LPS - A Logic-based Production System Framework

Robert Kowalski and Fariba Sadri
Department of Computing
Imperial College London
{rak, fs}@doc.ic.ac.uk
12 January 2009

Abstract

Production rules and logic programs can be combined in a single logic-based framework. The framework gives both an operational and model-theoretic semantics to production rules, as well as to logic programs extended with a database of facts that is modified by destructive assignment.
The model-theoretic semantics is obtained by separating the production system working memory into facts and goals. Logic programs are used to define ramifications of the facts and to reduce goals to sub-goals, including actions. The execution of actions generates a sequence of states, which serves as a candidate model of the production rules.

1 Introduction

Production rules and logic programs are among the main kinds of knowledge representation in Artificial Intelligence. Despite the fact that both represent knowledge in the form of rules, there has been little attempt to understand the relationships between them.

In this paper we present a framework that combines the two kinds of rules, eliminating their overlap, and reconciling their differences. In the course of doing so, we obtain a logic-based production system framework (LPS), in which production rules have a model-theoretic semantics, and logic programs destructively manipulate a database of facts.

LPS is inspired both by the abductive logic programming (ALP) agents of [Kowalski and Sadri, 1999] and by the teleo-reactive (TR) programs of [Nilsson 1994].

1.1 Confusions about rules in AI

Rules in production systems have the form *conditions* → *actions* and look like conditionals in logic. Indeed, the most popular textbook on Artificial Intelligence [Russell and Norvig 2003] views production rules as just conditionals used to reason forward (page 286). However, one of the main textbooks on Cognitive Science [Thagard, 2005] argues that production rules are not conditionals:

"Rules are if-then structures …very similar to the conditionals…, but they have different representational and computational properties." (page 43). "Unlike logic, rule-based systems can also easily represent strategic information about what to do. Rules often contain actions that represent goals, such as *IF you want to go home and you have bus fare, THEN you can catch a bus*." (page 45).

[Thagard, 2005] characterizes Prolog as "a programming language that uses logic representations and deductive techniques" (page40). [Simon,1999], on the other hand, includes Prolog "among the production systems widely used in cognitive simulation." (page 677).

The relationship between logic, logic programming and production systems has recently become the focus of attention as the result of the W3C Rule Interchange Format (RIF) Working Group recommendations for rules interchange (see http://www.w3.org/2005/rules/wiki/RIF_Working_Group).

The RIF proposals include a Horn clause language, RIF-BLD and a draft production rule language RIF-PRD. The definition of RIF-BLD has a model-theoretic semantics, but no operational semantics, whereas the definition of RIF-PRD has only an operational semantics. Both proposals ignore the fact that one of the main uses of both Horn clauses and production rules is to reduce goals to sub-goals.

Indeed, it is the elimination of this overlap that is one of the main features of our proposed framework.

In the remainder of the paper, we provide background material, describe the operational and model-theoretic semantics of LPS, present soundness results and then discuss conflict resolution and teleo-reactive behaviour.

2 Background

2.1 Production systems

A typical production system [Newell, 1973; Simon, 1999] combines a "working memory" consisting of atomic sentences, with condition-action rules of the form *conditions C* → *actions A*. Condition-action rules are also called production rules, just plain rules, or if-then rules.

Production rules are executed by means of a *cycle*, which uses *forward chaining* to match facts in the working memory with conditions of production rules, deriving the actions of the rules as *candidates* to be executed. If more than one rule is *triggered*, then *conflict-resolution* is performed to decide which rules should be *fired,* executing their actions by adding and deleting facts in the working memory. The

repeated execution of this *production system cycle* generates a sequence of state transitions of the working memory.

Conflict resolution can be performed in many different ways. Typical conflict resolution strategies include giving priority to more specific rules over more general rules, and writing rules in the order in which they are to be executed. For example, a timid agent might try the following rules in their given order:

someone attacks me → *try to escape*
someone attacks me → *attack them back.*

A more aggressive agent might try them in reverse order.

Compared with classical logic, which has both a model-theoretic semantics and diverse proof procedures, production systems arguably have only an "operational semantics" in the form of the production system cycle.

LPS has both a model-theoretic semantics and an operational semantics similar to the production system cycle, inspired by the ALP agent framework of [Kowalski and Sadri, 1999]. However, unlike the ALP agent cycle, the LPS cycle is closed to external updates and actions. This simplifies the formulation of the model-theoretic semantics and is consistent with the W3C draft proposal for RIF-PRD.

2.2 Three kinds of production rules

Arguably, in practice, there are three kinds of production rules: condition-action rules that implement stimulus-response associations; forward chaining logic rules; and goal-reduction rules.

The first kind of rule is arguably the most characteristic of production systems, and is responsible for their general characterisation as *condition-**action*** rules. They typically have implicit or *emergent* goals. For example, the rule

it is raining → *cover yourself with an umbrella.*

has the implicit goal *to stay dry*.

The second kind of rule, for example

human X → *mortal X*

adds to the working memory a fact that is a logical conclusion of other facts in the working memory. It is probably this kind of rule that gives the impression that production rules are just conditionals used to reason forward.

However, it is the third kind of rule, exemplified by the rule for taking a bus if you want to go home for the weekend, that causes the most mischief. Such rules have the form

goal G and conditions C → *add H as a goal.*

Forward chaining with such goal-reduction rules behaves the same as backward reasoning with the logic programming clause $G \leftarrow C, H$.

To represent goal-reduction in production rule form, the working memory contains, in addition to *"real" facts,* which describe the current state of some domain or world model, and *goal facts,* which describe some future desired state of the world. Similarly, production rules have both *"real" actions*, which update the world model, and *goal manipulation actions*, which maintain the relationship between goals and sub-goals.

Both types of actions are expressed as additions and deletions of facts in the working memory. There are no higher-level structures to ensure that these additions and deletions are "meaningful".

The logic-based production system LPS distinguishes between the two kinds of facts and the two kinds of actions. It uses structured definitions of actions in terms of their pre-conditions and the facts that that they initiate and terminate. It uses a structured representation of goals which distinguishes between goals that are conjoined together (joined by *and*) and sub-goals that are disjoined (related by *or*). It represents stimulus-response association rules by production rules, but represents forward chaining logic rules and goal-reduction rules by logic programming clauses.

2.3 Logic programming

Logic programs and production systems both suffer from their own confusions. Whereas production rules suffer from confusion about whether goals are emergent or explicit, logic programs suffer from confusion about whether they are declarative or procedural.

Logic programs are sets of *clauses* of the form:

$H \leftarrow B_1, \ldots, B_n$

where the comma stands for *and*, the conclusion H is an atomic formula (*atom*, for short) and the conditions B_i are *literals* (atoms or the negations of atoms). All variables are implicitly universally quantified in front of the clause. *Horn clauses* are the special case where all the conditions are atomic. *Facts* are the special case where there are no variables and n=0. Sometimes clauses that are not facts are also called rules, inviting confusion with production rules.

For example, the three sentences:

you will go home for the weekend ←
you have the bus fare, you catch a bus,
not something goes wrong with the bus journey.
you will go home for the weekend ←
you have the bus fare, you catch a bus.
you have the bus fare.

are a clause, a Horn clause and a fact, respectively.

Goal clauses are conjunctions of literals. The literals in a goal clause represent individual *goals*. All variables are implicitly existentially quantified in front of the goal clause, and the intention is to find an instantiation of the variables that makes all of the goals hold compatibly. This is normally performed by means of backward reasoning.

Given a goal clause $L_1, \ldots, L_i, \ldots, L_n$, backward reasoning is performed by applying the resolution rule [Robinson, 1965], selecting some atomic goal L_i, resolving it with some clause $H \leftarrow B_1, \ldots, B_m$ whose conclusion H unifies with L_i and generating a new goal clause

$(L_1, \ldots, L_{i-1}, B_1, \ldots, B_m, L_{i+1}, \ldots, L_n)\sigma$,

where σ is the most general unifier of H and L_i. This "selected" resolution reduces the goal L_i to the subgoals $(B_1, \ldots, B_m)\sigma$. If H is a fact, then resolution solves the goal without introducing sub-goals, If m=0 and n=i=1, then the new goal clause is equivalent to *true*.

Backward reasoning treats clauses as procedures:

to show/solve H, show/solve B_1 *and ... and* B_m.

For example, backward reasoning turns the clause:

you will go home for the weekend ←

you have the bus fare, you catch a bus.

into the procedure:

To go home for the weekend,
have the bus fare, and catch a bus.

A negative goal, *not G*, is solved by negation as failure: *not G* succeeds if and only if backward reasoning with goal *G* does not succeed.

For Horn clauses, the demonstration that the declarative and procedural interpretations coincide can be traced to the completeness result for SL-resolution [Kowalski and Kuehner, 1971]. Moreover, the declarative interpretation that a goal clause is true in all models is equivalent to its being true in a unique, minimal model [van Emden and Kowalski, 1976].

Similar results hold for logic programs with negation as failure [Apt and Bol, 1994]. In particular, for the model-theoretic semantics of LPS, we use the fact that any locally stratified set of clauses has a unique minimal model [Przymusinski, 1988]. Intuitively, a logic program is locally stratified if it does not contain recursion through negation. For example, the program *{p←not q, q←not r}* is locally stratified, whereas *{p←not p}* is not.

3 LPS – a logic-based production system

Given sets of clauses $\boldsymbol{A}$ and $\boldsymbol{L}$, set of rules $\boldsymbol{P}$, initial "world model" $\boldsymbol{W_0}$ and initial set of goal clauses $\boldsymbol{G_0}$, the framework determines a sequence of state transitions:

$$<\boldsymbol{W_0}, \boldsymbol{G_0}>, a_0, \ldots, <\boldsymbol{W_i}, \boldsymbol{G_i}>, a_i \ldots$$

$\boldsymbol{W_i}$ is a set of *facts* representing the current state of some "world model". These are, in effect, the extensional predicates of a deductive database. E.g. *on(a, b).*

a_i represents an action *a* executed in world $\boldsymbol{W_i}$, transforming $\boldsymbol{W_i}$ into $\boldsymbol{W_{i+1}}$. Actions *a* are specified by clauses in $\boldsymbol{A}$ defining the predicates *initiates(a, p)* and *terminates(a, p)* and *precondition(a, p)*, which, respectively, determine the facts *p* that they initiate and terminate, and that must hold as preconditions for their successful execution in $\boldsymbol{W_i}$. We impose the restriction that $\boldsymbol{A}$ is locally-stratified.

The transformation of $\boldsymbol{W_i}$ is implemented by destructive assignment. So $\boldsymbol{W_{i+1}} = \boldsymbol{W_i} \cup add(a_i) - delete(a_i)$ where:

$add(a_i)$ = {*p: initiates(a, p)* holds in $\boldsymbol{W_i} \cup \boldsymbol{A} \cup \boldsymbol{L}$}

$delete(a_i)$ = {*p: terminates(a, p)* holds in $\boldsymbol{W_i} \cup \boldsymbol{A} \cup \boldsymbol{L}$}

The set $\boldsymbol{L}$ contains three kinds of clauses: (1) Clauses that define state-independent predicates, like *human(X) ← mortal(X)*. (2) Clauses that define intentional predicates, whose values change as ramifications of the extensional predicates, like *above(X, Y) ← on(X, Z), above(Z, Y)*. (3) Clauses that define plans, including, but not restricted to plans in AgentSpeak(L) [Rao, 1996], like

quenched-thirst ← go-to-fridge, open-fridge,
get-drink, open-drink, drink.

Clausal syntax does not indicate the intended sequence of actions in plans. This can be remedied by introducing state variables, as in section 3.2. However, in the meanwhile, the reader may assume that actions are executed in the order they are written.

For simplicity, we impose the restriction that clauses of the second and third kind do not contain negation. However, the restriction can be removed, as in the ALP agent cycle [Kowalski and Sadri, 1999], at the expense of complicating the operational semantics. In any case, we require that clauses of the first kind are locally stratified.

To achieve their desired effect, the planning clauses in $\boldsymbol{L}$ should follow from the action specifications in $\boldsymbol{A}$, similar to the way that compound actions in [Shanahan, 2000] follow from the effects of their component actions. For example, the action *drink* in the plan for *quenched-thirst* should initiate the fact *quenched-thirst.* Moreover, earlier actions in a plan should achieve preconditions for later actions in the plan.

In contrast with [Shanahan, 2000], we do not require that plans have explicit conditions expressing that preconditions persist. Instead, the LPS cycle checks that action preconditions hold at the time of their execution, and exploits the use of destructive assignment to obtain persistence implicitly. If a precondition fails to persist, the cycle allows actions to be re-executed, reinstating their effects.

$\boldsymbol{P}$ is a set of production rules of the form *conditions* → *goals.* The intention is that $\boldsymbol{P}$ should be true in a minimal model $\boldsymbol{M}$ determined by the (possibly infinite) history

$\boldsymbol{W_0}, a_0 \ldots, \boldsymbol{W_i}, a_i \ldots$ augmented by $\boldsymbol{L}$.

(The minimal model is defined in sections 3.2 and 3.3.) The *conditions* of production rules are conjunctions of literals in the predicates of $\boldsymbol{L}$ and $\boldsymbol{W_i}$. In addition, one of the conditions of a rule can refer to the most recent action a_{i-1}, as in event-condition-action rules [Paton and Diaz, 1999]. The *goals* are conjunctions of atoms in the predicates of $\boldsymbol{L}$, $\boldsymbol{W_i}$ and $\boldsymbol{A}$, including actions to be performed in the future.

All variables occurring in the *conditions* of a rule are implicitly universally quantified in front of the rule. However, all variables occurring only in the *goals* of a rule are existentially quantified in front of the *goals*. For example,

X attacks me → Y defends me against X

is understood as

$\forall X$(*X attacks me* → $\exists Y$(*Y defends me against X*)).

$\boldsymbol{G_i}$ is a set of goal clauses, each of which represents a *partial plan* for achieving the goals generated by the production rules so far. The intention is that, for every $\boldsymbol{G_i}$, one of the goal clauses in $\boldsymbol{G_i}$ should be true in the model $\boldsymbol{M}$. $\boldsymbol{G_o}$ can be the empty set, as is typical of production systems.

3.1 The Operational Semantics

The operational semantics is a variant of the production system cycle, but can also be viewed as a model generator, which attempts to construct a model $\boldsymbol{M}$, in which $\boldsymbol{P}$ is true.

Given a current state $<\boldsymbol{W_i}, \boldsymbol{G_i}>$ where $i \geq 0$ and action a_{i-1} (if $i>0$), the LPS cycle executes the following steps:

Step 1. Every instance *conditions* σ → *goals* σ of a rule in $\boldsymbol{P}$ such that *conditions* σ hold in $\boldsymbol{W_i} \cup \boldsymbol{L} \cup \{a_{i-1}\}$ is *fired*, conjoining *goals* σ to every goal clause in $\boldsymbol{G_i}$.

Step 2. Backward reasoning is used an indefinite (possibly zero) number of times, to reduce non-action goals in $\boldsymbol{G_i}$ to sub-goals. For this purpose, facts in $\boldsymbol{W_i}$ are treated like clauses in $\boldsymbol{L}$, to solve goals in the extensional predicates.

The cycle *terminates successfully* if the empty goal clause is derived. Otherwise, backward reasoning continues until

one or more *candidate actions* are derived. If it is not possible to derive any candidate actions, then the cycle is *unsuccessful* (possibly non-terminating) and no further steps or iterations should be performed.

A *candidate action* is an action whose preconditions all hold in $\boldsymbol{W_i} \cup \boldsymbol{L}$ and that needs to be executed before all other actions are executed and goals are achieved in the same goal clause.

Step 3. *Conflict resolution* is performed to select a single candidate action *a* in some goal clause *C*.

Step 4. The selected action is executed, letting $a_i = a$, and updating $\boldsymbol{W_i}$ to $\boldsymbol{W_{i+1}}$, destructively adding the new facts in *add(a)* and deleting the old facts in *delete(a).*

A new goal clause is generated by resolving the selected goal *a* in *C* with the action a_i treated like a clause in $\boldsymbol{L}$. The resulting set of goal clauses is the new set $\boldsymbol{G_{i+1}}$.

Repeat the cycle.

Note:

- In step 1, finding a substitution σ such that *conditions* σ holds in $W_i \cup L \cup \{a_{i-1}\}$ can be performed by means of either forward or backward reasoning or by a combination of the two, exploiting the declarative nature of clauses.
- In step 2, treating the facts in $\boldsymbol{W_i}$ like clauses allows the opportunistic exploitation of the side effects of actions.
- In step 2, a form of conflict resolution is involved in deciding what goals to reduce, what clauses to use, and how long to continue backward reasoning. In particular, if a candidate action was derived in a previous cycle, then backward reasoning need not be performed at all.
- Checking that the preconditions of a candidate action *a* hold can instantiate variables in *a*. For example, checking the precondition *clear(X)* of the action *move(b, X)* instantiates *X* to *c* if *clear(c)* holds in $\boldsymbol{W_i} \cup \boldsymbol{L}$.
- In step 3 conflict resolution amounts to selecting between different ways of solving a goal. We will discuss this issue in greater detail in section 3.4.
- In step 4 treating the actions a_i like clauses allows the execution of a_i to solve different action goals. Retaining the goal clause *C* containing *a*, allows the action to be retried if its required effects are deleted by other actions later.
- There are numerous optimisations that can be made in an implementation of the operational semantics. These include deleting a goal clause if:
  - all of its resolvents have been generated,
  - it is subsumed by another goal clause,
  - it contains an action to be executed earlier,
  - it contains an action to be executed in the current cycle, but its preconditions do not hold,
  - it contains a goal that should hold in the current cycle, but does not.

The last three optimisations require an explicit representation of state.

3.2 Representation with explicit state

For simplicity, and because it is closer to conventional production system languages, the representation we have used until now does not have explicit state. This has the additional advantage that it makes it more obvious how to transform the world model from one state to another using destructive assignment. However, it has the disadvantage that it is ambiguous about states in rules and clauses. For example, the following clauses are ambiguous:

above(X, Y) ← *on(X, Z), above(Z, Y)*
quenched-thirst ← *go-to-fridge, open-fridge, get-drink, open-drink, drink*

In the first clause, the conclusion and conditions are intended to hold in the same state. But in the second clause, the goal and sub-goals are intended to hold in sequence.

This ambiguity can be resolved either by introducing explicit state variables, as additional arguments of predicates, or by employing a special-purpose notation for the three kinds of clauses in $\boldsymbol{L}$. The special-purpose notation can be compiled into the explicit state variable representation, in the same way that definite clause grammars are compiled into logic grammars in Prolog. In the sequel we use the representation with explicit state both for the sake of clarity and because it is necessary for the model-theoretic semantics.

With explicit representation of state, predicates *p* in $\boldsymbol{W_i}$ are written in the form *holds(p, i)* or *p(i).* Actions a_i are written as *happens(a, i)* or as *a(i)*. Predicates in rules and clauses are written similarly with explicit state arguments.

To represent rules and clauses with explicit states, we need inequalities. For example:

thirsty(I) → *quenched-thirst(I') & I ≤ I'*
above(X, Y, I) ← *on(X, Z, I), above(Z, Y, I)*
quenched-thirst(I5+1) ← *go-to-fridge(I1), open-fridge(I2), get-drink(I3), open-drink(I4), drink(I5), I1 < I2 < I3 < I4 < I5*

Here the inequalities impose a total ordering on the actions. However, they can also be used to impose a partial order.

$\boldsymbol{L}$ contains clauses defining the < and ≤ relations. For the purposes of specifying both the operational and model-theoretic semantics, we need not be concerned with efficiency. It is sufficient, therefore, to assume that $\boldsymbol{L}$ contains such defining clauses as:

0 < I *I ≤ J* ← *I < J*
I+1 < J+1 ← *I < J* *I ≤ J* ← *I = J*

The operational semantics needs little modification to deal with explicit representation of state. In particular, step 1 of the cycle needs no change, because the conditions of production rules are all verified relative to the current state $\boldsymbol{W_i}$.

The main impact on the operational semantics of having explicit states is that goal clauses in $\boldsymbol{G_i}$ now contain inequality goals relating state variables. These inequalities are treated just like any other goals and their variables are treated like any other variables. However, in an efficient implementation, inequalities would be processed as constraints, rather than by the clauses that define them.

For the model-theoretic semantics, we need a declarative specification of state transitions. This can be given by clauses similar to the situation calculus and event calculus, with explicit state arguments for *initiates* and *terminates:*

holds(P, I+1) ← *happens(A, I), initiates(A, I, P)*
holds(P, I+1) ← *holds(P, I), happens(A, I), not terminates(A, I, P)*

Call these clauses ***S/EC***. These clauses are needed only for the model-theoretic semantics and are not in ***L***.

The clauses ***A*** ∪ ***S/EC*** constitute a theory that can be used for planning from *first principles*. The planning clauses in ***L*** are intended for planning from *second principles*.

3.3 Model-theoretic semantics

Given some selection strategy for reducing goals to subgoals and some strategy for conflict resolution, the cycle determines a (possibly infinite) sequence $<\boldsymbol{W_0}, \boldsymbol{G_0}>, a_0, \ldots <\boldsymbol{W_i}, \boldsymbol{G_i}>, a_i, \ldots$ of states and actions.

Let $\boldsymbol{W_i}^*$, a_i^*, $\boldsymbol{G_i}^*$, $\boldsymbol{L}^*$, $\boldsymbol{A}^*$ and $\boldsymbol{P}^*$, respectively, be the representations of $\boldsymbol{W_i}$, a_i, $\boldsymbol{G_i}$, $\boldsymbol{L}$, $\boldsymbol{A}$ and $\boldsymbol{P}$ with explicit states. With this notation, the set $\sum$ of sentences

$\boldsymbol{W_0}^* \cup \ldots \cup \boldsymbol{W_i}^* \cup \ldots \{a_0^*, \ldots, a_i^*, \ldots\} \cup \boldsymbol{L}^* \cup \boldsymbol{A}^* \cup \boldsymbol{S/EC}$

is a locally-stratified logic program. Therefore, $\sum$ has a unique minimal model **M**. This model is like a Kripke possible worlds structure embedded in a single model. The following two theorems are "true by construction". For lack of space we do not present their proofs.

Theorem 1: (Finite case) If the operational semantics terminates successfully, then $\boldsymbol{G_0} \cup \boldsymbol{P}^*$ is true in **M**.

Theorem 2: (General case) $\boldsymbol{G_0} \cup \boldsymbol{P}^*$ is true in **M** if and only if for every i there exists a $k \geq i$, such that $\boldsymbol{G_i}^*$ is true in $\boldsymbol{W_0}^* \cup \ldots \cup \boldsymbol{W_k}^* \cup \{a_0^*, \ldots, a_{k-1}^*\}$.

Theorems 1 and 2 are soundness results for the operational semantics viewed as a model generator. It is also possible to prove completeness results, which ensure that, if there is a model then the operational semantics can construct it. These results are beyond the scope of this paper.

3.4 Conflict Resolution

The LPS cycle performs conflict resolution in the choice of actions to be executed, rather than in the choice of rules that are fired. Therefore, it is limited to choosing *how* goals are achieved, rather than whether they are achieved at all.

As a consequence, conventional ways of writing production rules are not always acceptable. For example, given a state in which *someone attacks me,* both the operational semantics and the model-theoretic semantics require that all of the goals of the rules:

someone attacks me → *attack them back*
someone attacks me → *escape*

are achieved in future states. Given the intended interpretation, not only is it unlikely that all the goals can be achieved, but just as importantly, it is unlikely that this is what the writer of the rules intended.

To obtain the intended effect of the rules in LPS we need to rewrite them, making their intended higher-level goal and the alternative ways of achieving it explicit. For example:

someone attacks me → *protect myself*
protect myself ← *attack them back*
protect myself ← *escape*

More generally, given rules:

$C \rightarrow G_1 \qquad C \rightarrow G_2$

where the intention is that when C is true one of G_1 and G_2 is to be achieved, rewrite them in the form:

$C \rightarrow G$
$G \leftarrow G_1 \qquad G \leftarrow G_2$

where G is the higher-level goal, which holds if G_1 or G_2.

There are also other, less problematic cases, where conflict resolution can be dealt with simply by assigning different priorities to the rules. In the particular case where the rules are written in order of priority:

$C_1 \rightarrow G_1 \quad C_2 \rightarrow G_2 \quad \ldots \quad C_n \rightarrow G_n$

conflict resolution can be incorporated by adding extra conditions to the rules:

$C_1 \rightarrow G_1 \quad C_2$, *not* $C_1 \rightarrow G_2 \quad \ldots$
C_n , *not* C_1 , ... , *not* $C_{n-1} \rightarrow G_n$

For example, the rules:

someone attacks me → *protect myself*
hungry → *eat*

can be replaced by

someone attacks me → *protect myself*
hungry , not someone attacks me → *eat*

There is also a more problematic case, where conflict resolution is used to deal with *refraction*, preventing the same rule from firing repeatedly when its conditions continue to hold in successive states. For example, the rule

thirsty(I) → *quenched-thirst(I'), I* ≤ *I'*

will fire repeatedly and redundantly in all states in which the condition *thirsty(I)* continues to hold.

This can be treated by replacing the condition by the action (or event) that initiates it, in this case by the rule

become-thirsty(I) → *quenched-thirst(I'), I* ≤ *I'*

More generally, to avoid firing the same rule

C(I)→*G(I'), I*≤*I'*

unnecessarily, replace it by

A(I)→*G(I'), I*≤*I'*,

if *C(I)* is an extensional predicate and *initiates(A, I, C).* A similar approach can be used when *C(I)* is an intensional predicate.

3.5 Teleo-Reactive Behaviour

The flexibility provided by the combination of production rules and logic programming clauses can also be used to obtain more flexible, opportunistic planning and problem-solving behaviour. For example, consider the clause:

quenched-thirst(I5+1) ← *go-to-fridge(I1),*
open-fridge(I2), get-drink(I3), open-drink(I4),
drink(I5), I1 < I2 < I3 < I4 < I5

The clause expresses an inflexible plan, in which all the actions need to be executed in the given order, whether or not they are all necessary. A more flexible plan, involving the same actions, can be obtained, following the style of teleo-reactive programs [Nilsson, 1994], by making the desired effects of the actions explicit:

quenched-thirst(I+1) ← *opened-drink(I), drink(I)*
opened-drink(I+1) ← *have-drink(I), open-drink(I)*
have-drink(I+1) ← *near-drink(I), get-drink(I)*
near-drink(I+1)← *near-fridge(I), open-fridge(I)*
near-fridge(I+1) ← *go-to-fridge(I)*

In this representation, earlier steps of the plan need not be executed, if all the preconditions already hold for some later step of the plan. This may be the case already in the state where the goal *quenched-thirst* was first introduced, or it

may arise as a side-effect of executing some other action, in pursuit of some other goal.

In general, instead of writing plans in the form

$G_n \leftarrow A_1, A_2, \ldots, A_n$

they can be written in the more flexible form:

$G_n \leftarrow G_{n-1}, A_n \quad \ldots \quad G_2 \leftarrow G_1, A_2 \qquad G_1 \leftarrow A_1$

where G_i is initiated by A_i and is a precondition of A_{i+1}. Note that the more general case, where an action has several preconditions, is straight-forward.

Another feature of TR programs, which we also obtain in LPS, is the ability to recover from failure. This is obtained by having the ability to re-execute actions and reinstate their effects.

3.6 Conclusions

LPS attempts to reconcile production systems and logic programs without reducing one to the other, building on the strengths of each. It retains the use of production rules for stimulus-response associations, but uses logic programs to obtain the effect of forward chaining logic rules and goal-reduction rules. It gives a model-theoretic semantics for production rules and a logical justification for destructively updating a database of facts.

To our knowledge, these contributions are novel. Other authors have explored the relationship between production rules and logic programs. [Raschid, 1994] and [Flesca and Greco, 2001] both translate production rules into logic programs and give them a model-theoretic semantics. But they do not distinguish between different kinds of facts, different kinds of actions, and different kinds of rules.

LPS can be viewed as a special case of the ALP agent language and cycle. The ALP agent model and its associated proof procedure [Fung and Kowalski, 1997] have a number of additional features, which can be exploited to extend LPS. These extensions include:

- rules with conditions that refer to the past history of actions and world states,
- planning clauses with conditions that are rules,
- negative goals solved by actions that terminate facts,
- deadlines for the achievement of goals.

Acknowledgments

We are grateful to Ken Satoh, Luis Moniz Pereira, Harold Boley, Thomas Eiter and Keith Stenning for helpful discussions.

References

[Apt and Bol, 1994] Krzysztof Apt and Roland Bol. Logic Programming and Negation: A Survey. *J. Log. Program.* 19/20: 9-71, 1994.

[van Emden and Kowalski, 1976] Maarten van Emden and Robert Kowalski. The Semantics of Predicate Logic as a Programming Language *JACM* , 23(4):733-742.

[Flesca and Greco, 2001] Sergio Flesca and Sergio Greco. Declarative semantics for active rules *Theory and Practice of Logic Programming*, 1(1):43-69.

[Fung and Kowalski, 1997] The IFF Proof Procedure for Abductive Logic Programming. *Journal of Logic Programming.*

[Kowalski and Kuehner, 1971] Robert Kowalski and Donald and Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2: 227-60.

[Kowalski and Sadri, 1999] Robert Kowalski and Fariba Sadri. From Logic Programming towards Multi-agent Systems. *Annals of Mathematics and Artificial Intelligence.* Vol. 25 391-419.

[Newell, 1973] Alan Newell. Production Systems: Models of Control Structure. In W. Chase (ed): *Visual Information Processing* 463-526 New York: Academic Press 463-526.

[Nilsson, 1994] Nils Nilsson. Teleo-reactive programs for agent control, *Journal of Artificial Intelligence Research*, 1, 1994, 139-158.

[Paton and Diaz, 1999] Paton N. and Diaz O. Active database systems, ACM Computing Surveys, 31(1):63-103, 1999.

[Przymusinski, 1988] Theodor Przymuszynski. On the Declarative Semantics of Deductive Databases and Logic Programs, in *Foundations of Deductive Databases and Logic Programming*, ed Minker J., Morgan Kaufman (1988), pp 193-216.

[Rao, 1996] Anand Rao. Agents Breaking Away, In *Lecture Notes in Artificial Intelligence, Volume 1038,* (eds Walter Van de Velde and John W. Perrame) Springer Verlag, Amsterdam, Netherlands.

[Raschid, 1994] Loquila Raschid. A Semantics for a Class of Stratified Production System Programs. *Journal of Logic Programming.* 21(1).

[Robinson, 1965] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *JACM 12*. Jan. 1965, 23-41.

[Russell and Norvig 2003] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.),* Upper Saddle River, NJ: Prentice Hall.

[Shanahan, 2000] An Abductive Event Calculus Planner, *The Journal of LogicProgramming*, vol. 44, 207–239.

[Simon, 1999] Herbert Simon. Production Systems. In Wilson, R. and Keil, F. (eds.): *The MIT Encyclopedia of the Cognitive Sciences.* The MIT Press. 676-677.

[Thagard, 2005] Paul Thagard. *Mind: Introduction to Cognitive Science.* Second Edition. MIT Press.