal{L}$ we have: $S^{inv}(I)(C) = I(C) \leq I'(C) = S^{inv}(I')(C)$. $\square$

We continue by proving that S and S^{inv} preserve $\neg$-supportedness. In the simplifying transformation S, a new atom non_A is invented to stand for $\neg A$. However, in Extended Logic Programming it is impossible to force non_A and $\neg A$ to have the same truth-value in every interpretation. It is therefore impossible to define a one-to-one mapping between the interpretations of $S(P)$ and P. However, if we only consider interpretations in which the explicitly negated atoms are supported, then A and $\neg non_A$, and $\neg A$ and non_A must have the same truth-value. We prove this in the following two propositions. For the first we do not need $\neg$-supportedness, for the second we do.

Proposition 5.8 *For any interpretation I of $\mathcal{L}$: $I = S^{inv}(S(I))$.*

Proof By definition of S^{inv} for interpretations, $S^{inv}(S(I)) = S(I)\dagger_{\mathcal{L}} = (S(I)_T \cap (\mathcal{B}_{\mathcal{L}} \cup \neg\mathcal{B}_{\mathcal{L}})) \cup \sim(S(I)_F \cap (\mathcal{B}_{\mathcal{L}} \cup \neg\mathcal{B}_{\mathcal{L}}))$. By defintion of S for interpretations the last is equal to $((I_T \cup \{non_A \mid \neg A \in I_T\}) \cap (\mathcal{B}_{\mathcal{L}} \cup \neg\mathcal{B}_{\mathcal{L}})) \cup \sim((I_F \cup \{non_A \mid \neg A \in I_F\}) \cap (\mathcal{B}_{\mathcal{L}} \cup \neg\mathcal{B}_{\mathcal{L}})) = I_T \cup \sim I_F$. □

Proposition 5.9 *Let P be an Extended Logic Program. For any $\neg$-supported interpretation I' of $S(P)$: $I' = S(S^{inv}(I'))$.*

Proof By definition of S and S^{inv} for interpretations, it is clear that for every classical literal C of $\mathcal{L}$: $I'(C) = S(S^{inv}(I'))(C)$. Furthermore, the $\neg$-supportedness of I' implies that $I'(\neg A) = I'(non_A)$ and $I'(\neg non_A) = I'(A)$. Combining the above and using the definitions of S and S^{inv}, we get for all new atoms non_A: $S(S^{inv}(I'))(non_A) = S^{inv}(I')(\neg A) = I'(\neg A) = I'(non_A)$. □

To complete the proof of Theorem 5.2 we see that for every Extended Logic Program P their is a Simple Extended Logic Program $S(P)$ such that $I'\dagger_{\mathcal{L}}$ is an interpretation of P for every $\neg$-supported interpretation I' of $S(P)$. And for every interpretation I of P, $S(I)$ is an interpretation of $S(P)$. So that $Int(P) = \neg - Int(S(P))\dagger_{\mathcal{L}}$. Since S and S^{inv} provide a one-to-one correspondence between the sets $Int(P)$ and $\neg - Int(S(P))\dagger_{\mathcal{L}}$, P and $S(P)$ are semantically equivalent for any semantics that satisfies $\neg$-supportedness for $S(P)$. Hence, SELP and ELP are semantically equivalent for any semantics that satisfies $\neg$-supportedness for SELP. Since most interesting semantics are a subset of the supported semantics, and since in [3] it has also been proven that the transformations S and S^{inv} preserve wellfoundedness and stability, the class of the Simple Extended Logic Programs is an important subclass of the Extended Logic Programs.

References

[1] K.R. Apt and R.N. Bol. Logic programming and negation: a survey. Technical Report Report CS-R9402, January 1994, 1994.

[2] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[3] C. M. Jonker. Constraints and Negations in Logic Programming. Quæstiones Infinitæ vol. 10, Dissertation, Dept. of Philosophy, Utrecht University, 1994.

[4] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987. Second edition.

[5] L.M. Pereira and J.J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *Proceedings 10th European Conference on Artificial Intelligence, ECAI'92*, pages 102–106, 1992.

Reasoning about Partially Ordered Events in the Event Calculus*

Luca Chittaro*, Angelo Montanari*+ , Alessandro Provetti−

*Dipartimento di Matematica e Informatica
Università di Udine
Via Zanon, 6 - 33100 Udine, Italy

+Institute for Logic Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24 - 1018 TV Amsterdam, The Netherlands

−CIRFID *"H. Kelsen"*
Università degli Studi di Bologna
Via Galliera, 3/a - 40121 Bologna, Italy

Abstract

The paper deals with the problem of reasoning about partially ordered events in Kowalski and Sergot's Event Calculus (EC). We home in on EC between its skeptical (SKEC) and credulous (CREC) variants. When only partial knowledge about event ordering is given SKEC and CREC compute the set of maximal validity intervals (MVIs) over which the considered properties are necessarily and possibly true, respectively. The results obtainable with these two variants are amenable of a rather intuitive modal logic interpretation: SKEC computes all MVIs that will be true in whatever completion of the given partial ordering, while CREC derives those MVIs that are true in at least one completion of the given ordering. Finally, the practical relevance of the calculi is shown on a diagnostic case study.

1 Introduction

This paper studies the application of Sergot and Kowalski's Event Calculus [7, 11] (hereinafter EC) to situations where information about the ordering of events is incomplete [1, 12]. For example, complete ordering information can be impossible to acquire (this is often the case in diagnosis) or it can arrive asynchronously with respect to the recording of event occurrences (this is the case of management of database updates).

EC is a formalism for reasoning about actions and time in a logic programming framework. In EC, the occurrences of events affect the state of the world by starting or terminating intervals of validity of properties holding in the world. Given a set of event occurrences, EC allows to derive the maximal time intervals (MVIs hereinafter) over which properties hold. However, when only partial knowledge about events' ordering is given and events are therefore not completely ordered, EC is neither able to derive all possible MVIs nor to distinguish which of the derived intervals are defeasible and which are not. On the contrary, a typical human reasoner who is told a narrative where the ordering of events is not completely known is able to distinguish among conclusions which can be certainly or possibly drawn. Moreover, he/she is able to change his/her beliefs when provided with new information increasing the ordering. In order to produce comparable results, we developed two variants of EC, namely the skeptical EC and the credulous EC. In the presence of a partially ordered

*A short version of this paper has been published in the Proceedings of ECAI'94.

sequence of events, skeptical and credulous EC derive which intervals are necessarily and possibly true, respectively.

The paper is organized as follows. In Section 2 we introduce the main features of plain EC. In Section 3 we define its skeptical and credulous variants, and compare them with the original calculus. In Section 4 we provide both calculi with a modal logic interpretation, and show how they can be used to express more complex conditions on MVIs involving logical connectives and/or temporal constraints. Finally, in Section 5 the proposed calculi are applied to a diagnostic case study concerning temporally distributed information about device behaviour.

2 The plain Event Calculus

EC proposes a general approach to represent and reason about events and their effects in a logic programming framework. It takes the notions of event, property, time-point and time-interval as primitives and defines a model of change in which *events* happen at *time-points* and initiate and/or terminate *time-intervals* over which some *property* holds. EC also embodies a notion of *default persistence* according to which properties are assumed to persist until an event occurs that interrupts them. Formally, we represent an event occurrence by means of the *happens* predicate and we specify the type of event that occurred by means of the *type* predicate:

```
happens(event).                    type(event,type).
```

A time-point can be attached to an event, univocally identifying an event occurrence provided that two events of the same type can not simultaneously happen. In this work, we will focus our attention on situations where precise date information for event occurrences is not available. The relation between events and properties is defined by means of *initiates* and *terminates* predicates which express the effects of events on properties:

```
initiates(event1,property1):-          terminates(event2,property2):-
  type(event1,type1).                    type(event2,type2).
```

This *initiates* (*terminates*) predicate states that each instance of *event1* (*event2*) initiates (terminates) a period of time during which *property1* (*property2*) holds, respectively.

The plain EC model of time and change is defined by means of a set of axioms. The first axiom we introduce is *holds*. It allows us to state that a property P holds maximally between events *Ei* and *Et* if *Ei* initiates *P* and occurs before *Et* that terminates *P*, provided there is no known interruption in between:

```
holds(period(Ei,P,Et)):-
  happens(Ei), initiates(Ei,P), happens(Et),
  terminates(Et,P), before(Ei,Et), \+ broken(Ei,P,Et).
```

The negation involving the *broken* predicate is interpreted using negation-as-failure. This means that properties are assumed to hold uninterrupted over an interval of time on the basis of failure to determine an interrupting event. Should we later record an initiating or terminating event within this interval, we can no longer conclude that the property holds over the interval. This gives us the non-monotonic feature of the calculus that deals with default persistence. The predicate *broken* is defined as follows:

```
broken(Ei,P,Et):-
  happens(E), before(Ei,E), before(E,Et),
  (initiates(E,P1);terminates(E,P1)), (exclusive(P,P1);P=P1).
```

This axiom states that a given property P ceases to hold if there is an event E that happens between Ei and Et and initiates or terminates a property $P1$ that is exclusive with P. The *exclusive(P,P1)* predicate [7] has been introduced as a constraint to force the derivation of P to fail when it is possible to conclude that P1 holds at the same time, and viceversa. Finally, the condition $P = P1$ constrains interferences due to incomplete sequences of events relating to the same property. It indeed guarantees that the axiom *broken* succeeds also when an initiating or terminating event for property P is found between the pair of events Ei and Et starting and initiating P respectively.

These axioms constitute the kernel of basic (typed) EC. They provide a simple and effective tool to reason about events and their effects, but have a limited expressive power. Elsewhere, we have shown how to extend it with time granularity, context-dependency and discrete processes [3, 4, 9]. In this work, we consider the management of incomplete ordering information. To this purpose we introduce the possibility of providing factual knowledge about the relative ordering of events, expressed with the predicate *beforeFact*:

```
beforeFact(event1,event2).
```

The predicate *before* used in *holds* and *broken* is defined as the transitive closure of *beforeFact*:

```
before(E1,E2):-
  beforeFact(E1,E2).
```

```
before(E1,E2):-
  beforeFact(E1,E3), before(E3,E2).
```

3 Managing the temporal ordering of events

Database updates in EC provide information about the happening of events and their occurrence times, and are of additive nature only [8]. Since EC computes MVIs by applying a default persistence rule, an upgrading of its knowledge about events may result in some MVIs no longer derivable. Therefore, the computed set of MVIs, i.e. the collection of $Holds(period(ei, f, et))$ queries that succeeds against EC, changes cardinality and composition in response to updates.

In the general case, information about event happenings is entered in the database together with information about their occurrence times. In this work we focus on the special case in which the set of event happenings has been fixed once and for all, and the updates only regard the ordering of events. In such case the input process consists of the addition of ordering pieces of information in the form of $beforeFact$ facts. In [1] we introduced a monotonic version of EC such that the set of MVIs it computes monotonically increases with respect to updates. In this work also the opposite view is taken: a variant of EC is defined which derives MVIs any time there is no evidence that the temporal ordering makes them unviable, i.e. when the terminating event precedes that initiating. The set of MVIs it computes monotonically decreases with respect to updates.

The results obtainable with these two variants of EC are amenable of a rather intuitive modal interpretation. The first variant derives MVIs that will be true in whatever final realization of the ordering, while the second one derives those that may be true at least in one possible developement of the ordering. In [5] we provided both variants with a modal logic interpretation that allows us to formally characterize the state of knowledge about event ordering and its update as well as queries about maximal validity intervals. Furthermore, we discussed an example of their application to a diagnostic case study concerning temporally distributed information about device behaviour.

3.1 Partially specified orderings on events

Let us consider a finite set E of n events and assume that all the events happened at different times. We look at the input of ordering pieces as the process of putting new constraints that narrow the specification of the actual event ordering, thus reducing the number of admissible total orderings that can be consistently foreseen. Let us call each of these admissible orderings an *extension* of the given partial specification. As the ordering specification becomes complete, the "surviving extensions" reduce to just one.

Let $Ord(E)$ be the set of all possible total orderings over E. $Ord(E)$ contains $n!$ elements, each one consisting of $n \cdot (n-1)/2$ mutually consistent ordering pieces. Any set of ordering pieces with less than $n \cdot (n-1)/2$ elements thus provides only a partial specification of the actual ordering, and it univocally identifies the subset of $Ord(E)$ consisting of all its admissible extensions. On the basis of the correspondence between ordering pieces and $Before$ statements, we can equivalently define as *partial specification* the transitive closure of a set of $Before$ statements. The input process can therefore be seen as a sequence of partial specifications $\prec_0 = \emptyset, \prec_1, \prec_2, \prec_3, \ldots, \prec_m$, such that:

$$\prec_{i+1} = (\prec_i \cup \{Before(e_j, e_k)\})^* \; and \; \forall i \; \prec_i \; is \; consistent$$

where $()^*$ denotes the transitive closure and $\prec_i$ is consistent if and only if the ordering pieces belonging to it are mutually consistent. Of course, for each i, the set of admissible extensions of $\prec_{i+1}$ is a subset of those of $\prec_i$. Moreover, it is easy to prove that, for all i, the admissible extensions are less than or equal to $\lceil n!/2^i \rceil$.

3.2 The skeptical Event Calculus

We nickname *skeptical Event Calculus* (SKEC) a weaker variant of EC that forces the monotonicity of the calculus. It implements a sort of absolute persistence so as to exclude the possibility of deriving information that could be later retracted, provided that the given set of event occurrences does not change. The idea is to transform the definition of *Holds* so that *Holds(period(ei,r,et))* succeeds if and only if it is possible to conclude that no event affecting r may have occurred after *ei* and before *et*. In such a way computed MVIs are indefeasible with respect to refinements of the ordering specification: new ordering pieces coming in may result in new MVIs being derived, but every old MVI is still available. Thus, the set of MVIs computed by SKEC monotonically increases.

SKEC replaces predicates *holds* and *broken* of EC with predicates *skeHolds* and *skeBroken*, respectively, which are defined by the following axioms:

```
skeHolds(period(Ei,R,Et)):-
  happens(Ei),
  initiates(Ei,R),
  happens(Et),
  terminates(Et,R),
  before(Ei,Et),
  \+ skeBroken(Ei,R,Et).
```

```
skeBroken(Ei,R,Et):-
  happens(E),
  E \== Ei, E \== Et,
  \+ before(E,Ei),
  \+ before(Et,E),
  (initiates(E,R1);terminates(E,R1)),
  exclusive(R,R1).
```

The relationship between a partially specified ordering $\prec_i$ and the set of MVIs computed by SKEC given the corresponding ordering pieces as input is expressed by the following proposition:

Proposition 3.1
For any given set of events $E = \{e1, e2, \ldots, en\}$, let $H(E)$ be the corresponding set of *happens* and *type* definitions $\{happens(e1), type(e1,t1), happens(e2), type(e2,t2), \ldots\}$ and let $\prec_i$ be a set of mutually consistent *before* elements. Furthemore, let $SKEC \cup Hi \cup \prec_i$ the database obtained by the union of $SKEC$, Hi and $\prec_i$.
For any pair of events ei, ej belonging to E and any relationship r:
$SKEC \cup Hi \cup \prec_i \vdash holds(period(ei,r,ej))$ if and only if
$period(ei,r,ej)$ holds with respect to every admissible extension of $\prec_i$.

Furthermore, it is quite straightforward to prove that every interval derived by SKEC is also derived by EC. This result is formally stated by the following theorem:

Theorem 3.2
For any pair of events ei, ej belonging to E and any relationship r:
if $SKEC \cup Hi \cup \prec_i \vdash holds(period(ei,r,ej))$ then $EC \cup Hi \cup \prec_i \vdash holds(period(ei,r,ej))$.

The proof is given in [1]. It is easy to see that the opposite implication does not hold in general.

3.3 The credulous Event Calculus

The *credulous Event Calculus* (CREC) is a stronger variant of EC that stresses its nonmonotonism. Whenever it is not possible to derive that a terminating event *et* precedes an initiating event *ei*, CREC assumes that *ei* precedes *et*. Such an assumption allows CREC to compute all MVIs which are not incompatible with a given set of partially ordered events. If no information about event ordering is given, CREC computes every MVI that holds with respect to at least one possible total ordering of events. When new information about event ordering is added, this set of computed MVIs monotonically decreases. Further constraining the ordering of events may indeed invalidate previously computed MVIs, but it never forces CREC to compute new MVIs. The input process can thus be

viewed as a way of progressively selecting the subset of MVIs derivable from a totally ordered set of events from the initial set of all possible MVIs.

The axioms of CREC are the same of EC but the replacement of $before(Ei, Et)$ with the negation of $before(Et, Ei)$ in the definition of $Holds$. The resulting predicate $creHolds$ is defined as follow:

```
creHolds(period(Ei,R,Et)):-
  happens(Ei), initiates(Ei,R),
  happens(Et), terminates(Et,R),
  \+ before(Et,Ei), \+ broken(Ei,R,Et).
```

The relationship between a partially specified ordering $\prec_i$ and the set of MVIs computed by CREC given the corresponding ordering pieces as input is expressed by the following proposition:

Proposition 3.3
For any pair of events ei, ej belonging to E and any relationship r:
$CREC \cup Hi \cup \prec_i \vdash holds(period(ei, r, ej))$ if and only if
$period(ei, r, ej)$ holds with respect to at least one admissible extension of $\prec_i$.

Furthermore, it is straightforward to prove that every MVIs derived by EC is also derived by CREC. This result is formally stated by the following theorem:

Theorem 3.4
For any pair of events ei, ej belonging to E and any relationship r:
if $EC \cup Hi \cup \prec_i \vdash holds(period(ei, r, ej))$ then $CREC \cup Hi \cup \prec_i \vdash holds(period(ei, r, ej))$.

The proof is similar to the proof of *Theorem 3.2*. Moreover, it is easy to see that the opposite implication does not hold in general.

Putting together *Theorems 3.2* and *3.4* we obtain the following expression for the relations between the sets of MVIs computed by EC, SKEC and CREC:

$$MVI_{SKEC} \subseteq MVI_{EC} \subseteq MVI_{CREC}$$

where the three sets of MVIs coincide if and only if the ordering of events is completely specified.

3.4 A comparison of the calculi

Let us analyze and compare the behaviour of the three calculi with respect to a sequence of updates which progressively constrain the ordering of a given set of events. Suppose that the database includes the following domain description (for the sake of simplicity, we define domain axioms at the instance level, rather than at the type one):

$initiates(e1, r), terminates(e2, r), initiates(e3, p), terminates(e4, p),$
$initiates(e5, r), terminates(e6, r), exclusive(r, p), exclusive(p, r)$

together with the following set of event occurrences:

$$Hi = \{happens(e1), happens(e2), happens(e3), happens(e4), happens(e5), happens(e6)\}$$

We analyze the behaviour of the different calculi when the following sequence of ordering pieces is entered in the database:

$\sigma_1 = \{beforeFact(e1, e4)\}, \sigma_2 = \{beforeFact(e1, e6)\}, \sigma_3 = \{beforeFact(e2, e4)\},$
$\sigma_4 = \{beforeFact(e1, e2)\}, \sigma_5 = \{beforeFact(e3, e4)\}, \sigma_6 = \{beforeFact(e4, e5)\},$
$\sigma_7 = \{beforeFact(e2, e3)\}, \sigma_8 = \{beforeFact(e2, e6)\}, \sigma_9 = \{beforeFact(e5, e6)\}$

The input process gives rise to a sequence of partial ordering specifications $\prec_0 = \emptyset, \prec_1 = (\prec_0 \cup \sigma_1)^*, \prec_2 = (\prec_1 \cup \sigma_2)^*, \ldots$ that refines the original empty specification $\prec_0$ into the final complete specification $\prec_9$ according to which event ei precedes event $e(i+1)$, for $i = 1, ..5$.

The MVIs computed by the three calculi at each step of the input process are reported in the table below, where, for each pair of events ei, et, $r(ei, et)$ and $p(ei, et)$ stand for $period(ei, r, et)$ and

period(ei, p, et), respectively. As expected, skeptical and credulous EC compute a monotonically increasing and decreasing set of MVIs, respectively, while the set of MVIs computed by plain EC sometimes increases (e.g. when σ_2 is added) and sometimes decreases (e.g. when σ_8 is added). When the ordering of events is completely specified, the three calculi compute the same set of MVIs. When the ordering is only partially specified, plain EC neither derives all necessary MVIs nor derives all possible ones. Furthermore, it does not distinguish between defeasible and indefeasible MVIs. These drawbacks are overcome by SKEC and CREC. Given a set of ordering pieces, SKEC derives the set of all and only indefeasible MVIs, and as soon as new ordering pieces are entered it determines new indefeasible MVIs (if any), while CREC derives the set of all defeasible and indefeasible MVIs, and as soon as new ordering pieces are entered it automatically withdraws defeasible MVIs which are no longer derivable (if any). Finally, the set of defeasible MVIs can be determined by subtracting the set of MVIs computed by SKEC from the set of MVIs computed by CREC.

	holds(X)	*skeHolds*(X)	*creHolds*(X)
$\prec_0$	$\emptyset$	$\emptyset$	$X = r(e1,e2); r(e1,e6);$ $p(e3,e4); r(e5,e2); r(e5,e6)$
$\prec_1$	$\emptyset$	$\emptyset$	$X = r(e1,e2); r(e1,e6);$ $p(e3,e4); r(e5,e2); r(e5,e6)$
$\prec_2$	$X = r(e1,e6)$	$\emptyset$	$X = r(e1,e2); r(e1,e6);$ $p(e3,e4); r(e5,e2); r(e5,e6)$
...	...	...	...
$\prec_7$	$X = r(e1,e2);$ $r(e1,e6); p(e3,e4)$	$\emptyset$	$X = r(e1,e2); r(e1,e6);$ $p(e3,e4); r(e5,e6)$
$\prec_8$	$X = r(e1,e2);$ $p(e3,e4)$	$X = r(e1,e2)$	$X = r(e1,e2);$ $p(e3,e4); r(e5,e6)$
$\prec_9$	$X = r(e1,e2);$ $p(e3,e4); r(e5,e6)$	$X = r(e1,e2);$ $p(e3,e4); r(e5,e6)$	$X = r(e1,e2);$ $p(e3,e4); r(e5,e6)$

4 A modal logic intepretation of SKEC and CREC

In this section, SKEC and CREC are provided with a modal logic interpretation that allows us to formally characterize the state of knowledge about event ordering and its update as well as queries about maximal validity intervals. Database states and updates are modeled in terms of worlds and accessibility relation, respectively, and SKEC and CREC computations are interpreted as modal queries executions.

Since each set of mutually consistent ordering pieces $\prec_i$ can be interpreted as a partially specified ordering which univocally identifies a set of admissible extensions *Est*, it is possible to provide both SKEC and CREC with a Kripke-like semantics over sets of admissible extensions, and then to support modal queries about what is true with respect to all admissible extensions (necessity) and what is true with respect to at least one admissible extension (possibility). Formally, a modal logic interpretation of SKEC is a tuple $((\mathcal{W}, \mathcal{R}), \Im)$ where $\mathcal{W}$ is a non-empty finite set of worlds, $\mathcal{R}$ is a binary accessibility relation between worlds, and $\Im$ is the interpretation function.

Each world $w \in \mathcal{W}$ denotes a set of admissible extensions corresponding to a set of mutually consistent ordering pieces. This provides us with a twofold characterization of $\mathcal{W}$ in terms of both the set of sets of admissible extensions $Ext_{\mathcal{W}}$ and the set of sets of mutually consistent ordering pieces $\prec_{\mathcal{W}}$. The accessibility relation $\mathcal{R}$ is defined in such a way that a world w_2, denoting a set Ext_{w_2} of admissible extensions corresponding to a set $\prec_{w_2}$ of ordering pieces, can be accessed from a world w_1,

characterized by Ext_{w_1} and $\prec_{w_1}$, if and only if there exists a possibly empty set of ordering pieces o such that $\prec_{w_2} = (\prec_{w_1} \cup \{o\})^*$ and there exist no sets o' such that $o' \subset o$ and $\prec_{w_2} = (\prec_{w_1} \cup \{o'\})^*$. It can be shown that:

$$\forall w_1, w_2 (if\ \mathcal{R}(w_1, w_2)\ then\ (\prec_{w_1} \subseteq \prec_{w_2}\ \ and\ Ext_{w_2} \subseteq Ext_{w_1}))$$

as well as

$$\forall ext_w \in Ext_{w_1} \exists w_2 (\mathcal{R}(w_1, w_2)\ and\ e_w \in Ext_{w_2})$$

It is straightforward to show that $\mathcal{R}$ is a reflexive, antisymmetric, and transitive ordering relation over $\mathcal{W}$. Finally, the interpretation function states that a MVI holds in a given world w if and only it holds over all the admissible extensions it denotes. The modal logic interpretation of CREC is equal to the one of SKEC but the replacement of the interpretation function. In the case of CREC, the interpretation function states that a MVI holds in a given world w if and only it holds over at least one of the admissible extensions it denotes.

The satisfiability relation is defined as usual [6]. Furthermore, from the given definition of $\mathcal{R}$, it follows that, for each world w_1, if a given MVI holds in every w_2 such that $\mathcal{R}(w_1, w_2)$, then it holds in w_1. This implies that, for each world w_1, if a given MVI holds in w_1 then there exists at least one world w_2 such that $\mathcal{R}(w_1, w_2)$ and the given MVI holds in w_2. Moreover, as far as we only consider atomic formulae of the type $holds(period(ei, r, et))$ the opposite implication holds too. It states that, for each world w_1, if a given MVI holds in at least one world w_2 such that $\mathcal{R}(w_1, w_2)$ then it holds in w_1.

Given the correspondences established by *Propositions 3.1* and *3.3*, necessity and possibility over sets of admissible extension can then be implemented by means of SKEC and CREC, respectively.

```
holdsL(period(Ei,P,Et)):-
  skeHolds(period(Ei,P,Et)).

holdsM(period(Ei,P,Et)):-
  creHolds(period(Ei,P,Et)).
```

Besides proving that a single MVI holds with respect to one/all admissible extensions, we are interested in proving that the conjunction or the disjunction of a given set of MVIs holds as well as in showing that a given MVI does not hold. This can be done by modeling logical connectives as functors operating on MVIs. For example, in order to prove that the conjunction (functor and) of two MVIs is necessary, we simply ask that the two MVIs necessarily hold:

```
holdsL(period(Ei1,P1,Et1) and period(Ei2,P2,Et2)):-
  holdsL(period(Ei1,P1,Et1)), holdsL(period(Ei2,P2,Et2)).
```

Proving that the conjunction possibly holds is more complex and requires to perform a sort of update simulation. More precisely, after proving that the first MVI possibly holds between events $Ei1$ and $Et1$, the query concerning the second MVI has to be evaluated assuming that $beforeFact(Ei1, Et1)$ holds. This is implemented by introducing new versions of predicates *before*, *creHolds* and *broken* which accept also a list of assumptions about event ordering, besides using the facts in the database. This new predicates are called *hypBefore*, *hypHoldsM* and *hypBroken* and their definition is given below. In case the two MVIs refer to exclusive properties it is also necessary to guarantee that the two MVIs (which are possible in isolation) do not intersect. This is implemented by the *hypIncompatible* and *hypIntersect* predicates.

```
holdsM(period(Ei1,P1,Et1) and period(Ei2,P2,Et2)):-
  holdsM(period(Ei1,P1,Et1)),
  hypHoldsM(period(Ei2,P2,Et2),[beforeFact(Ei1,Et1)]),
  \+ hypIncompatible(period(Ei1,P1,Et1),period(Ei2,P2,Et2),
                     [beforeFact(Ei1,Et1),beforeFact(Ei2,Et2)]).

hypHoldsM(period(Ei,R,Et),Hypotheses):-
  happens(Ei), initiates(Ei,R),
```

```
  happens(Et), terminates(Et,R),
  \+ hypBefore(Et,Ei,Hypotheses),
  \+ hypBroken(Ei,R,Et,Hypotheses).

hypIncompatible(period(Ei1,P1,Et1),period(Ei2,P2,Et2),Hypotheses):-
  exclusive(P1,P2),
  hypIntersect([Ei1,Et1],[Ei2,Et2],Hypotheses).

hypBroken(Ei,R,Et,Hypotheses):-
  happens(E), hypBefore(Ei,E,Hypotheses), hypBefore(E,Et,Hypotheses),
  (initiates(E,R1);terminates(E,R1)), exclusive(R,R1).

hypIntersect([Ei1,Et2],[Ei2,Et2],Hypotheses):-
   hypBefore(Ei1,Et2,Hypotheses), hypBefore(Ei2,Et1,Hypotheses).

hypIntersect([Ei1,Et1],[Ei2,Et2],Hypotheses):-
   hypBefore(Ei2,Et1,Hypotheses), hypBefore(Ei1,Et2,Hypotheses).

hypBefore(E1,E2,Hypotheses):-
  (beforeFact(E1,E2);member(beforeFact(E1,E2),Hypotheses)).

hypBefore(E1,E2,Hypotheses):-
 (beforeFact(E1,E3);member(beforeFact(E1,E3),Hypotheses)),
  hypBefore(E3,E2,Hypotheses).
```

Further functors can be added to model temporal constraints between MVIs. As an example, logical conjunction only imposes that involved MVIs hold with respect to the same admissible extension ('asynchronous and'), but it does not require that they overlap ('synchronous and'). An overlap operator for holdsL can be easily introduced by requiring that the two MVIs actually intersect and the corresponding properties are not exclusive:

```
holdsL(period(Ei1,P1,Et1) overlap period(Ei2,P2,Et2)):-
   holdsL(period(Ei1,P1,Et1) and period(Ei2,P2,Et2)),
   \+ exclusive(P1,P2),
   hypIntersect([Ei1,Et1],[Ei2,Et2],[]).
```

In the case of holdsM, the requirement is weaker: it specifies that the two MVIs *may* intersect. This is done by the *hypCreIntersect* and *hypCreBefore* predicates:

```
holdsM(period(Ei1,P1,Et1) overlap period(Ei2,P2,Et2)):-
  holdsM(period(Ei1,P1,Et1)),
  hypHoldsM(period(Ei2,P2,Et2),[beforeFact(Ei1,Et1)]),
  \+ exclusive(P1,P2),
  hypCreIntersect([Ei1,Et1],[Ei2,Et2],
                  [beforeFact(Ei1,Et1),beforeFact(Ei2,Et2)]).

hypCreIntersect([Ei1,Et1],[Ei2,Et2],Hypotheses):-
   hypCreBefore(Et2,Ei1,Hypotheses), hypCreBefore(Et1,Ei2,Hypotheses).

hypCreIntersect([Ei1,Et1],[Ei2,Et2],Hypotheses):-
   hypCreBefore(Et1,Ei2,Hypotheses), hypCreBefore(Et2,Ei1,Hypotheses).

hypCreBefore(E1,E2,Hypotheses):-
  E1\==E2, \+ hypBefore(E1,E2,Hypotheses).
```

5 Application to a diagnostic case study

In this section we apply our approach to a real-world case study taken from the diagnostic domain. We focus our attention to the representation and processing of information about fault symptoms that is spread out over periods of time and for which current expert system technology is particularly deficient [10]. An example of such a dynamic behavior taken from the domain of a computerized numerical control (CNC) machining center is the following: "One possible cause for an undefined position of the tool magazine is a faulty limit switch. This cause can be ruled out if the status registers IN29 and IN30 of the CNC control system show the following behavior: at the beginning both registers contain the value 1. Then IN29 drops to 0, followed by IN30. Finally, both return to their original values in the reverse order"[10]. Figure 2 represents graphically the situation. In order to apply the rule, measurements have to be taken in the real world.

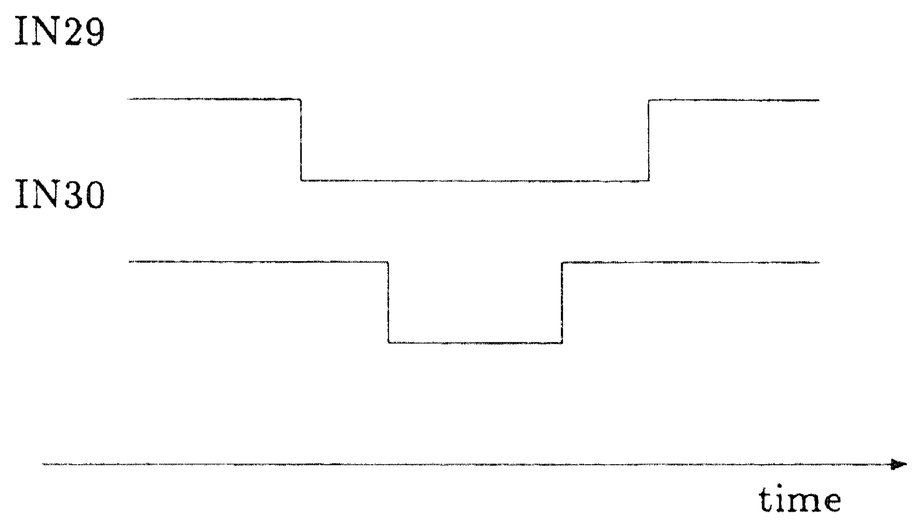


Figure 1. The expected behaviour of status registers.

However, while measurements should be taken frequently enough to guarantee that signal transitions are not lost, it is generally impossible to locate exactly the instants where a quantity assumes or loses a value. Consider, for example, figure 2, showing a set of taken measurements: from measurement m1 and m2, we can conclude that IN29 and IN30 assumed the value 1, but we are not able to conclude anything about when the two causing events e1 and e2 exactly happened (in particular, we cannot determine their relative ordering). From measurements m3 and m4 we can conclude that both IN29 and IN30 dropped to 0, and while we are not able to conclude anything about the exact location in time and the relative order of their causing events e3 and e4, we are sure that they happened after e1 and e2.

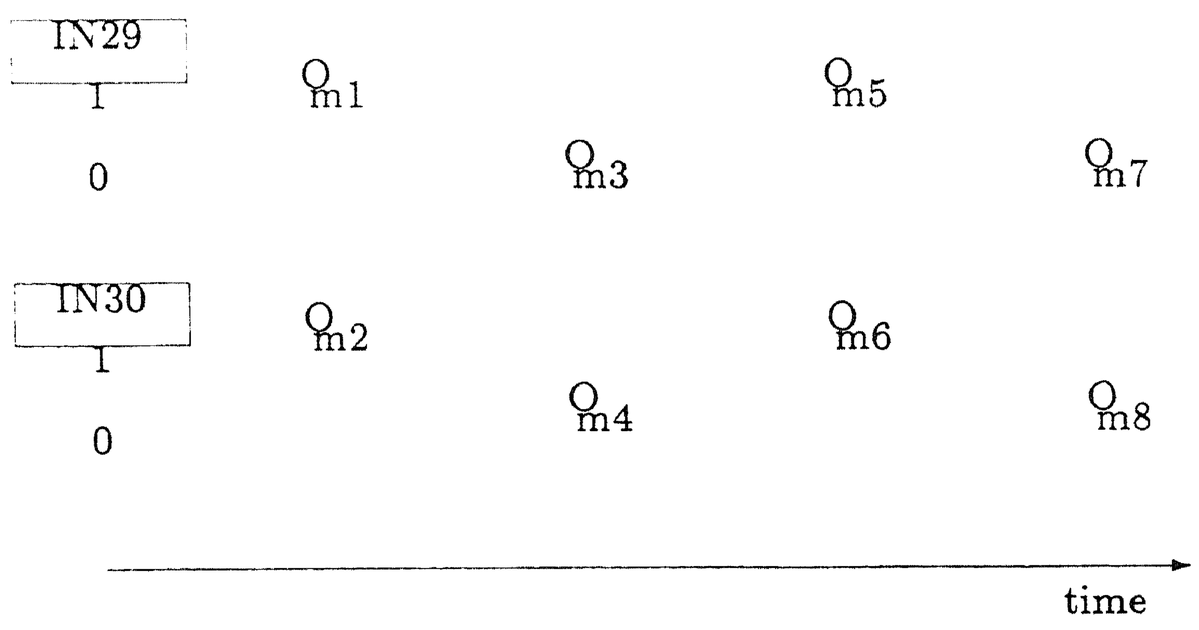


Figure 2. A possible set of measurements

Analogous considerations hold for measurements m5, m6, m7, m8 and events e5, e6, e7, e8. The situation can be formally represented as follows. The general definitions of event types and their

effects are:

```
initiates(E,value(Signal,0)):-
  type(E,zero(Signal)).

terminates(E,value(Signal,0)):-
  type(E,one(Signal)).

initiates(E,value(Signal,1)):-
  type(E,one(Signal)).

terminates(E,value(Signal,1)):-
  type(E,zero(Signal)).
```

The description of the specific situation is given by the following database (we will refer to it as DB1):

```
type(e1,one(in29)).     type(e4,zero(in30)).    type(e7,zero(in29)).
happens(e1).            happens(e4).            happens(e7).

type(e2,one(in30)).     type(e5,one(in29)).     type(e8,zero(in30)).
happens(e2).            happens(e5).            happens(e8).

type(e3,zero(in29)).    type(e6,one(in30)).
happens(e3).            happens(e6).

beforeFact(e1,e3).      beforeFact(e1,e4).      beforeFact(e2,e3).
beforeFact(e2,e4).      beforeFact(e3,e5).      beforeFact(e3,e6).
beforeFact(e4,e5).      beforeFact(e4,e6).      beforeFact(e5,e7).
beforeFact(e5,e8).      beforeFact(e6,e7).      beforeFact(e6,e8).
```

Suppose we want to know if on the basis of available knowledge, we can conclude that the limit switch is certainly faulty. Since abnormal behavior for the limit switch is given by IN29 dropping to zero after IN30 or alternatively IN29 rising to one before IN30, we can pose a query to check that this behavior certainly occurred. The following query captures the abnormal behavior by searching for an overlap between a period where IN29 contains 1 and a period where IN30 contains 0:

```
?-holdsL(period(EA,value(in29,1),EB) overlap period(EC,value(in30,0),ED)).
```

If such a query is satisfied, the limit switch is faulty. Considering the database DB1, this query fails and thus we can not conclude that the limit switch is faulty. If we use holdsM instead of holdsL the result is the following[1]:

```
?-holdsM(period(EA,value(in29,1),EB) overlap period(EC,value(in30,0),ED)).

EA = e1,  EB = e3,  EC = e4, ED = e6 ;

EA = e5,  EB = e7,  EC = e4, ED = e6
```

Therefore, although we can not be certain that the limit switch is faulty, there exist two extensions in which it is and therefore the limit switch is a component that may be faulty.

Suppose now to enrich database DB1 with an additional measurement about the ordering of e3 and e4 (we will refer to this enriched database as DB2):

```
beforeFact(e4,e3).
```

If we consider DB2 and pose again the two previous queries, while the result of holdsM does not change, holdsL now succeeds returning the assignment (EA=e1, EB=e3, EC=e4, ED=e6). Therefore, we can conclude that the limit switch is faulty.

If instead of proving that the switch is faulty, we would want to prove that it is correctly behaving, the query has to specify the entire correct temporal behavior pattern:

---

[1]It is worth noting that the simple version of EC we adopted does not allow us to model properties which hold forever from the occurrence of a given event. This confines the detection of possible faults to those concerning the ordering of events *e3*, *e4*, *e5*, and *e6*. However, it is straightforward to generalize the approach to the case of persistence in the future [4].

```
?-holdsL(period(EA,value(in29,1),EB) overlap period(EC,value(in30,1),ED)),
  holdsL(period(EB,value(in29,0),EE) overlap period(ED,value(in30,0),EF)),
  before(EB,ED), before(EF,EE),
  holdsL(period(EE,value(in29,1),EG) overlap period(EF,value(in30,1),EH)).
```

This query does not succeed considering either database DB1 or database DB2. In both situations we are therefore unable to prove that the limit switch is certainly normal. If we instead would want to prove that it is possible that the switch is normal, we can substitute holdsL with holdsM and before whith creBefore in the previous query. In this case, the query obviously fails in DB2, while it instead returns a solution for database DB1 (EA=e1, EB=e3, EC=e2, ED=e4, EE=e5, EF=e6, EG=e7, EH=e8) and thus the switch may be correctly functioning there.

Finally, if we suppose to enrich database DB1 with the additional facts **beforeFact(e3,e4)**, and **beforeFact(e6,e5)**, we become able to prove that the switch is certainly behaving correctly.

6 CONCLUSION

This paper proposed two variants of EC in order to improve it with the capability of handling modal queries about properties in the presence of incomplete ordering information. Both variants have been formally defined and implemented on a Sun Sparc2 in Quintus Prolog. While significantly extending the applicability of EC, they still suffer from its inefficiency. We addressed this problem in [2, 4].

References

[1] I. Cervesato, A. Montanari, A. Provetti, *On the Non-monotonic Behavior of Event Calculus for Deriving Maximal Time-Intervals*; *Internat. Journal of Interval Computations*, 2/1993, 83–119.

[2] I. Cervesato, L. Chittaro, A. Montanari, *What the Event Calculus actually does and how to do it efficiently*; *Proc. of GULP-PRODE Joint Conference on Declarative Programming*, Peniscola, Spain, 1994.

[3] L. Chittaro, A. Montanari, *Reasoning about Discrete Processes in a Logic Programming Framework*; Proc. of *GULP'93 - Eight Conference on Logic Programming*, Gizzeria Lido (CZ), Italy, June 1993, 407–421.

[4] L. Chittaro, A. Montanari, *Efficient Handling of Context-Dependency in the Cached Event Calculus*; Proc. of *TIME'94 - International Workshop on Temporal Representation and Reasoning*, Pensacola Beach, Florida, 1994, 103–112.

[5] L. Chittaro, A. Montanari, A. Provetti, *Skeptical and Credulous Event Calculi for Supporting Modal Queries*; Proc. of *ECAI'94 - 11th European Conference on Artificial Intelligence*, Amsterdam, The Netherlands, 1994, 361–365.

[6] G. Hughes, M. Cresswell, *A Companion to Modal Logic*; Methuen, London, 1984.

[7] R. Kowalski, M. Sergot, A Logic-based Calculus of Events; *New Generation Computing*, Vol. 4, Ohmsha Ltd and Springer Verlag, 1986, 67–95.

[8] R. Kowalski, *Database Updates in the Event Calculus*; *Journal of Logic Programming*, Vol. 12, June 1992, 121–146.

[9] A. Montanari, E. Maim, E. Ciapessoni, E. Ratto, *Dealing with time granularity in the Event Calculus*; Proc. of *FGCS'92 Conference*, Tokyo, Japan, June 1992, 702–712.

[10] K. Nokel, *Temporally distributed symptoms in Technical Diagnosis*; Springer-Verlag, 1991.

[11] M. J. Sergot, *(Some topics in) Logic Programming in AI*; Lecture notes of the GULP Advanced School on Logic Programming, Alghero, Italy, 1990.

[12] M. P. Shanahan, *Prediction is Deduction but Explanation is Abduction*; Proc. of *IJCAI'89 Conference*, Detroit, 1989. pp. 1055–1050.

Revising Incoherent and Contradictory Logic Programs

Cees Witteveen
Delft University of Technology,
Dept of Mathematics and Computer Science,
P.O.Box 356, 2600 AJ Delft, The Netherlands,
e-mail: witt@cs.tudelft.nl

Abstract

We discuss a revision method for logic programs which are classically satisfiable but do not have a stable model (incoherent programs) or do have stable models which are not acceptable (contradictory programs). We discuss the application of the method for the class of normal logic programs with constraints.

The method we propose is a revision by expansion method: if a program P is contradictory or incoherent, we expand the program, i.e., we add some rules to P in such a way that its classical meaning is not affected but the expanded program is *well-behaved*, i.e. has at least one (acceptable) stable model.

This method is a strict generalization of the well-known dependency-directed backtracking method, which could not be applied successfully to incoherent programs.

Keywords: Revision, Non-monotonic Reasoning, Logic Programming.

1 Introduction

Non-monotonic logics can be characterized by inference relations which are not based on the collection of all ordinary models of a given theory, but on a distinguished subset of this collection.
This has some consequences for a theory of belief revision applied to non-monotonic theories. For, unlike belief revision in classical monotonic theories, in non-monotonic theories we can be forced to revise our beliefs even if the theory is classically satisfiable. As a simple example, take the logic program P containing the rules $b \leftarrow\sim a$ and the constraint $\bot \leftarrow b$, the latter expressing that b cannot be true. If we base our selection upon stable models of the program,

on the basis of the single rule $b \leftarrow\sim a$ we would consider b as true, while the constraint $\bot \leftarrow b$ forbids such an interpretation: so the (unique) stable model of P is a contradictory model. The program, however, is classically satisfiable: it has an ordinary model in which a is true and b is false.

It is also possible that a program simply does not have any model meeting our selection criteria. This latter situation arises when there seems to be no *acceptable* (non-monotonic) interpretation of the program, although according to a classical reading of the theory it is satisfiable. For example, take the simple program $a \leftarrow\sim a, \sim b$. It has a classical model in which b is true and a is false, but it has no stable models at all.

Following [7, 4] we will call the first kind of programs *contradictory* and the second kind *incoherent*. For such programs the classical reading of the program might give no problems, i.e. the set of classical models for such a program might is not empty, but a non-monotonic reading of the same program would give us no *acceptable model.*

In order to find acceptable models for such programs we propose to use the set of acceptable models of a *revision* of the program. Unlike revision of (inconsistent) classical theories, however, *retraction* of some beliefs is not the only option we have, since the problem arises not because of a lack of *classical* models but a lack of *acceptable* models.

In such a situation then, a lack of acceptable models means that our model-selection criteria have to be changed. Therefore, we propose to perform revision by changing our selection criteria, i.e., we select other models of the original theory as being acceptable.

In this paper we propose a special revision method to perform such a change of selection criteria. Instead of adapting the non-monotonic semantics, we adapt the program and we will use the stable models of the adapted theory as the acceptable models of the original theory.

Syntactically, this revision method is a *theory-expansion* method: we expand the original theory by adding some extra statements in such a way that acceptable models of the expanded theory are always models of the original theory.

We can show that expanding the program does not affect the set of minimal models of the original program: the set of minimal models of the revised theory equals the set of minimal models of the original theory. This means that, unlike other expansion methods proposed in the literature, we have a guarantee that the stable models of the expanded theory are minimal models of the original theory.

2 Revising Logic Programs

In logic programming some revision methods based on theory-expansion have been proposed. Generally speaking, two different approaches can be distinguished: the first one, based on a kind of *dependency-directed backtracking*, aims at enlarging the space of stable models but at the same tries to avoid affecting the space of classical models of the original program.

This approach, originating with Doyle ([1]) and later on reconstructed as adding logical contrapositives to the original program by Giordano and Martelli ([3]), however, can be applied to *contradictory* programs only. For incoherent programs such as $a \leftarrow\!\sim a$ it does not offer much help: the contrapositive of this rule is identical to the original rule.
A further disadvantage of this approach is that, for normal logic programs, we are forced to introduce extra literals in order to represent the logical contrapositive of a rule. For example, the contrapositive of $b \leftarrow a$ is $\sim a \leftarrow\!\sim b$. Such rules, however, can be expressed in normal logic programs in an indirect way by adding extra atoms to encode the meaning of the negation in the head of a rule. This means that the models of the expanded theory are *extensions* of the models of the original program.
The second approach, called the *contradiction-removal approach* [7], originally was used to revise contradictory programs, but can be used successfully revise incoherent programs, too. Basically, a contradiction-removal approach adds simple, unary, rules (inhibition rules) of the form $a \leftarrow$ to the program for (some) atoms a occurring in the program. The effect of adding such a rule $a \leftarrow$ to the program is that a cannot assumed to be false by CWA-reasoning, hence the rules in which a occurs negatively can be ignored. It is very simple to see that such a strategy always can be applied successfully to both incoherent and contradictory normal programs + constraints: just by adding enough inhibition rules, only the *positive* rules of the program will determine its meaning. Now clearly, the subprogram consisting of these positive rules (including constraints) must have an acceptable stable model, for else the original program is classically inconsistent.
Those contradiction removal strategies which succeed by adding a minimal number of rules to the program are preferred. This means that the selection of the expansion is based on a *syntactical* criterion. It is not difficult, therefore, to come up with examples showing that a stable model of the expanded program is a non-minimal or even maximal model of the original program[1].
This seems to be a disadvantage of the approach: if a program is not well-behaved, we should change our model-selection criteria as little as possible, i.e. instead of proposing a *stable* model of the program we could propose a *minimal* model. The selection of such a minimal model then, could be based on syntactical criteria, like in the contradiction-removal strategy. Therefore, in this paper we will sketch a revision strategy based on a generalization of a dependency-directed backtracking strategy, in the sense that

- the method can be successfully applied to both contradictory and incoherent normal programs,

[1] For example, take the program P consisting of the rule $\bot \leftarrow\!\sim a_1$ and the rules $a_i \leftarrow a_{i+1}$ for $i = 1, \ldots n$. A minimal contradiction-removal strategy adding the rule $a_{n+1} \leftarrow$, would select the maximal model of the original program.

- the original program will be modified in such a way that its set of classical models will be unaffected and we select the set of stable models of the revised program. This implies that every model selected is a minimal model of the original program.

2.1 Preliminaries

We assume the reader to be familiar with some basic terminology used in logic programming, as for example in Lloyd [5]. A *normal program rule* is a directed propositional clause of the form $A \leftarrow B_1, \ldots, B_m, \sim D_1, \ldots, \sim D_n$, where $A, B_1, \ldots, D_1, \ldots, D_n$ are all atomic propositions. Often such a rule is denoted as $A \leftarrow \alpha$, where $\alpha = \alpha^+ \wedge \alpha^-$; α^+ denotes the conjunction of the B_i atoms and α^- denotes the conjunction of the negative atoms $\sim D_j$. The head of a rule r is denoted by $hd(r)$. A program is called *positive* if for every rule r, $\alpha^-(r) = \top$ (the empty conjunction). As usual, interpretations and models of a program are denoted by maximal consistent subsets of literals over the Herbrand base B_P of P. If L is a set of literals over B_P, $atoms(L)$ denotes the set of atoms $a \in B_P$ occurring positively or negatively in L and L^+ denotes the subset of atoms occurring positively in L.
Constraints are special rules of the form $\bot \leftarrow \alpha$, expressing that the conjunction of the literals occurring in α cannot be true. We do not allow $\bot$ (*falsum*) to occur in the antecedent $\alpha(r)$ of any rule r. T_P denotes the immediate consequence operator associated with the program P and $lfp T_P = T_P \uparrow \omega$ denotes the least fixpoint of this operator.
The intended meaning of a normal logic program is given by the two-valued *stable model* semantics ([2]):

Definition 2.1 *A model M of a logic program P is called stable iff M^+ equals the least fixpoint of the Gelfond-Lifschitz reduction $G(P, M)$ of P, where*

$$G(P, M) = \{c \leftarrow \alpha^+ \mid c \leftarrow \alpha \in P,\ M \models \alpha^-\}.$$

Note that $G(P, M)$ is a positive program, i.e., does not contain any rule with a negative antecedent.
If P also contains constraints, we say that a model M is *acceptable* iff it satisfies every constraint, i.e., M does not contain the special atom $\bot$.
Let P be a program having a non-empty set $Mod(P)$ of acceptable classical models.
We say that P is *incoherent* if P does not have a stable model. We say that P is contradictory if P has stable models, but none of them is acceptable.
We will need the following lemma's pertaining to properties of stable models:

Lemma 2.2 (Marek & Truszczyński [6]) *Let M be a model of a program P. Then $T_{G(P,M)} \uparrow \omega \subseteq M$.*

Lemma 2.3 *If M is a stable model of P and $M \models r$ for some program rule r over B_P, then M is also a stable model of $P + r$.*

PROOF If P_1 and P_2 are positive programs, $P_1 \subseteq P_2$ implies $lfp(T_{P_1}) \subseteq lfp(T_{P_2})$. Hence, $M^+ = lfpT_{G(P,M)} \subseteq lfp(T_{G(P+r,M)})$. Since M is a model of $P + r$, by the previous lemma, it follows that $lfp(T_{G(P+r,M)}) \subseteq M^+$. So $M^+ = lfp(T_{G(P+r,M)})$ and by definition of stability, M is a stable model of $P + r$. □

We will denote the set of stable models of a program P by $ST(P)$.

3 The expansion method

It is well-known that the stable model semantics for logic programs is not supra-classical in the sense that a program P may not have an acceptable stable model while it has a non-empty set of ordinary models.
The revision method we will introduce can be seen as an attempt to remedy this defect: we will revise P such that the resulting program P' has at least one acceptable stable model.
The idea we will develop is a strict generalization of the *dependency-directed backtracking* approach used in truth-maintenance and logic programming, in the sense that rules are added to the programming in such a way that its classical meaning is not affected.

The idea essentially is the following:

1. We represent an incoherent or contradictory program P by a classically equivalent set of propositional clauses Σ_P. Then Σ_P is reduced to an equivalent set of *minimally satisfiable* clauses Π.

2. We use the set Π to construct a *hierarchical* program P_Π *without constraints*, i.e., a normal logic program without circular dependencies. It is well-known that such a program has a unique stable model M_Π.

3. We show that M_Π is a model of P. Then, by Lemma 2.3, it follows that M_Π is also a *stable* model of the expansion $P' = P + P_\Pi$ of P. This shows that $ST(P') \neq \emptyset$.

4. Finally, we show that every model of P is also a model of P', i.e. $Mod(P) \subseteq Mod(P')$. Since $P' \supseteq P$, we also have $Mod(P') \subseteq Mod(P)$, hence, $Mod(P) = Mod(P')$.

This show that the method is a strict generalization of the dependency-directed backtracking method: beside contrapositives of rules we do allow for the addition of other rules as far as they do not affect the classical meaning of the program.

To discuss this method, we need the following concepts and definitions.
A *literal* is an atomic proposition (atom) or its negation. A *clause* is a finite repetition-free disjunction of literals. A clause C is called *positive* if it contains only positive atoms. We use $\sim C$ to denote the *conjunction* of the negations of literals occurring in C.
A clause C is said to be a *prime implicant* of a set of clauses Σ iff $\Sigma \models C$ and there is no proper subset $C' \subset C$ such that $\Sigma \models C'$.
Given a set of clauses Σ, let $\Pi(\Sigma)$ denote a (smallest) subset of prime implicants of Σ such that for every $C \in \Sigma$ there is a prime implicant $C' \subseteq C$ occurring in $\Pi(\Sigma)$. The following proposition is immediate:

Proposition 3.1 $Mod(\Sigma) = Mod(\Pi(\Sigma))$.

We will denote the set of clauses associated with a program P by Σ_P. It is obvious that Σ_P and $\Pi(\Sigma_P)$ are always satisfiable and $Mod(P) = Mod(\Pi(\Sigma_P))$.
The following observations are essential for the discussion of the expansion method:

Observation 3.2 *Let* $M_e = \{\sim a \mid a \in B_P\}$. *Then* M_e *is a stable model of* P *iff* $\Pi(\Sigma_P)$ *contains no positive clause.*

PROOF Suppose that M_e is a stable model of P and $\Pi(\Sigma_P)$ contains a positive clause C. Since $Mod(\Pi(\Sigma_P)) = Mod(P)$, it follows that every model M of P has to satisfy at least one positive literal in C; contradiction.
Conversely, suppose that $\Pi(\Sigma_P)$ contains no positive clause. Then, of course, M_e is a model of $\Pi(\Sigma_P)$ and hence, a model of P. Since $G(P, M_e)$ does not contain any unary rule $a \leftarrow$, $lfp(T_{G(P,M_e)}) = \emptyset$ and it follows immediately that M_e is a stable model of P. □

Observation 3.3 *For every positive clause* C *in* $\Pi(\Sigma)$ *and for every atom* c *in* C *there exists at least one model* M *of* Σ *such that* M *minimally satisfies* C *and* M *makes* c *true.*

PROOF Obvious, since $\Pi(\Sigma)$ is a set of prime implicants. □

We will now describe the method to extract a program P_Π having a stable model M_Π from the set of clauses $\Pi = \Pi(\Sigma_P)$.
First of all, we can assume that Π contains at least one positive clause $C = \{c_1, \ldots, c_m\}$. For else, according to Observation 3.2, M_e would be a model of P and, since M_e is the stable model of the empty program over B_P, by Lemma 2.3, M_e would be an acceptable stable model of P.

According to Observation 3.3 then, there is at least one model M of Π, minimally satisfying C, for example by making true c_1 and making false the other literals c_j. Now consider the rule

$$r_C : c_1 \leftarrow \sim c_2, \ldots, \sim c_m$$

and the model

$$M = \{c_1, \sim c_2, \sim c_3, \ldots, \sim c_m\}.$$

Note that M is the unique stable model of r_C. So, if we extract r_C from Π, we let P_Π contain r_C and set M_Π equal to M, we have a model M_Π which is:

1. a stable model of P_Π and

2. a *partial* model of Π, i.e., it can be extended to a model of Π by adding literals to it.

If we can extend both M_Π and P_Π, such that both (1) and (2) hold, finally, M_Π will be a complete model of Π and a stable model of P_Π.
Then, of course, by applying Lemma 2.3, it follows immediately that M_Π is a stable model of the expansion $P + P_\Pi$ of P.

To extend M_Π and P_Π, we have to consider the remaining part of Π after the extraction of r_C.
Since, eventually, M_Π has to be a model of Π, we can remove every *clause* C' from Π containing the literal c_1 or the literals $\sim c_j$ for $j \geq 2$: such a clause is already satisfied by the partial realization of M_Π.
On the other hand, in the remaining clauses C', every *literal* of the form $\sim c_1$ or c_j, $j \geq 2$, has to be removed, since such a literal is evaluated false by the current partial model M_Π.
Let Π' be the resulting set of clauses. We would like to continue with this set Π' to extract new rules from it to add them to P_Π and to extend M_Π.
There is, however, a slight problem here: the set Π', in general, is not a set of prime implicants. This implies that we cannot always select positive clauses from Π' and assume that they can be minimally satisfied.
Therefore, for every $C' \in \Pi'$ we replace C' by a prime implicant $\pi(C') \subseteq C'$ of Π'. Let $R(\Pi, M_\Pi)$ denote this resulting set of prime implicants.
Now the extraction process can be continued as long as $\Pi' = R(\Pi, M_\Pi)$ contains a positive clause $D' = \{d_1, \ldots d_k\}$.
Hence, there must exist a minimal model M' making d_1 true and false every d_j, $j \geq 2$.
However, instead of adding the rule $d_1 \leftarrow \sim d_2, \ldots, \sim d_k$ to P_Π, we now add the rule

$$r_{D',M_\Pi} = d_1 \leftarrow \sim d_2, \ldots, M_\Pi$$

to P_Π, where M_Π is a shorthand for the conjunction of all literals occurring in M_Π. Finally, we add $d_1, \sim d_2, \ldots, \sim d_k$ to M_Π.
The reason for adding M_Π to the body of the rule obtained from the positive clause D' is that at the end of the extraction process, we would like to ensure that every model of P also is a model of P_Π, in order to guarantee that $Mod(P) = Mod(P + P_\Pi)$.
Suppose we would omit M_Π from the body of the rule r_{D',M_Π}. Note that D' has been obtained from a clause D occurring in P by removing some literals evaluated false in M_Π. Now it might occur, that a model M of P satisfies D by making true a literal which no longer occurs in D'. Clearly, then such a literal must have been evaluated false in M_Π. Hence, M still satisfies r_{D',M_Π} but not $d_1, \sim d_2, \ldots, \sim d_k$.
This should give enough intuition to give a more formal verification of the following proposition:

Proposition 3.4 *Let M be a model of P and P_Π Let M_Π be the stable model of P_Π and a partial model of P.*
If C is a positive clause occurring in P_Π, then M is also a model of $P_\Pi \cup r_{C,M_\Pi}$. and $M_\Pi \cup \{d_1, \sim d_2, \ldots, \sim d_k\}$ is a stable model of $P_\Pi \cup r_{C,M_\Pi}$.

The following procedure is a succinct description of our method to find a stable model of an expansion P' of P:

---

input: the set $\Pi = \Pi(\Sigma_P)$.
output: a program P_Π and a model M_Π.
begin
Let $P_\Pi := \emptyset$; let $M_\Pi := \emptyset$;
while Π contains a positive clause $C = \{c_1, c_2, \ldots, c_m\}$
$r_{C,M_\Pi} := c_1 \leftarrow \sim c_2, \ldots, \sim c_m, M_\Pi$;
$P_\Pi := P_\Pi + r_{C,M_\Pi}$;
$M_\Pi := M_\Pi \cup \{c_1, \sim c_2, \ldots, \sim c_m\}$;
$\Pi := R(\Pi, M_\Pi)$
wend
$M_\Pi := M_\Pi \cup \sim(atoms(P) - atoms(M_\Pi))$;
return (P_Π, M_Π);
end;

It is easy to see that this procedure terminates for every normal program P since in every iteration of the while-loop the total number of literals occurring in Π is decreased.

The following proposition is easily verified:

Proposition 3.5 *P_Π is a hierarchical program and M_Π is the unique stable model of P_Π.*

PROOF If P_Π contains a rule r with head c, there is a first time r is added to P_Π. This implies that

1. c did not occur in (the body of) any rule added before r and
2. Since clearly, $atoms(P_\Pi) \cap atoms(R(\Pi, M_\Pi) = \emptyset$, every literal occurring in the body of r can never occur in the head of a rule r' added to P_Π thereafter.

This implies that P_Π does not contain any circular dependencies. Hence P is a hierarchical program.
By construction of P_Π, it is not difficult to see that at any stage during the extraction process, M_Π is a stable model of P_Π. Since every hierarchical normal program has a unique stable model, the proposition follows. □

Example 3.6 Consider the following incoherent program:

$$\begin{array}{lcl} P: & b & \leftarrow \sim a \\ & c & \leftarrow \sim b \\ & a & \leftarrow \sim c \end{array}$$

Since $ST(P) = \emptyset$, we transform P into a set of clauses $\Sigma_P = \{\{a, b\}, \{b, c\}, \{c, a\}\}$.
Now $\Pi = \Pi(\Sigma) = \Sigma$ and there is a positive clause $\{a, b\}$; so we add the rule $r : a \leftarrow \sim b$ to P_Π and we add $a, \sim b$ to M_Π. Now constructing $R(\Pi, M_\Pi)$, the clauses $\{a, b\}$ and $\{c, a\}$ can be removed since a occurs in both. In the clause $\{b, c\}$, b is removed, so, $\Pi' = R(\Pi, \{a, \sim b\})$ contains one positive clause $C = \{c\}$. Therefore, $c \leftarrow a, \sim b$ is added to P_Π and c to M_Π. Since $R(\Pi', \{a, \sim b, c\}) = \emptyset$, we have

$$P_\Pi = \{a \leftarrow \sim b, c \leftarrow a, \sim b\}$$

and $M_\Pi = \{a, \sim b, c\}$.
We see that M_Π is the unique stable model of the hierarchical program P_Π and that $M_\Pi \models C$ for every $C \in \Sigma_P$. Hence, M_Π is a stable model of $P + P_\Pi$.

Example 3.7 Consider the contradictory program P:

$$\begin{array}{lcl} \bot & \leftarrow & c, \sim a \\ c & \leftarrow & \sim b, d \\ d & \leftarrow & \\ b & \leftarrow & a \end{array}$$

Since $ST(P) = \{\{\bot, c, d\}\}$ and this stable model is contradictory, we transform P into an equivalent set of clauses

$$\Sigma_P = \{\{\sim c, a\}, \{c, b, \sim d\}, \{d\}, \{b, \sim a\}.$$

Now

$$\Pi = \Pi(\Sigma) = \{\{\sim c, a\}, \{b\}, \{d\}, \}.$$

Let us add the rule $r : b \leftarrow$ to P_Π and b to M_Π.
In the construction of $R(\Pi, M_\Pi)$, we can remove the clause $\{b\}$. Next, we select $\{d\}$, since it is positive. We add $d \leftarrow b$ to P_Π and d to M_Π. After the construction of $R(\Pi, M_\Pi)$, only the clause $\{\sim c, a\}$ remains. So this set of clauses does not contain a positive clause. Therefore the procedure returns $P_\Pi = \{b \leftarrow, d \leftarrow b\}$ and $M_\Pi = \{b, d, \sim a, \sim c, \}$.
It is easy to see that indeed M_Π is a stable model of $P + P_\Pi$.

To show the correctness of the procedure, consider the expanded program $P' = P + P_\Pi$ and the set M_Π. It suffices to show that

1. M_Π is a stable model of P'.
   This is Proposition 3.5 and Lemma 2.3.

2. $Mod(P') = Mod(P)$.
   Using Proposition 3.4, we can easily show that

$$Mod(P) \subseteq Mod(\Sigma_{P_\Pi}) = Mod(P_\Pi)$$

   Therefore,

$$Mod(P') = Mod(P + P_\Pi) \subseteq Mod(P) \subseteq Mod(P')$$

The following theorem[2] summarizes the results we have obtained:

Theorem 3.8 *For every classically consistent normal program P there is a classically equivalent program P' containing P such that $St(P') \neq \emptyset$.*

[2] As Wiktor Marek pointed out to me, in [8] Marco Schaerf describes a similar result, although quite different in motivation.

4 Discussion

We have discussed a generalization of the dependency-directed backtracking method to revise both contradictory and incoherent normal logic programs.
For every consistent program P, this expansion method generates a program P' containing P, with the following properties:

1. $Mod(P) = Mod(P')$, i.e., P and P' are classically equivalent,

2. $ST(P') \neq \emptyset$, i.e., P' is well-behaved.

Since P and P' are classically equivalent, they also have the same set of *minimal* models. Moreover, since every stable model of P' is a minimal model of P', this implies that the stable models of P' are minimal models of P.
Since our method is a non-deterministic one, the question arises which minimal models of P can occur as stable models of P', where P' is the result of applying the expansion method. It is not difficult to see that in principle every minimal model of P can occur as a stable model of some P': by selecting positive literals in a minimal model M minimally satisfying clauses of P, we can "reconstruct" such a model as a stable model of an expansion P'.

Proposition 4.1 *Let P be a program such that $ST(P) = \emptyset$ and let M be a minimal model of P. Then there exists an expansion P' of P such that $M \in ST(P')$.*

Hence, we would like to select those minimal models of P which occur as stable models of *minimal expansions* of P as the preferred models of a contradictory or incoherent program. Intuitively, such a semantical and syntactical constrained revision process seems to be more attractive than revisions based on syntactical criteria alone.

References

[1] Doyle, J., A Truth Maintenance System, *Artificial Intelligence* 12, 1979

[2] M. Gelfond and V. Lifschitz, The Stable Model Semantics for Logic Programming. In: *Fifth International Conference Symposium on Logic Programming*, pp. 1070-1080, 1988.

[3] L. Giordano and A. Martelli, Generalized Stable Models, Truth Maintenance and Conflict Resolution, in: D. Warren and P. Szeredi (eds) *Proceedings of the 7th International Conference on Logic Programming*, pp. 427-441, 1990.

[4] K. Inoue, Hypothetical Reasoning in Logic Programs, *Journal of Logic Programming*, **18**, 3, 191–227, 1994

[5] J. W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, Heidelberg, 1987.

[6] V. W. Marek and M. Truszczyński, *Nonmonotonic Logic*, Springer Verlag, Heidelberg, 1993.

[7] L. M. Pereira, J. J. Alferes and J. N. Aparicio, Contradiction Removal within well-founded semantics. In: A. Nerode, W. Marek and V. S. Subrahmanian, (eds.),

[8] M. Schaerf, Negation and minimality in non-Horn databases. In: C. Beeri (ed.), *Proceedings of the Twelfth Conference on Principles of Database Systems (PODS-93)*, pp. 147–157, ACM-Press, 1993.

L50-BWL-006

6g D16
6g F41
6g K13