Flexible Rule-Based Programming for Autonomic Computing

José Oscar Olmedo-Aguirre[1], Marisol Vázquez-Tzompantzi[2]

[1] Cinvestav-IPN, Department of Electrical Engineering, Mexico City, Mexico

[2] Cinvestav-IPN, DCTS, Mexico City, Mexico

oolmedo@cinvestav.mx, mvazquezt@cinvestav.mx

Abstract. The ECAP rule programming language DLRL is currently being developed for architecting autonomic systems by coupling deduction and interaction. Three of the fundamental properties of autonomic systems, namely self-configuration, self-optimization and self-healing, are provided by DLRL: high-level program specification that can be user-defined for self-configuration; program introspection that allows to reactively adapt on-line program behavior for self-optimization, and program interaction that provides communication and coordination with the surrounding environment in order to detect deviations from their expected behavior for self-healing. The DLRL, programming model extends pure Prolog by including the modal actions of dynamic logic in the consequent predicates of conditional forward rules. It combines some the well-known refinements of resolution along with syntactically guided control strategies to represent and enact problem specifications dealing with stateless and state-based descriptions. The main contribution of this work consists in showing the benefits for architecting autonomic systems in a single, uniform and expressive mulit-paradigm programming language for rapidly changing demands of complex problems in distributed settings.

Keywords: Logic programming, interaction, autonomic computing, self-management, dynamic logic.

1 Introduction

As the complexity of developing and maintaining distributed applications has steadily increased, new approaches for software design are needed to provide a simple yet resourceful conceptual framework. The traditional off-line feedback for reconfiguring and reprogramming software needs to be replaced by an on-line self-management software. The software needs to adapt to the rapidly changing demands of modern applications and still keeping the desired quality of service within reasonable costs during operation. IBM introduced the main properties

of self-management, known as the self-* properties of an autonomic system: self-configuration, self-optimization, self-healing and self-protection [1, 2]:

- Self-configuration. An autonomic system has the ability to configure itself according to high-level goals stated in a declarative manner, by specifying what is desired, not necessarily how to accomplish it.
- Self-optimization. An autonomic system pursues the best use of resources. After some automated reasoning, the autonomic system may conclude to initiate a change to the system behavior, either reactively by pursuing the user demands or pro-actively by systematically satisfying its own goals, in an attempt to improve performance or quality of service.
- Self-healing. An autonomic system detects, analyzes and diagnoses (potential) problems. The kind of problems detected can be low-level like hardware malfunctioning or high-level like a non-responsive software module. Being distributed, the autonomic system may take advantage of the available redundancy in both hardware and software to fix the problem, by switching to a trusted redundant component or by downloading and installing a software update. Because this replacement process is a potential risk for various kinds of malware intrusion, self-healing must perform exhaustive checking that ensures the system exhibits the expected trusted behavior before becoming an operational component in its distributed setting.
- Self-protection. An autonomic system protects itself not only from intrusion attacks but also from users who inadvertently make changes that opposes to the overall system purposes and behavior. The system opportunely verifies and adjusts its settings on security, privacy and data protection based on the rise of unexpected patterns of external activity.

`DLRL` is a symbolic rule-based programming language where self-configuration, self-optimization and self-healing, are respectively provided by:

- high-level program specification that can be user-defined
- program introspection that allows to reactively adapt on-line program behavior
- program interaction that provides communication and coordination with the surrounding environment in order to detect deviations from their expected behavior

The property of self-protection is believed by the authors that can be derived from the others three in the setting of an appropriate architecture. `DLRL` is an ECAP programming language with deductive, introspective and interactive capabilities that provide a run-time system that is able to reconfigure and recompile program code on demand whenever the changes of the surrounding environment compromises the goals of the program purpose. ECAP stands for Event-Condition-Action-Postcondition an extension of usual ECA rules commonly found in database management systems where the Postcondition imposes additional constraints to the result of the Action. `DLRL` borrows from `Indeed` [5] its state-based forward conditional rules and from `DLProlog` [6, 7] the

dynamic logic modalities [4] for efficient imperative program execution. Thus, it is able to combine efficiently stateless and state-based reasoning along with coordinated interaction. The computational model comprises various forms of resolution-based inference procedures, like SLD-resolution, UR-resolution and positive hyper-resolution [10, 11], to describe respectively stateless deduction and state-based transitions. The coordination model consists of a transactional global memory of ground predicates 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 [11] coordinates the input and output of facts with the shared memory, maintaining the coherence of the current state of the program.

The paper is organized as follows. In section 2, a brief revision of the related work is presented, In section 3, we illustrate the forward and backward reasoning schemes that arise from the computational model with a programming example. Next, in section 4, the syntax and the declarative semantics of the the `DLRL` programming language is presented. Finally, in section 5 some remarks are given to conclude.

2 Related Work

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 *Prover9* [10, 11] comprises a number of refinements of resolution along with a set of control strategies to prune the explosive generation of intermediate clauses. However, Prover9 does not account for interaction. The set of all instantaneous descriptions essentially corresponds to the set of support strategy. In Prover9, 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) [8] 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 `DLRL` 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 DLRL 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.

3 A Programming Example

As autonomic systems perceive the surrounding environment through *sensors* and act upon it through *effectors*, their interaction can effectively be decoupled by a global shared transactional memory consisting of a multiset of basic facts (ground predicates) that can be implemented following the interaction approach of Wegner [10]. By abstracting away interaction from deduction, the inherently complex operational details of sensors and effectors become irrelevant. The behavior of each user-defined system is described by a set of backward and forward rules that describe the exchange of information through the global memory. As an example, consider the problem of converting a sequence of digits to obtain its numeric value where the sequence is fragmented in subsequences that are dispersed across a large partitioned region. This problem is representative of a class of problems that deal with summarizing information coming from disperse geographical regions, like those that deal with calculating the average temperature, humidity or atmospheric pressure of a large region, for example. In order to perform the conversion, the subsequences need to be retrieved and incorporated into the calculations in a coordinated manner. Because of the underlying indeterminism, sometimes one method may lead to shorter processing time for some regions than for others. The problem addressed here can be stated as finding and applying the faster method suitable for some region and to incorporate this partial result into the overall calculation.

For the conversion, a recursive definition of the basic arithmetic operators is given next. Figure 1 shows theory *Natural* for the natural numbers written in DLRL, closely similar to those written in pure Prolog. Similar rules, not presented here, can be given for arithmetic multiplication. This theory uses *backward rules* that have the general form $P \Leftarrow P_1, \ldots, P_n$, where $P, P_1, \ldots, P_n$ are atomic predicates, P is the consequent, or simply the head of the rule and $P_1, \ldots, P_n$ is the antecedent, or simply the body of the rule, with $n \geq 0$. The ellipsis stands for the conjunction of predicates $P_1, \ldots, P_n$ if $n > 0$ and for `true` if $n = 0$. The logical propositions of the theory are built upon infix predicates $=$, $<$, and $\leq$, whose recursive definitions are given by backward rules N_1 to N_5. *Natural* represents the deductive component of the interactive parser. Figure 2 shows a theory written in DLRL for a parallel *Parser* that extends *Natural*. This theory uses *forward rules* that have the general form $E_1, \ldots, E_n \,|\, C \Rightarrow [A]\, P$ 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 shared memory such that each E_i $(i = 1, \ldots, n)$ matches a distinct P_j 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 conversion methods use forward rules to define a simple bottom-up parser whose syntactic entities are represented by ground atomic predicates. $T(n,t)$ 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 three conversion methods considered here are presented next. Figure 2 shows the rules of a method based on a parallel composition from smaller fragments that are adjacent to compose a larger one. Figure 3 shows the rules of a method based on a sequential conversion of digits, proceeding from left to right, starting with the delimiter '[' and ending with the delimiter ']'. Finally Figure 4 shows the single rule of a method that uses a sequential implementation of the method shown in Figure 3. The rules have been named consecutively as $R_1, R_2, \ldots, R_6$. In that follows, the user-defined rule $R_i : E_i | C_i \Rightarrow [A_i]P_i$ will be simply designated by its index i instead of R_i for brevity.

$$
\begin{array}{ll}
N_1: & 0 + y = y \Leftarrow \\
N_2: & (x+1) + y = (x+y) + 1 \Leftarrow \\
N_3: & 0 \leq y \Leftarrow \\
N_4: & (x+1) \leq (y+1) \Leftarrow x \leq y
\end{array}
$$

Fig. 1. Natural numbers using backward rules.

$$
\begin{array}{l}
R_1 : T(n,x) \mid \mathit{digit}(x) \Rightarrow [z := \mathit{toInt}(x)]\, N(n,n,z). \\
R_2 : N(n_1,n_2,x), N(n_3,n_4,y) \\
\qquad \mid n_1 \leq n_2, n_2 + 1 = n_3, n_3 \leq n_4 \\
\qquad \Rightarrow [z := x \times 10^{n_4 - n_3 + 1} + y] N(n_1,n_4,z). \\
R_3 : T(n_1, '['), N(n_2,n_3,x), T(n_4, ']') \\
\qquad \mid n_1 + 1 = n_2, n_2 \leq n_3, n_3 + 1 = n_4 \\
\qquad \Rightarrow E(n_1,n_4,x).
\end{array}
$$

Fig. 2. Parallel parser of numbers.

The scheduler selects and applies rules according to the rules S_1, EC_i , AP_i and M_1 shown in Figure 5. Rule S_1 applies to a set of user-defined rules that are in conflict like the set $\{R_3, R_4, R_6\}$. The action of rule S_1 performs a linear search for those userd-defined rules that can be selected. From them, the one selected has the minimum average time of execution. This criterion helps not only to select the historically faster rule but also to avoid oscillations caused by

$$
\begin{array}{l}
R_4 : T(n, \,'[') \Rightarrow M(n, n, 0). \\
R_5 : N(n_1, n_2, x), T(n_3, t) \\
\qquad\qquad \mid n_1 \leq n_2, n_2 + 1 = n_3, digit(t) \\
\qquad\qquad \Rightarrow [z := x \times 10 + toInt(t)]\, N(n_1, n_3, z). \\
R_5 : N(n_1, n_2, x), T(n_3, \,']') \\
\qquad\qquad \mid n_1 \leq n_2, n_2 + 1 = n_3 \\
\qquad\qquad \Rightarrow E(n_1, n_3, x).
\end{array}
$$

Fig. 3. Sequential parser of numbers with three rules.

$$
\begin{array}{l}
R_6 : T(n_1, \,'[') \\
\qquad \Rightarrow \left[\begin{array}{l}
\texttt{int}\, x, t, n, v : \\
\left(\begin{array}{l}
x, n := 0, n_1 + 1; \\
\left(\begin{array}{l}
T(n, t)\texttt{?}; digit(t)\texttt{?}; \\
\quad v := toInt(t); \\
\quad x, n := 10 \times x + v, n + 1; \\
\texttt{retract}(T(n, t))
\end{array}\right)\texttt{*}; \\
t = \,']'\texttt{?}; \\
z, n_2 := x, n
\end{array}\right)
\end{array}\right] E(n_1, n_2, z).
\end{array}
$$

Fig. 4. Sequential parser of numbers with a single rule.

sporadic fluctuations. Note that rule evaluates asserted predicates $ready(i)$ and $average(i, t)$ to determine if rule i can be selected for execution and to obtain the average execution time t, respectively. Once obtained the statistically faster rule m, the predicate $ready(m)$ is discarded to avoid applying the rule once more. A request for applying the faster rule is also established by asserting the predicate $do(m)$. shown in Figure 5.

The rule conflict solver and the rule executor are given by rules EC_i derived from the event-condition parts and rules AP_i derived from the action-postcondition parts of each user-defined rule i. The purpose of rules EC_i and AP_i is to produce the same effect caused by rule $E_i \,|\, C_i \Rightarrow [A_i]\, P_i$ though in this case by mediation of the rule scheduler S. In order to produce this effect, the user-defined rule i is split into two parts at compilation-time, the event-condition and the action-postcondition that are later rejoined at run-time by the scheduler. The event-condition parts $E_i \,|\, C_i$ are embedded in rule EC_i, while the action-postcondition parts $[A_i]\, P_i$ are embedded in rule AC_i, producing in fact two set of rules. The purpose of the rules EC_i is to determine for which of them there are available asserted predicates that unifies with the event part of rule and if such unifier satisfies the condition part of the user-defined rule $E_i \,|\, C_i \Rightarrow [A_i]\, P_i$. If such a unifier exists, the values bound to the variables are orderly passed to the corresponding rule executor in order to instantiate the variables used in the action-postcondition parts of the corresponding rule, producing in this way the same effect.

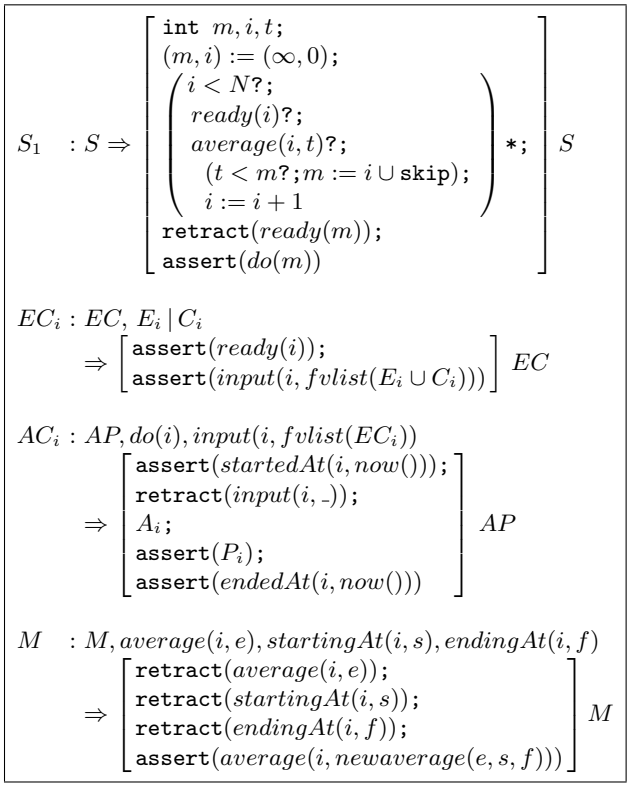

Fig. 5. Scheduler rules.

In rule EC_i, the expression $fvlist(\{T(n,x), digit(x)\}) = [n,x]$, determined at compilation-time, produces the list of variable names in the textual order in which they appear in both the event and the condition parts of rule i. Thus for example, rule EC_1 shown in Figure 6 is obtained from rule R_1 in this manner. where for rule P_1, $fvlist(\{T(n,x), digit(x)\}) = [n,x]$. Note that neither this rule nor any other rule of this set modifies the knowledge base of the asserted predicates as they only obtains the values bound to the variables and evaluates the condition with such values. The asserted predicate $ready(i)$ tells the scheduler that the user-defined rule i is selectable, while the asserted predicate $input(i, vs)$ tells the executor AP_i to bind its orderly list of variables with the list vs of values by means of unification. The purpose of the rule executor AP_i is to perform the action part of the user-defined rule i, given the values passed through the asserted predicate $input(1, [x,n])$ and then binding the variables to their respective values in the so-called input substitution. Then, after the action terminates with a binding of values to variables, the so-called output substitution, rule AP_i asserts the instance of the postcondition $N(n,n,z)$ under the composition of both the input and the output substitutions. For example,

$$EC_1 : EC, T(n,x) \,|\, digit(x) \Rightarrow \begin{bmatrix} \texttt{assert}(ready(1)); \\ \texttt{assert}(input(1,[n,x])) \end{bmatrix} EC$$

$$AP_1 : AP, do(1), input(1,[n,x]) \Rightarrow \begin{bmatrix} \texttt{assert}(startedAt(1, now())); \\ \texttt{retract}(input(1,\_)); \\ z = toInt(x); \\ \texttt{assert}(N(n,n,z)); \\ \texttt{assert}(endedAt(1, now())) \end{bmatrix} AP$$

Fig. 6. From rule P_1, rules EC_1 and AP_1 are extracted and handled by the scheduler.

for the user-defined rule R_1, the rule AP_1, obtained as indicated before, is shown in Figure 6, where the asserted predicate $input(1,[x,n])$ helps to instantiate the variables of the input substitution rule AP_1.

Finally, rule M_1 describes the simple behavior of the monitor. The monitor simply determines the average execution time for the actions of each of the methods described in Figure 2 to 4. The execution time is calculated as the difference between the final time f and the initial time s of execution. The new average is recalculated from the previous one e along with s and f using function $newaverage(e,s,f)$.

4 DLRL Formal Description

An experimental system for DLRL 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 following syntactic categories are built upon the signature (Σ, Ξ):

Terms $T(\Sigma,\Xi)$	$T ::= x \mid c \mid c(T_1,\ldots,T_n)$
Predicates $P(\Sigma,\Xi)$	$P ::= \texttt{false} \mid \texttt{true} \mid T_1 = T_2 \mid p(T_1,\ldots,T_n)$
Goals $G(\Sigma,\Xi)$	$G ::= P \mid G_1 \wedge G_2$
Horn clauses $B(\Sigma,\Xi)$	$B ::= P \mid P \Leftarrow G \mid \forall x.B$
Events $E(\Sigma,\Xi)$	$E ::= P \mid E_1 \ , \ E_2$
Actions $A(\Sigma,\Xi)$	$A ::= \texttt{skip} \mid \texttt{fail} \mid G\texttt{?} \mid (A) \mid A_1\,\texttt{;}\,A_2 \mid A_1 \cup A_2 \mid A\texttt{*} \mid \texttt{int}\ x_1,\ldots,x_n : A \mid x_1,\ldots,x_n := T_1,\ldots,T_n \mid \texttt{assert}(p(T_1,\ldots,T_n)) \mid \texttt{retract}(p(T_1,\ldots,T_n))$
Modal actions $A(\Sigma,\Xi)$	$M ::= P \mid [A]\ M \mid \langle A\rangle\ M$
Forward rules $F(\Sigma,\Xi)$	$F ::= M \mid E \,|\, G \Rightarrow M \mid \forall x.F$

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. The action `assert`$(p(T_1, \ldots, T_n))$ introduces the predicate $p(T_1, \ldots, T_n)$ into the knowledge base making it valid, whereas the action `retract`$(p(T_1, \ldots, T_n))$ removes the predicate $p(T_1, \ldots, T_n)$ from the knowledge base making it invalid. Being these actions borrowed from standard Prolog, they are not intended to be blocking actions, like in other Linda-like coordination models, and therefore they can only succeed or fail. 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 $x\sigma$:

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

where notation $[\sigma_{:=}]$ stands for the multiple assignment $x_1, \ldots, x_n := T_1, \ldots, T_n$ obtained from 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.

The *backward computation* relation $\triangleleft \subset G(\Sigma, \Xi) \times (\Xi \to T(\Sigma, \Xi))$ 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 Figure 7.

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

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

Fig. 7. Operational semantics of backward and forward rules.

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 $E_1, \ldots, E_n, P \mid G \Rightarrow [A]\, P'$, with $n > 0$, can be selected for deducing the ground predicate $P'\iota o$ from $P\sigma$ only if the following three conditions hold: (i) there are n ground predicates $P_1, \ldots, P_n$ already asserted in the shared 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 $\sigma_1 \cdots \sigma_n\sigma$ of the n substitutions along with σ satisfies the goal G. Whenever these conditions are met, the forward rule can be applied. With the ground substitution $\sigma_1 \cdots \sigma_n\sigma$ starts the backward computation rule that leads to the input substitution ι with bindings for the new

variables that G may introduce. The instance under ι of the modal action $[A]\,P$ is then executed following the standard interpretation of the action connectives [4]. Assuming that A terminates starting with the initial values given by ι, the postcondition P' becomes satisfied by the substitution ιo, where o is the output substitution produced by A on the output variables. 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 system would appear non-responsive until another predicate assertion were eventually produced in the shared memory.

5 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 `DLRL` 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. P. Horn: Autonomous Computing: IBMs perspective on the state of Information Technology. IBM Research (2001)
2. IBM: An architectural blueprint for autonomic computing. Tech. Rep., IBM (2003)
3. P. Ciancarini: Coordinating Rule-Based Software Processes with ESP. ACM Trans. on Software Engineering and Methodology, 2(3), 203–227 (1993)
4. D. Harel, J. Tiuryn, D. Kozen: Dynamic Logic. Cambridge, MA, USA, MIT Press (2000)
5. O. Olmedo-Aguirre, G. Morales-Luna: Indeed: Interactive Deduction on Horn Clause Theories. In: Proceedings IBERAMIA 2002, LNAI, vol. 2527, pp. 151–160, Springer-Verlag (2002)
6. J. O. Olmedo-Aguirre, G. Morales-Luna: A Dynamic Logic-based Modal Prolog. In: Proceedings MICAI 2012, CPS IEEE Computer Society, pp. 3–9 (2012)
7. J. O. Olmedo-Aguirre: DL Prolog: Another Unifying Programming Language. Research in Computing Science, vol. 60, pp. 23–33 (2012)
8. V.A. Saraswat: Concurrent Constraint Programming. In: Records of 17th ACM Symposium on Principles of Programming Languages, San Franciso, CA, pp. 232–245 (1990)
9. P. Wegner: Interactive Software Technology. In: CRC Handbook of Computer Science and Engineering (1996)
10. L. Wos, R. Overbeek, E. Lusk, J. Boyle: Automated Reasoning. Introduction and Applications. McGraw-Hill (1992)
11. L. Wos, G. Pieper: A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning. World Scientific Publishing Co. (1999)