Opportunistic Reasoning in DL Prolog

José Oscar Olmedo-Aguirre

Electrical Engineering, Cinvestav-IPN,
Av. Instituto Politécnico Nacional 2508, 07300 Mexico City, MEXICO,
oolmedo@cinvestav.mx

Abstract. The declarative programming language DL Prolog is currently being developed for coupling deduction and interaction for multi-agent system applications. The language design is addressed by providing a uniform programming model that combines the well-known refinements of resolution, SLD-resolution, UR-resolution and positive hyper-resolution, along with control strategies for problems dealing with stateless and state-based descriptions. The computational model of the language is shown to be correct with respect to resolution-based refutation. The main contribution of this work is in introducing a computational model that uses dynamic logic modalities in forward rules, leading to more efficient versions with respect to the reduced number of rules required to deal with the same problem.

Keywords. Logic programming, interaction, automated theorem proving, Horn clause theories, Dynamic Logic.

1 Introduction

Dynamic acquisition of interactive knowledge is an effective approach to deal with complex system design and development [7]. Blackboard systems [1, 2] were among the first interactive knowledge systems that imitates the opportunistic reasoning that arise in the brainstorming sessions of human experts while solving complex problems. Opportunistic reasoning allows to draw conclusions from a given set of facts placed in the blackboard, by reasoning in either forward or backward manner. The logic programming language `DL Prolog` [5] is being developed to combine efficiently both forms of reasoning. The computational model comprises SLD-resolution, UR-resolution and positive hyper-resolution [8, 9], to describe respectively stateless deduction and state-based transitions. The coordination model consists of a transactional global memory of ground facts along with a strategy for the theorem prover to control program execution by syntactically guided rule selection. In addition, the set of support restriction strategy coordinates the input and output of facts with the shared memory, maintaining the coherence of the current state of the computing agent.

Let us briefly explore other approaches that can be compared with ours: resolution theorem provers, constraint logic programming and coordination logic programming. The resolution-based theorem prover *OTTER* [8, 9] comprises a number of refinements of resolution along with a set of control strategies to

Advances in Computer Science and Engineering
Research in Computing Science 58, 2012 pp. 197–207
Paper Recived 01-10-2012 and Acepted 07-11-2012

prune the explosive generation of intermediate clauses. However, OTTER does not account for interaction. The set of all instantaneous descriptions essentially corresponds to the set of support strategy. In OTTER, a clause is selected and removed from the set of support to produce a new set of clauses deduced from the axioms of the theory. Then, after simplifying a new clause by demodulation and possibly discarding it by either weighting, backward or forward subsumption, the new clause is placed back to the set of support.

Concurrent Constraint Programming (CCP) [6] proposes a programming model centered on the notion of constraint store that is accessed through the basic operations 'blocking *ask*' and 'atomic *tell*'. Blocking $ask(c)$ corresponds to the logical entailment of constraint c from the contents of the constraint store: the operation blocks if there is not an enough strong valuation to decide on c. In this respect, the blocking mechanism is similar to the one used in `DL Prolog` to obtain the set of ground facts that match with the left-hand side of some rule. Besides, the constraint store shares some similarities with the global memory of ground facts. However, operation $tell(c)$ is more restrictive than placing ground atoms in the global memory because constraint c must be logically consistent with the constraint store.

Extended Shared Prolog (ESP) [3] is a language for modeling rule-based software processes for distributed environments. ESP is based in the PoliS coordination model that extends Linda with multiple tuple spaces. The language design seeks for combining the PoliS mechanisms for coordinating distribution with the logic programming Prolog. Coordination takes place in ESP through a named multiset of passive and active tuples. They correspond to the global memory of facts in `DL Prolog`, although no further distinction between passive and active ground facts is made. ESP also extends Linda by using unification-based communication and backtracking to control program execution.

The paper is organized as follows. First we illustrate the forward and backward reasoning schemes that arise from the computational model with a programming example. Next, the syntax and the declarative semantics of the the `DL Prolog` programming language is presented. Finally, some remarks are given to conclude.

2 A programming example

As agents perceive the surrounding environment through *sensors* and act upon it through *effectors*, their interaction can effectively be decoupled by a shared coordination medium consisting of a multiset of ground facts that can be implemented by a blackboard system. By abstracting away interaction from deduction, the inherently complex operational details of sensors and effectors become irrelevant. The behavior of each individual agent is described by a set of backward and forward rules that describe the exchange of information through the coordination medium. As an example, consider the problem of parsing and evaluating simple arithmetic expressions. The parser uses the context free grammar (CFG):

$$E \to i \mid (E) \mid E + E \mid E \times E$$

where non-terminal E stands for a well-formed integer expression.

Table 1 shows theory *Natural* for the natural numbers written in `DL Prolog`, closely similar to those written in pure Prolog. This theory uses *backward rules* that have the general form $P \Leftarrow P_1, \ldots, P_n$ with $n \geq 0$. The logical propositions of the theory are built upon infix predicates $=$, $<$, and $\leq$, whose recursive definitions are given by clauses N_1 to N_5. *Natural* represents the deductive component of the interactive parser.

theory *Natural*
axioms
N_1 : $0 + y = y \Leftarrow$
N_2 : $(x + 1) + y = (x + y) + 1 \Leftarrow$
N_3 : $0 \leq y \Leftarrow$
N_4 : $(x + 1) \leq (y + 1) \Leftarrow x \leq y$
N_5 : $x < y \Leftarrow (x + 1) \leq y$
end

Table 1. Natural numbers using backward rules.

Table 2 shows a theory written in `DL Prolog` for a parallel *Parser* that extends *Natural*. This theory uses *forward rules* that have the general form $P_1, \ldots, P_n \,|\, C \Rightarrow [A]\, P$ with $n \geq 0$. The declarative reading of the forward rule is that, if appropriate predicates $P_1, \ldots, P_n$ have been placed in the common memory and their contents satisfy the condition C, then the action (i.e. imperative program) A is executed to obtain the values bound to the variables occurring in the postcondition P. The rules of *Parser* define a bottom-up parser for simple arithmetic expressions whose syntactic entities are represented by ground atoms. $T(n, t)$ asserts that symbol t occurs at position n, while $E(n_1, n_2, x)$, with $n_1 \leq n_2$, asserts that the sequence of symbols from n_1 to n_2 forms a well-formed arithmetic expression whose evaluation is the integer value x. The forward rules are remarkably similar to those given in the CFG. Table 3 shows the forward rules P_1' through P_3' that respectively replace P_1 through P_3 from the theory $Parser$ to produce a sequential parsing and evaluation of the number enclosed in brackets. The parsing goes from left to right, starting as soon as the left-bracket $'['$ is detected and terminating after the right-bracket $']'$ is detected. Table 4 shows the sole forward rule P_{123}'' that replaces rules P_1 through P_3 from the theory $Parser$ to produce an alternative sequential parser along with its evaluation. This version of the parser differs from the previous sequential version in that it only uses one forward rule for the parsing of the numbers, leading to a more efficient parsing and evaluation. In rule P_{123}'', the action resembles the usual sequential imperative program that parses and converts the sequence of digits to a number whose value is bound to the output variable z of the postcondition $E(n_1, n_2, z)$. The meaning of the action connectives is explained in section devoted to the formal description of the language.

theory *Parser*
extends *Natural*
rules
$P_1 : T(n,t) \mid digit(x) \Rightarrow [x := toInt(t)]\, N(n,n,x).$
$P_2 : N(n_1,n_2,x), N(n_3,n_4,y)$
$\quad \mid n_1 \leq n_2, n_2 = n_3, n_3 \leq n_4$
$\quad \Rightarrow [z := x \times 10^{n_4 - n_3 + 1} + y] N(n_1,n_4,z).$
$P_3 : T(n_1, '['), N(n_2,n_3,x), T(n_4, ']')$
$\quad \mid n_1 + 1 = n_2, n_2 \leq n_3, n_3 + 1 = n_4$
$\quad \Rightarrow E(n_1,n_4,x).$
$P_4 : T(n_1, '('), E(n_2,n_3,x), T(n_4, ')')$
$\quad \mid n_1 + 1 = n_2, n_2 \leq n_3, n_3 + 1 = n_4$
$\quad \Rightarrow E(n_1,n_4,x).$
$P_5 : E(n_1,n_2,x), T(n_3, '+'), E(n_4,n_5,y)$
$\quad \mid n_1 \leq n_2, n_2 + 1 = n_3, n_3 + 1 = n_4, n_4 \leq n_5$
$\quad \Rightarrow [z := x + y] E(n_1,n_5,z).$
$P_6 : E(n_1,n_2,x), T(n_3, '\times'), E(n_4,n_5,y)$
$\quad \mid n_1 \leq n_2, n_2 + 1 = n_3, n_3 + 1 = n_4, n_4 \leq n_5$
$\quad \Rightarrow [z := x \times y] E(n_1,n_5,z).$
end

Table 2. Bottom-up parallel parser for arithmetic expressions.

$P'_1 : T(n, '[') \Rightarrow N(n,n,0).$
$P'_2 : N(n_1,n_2,x), T(n_3,t)$
$\quad \mid n_1 \leq n_2, n_2 + 1 = n_3, digit(t)$
$\quad \Rightarrow [z := x \times 10 + toInt(t)]\, N(n_1,n_3,z).$
$P'_3 : N(n_1,n_2,x), T(n_3, ']')$
$\quad \mid n_1 \leq n_2, n_2 + 1 = n_3$
$\quad \Rightarrow E(n_1,n_3,x).$

Table 3. Sequential parsing of numbers.

$$P''_{123} : T(n_1, '[') \Rightarrow \left[\texttt{int}\, x,t,n : \left(\begin{array}{l} x,n := 0, n_1 + 1; \\ \left(\begin{array}{l} T(n,t)\texttt{?}; digit(t)\texttt{?}; \\ \quad x,n := 10 \times x + toInt(t), n + 1; \end{array} \right)\texttt{*}; \\ t = ']'\texttt{?}; \\ z,n_2 := x,n \end{array} \right) \right] E(n_1,n_2,z).$$

Table 4. Sequential parsing of numbers with a dynamic logic modal action.

Table 5 sketches the interaction that produce both sequential parsers in the common memory, when dealing with the input $([\texttt{3141}] + [\texttt{79}]) \times [\texttt{2}]$. In the table, the time increases downwards with each row, whereas the symbols of the input are disposed horizontally. At the top of the table, the input consists of two rows: the first row corresponds to the indexes of the input, starting with 0, whereas the second row corresponds to all the symbols of the input occurring at the corresponding position given by the index. The entire input is not available immediately, but rather unpredictably and individually each symbol from another. When a sensor detects a symbol t at position n, it asserts the ground predicate $T(n, t)$, as shown in the first column of the table. For example the symbols $'4'$ and $'['$, at positions 4 and 14, respectively, were early detected, while the symbol $']'$ at position 6 was detected soon after. When there are enough symbols in the memory to activate a forward rule, those symbols are enclosed in a box and the rule applied is shown in the first column of the table. For example, at the row 8, the rules P_1' and P_2' are applied one after the other. However due to there is no symbol at index position 3, the parsing stops there. The parsing resumes immediately after the symbol $'1'$ at 3 becomes available, continuing the parsing of the entire number, from positions 1 to 6 with value 3141. The parsing and evaluation process continues until all the symbols of the input expression are eventually analyzed. Then an effector may inform to a client agent that the given expression is well-formed by placing the term $E(0, 16, 6440)$ that include its evaluation.

3 `DL Prolog` formal description

An experimental system for DL Prolog has been built to evidence the viability of the approach. The system consists of a parser with integrated type inference to decide whether the program constructs are well-formed. The computational model is described as a structured-operational semantics interpreter that calculates the next state of the shared memory.

Let $\Sigma = \bigcup_\alpha \Sigma_\alpha$ be a set of *constructor* (constant) names and let $\Xi = \bigcup_\beta \Xi_\beta$ be a set of *variable* names, each partitioned by the basic types `bool`, `int`, and `act`, among others. The set $T(\Sigma, \Xi)$ of *terms with variables* is the minimal set of phrases that is closed under composition of a constructor with a (possibly empty) previously constructed sequence of terms. The set $T(\Sigma) = T(\Sigma, \emptyset)$ of *ground terms* consists of the terms with no variables. Type judgments are embedded in the grammar rules of the language to ensure that clauses and programs are well-formed. In particular, the grammar rule shown below describes the syntactic structure of a well-formed term T_β of type β:

$$T_\beta ::= x_\beta \mid c_\beta \mid c_{\beta_1 \cdots \beta_n \to \beta}(T_{\beta_1}, \ldots, T_{\beta_n})$$

The set $P(\Sigma, \Xi)$ of *atomic predicates with variables* is the minimal set closed under composition of predicate symbols with (possibly empty) sequences of terms. The set $P(\Sigma) = P(\Sigma, \emptyset)$ of *ground atoms* consists of all atoms with no variables.

	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
	(	[	3	1	4	1	]	+	[	7	9	]	)	×	[	2	]
$\Rightarrow T(4,\,'4'), \Rightarrow T(16,\,'[')$					4										[		
$\overset{P'_1}{\Rightarrow} N(16,16,0)$															[		
$\Rightarrow T(6,\,']')$							]										
$\Rightarrow T(2,\,'3'), \Rightarrow T(10,\,'7')$			3							7							
$\Rightarrow T(12,\,')')$													)				
$\Rightarrow T(10,\,'9'), \Rightarrow T(11,\,']')$											9	]					
$\Rightarrow T(1,\,'[')$		[															
$\overset{P'_1}{\Rightarrow} N(1,1,0), \overset{P'_2}{\Rightarrow} N(1,2,3)$		[	3														
$\Rightarrow T(3,\,'1'), \Rightarrow T(13,\,'\times')$				1										×			
$\overset{P'_2}{\Rightarrow} N(1,4,314)$		[	3	1	4												
$\Rightarrow T(5,\,'1'), \Rightarrow T(18,\,']')$						1											]
$\overset{P'_2}{\Rightarrow} N(1,5,3141)$		[	3	1	4	1											
$\overset{P'_3}{\Rightarrow} E(1,6,3141)$		[	3	1	4	1	]										
$\Rightarrow T(7,\,'+'), \Rightarrow T(15,\,'2')$								+								2	
$\overset{P'_2}{\Rightarrow} N(14,15,2)$															[	2	
$\overset{P'_3}{\Rightarrow} E(14,16,2)$															[	2	]
$\Rightarrow T(8,\,'[')$									[								
$\overset{P'_1}{\Rightarrow} N(8,8,0)$									[								
$\overset{P'_2}{\Rightarrow} N(8,10,79)$									[	7	9						
$\overset{P'_3}{\Rightarrow} E(8,11,79)$									[	7	9	]					
$\overset{P_5}{\Rightarrow} E(1,11,3220)$		[	3	1	4	1	]	+	[	7	9	]					
$\Rightarrow T(0,\,'(')$	(																
$\overset{P_4}{\Rightarrow} E(0,12,3220)$	(	[	3	1	4	1	]	+	[	7	9	]	)				
$\overset{P_6}{\Rightarrow} E(0,16,6440)$	(	[	3	1	4	1	]	+	[	7	9	]	)	×	[	2	]

Table 5. Interactive parsing.

A *literal* is an atom or a negated atom. A *clause* is a disjunction of literals. The set of clauses with variables is denoted $C(\Sigma, \Xi)$. Clauses are usually written in implication form $P \Leftarrow Q$, where P, called the *consequent,* is a disjunction of atoms and Q, called the *antecedent*, is a conjunction of atoms. A *unit clause* contains only one literal. A *positive clause* contains no negated atoms, whereas a *negative clause* contains no positive atoms. A *Horn clause* contains at most a positive atom. A *goal* consists only of negative atoms that can be represented by an implication with false as consequent. The *set of goals with variables* is denoted $G(\Sigma, \Xi)$.

Terms $T ::= x \mid c \mid c(T_1, \ldots, T_n)$
Predicates $P ::= \texttt{false} \mid \texttt{true} \mid T_1 = T_2 \mid p(T_1, \ldots, T_n)$
Goals $G ::= P \mid G_0 \wedge G_1$
Horn clauses (Backward rules) $B ::= P \mid P \Leftarrow G \mid \forall x.B$
Events $E ::= P \mid E_1 \ , \ E_2$
Actions (Programs)

$$A ::= \texttt{skip} \mid \texttt{fail} \mid G\texttt{?} \mid (A) \mid A_1\texttt{;}A_2 \mid A_1 \cup A_2 \mid A\texttt{*} \mid \texttt{int}\ x_1, \ldots, x_n : A \mid x_1, \ldots, x_n := T_1, \ldots, T_n$$

Modal actions $M ::= P \mid [A]\ M \mid \langle A \rangle\ M$
Modal clauses (Forward rules) $F ::= M \mid E \,|\, G \Rightarrow M \mid \forall x.F$

The DL modal extensions require the introduction of well-formed imperative programs in clauses, hereinafter called *actions*. The set $A(\Sigma, \Xi)$ *of actions with variables* is the minimal set of phrases that is closed under composition of actions with action connectives. A basic action is either the null-action ($\texttt{skip}_{\texttt{act}}$), the failure ($\texttt{fail}_{\texttt{act}}$), or the assignment of ground terms to variables (using the binary operator $:=_{\texttt{int int}\rightarrow\texttt{act}}$). Action connectives are the postfix unary operator for testing a condition ($\texttt{?}_{\texttt{act}}$), the postfix unary operator for iteration ($\texttt{*}_{\texttt{act}\rightarrow\texttt{act}}$), the infix binary operators for sequential composition ($\texttt{;}_{\texttt{act}\rightarrow\texttt{act}}$), and the infix binary operator for non-deterministic choice ($\cup_{\texttt{act}\rightarrow\texttt{act}}$). The mixfix modal connectives of modal necessity ($[]_{\texttt{act}\rightarrow\texttt{bool}}$) and possibility ($\langle\rangle_{\texttt{act}\rightarrow\texttt{bool}}$) compose actions along with their postconditions. The following definitions provide an interpretation of the usual imperative programming constructs in terms of DL actions:

$$\begin{aligned}
\texttt{skip} &\equiv \texttt{true?} \ , \ \texttt{fail} \equiv \texttt{false?} \\
\texttt{if}\ F\ \texttt{then}\ A\ \texttt{fi} &\equiv F\texttt{?;}A \\
\texttt{if}\ F\ \texttt{then}\ A_1\ \texttt{else}\ A_0\ \texttt{fi} &\equiv (F\texttt{?;}A_1) \cup (\neg F\texttt{?;}A_0) \\
\texttt{while}\ F\ \texttt{do}\ A\ \texttt{od} &\equiv (F\texttt{?;}A)\texttt{*;}\neg F\texttt{?}
\end{aligned}$$

The precedence in decreasing order among the action connectives is the following: test (?), iteration (*), sequence (;) and non-deterministic choice ($\cup$). Parenthetical expressions are allowed to modify the precedence order of the action connectives. Variables occurring in an action A are either *logical variables*

or *local imperative variables*. Logical variables occurring in a clause are universally quantified, whereas local variables are introduced by declaration within an action. A declaration of local variables `int` $x_1, \ldots, x_n : A$ creates new local imperative variables whose scope and duration are restricted to the block A. A simple assignment $x := T$ evaluates the term T in the current state and the resulting constant value is assigned to x. Logical and imperative variables are compatible in assignments of the same type, so they can appear in both sides of the assignment. Note however that logical variables can be defined at most once, whereas imperative variables can be redefined. A multiple assignment $x_1, \ldots, x_n := T_1, \ldots, T_n$ evaluates all the terms at the right-hand side in the current state and the resulting values are assigned to the corresponding variables at the left-hand side of the assignment. Modal necessity composition $[A]\ P$ means that after executing action A, postcondition P is necessarily true.

In a signature (Σ, Ξ) with variables, a *substitution* is a partial function $\sigma : \Xi \to T(\Sigma, \Xi)$, where $\sigma(x) \neq x$ for any variable $x \in \Xi$. $\{\}$ denotes the empty substitution. A *ground substitution* is a substitution $\sigma : \Xi \to T(\Sigma)$ valued on ground terms. For any variable $x \in \Xi$ and any substitution σ, let $x\sigma = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$ and $x\sigma = x$ otherwise. For any term $t \in T(\Sigma, \Xi)$, let $t\sigma$ be the term obtained by substituting any variable x appearing in T by $x\sigma$:

$$
\begin{array}{rr}
x\{\} = x & x\,\sigma = \begin{cases} x & \text{if } x \notin \mathrm{dom}(\sigma) \\ (x\sigma) & \text{if } x \in \mathrm{dom}(\sigma) \end{cases} \\
c\,\sigma = c & c(T_1, \ldots, T_n)\,\sigma = c(T_1\,\sigma, \ldots, T_n\,\sigma) \\
[A]\,p\,\{\} = [A]\,p & [A]\,p\,\sigma = [\sigma_{:=}\,;A]\,p
\end{array}
$$

where notation $[\sigma_{:=}]$ stands for the multiple assignment $x_1, \ldots, x_n := T_1, \ldots, T_n$ given the substitution $\sigma = \{x_1 \mapsto T_1, \ldots, x_n \mapsto T_n\}$, for $1 \leq n$. Thus the substitution for a modal action A is defined as the initial value that the variables take before the action starts its execution. The *composition* of two substitutions $\sigma_0, \sigma_1 \in \Xi \to T(\Sigma, \Xi)$, written $\sigma_0 \cdot \sigma_1$, is defined as

$$
\sigma_0 \cdot \sigma_1 : x \mapsto \begin{cases} (x\sigma_0)\sigma_1 & \text{if } x\sigma_1 \notin \mathrm{dom}(\sigma_1) \\ x\sigma_1 & \text{if } x \in \mathrm{dom}(\sigma_1) - \mathrm{dom}(\sigma_0) \\ failure & \text{otherwise} \end{cases}
$$

Besides the natural extension to terms $T(\Sigma, \Xi) \to T(\Sigma, \Xi)$, substitutions are also extended to predicates, goals, and both backward and forward rules.

3.1 Computational model

The *backward computation* relation $\lhd \subset G(\Sigma, \Xi) \times (\Xi \to T(\Sigma, \Xi))$ consists of pairs relating goals and substitutions, where the substitutions are defined upon the variables occurring in a renamed variant of the rule. An *instantaneous description* $I \subset P(\Sigma) \times (\Xi \to T(\Sigma))$ relates ground predicates and ground substitutions, describing a portion of the current state of the shared memory. The substitutions keep a track of the bindings for all the variables that occurred in

the renamed variant of each forward rule applied. The *forward computation* relation $\triangleright \subset \mathcal{P}(I) \times \mathcal{P}(I)$ relates pairs of instantaneous descriptions. The transition relations are defined in Table 6.

Backward computation

$$\frac{P' \Leftarrow G' \in B(\Sigma, \Xi) \quad P\sigma' = P'\sigma'}{(\{P\} \cup G, \sigma) \triangleleft (G'\sigma' \cup G\sigma', \sigma\sigma')}$$

Forward computation

$$\frac{E_1, \ldots, E_n \mid G \Rightarrow [A]\, P \in F(\Sigma, \Xi) \quad E_i\sigma_i = P_i\sigma_i,\ i \in 1, ..., n \quad (G, \sigma_1 \cdots \sigma_n) \triangleleft^* (\{\}, \sigma)}{\{(P_1, \sigma_1), \ldots, (P_n, \sigma_n)\} \cup I \triangleright \{(P_1, \sigma_1), \ldots, (P_n, \sigma_n), ([\sigma_{:=}; A]\, P, \sigma\sigma')\} \cup I}$$

Table 6. Operational semantics of backward and forward computation.

The backward computation rule describes a *refutation step* from $(\{P\} \cup G, \sigma)$ to $(G'\sigma' \cup G\sigma', \sigma\sigma')$ by replacing the head $P\sigma'$ with the body $G'\sigma'$ of the instance of the backward rule $P \Leftarrow G'$ under a suitable substitution σ' such that $P\sigma' = P'\sigma'$. The new goal is an instance under σ' of the body G' and the remaining goal G, along with the new answer substitution obtained from the composition of σ' with the previous one σ. In case that the application of the rule leads to a failure, another backward rule if any is selected and applied after backtracking to the previous goal and the previous substitution; otherwise, if no more rules can be selected, the backward computation terminates in failure.

The forward computation rule $E_1, \ldots, E_n \mid G \Rightarrow [A]\, P$, with $n > 0$, can be selected for deducing the ground predicate $P\sigma\sigma'$ only if the following three conditions hold: (i) there are n ground predicates $P_1, \ldots, P_n$ already asserted in the working memory, (ii) there are n ground substitutions $\sigma_1, \ldots, \sigma_n$ that makes syntactically identical the corresponding instances of each event E_i with an appropriate predicate P_i, i.e. equation $E_i\sigma_i = P_i\sigma_i$ holds for $1 \leq i \leq n$, and (iii) the composition $\sigma_1 \cdots \sigma_n$ of the n substitutions satisfies the goal G. Whenever these conditions are met, the forward rule can be applied. In the rule, because the variables occurring in any event E_i does not occur in any other E_j $(i \neq j)$, the composition of their corresponding ground substitutions simply consists of the union of all of them. The ground substitution σ produced by the backward computation rule may extend the composition $\sigma_1 \cdots \sigma_n$ with bindings for the new variables that G may introduce. Hence, the equation $E_i\sigma = P_i\sigma$ also hold for σ with $1 \leq i \leq n$. The instance under σ of the modal action $[A]\, P$ is then executed following the standard interpretation of the action connectives [4]. Assuming that A terminates with the initial values given by σ, the postcondition P becomes

satisfied by the substitution $\sigma\sigma'$, i.e. by the values computed by A assigned to the output variables occurring in σ'. The truth of the ground predicate $P\sigma\sigma'$ leads to the instantaneous description $(P, \sigma\sigma')$. However, if the guard $G\sigma$ fails, another set of predicates asserted in the shared memory must be considered. If no more possible selections of predicates were possible for the forward rule, another rule is selected if any. If no more forward rules were applicable, the agent would appear as non-responsive until another predicate assertion were eventually produced in the shared memory.

The correctness of the computational model can be stated as follows:

Proposition 1 (Correctness).

$\{(P_1, \sigma), \ldots, (P_n, \sigma)\} \triangleright \{(P_1, \sigma), \ldots, (P_n, \sigma), (P, \sigma\sigma')\}$ *implies* $P_1\sigma \wedge \ldots \wedge P_n\sigma \Rightarrow P\sigma\sigma'$

Note that a predicate asserted by the forward rule monotonically increases the content of the shared memory as the events are never retracted by the rule.

4 Conclusions

The problem of coupling interaction in a resolution theorem prover with syntactically guided selection of the control strategy to be used has been presented in this paper. The experimental programming language `DL Prolog` has been designed to deal with state-based descriptions using forward rules and stateless deduction using backward rules. The programming model allows to combine backward and forward rule chaining in a simple and more efficient manner.

References

1. D. D. Corkill Collaborating Software: Blackboard and Multi-Agent Systems and the Future. In Proceedings of the International Lisp Conference, New York, New York, Oct, 2003.
2. D. D. Corkill GBBopen Tutorial. The GBBopen Project, March 2011. online: http://gbbopen.org/hypertutorial/index.html
3. P. Ciancarini Coordinating Rule-Based Software Processes with ESP, *ACM Trans. on Software Engineering and Methodology*, 2(3):203-227, July, 1993.
4. D. Harel, J. Tiuryn, and D. Kozen Dynamic Logic. Cambridge, MA, USA: MIT Press, 2000.
5. J. O. Olmedo-Aguirre and G. Morales-Luna. A Dynamic Logic-based Modal Prolog. In Proceedings MICAI 2012, San Luis Potosi, Meẋico, Oct, 2012.[To appear]
6. V.A. Saraswat Concurrent Constraint Programming. *Records of 17th ACM Symposium on Principles of Programming Languages*, 232-245. San Franciso, CA. 1990.
7. P. Wegner, Interactive Software Technology, *CRC Handbook of Computer Science and Engineering*, May 1996.
8. L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning. Introduction and Applications*, McGraw-Hill, Inc., 1992.
9. L. Wos and G. Pieper, *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning*, World Scientific Publishing Co., 1999.