Towards a Logic-based Production System Language

Robert Kowalski and Fariba Sadri

Department of Computing, Imperial College London, 180 Queens Gate, London SW7 2AZ

{rak, fs}@doc.ic.ac.uk

Abstract. In this paper we present a system, called LPS, that combines production rules and logic programs in a single logic-based framework. The framework gives both an operational and model-theoretic semantics to production rules, and 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 macro-actions, ramifications of the facts and to reduce goals to sub-goals, including actions. The execution of actions generates a sequence of database states. In the case of successful termination, this sequence, taken as a whole, serves as the basis for a model of the production rules.

Keywords – logic programming, production systems, model-theoretic semantics

1 Introduction

what is production system and why LPS

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 seems to have been little attempt to study the relationships between them. In this paper we present a framework that combines the two kinds of rules, eliminates their overlap, and reconciles their differences. For this purpose, we define a logic-based framework (LPS), in which production rules have a model-theoretic semantics, and logic programs destructively manipulate a database of facts.

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 [13] views production rules as just conditionals used to reason forward (page 286). However, one of the main textbooks on Cognitive Science [14] maintains that "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 for the weekend and you have bus fare, THEN you can catch a bus*." (page 45).

Thagard [15] characterizes Prolog as "a programming language that uses logic representations and deductive techniques". However, Simon [14] includes Prolog "among the production systems widely used in cognitive simulation."

The relationship between logic, logic programming and production systems has recently become the focus of attention in the W3C RIF Working Group drafts on rule 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. RIF-BLD has model-theoretic, but no operational semantics. Although the conditions of production rules in RIF-PRD have model theoretic semantics, production rules themselves have 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 goals of LPS.

Whereas production systems manipulate a working memory that combines both facts and goals, and logic programs manipulate only goals, LPS separates facts from goals and treats them differently. The facts are updated destructively by structured actions, and the goals are updated by firing production rules and by goal-reduction. LPS has both an operational semantics and a model-theoretic semantics.

In the remainder of the paper, we present background, the syntax, operational semantics and model-theoretic semantics of LPS, and discuss conflict resolution. We assume the reader is familiar with the basic concepts of logic programming.

1.2 Motivating Example

We consider an online shopping scenario, similar to the RIF-PRD draft running example. The following informal rules and clauses illustrate the LPS approach.

Some condition-goal rules:

When a customer logs in, he is welcomed.

When a customer checks out, payment is taken, and confirmation is issued.

Some logic programming clauses (A customer is w*elcomed* with an appropriate offer):

A customer is welcomed if his status is new and there is a promotional item in the store and the promotional item is offered to him.

A customer is welcomed if his status is gold and there is a promotional item in the store and there is an item in his profile and the item is similar to the promotional item and the promotional item is offered to him.

2 Background

2.1 Production systems

A typical production system combines a *working memory* of atomic sentences, also called *facts*, with condition-action rules of the form *condition→ actions.* Condition-action rules are also called production rules, if-then rules, or just plain rules.

In RIF-PRD and many other production systems, the working memory can be viewed as a model-theoretic structure. An atomic sentence is true in the structure if it is a fact in the working memory and false otherwise. The language of the *condition* of a rule is generally a subset of first-order logic. Roughly speaking, a rule is *fireable*, if there exists an instance of the rule such that the condition of the instance is true in the current state of the working memory, viewed as a model-theoretic structure.

If more than one rule is *fireable*, then *conflict-resolution* is performed to select an instance of a rule to *fire*, by executing its actions. The *actions* part of a rule can be a sequence of actions. Typical actions are to *add*, *delete* or *modify* a fact in the working

memory. An action can also be an external action, such as printing. Executing production rules in this way is called "forward chaining".[1]

The repeated firing of production rules generates a sequence of state transitions of the working memory. It is partly because state transitions are performed by destructive assignment and partly because of conflict resolution that production systems do not have a model-theoretic semantics.

The following formulation of the *production system cycle* will facilitate comparison with the LPS cycle later in the paper. Both cycles can be formalised by means of the transitive closure of a labelled transition system, as in the RIF-PRD formal operational semantics.

Definition 1. A *production system (PS) framework* is a tuple <***P, S₀,*** *Con*> where ***P*** is a set of production rules, S_0 is the initial state of the working memory, and *Con* is a conflict resolution strategy. Given such a framework, the *PS cycle* generates a sequence of state transitions, S_0, …, S_i , S_{i+1} ….., where each S_i is a working memory state. It generates S_{i+1} from S_i by means of the following steps:

PS1. For every instance *condition* $\sigma \rightarrow$ *actions* σ of a rule *condition* $\rightarrow$ *actions* in ***P*** such that *condition* σ holds in S_i , all the *actions* σ are collected into a set G_i.

PS2. If the set G_i is not empty, *conflict resolution* is performed according to the policy of *Con* to select a single a= *actions* σ in G_i. The cycle *terminates* if there is no such a selected by *Con* or if the set G_i is empty.

PS3. The actions of a are executed in sequence, generating S_{i+1} from S_i.

Conflict resolution can be performed in many different ways, for example by giving priority to rules in the order in which they are written. For example, a timid agent might try the following rules in the order:

someone attacks me $\rightarrow$ *add(run away)*
someone attacks me $\rightarrow$ *add(attack them back)*

Other conflict resolution strategies will be discussed after the LPS cycle.

2.2 Production Rules in Practice

Arguably, there are three kinds of production rules: reactive rules, forward chaining logic rules, and goal-reduction rules.

The first kind of rule is the most characteristic type of production rule, and is responsible for their general characterisation as *condition-action* rules. This kind of rule typically has implicit or *emergent* goals. For example, the rule *it is raining* $\rightarrow$ *cover yourself with an umbrella* has the implicit goal *to stay dry*.

The second kind of rule, for example *human X*$\rightarrow$ *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 goal-reduction rule "*IF you want to go home for the weekend and you have bus fare, THEN you can catch a bus*." that causes the most trouble for a model-theoretic semantics. This is because, to represent goal-reduction in production rule form, the working memory needs to contain, in addition to *"real" facts*, which describe the current state of a database, also

[1] It is often claimed that production rules can be executed either forwards or backwards. But this claim seems to be based on a confusion with logical implications and possibly with production rules in formal grammars.

goal facts, which describe some future desired state. Correspondingly, production rules have both *"real" actions*, which update the database, and *goal actions*, which maintain 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".

To obtain a model-theoretic semantics, LPS represents the third kind of rule, which has the general form: *goal G and conditions C → add H as a goal*
as a logic programming clause of the form: $G \leftarrow C, H$. Backward reasoning with the clause simulates forward chaining with the production rule.

To obtain the model theoretic semantics, LPS divides the working memory into two components, one containing real facts, and the other containing goals. The real facts act as a database representing the current state of a domain. LPS uses destructive assignment with structured definitions of actions, to transform one state of the database into another. The collection of all such states determines a model-theoretic structure, which makes the production rules true.

LPS updates and manipulates goals using backward reasoning, representing the current state of the goals as a disjunction of conjunctions of atomic goals, which is a kind of flattened and-or tree with existentially quantified variables. The LPS cycle terminates successfully when one of the conjunctions becomes empty.

LPS also represents forward chaining production rules of the second kind, above, as logic programming clauses. However, it extends logic programming by using reactive rules (and *condition-goal* rules, more generally) dynamically to add *new goals* to the current state of the goals whenever an instance of the *condition* of a rule holds in the current state of the database. The model-theoretic semantics ensures that these rules are true in the model-theoretic structure determined by the sequence of databases that is generated if the cycle terminates successfully.

2.3 Logic Programming

In this section we review the main concepts of logic programming needed for LPS. We restrict logic programs and goals in LPS to Horn clauses (without negation). For simplicity, in the remainder of the paper, when we use the terminology "clause" and "goal clause" we mean Horn clauses without negation.

Definition 2. *Logic programs* are sets of *clauses* of the form: $H \leftarrow B_1, \ldots, B_n$ where $\leftarrow$ stands for *if* and the comma stands for *and*. The *conclusion H* and the *conditions* B_i are atomic formulae (*atoms*, for short). All variables are implicitly universally quantified in front of the clause. *Facts* are clauses with no variables and with n=0.

Thus clauses that are facts are like facts in the working memory of a production system. Clauses that are not facts are often called *rules*. We do not use this terminology in this paper, because this invites confusion with production rules.

Definition 3. *Goal clauses* are conjunctions of atoms, the individual atoms of which represent individual *goals*. All variables are implicitly existentially quantified in front of the goal clause.

Backward reasoning uses program clauses to reduce goals to sub-goals and is formalized by SLD-resolution [9].

Definition 4. A *selection function*, ***s***, is a function from goal clauses to atoms, such that when $L = \boldsymbol{s}(C)$, L is an atom in C and is said to be the *selected literal* in C.

Given a goal clause $L_1,\ldots, L_i,\ldots, L_n$, with selected atom L_i and given a clause $H \leftarrow B_1,\ldots, B_m$ whose conclusion H unifies with L_i with most general unifier σ, *SLD-resolution* reduces the goal L_i to the subgoals $(B_1, \ldots, B_m)\sigma$ and generates the SLD-resolvent, which is a new goal clause $(L_1, \ldots, L_{i-1}, B_1, \ldots, B_m, L_{i+1}, \ldots, L_n)\sigma$.

If m=0 and n=i=1, then the new clause is called the *empty clause*, written {}, which is logically equivalent to *true*.

Definition 5. Given a set of clauses ***L***, a selection function ***s***, and an initial goal clause C_0, an *SLD-derivation*, or *successful SLD-computation of C_0 from **L** wrt **s***, is a sequence of goal clauses $C_0, \ldots, C_i, C_{i+1} \ldots C_n$ such that C_{i+1} is generated from C_i by means of SLD-resolution and C_n is the empty clause.

The selection function determines a search tree whose nodes are goal clauses, each branch of which is an attempted construction of a successful computation. To obtain a proof procedure, it is necessary to employ a search strategy $\sum$, which chooses goal clauses and program clauses, in the search for an SLD-derivation of the empty clause. Different search strategies, including depth-first, breadth-first, and best-first strategies, are all possible. We will see later, that the selection function ***s*** and the search strategy $\sum$ in LPS perform some of the functions of conflict resolution in production systems.

Backward reasoning treats clauses $H \leftarrow B_1,\ldots, B_m$ as goal-reduction procedures: *to show/solve H, show/solve B_1 and … and B_m*. For example, it 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.*

Horn clauses inherit the model-theoretic semantics of classical logic. However, they also have an alternative, minimal model semantics:

Theorem 1: Given a set of Horn clauses ***L***, there exists a unique minimal model ***M*** such that for all goal clauses *C*, *C* is a logical consequence of ***L*** (i.e. *C* is true in all models of ***L***) if and only if *C* is true in ***M.***

The theorem is a direct consequence of the theorem proved in [4] for the case where *C* is an atomic fact. Note that the theorem does not hold for more general consequences of ***L***, containing negation and universal quantification. The following theorem follows from Theorem 1 and the well-known soundness and completeness results for SLD-resolution:

Theorem 2: Given a set of Horn clauses ***L***, a goal clause *C*, and a selection function ***s***, there exists a successful SLD computation of *C* from ***L*** wrt ***s***

- if and only if *C* is a logical consequence of ***L***
- if and only if *C* is true in the minimal model of ***L.***

2.4 Abductive Logic Programming

LPS is partly inspired by abductive logic programming (ALP) [8] and by abductive logic programming agents (ALP agents) [10]. ALP extends LP by allowing some predicates, ***Ab***, the *abducibles*, to be undefined, in the sense that they do not occur in the conclusions of clauses. Instead, they can be assumed, but are constrained directly or indirectly by a set ***IC*** of *integrity constraints*.

Thus an *ALP framework* <***L***, ***Ab***, ***IC***> consists of a logic program ***L***, a set of abducibles ***Ab***, and a set of integrity constraints ***IC***. The predicates in the conclusions of clauses in ***L*** are disjoint from the predicates in ***Ab***. An atom whose predicate is in

Ab is called *abducible*. In LPS, the abducible atoms represent actions, and the integrity constraints generalise condition-action rules.

Here we define a simplified variant of ALP that is the basis for the LPS operational and model-theoretic semantics. For this purpose, we restrict the integrity constraints to implications of the form *condition* $\rightarrow$ *conclusion* where *condition* and *conclusion* are conjunctions of atoms, and all the variables occurring in *condition* are universally quantified over the implication, and all variables occurring only in the *conclusion* are existentially quantified over the *conclusion*. Thus the implication *customer(C)* $\rightarrow$ *status(C, S)* stands for the quantified sentence $\forall C[customer(C) \rightarrow \exists S[status(C, S)]]$.

Definition 6. Given an ALP framework <***L***, ***Ab***, ***IC***> and a goal clause *C* (which can be the empty clause), a *solution* is a set of atomic sentences Δ in the predicates ***Ab***, such that both *C* and ***IC*** are true in the minimal model of $\boldsymbol{L} \cup \Delta$.

This semantics is one of several that have been proposed for ALP and for integrity constraints more generally. It has the advantage that it extends naturally to provide a semantics for production rules in LPS. The following proof procedure for ALP, which extends SLD-resolution, also provides a natural basis for the LPS cycle. To ensure that Δ_i, below, is a set of atomic sentences, the selection function must be restricted so that when it selects an abducible atom *L* then *L* is ground (contains no variables). We call such a selection function *safe* (by analogy with the terminology "safe selection function" for logic programs with negation).

Definition 7. Given an ALP framework <***L***, ***Ab***, ***IC***>, safe selection function ***s***, and initial goal clause C_0, an *ALP-derivation*, *wrt* ***s***, is a sequence of pairs $<\Delta_0, C_0>$, …, $<\Delta_n, C_n>$, where $\Delta_0 = \{\}$ and C_n is the empty clause, $\Delta_n = \Delta_{n-1}$ and for all i, $0 \leq i \leq n-1$, $<\Delta_{i+1}, C_{i+1}>$ is obtained from $<\Delta_i, C_i>$ by the following steps:

ALP1. For every instance *condition* $\sigma \rightarrow$ *conclusion* σ of an integrity constraint in ***IC*** such that *condition* σ holds in $\boldsymbol{L} \cup \Delta_i$, add the conjunction of atoms *conclusion* σ to C_i. Let C_i* be the resulting goal clause.

ALP2. Let $L = \boldsymbol{s}(C_i^*)$. If *L* is not abducible, then C_{i+1} is the SLD-resolvent of C_i* with some clause in ***L*** and $\Delta_{i+1} = \Delta_i$. If *L* is abducible and $L \in \Delta_i$ then C_{i+1} is the SLD-resolvent of C_i* with *L* and $\Delta_{i+1} = \Delta_i$. If *L* is abducible and $L \notin \Delta_i$ then C_{i+1} is the SLD-resolvent of C_i* with *L* and $\Delta_{i+1} = \Delta_i \cup \{L\}$.

The condition "C_n is the empty clause and $\Delta_n = \Delta_{n-1}$" is needed to ensure that, not only are there no more goals to solve, but there are no further instances of the integrity constraints that need to be made true. Note the similarity between ALP1 and PS1.

The verification that *condition* σ holds in $\boldsymbol{L} \cup \Delta_i$ can be performed in different ways, for example by showing there is an SLD derivation of *condition* σ from $\boldsymbol{L} \cup \Delta_i$. Alternatively, the addition of an abducible atom to Δ_i can be used to "trigger" the *condition* by forward reasoning, and the remaining conditions can then be verified by backward reasoning.

To ensure ALP1 and ALP2 fully instantiate the universally quantified variables of integrity constraints, and to ensure that the abducible atoms in ***L*** and ***IC*** can always be made safe for selection, the abductive framework and initial goal clause C_0 have to be *range-restricted*. We will define a similar restriction for LPS later. The proof of the following theorem can be adapted from the proof for the IFF proof procedure [7].

Theorem 3. Soundness and Completeness. Let <***L***, ***Ab***, ***IC***> be a range-restricted ALP framework and C_0 a range-restricted goal clause. If there exists an *ALP-derivation* $<\{\}, C_0>, \ldots, <\Delta_n, \{\}>$, then C_0 and ***IC*** are true in the minimal model of

$L \cup \Delta_n$. If there exists a set Δ of atomic sentences with predicates in Ab such that C_0 and IC are true in the minimal model of $L \cup \Delta$, then for every safe selection function there exists an *ALP-derivation* $<\{\}, C_0>, \ldots, <\Delta_n, \{\}>$ where $\Delta_n \subseteq \Delta$.

LPS is based on ALP agents [10], which embed ALP in an observation-thought-action cycle, in which abducible atoms represent observations and actions. Integrity constraints give the reactive behaviour of condition-action rules. The ALP agent cycle generates a sequence of actions in the attempt to make an initial goal and the integrity constraints true in the agent's environment.

In ALP agents, the agent's environment is an external, destructively changing semantic structure. The set Δ, on the other hand, is the agent's internal representation of its interactions with the environment. This internal representation is monotonic in the sense that observations and actions are time-stamped and state representations are derived by an action theory, such as the situation or event calculus. In contrast, in both production systems and LPS, the environment is simulated by an internal destructively changing representation. In LPS this representation is associated with a Kripke-like model-theoretic structure.

3 LPS – A Logic-based Production System Framework

Like ALP agents, LPS combines logic programs and integrity constraints, and represents actions by abducible atoms. Moreover, like the ALP agent cycle, the LPS cycle generates actions in the attempt to make an initial goal and the set of integrity constraints true. However, unlike ALP agents, which use an internal, monotonic representation of states, LPS uses destructive assignment to maintain an internal representation of only the current state. In both ALP agents and LPS, if the cycle terminates successfully, then the actions and the totality of all the states generated by the cycle determine a model of the initial goal and integrity constraints. This model is analogous to a Kripke model, in which the individual states are like possible worlds.

Definition 8. An *LPS framework* is a tuple $< V, P, L, A, W_0, G_0 >$, where

- V is itself a tuple $<A, M, E, I, O, \leq, <, initiates, terminates, precondition>$ dividing the vocabulary of L and P into predicates representing atomic actions A, macro-actions M, extensional predicates E, directly affected by atomic actions, intensional predicates I, indirectly affected by changes to the extensional predicates, and "ordinary" stateless predicates O not affected by change of state. In addition, the vocabulary contains the inequality predicates $\leq$ and $<$ to compare states, the special predicates *initiates* and *terminates*, which represent the effects of actions on the extensional predicates, and *precondition*, which represents the preconditions of actions. Let AFT be the set $A \cup M \cup E \cup I \cup O \cup \{\leq, <\}$. Only the predicates in AFT can occur in rules and clauses. The other predicates are used for state transitions.
- P is a set of condition-conclusion rules.
- L is a set of Horn clauses, made up of five disjoint subsets $L_{stateless}$, L_{ram}, L_{plan}, L_{temp}, L_{macro}, defining stateless predicates, intentional predicates, plans, the inequality predicates, and macro-actions, respectively.
- A is a set of Horn clauses, called *action clauses*, made up of two disjoint subsets A_{pre} and A_{post}, defining preconditions and postconditions of actions, respectively.
- W_0 is a set of atomic sentences, in the predicates of E.
- G_0 is a set of goal clauses whose predicates belong to AFT.

Given such a framework, the LPS cycle, defined below, determines a sequence of state transitions $<W_0, G_0>, a_0, \ldots, <W_i, G_i>, a_i \ldots$, where the a_i represent actions, the

$\boldsymbol{W_i}$ represent the successive states of the database, and the $\boldsymbol{G_i}$ represent the successive states of the goals. The cycle terminates successfully in state *n*, if $\boldsymbol{G_n}$ contains the empty clause. In this case, the sequence of states and actions $\boldsymbol{W_0}$, a_0, $\boldsymbol{W_1}$, a_1, …, $\boldsymbol{W_n}$ augmented with the clauses in $\boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}} \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}$ determines a minimal model $\boldsymbol{M}$, in which $\boldsymbol{G_0}$ and $\boldsymbol{P}$ are true. Here we give greater detail:

$\boldsymbol{V}$ **- the vocabulary.** For clarity in the LPS cycle, except for facts in the database states $\boldsymbol{W_i}$, the extensional, intension and atomic and macro-action predicates have a single explicit state argument, which we write as their last argument. We sometimes use the notation *p(t)* to refer to an atom with state argument *t* (represented by non-negative integers or variables *T*, T_1, T_2) when the other arguments of the atom are irrelevant to the discussion.

This state argument is always a variable in $\boldsymbol{G_0}$, the rules in $\boldsymbol{P}$ and the clauses in $\boldsymbol{L}$ and $\boldsymbol{A}$. Moreover, different occurrences of state variables in the same rule or clause are so highly constrained that they could easily be made implicit. For example, all state arguments in clauses in $\boldsymbol{L_{ram}}$ and $\boldsymbol{A}$ are a single (universally quantified) variable.

On the other hand, the state variables and inequalities in $\boldsymbol{G_0}$, in the conclusions of rules in $\boldsymbol{P}$ and in the conditions of clauses in $\boldsymbol{L_{plan}}$ and $\boldsymbol{L_{macro}}$ are needed only to indicate the order in which actions and goals are to be achieved. These variables and inequalities could be made implicit, by employing a syntax in which the intended order of achievement is indicated by the order in which the goals and actions are written.

Whether states are implicit or explicit, we impose the restriction that no two actions (whether from *A* or *M*) have the same state argument. This is because, for simplicity, the LPS operational semantics executes a single action in each cycle/state.

$\boldsymbol{W_i}$ **- the database.** $\boldsymbol{W_i}$ is a set of *facts* representing the current state of the database. These correspond to the extensional predicates of a deductive database, e.g. *customer(john-smith), spent-to-date(john-smith, 500).* Because the transformation of $\boldsymbol{W_i}$ into $\boldsymbol{W_{i+1}}$ is implemented by destructive assignment, the facts in $\boldsymbol{W_i}$ are written without state arguments. This means that facts that are not affected by the transformation simply persist without being copied explicitly from one state to the next. However, for the model-theoretic semantics, to combine all the states into a single model, the facts in $\boldsymbol{W_i}$ need to be understood as containing the implicit state argument *i*. We use the notation $\boldsymbol{W^*_i}$ when we need to refer to facts containing explicit state arguments: $\boldsymbol{W^*_i} = \{p(i) : p \in \boldsymbol{W_i}\}$.

$\boldsymbol{A}$ **– the action clauses.** The clauses in $\boldsymbol{A}$ are divided into clauses $\boldsymbol{A_{pre}}$ defining the preconditions of actions and clauses $\boldsymbol{A_{post}}$ defining the post-conditions of actions. Unlike the clauses in $\boldsymbol{L}$, the clauses in $\boldsymbol{A}$ do not contribute to checking the conditions of rules or to the reduction of goals to subgoals. They have the form:

initiates(a, p, T) ← *init-conditions(T)*
terminates(a, p, T) ← *term-conditions(T)*
precondition(a, q, T) ← *pre-conditions(T)*

where *a* represents an atomic action, *p* represents an extensional predicate and *q* represents any predicate in $E \cup I \cup O$. The conditions *init-conditions(T), term-conditions(T)* and *pre-conditions(T)* are (possibly empty) conjunctions of atoms in the predicates $E \cup I \cup O$. Each clause in $\boldsymbol{A}$ contains a single state variable. For example:

initiates(take-payment(X, ID, Value), spent-to-date(X, New), T) ←
spent-to- date(X, Old, T), New = Old + Value.
terminates(take-payment(X, ID, Value), spent-to-date(X, Old), T) ←
spent-to-date(X, Old, T).

An action *a* can be executed in state W_i provided that for all *q*, if *precondition(a, q, i)* holds in $W^*_i \cup A_{pre} \cup L_{stateless} \cup L_{ram}$ then *q* holds in $W^*_i \cup L_{stateless} \cup L_{ram}$.
Since $W^*_i \cup A_{pre} \cup L_{stateless} \cup L_{ram}$ and $W^*_i \cup L_{stateless} \cup L_{ram}$. are sets of Horn clauses, the notion of *holding* can be understood equivalently either in terms of truth in the associated minimal model, in terms of the existence of an SLD-derivation, or in terms of any other complete proof procedure for Horn clauses. The execution of *a* updates the state of the database: $W_{i+1} = (W_i - delete(a)) \cup add(a)$
where $delete(a) = \{p: terminates(a, p, i)$ holds in $W^*_i \cup A_{post} \cup L_{stateless} \cup L_{ram}\}$
$add(a) = \{p: initiates(a, p, i)$ holds in $W^*_i \cup A_{post} \cup L_{stateless} \cup L_{ram}\}$.
Not every action needs to initiate or terminate database facts. In particular, we can have external actions, as in RIF-PRD, which have no impact on the database.

- G_i - **the goal states.** G_i is a set of goal clauses, each of which can be regarded as a *partial plan* for achieving the initial goals G_0 and the additional goals generated by the cycle so far. The intention is that, for every G_i, one of the goal clauses in G_i should be true in the model that is generated by the LPS cycle. G_0 can contain only the empty clause, as is typical of production systems. The predicates in G_i all belong to *AFT*.
- ***P*** – **the rules.** The set ***P*** of rules has the same form *condition*→ *conclusion* and the same implicit quantification as ALP integrity constraints, where both *condition* and *conclusion* are conjunctions of atoms[2]. The predicates in the *condition* belong to $E \cup I \cup O \cup A$, and the predicates in the *conclusion* belong to *AFT*. For example:

customer(X, T), login(X, T) → *welcome(X, T'), T≤T'*
customer(X, T), checkout(X, T), shop-cart(X, ID, Value, T) →
take-payment(X, ID, Value, T1), confirm(X, ID, Value, T2), T<T1, T1<T2

All atoms in the *condition* with predicates belonging to $E \cup I$ contain a single state variable, say *T*. The *condition* can also include a single atom representing an action in *A* of the form *a(T-1)*. This allows ***P*** to include a kind of action-condition-conclusion rule. All state variables in the *conclusion* refer to states later than or equal to *T*, as indicated by inequalities among state variables in the *conclusion*.

L **- the clauses.** The set ***L*** of clauses is made up of four disjoint subsets $L_{stateless}$, L_{ram}, L_{plan}, L_{temp}. The set $L_{stateless}$ consists of clauses not containing state arguments, with all predicates from *O*. For example: *similar(X, Y)* ← *cd(X), dvd(Y)*

L_{ram} defines intensional predicates, which are ramifications of the extensional predicates. For example :

status(X, gold, T) ← *spent-to-date (X, V, T), 500≤V*
status(X, new, T) ← *spent-to-date (X, V, T), V <500*

Here *spent-to-date* is an extensional predicate, which changes directly as the result of actions, such as *take-payment*, and *status* is an intensional predicate, which changes as a ramification of changes to the predicate *spent-to-date*. In general, clauses in L_{ram} have the form *conclusion* ← *condition* where *conclusion* is an intensional atom in *I*, and the atoms in *condition* are all extensional, intensional or state-independent atoms in $E \cup I \cup O$. The clauses in L_{ram} contain only a single state variable.

L_{plan} represents plans for achieving future states of the database, including plans similar to those in AgentSpeak(L) [11]. For example:

quenched-thirst(T5+1) ← *go-to-fridge(T1), open-fridge(T2), get-drink(T3), open-drink(T4), drink(T5), T1 < T2, T2< T3, T3 < T4, T4 < T5*

In general, clauses in L_{plan} have an extensional predicate in the conclusion, and predicates from *AFT* in the conditions. States in the conditions are constrained to be

[2] Later we will see that, to deal with conflict resolution, we allow negative conditions in rules.

earlier than the state in the conclusion. The conditions must have one action whose state is immediately prior to the state in the conclusion.

$\boldsymbol{L_{temp}}$ defines the inequality relations. These clauses are needed only to check that one state comes before another. Therefore any such definition will serve the purpose including, for example:

$0 < I+1$ $\qquad\qquad$ $I \leq J \leftarrow I < J$

$I+1 < J+1 \leftarrow I < J$ $\qquad\qquad$ $I \leq J \leftarrow I = J$

$\boldsymbol{L_{macro}}$ defines the macro-actions. Clauses in $\boldsymbol{L_{macro}}$ have a predicate from *M* in the conclusion, and predicates from *AFT* in the conditions. States in the conditions are constrained to be earlier than or the same as the state in the conclusion. The conditions must have one action (from *A* or *M*) whose state is the same as the state in the conclusion. For example:

welcome(X, T) ← status(X, new, T), promotional-item(Y, T), offer(X, Y, T)

welcome(X, T) ← status(X, gold, T), promotional-item(Y, T), profile(X, Z, T), similar(Y, Z), offer(X, Y, T)

In the LPS cycle, the state argument of an action is always a variable when it is chosen for execution. Otherwise, all of its other arguments need to be variable-free. This is a generalisation of the safety requirement of ALP. In addition, the selection function needs to be *timely* in the sense that, when an action is selected, the goal clause in which it occurs contains no other atoms that need to be solved or executed earlier.

Definition 9. A selection function is *safe* if and only if, when it selects an action, the action is ground except for state variables; and, when it selects an inequality, the inequality is ground. A selection function is *timely* if and only if, when it selects an action *a(t)* in a goal clause *C*, then *C* contains no other atom whose state is *constrained* to be before or the same as *C*, i.e. no other atom of the form *p(t)*, no pair of atoms of the form *p(t')*, $t' \leq t$ and no pair of atoms of the form *p(t')*, $t' < t$.

To ensure the existence of safe selection functions and to ensure that quantifiers in rules are correctly interpreted, LPS frameworks need to be range-restricted. We define range-restriction after the LPS cycle.

The operational semantics is a synthesis of the PS cycle and ALP derivations.

3.1 The Operational Semantics

Definition 11. LPS cycle: Given a range-restricted LPS framework $< \boldsymbol{V, P, L, A, W_0}, \boldsymbol{G_0} >$, a safe selection function $\boldsymbol{s}$, and search strategy $\sum$, the LPS cycle determines a sequence of state transitions $<\boldsymbol{W_0}, \boldsymbol{G_0}>, a_0, \ldots, <\boldsymbol{W_i}, \boldsymbol{G_i}>, a_i \ldots$ where for all i, $0 \leq i$, a_i and $<\boldsymbol{W_{i+1}}, \boldsymbol{G_{i+1}}>$ are obtained from a_{i-1} and $<\boldsymbol{W_i}, \boldsymbol{G_i}>$ by the following steps:

LPS1. For every instance *condition* $\sigma \rightarrow$ *conclusion* σ of a rule in $\boldsymbol{P}$ such that *condition* σ holds in $\boldsymbol{W^*_i} \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}$, add *conclusion* σ to every clause in $\boldsymbol{G_i}$. Let $\boldsymbol{G_i}^*$ be the resulting set of goal clauses.

LPS2. Using the selection function $\boldsymbol{s}$ and search strategy $\sum$, let $\boldsymbol{G_{i+1}}$ be a set of goal clauses, starting from $\boldsymbol{G_i}^*$, derivable by SLD-resolution using the clauses in $\boldsymbol{W^*}_i \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}} \cup \boldsymbol{L_{plan}} \cup \boldsymbol{L_{temp}}$ such that either $\boldsymbol{G_{i+1}}$ contains the empty clause or $\boldsymbol{G_{i+1}}$ contains at least one clause whose selected literal is an executable action. (Recall that an action is executable in state $\boldsymbol{W_i}$ if and only if all the preconditions of the action hold in the current state.)

- If there is no such G_{i+1} then the cycle terminates unsuccessfully. (Note that SLD-resolution might loop without generating a selected executable action, in which case unsuccessful termination is non-constructive.)
- If G_{i+1} contains the empty clause and $G_i = G_i^*$, then the cycle *terminates successfully* and $W_{i+1} = W_i$.

LPS3. If the cycle does not terminate in step LPS2, then let *a(t)* be the selected, executable action in a goal clause in G_{i+1} chosen by the search strategy $\sum$. (Note that *a(t)* might have been generated and selected in an earlier cycle, but not have been executable before. Moreover, even if it was selected and executable before, the search strategy might have chosen some other action.)

LPS4. The chosen action is executed, letting $a_i = a(i)$. The current state of the database W_i is updated to W_{i+1}, by destructively deleting the old facts in *delete(a(i))* and adding the new facts in *add(a(i))*.

Range-restriction. The following definition generalises the definition of range-restriction for normal logic programming and ALP:

Definition 10. An LPS framework < ***V, P, L, A, W₀ , G₀*** > is *range-restricted* if and only if all rules in ***P*** and all clauses in ***L, A*** and G_0 are range-restricted, where:

- A goal clause *C* in G_0 (and more generally any conjunction *C* of atoms in the predicates *AFT* is range-restricted if and only if every variable occurring in an atomic action atom in *C*, except for state variables, occurs in an atom in *C* whose predicate does not belong to $A \cup \{\leq , <\}$, and whose state is not constrained to be after the state of the action, and every variable in an inequality (with predicate in $\{\leq , <\}$) occurs in a non-inequality atom of *C*.
- A clause *conclusion* ← *conditions* in ***L*** is range-restricted if and only if *conditions* is range-restricted and every variable in *conclusion* occurs in *conditions.*
- A clause *conclusion* ← *conditions* in ***A***, where *conclusion* is *initiates(a, p, t*), *terminates(a, p, t),* or *precondition(a, p, t),* is range-restricted if and only if every variable in *p* occurs either in *conditions* or in *a*.
- A rule *condition* → *conclusion* in ***P*** is range-restricted if and only if every variable occurring in an atomic action atom in *conclusion*, except for a state variable, occurs either in the *condition* or in an atom in the *conclusion* whose predicate does not belong to $A \cup \{\leq , <\}$, and whose state is not constrained to be after the state of the action, and every variable in an inequality (with predicate in $\{\leq , <\}$) occurs in a non-inequality atom in the rule.

Derivation versus search. Both ALP and LPS generate a sequence of paired database and goal states – of the form $<\Delta_i, C_i>$ in the case of ALP, and of the form $<W_i, G_i>$ in the case of LPS. However, ALP derivations include only inferences necessary to solve the initially given problem, whereas the LPS cycle also includes a search for the solution, determined by the search strategy $\sum$.

The cycle granularity. To maximise similarity with the PS cycle, except for the last transition of a successfully terminating cycle, the transition from one state $<W_i, G_i>$ to the next $<W_{i+1}, G_{i+1}>$ is determined by the execution of an action. As a result, a single iteration of the cycle needs to perform an indeterminate number of SLD-resolution steps until a selected action becomes executable and is chosen for execution. A single action in LPS typically involves the deletion and addition of several facts. In PS, each such deletion and addition is a separate action.

Notice that for the shopping example, it would be useful to allow input actions by customers. It is easy to extend the LPS cycle to allow such external input actions, so

that the transition from one state to another results from the collective effects of these and the internally chosen action. We omit the details for lack of space.

The selection function and choice of actions to be executed. There is no requirement that an action must be selected if it is executable. In fact, the selection of an executable action can be delayed, to plan future goals, Nor is there any requirement that an action that is selected and executable be chosen for execution. This is because several such actions might be executable in the same iteration of the cycle.

The evaluation of conditions of rules and clauses in *A*. In LPS1, an instance *condition* $\sigma \rightarrow$ *conclusion* σ of a rule in $\boldsymbol{P}$ is fired when *condition* σ holds in $\boldsymbol{W^*_i} \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}$. Here "holds" means that *condition* σ is *true* in the minimal model of $\boldsymbol{W^*_i} \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}$. To ensure that truth can be verified effectively, we have restricted the form of *condition* to conjunctions of atoms. However, if the minimal model is finite, then this restriction can be liberalised, and *condition* can be any sentence of first-order logic. There are many simple, syntactic restrictions, such as Datalog, that can be imposed to guarantee finiteness. Similarly, the syntax of the conditions of clauses in $\boldsymbol{A}$ can be liberalised for the same reason.

3.2 Conflict Resolution

Unlike the PS cycle, the LPS cycle does not perform explicit conflict resolution. Instead, it uses the selection function, to sequence goals, and the search strategy, to choose among the alternative ways of solving goals and sub-goals. However, like conflict resolution in production systems, the execution of an action in LPS is a *committed choice*, from which there is no backtracking.

Production systems use conflict resolution to decide which rules should be fired when several rules are fireable. In contrast, LPS adds to its goals the conclusions of all instances of rules whose conditions hold in the current state. This feature of LPS is necessary for the model-theoretic semantic, to ensure that the rules in $\boldsymbol{P}$ will be true in the model determined by the LPS cycle.

As a consequence, conventional ways of writing production rules are not always acceptable in LPS. Consider, for example, the production rules:

Someone-attacks-me $\rightarrow$ *attack-them-back*
Someone-attacks-me $\rightarrow$ *run-away*

Both rules are fireable in a state in which *someone attacks me*. In conventional production systems, conflict resolution would be necessary to fire only one of them.

However, in the same situation in LPS, both rules would fire and both actions would need to be executed. To obtain the effect of conflict resolution, LPS would need to rewrite the two rules, making their intended higher-level goal and the alternative ways of achieving it explicit. For example (with explicit state variables):

In $\boldsymbol{P}$: *someone-attacks-me(T)* $\rightarrow$ *protect-myself(T'), T*$\leq$*T'*
In $\boldsymbol{L_{Plan}}$: *protect- myself(T+1)* $\leftarrow$ *attack -them -back(T)*
protect myself(T+1) $\leftarrow$ *run-away(T)*

More generally, to obtain the effect of the production rules:

$C \rightarrow A_1$ $\quad\quad$ $C \rightarrow A_2$

where the intention is that, when C is true, then one of the actions A_1 or A_2 should be made true in the future, rewrite the rules in the form:

In $\boldsymbol{P}$: $C(T) \rightarrow G(T'), T \leq T'$
In $\boldsymbol{L_{Plan}}$: $G(T+1) \leftarrow A_1(T)$ $\quad\quad$ $G(T+1) \leftarrow A_2(T)$

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:

$C_1 \rightarrow G_1$ $C_2 \rightarrow G_2$ … $C_n \rightarrow G_n$

the effect of conflict resolution can be obtained in LPS by adding extra conditions to the rules:

$C_1(T) \rightarrow G_1(T'), T \leq T'$ $C_2(T)$, *not* $C_1(T) \rightarrow G_2(T'), T \leq T'$ …

$C_n(T)$, *not* $C_1(T)$,… , *not* $C_{n-1}(T) \rightarrow G_n(T'), T \leq T'$

Notice that this is a case where we need the liberalisation of the syntax of the conditions of the rules in ***P***, discussed at the end of the previous section

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

customer(X,T), online(X,T) $\rightarrow$ *present-advert(X, T'), T* $\leq$ *T'*

will fire repeatedly in all states in which the condition *online(X,T)* continues to hold. If required, this can be avoided by replacing the *online* condition by the action (or event) that initiates it, in this case obtaining the rule

customer(X,T), login(X,T) $\rightarrow$ *present-advert(X, T'),T* $\leq$ *T'*

More generally, to avoid firing the same rule $C(T) \rightarrow G(T'), T \leq T'$

unnecessarily, replace it by $A(T) \rightarrow G(T'), T \leq T'$,

where *C(T)* is an extensional predicate and *initiates(A, C, T)*. A similar approach can be used when *C(T)* is an intensional predicate. Dealing with refraction by representing it in an appropriate formulation of the rules has the advantage that different policies regarding refraction can be combined in the same application.

3.3 Model-theoretic semantics

The set ***S*** of sentences $\boldsymbol{W^*_0} \cup \ldots \cup \boldsymbol{W^*_i} \cup \ldots \{a_0, \ldots, a_i, \ldots\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}} \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}$ is a Horn clause logic program. Therefore, ***S*** has a unique minimal model ***M***. This model is like a Kripke structure of possible worlds $\boldsymbol{M}_i = \boldsymbol{W^*_i} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}} \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}$ embedded in a single model ***M***, where the actions $\{a_0, \ldots, a_i, \ldots\}$ are like the transition relation from one possible world to another.

Soundness. To prove the soundness of the LPS cycle, the rules in $\boldsymbol{L_{plan}}$ need to be compatible with the action clauses in ***A***:

Definition 12. $\boldsymbol{L_{plan}}$ is *compatible* with ***A*** if every clause in $\boldsymbol{L_{plan}}$ has the form :

p(T+1) $\leftarrow$ *init-conditions(T), other-conditions, a(T)*

where *initiates(a, p, T)* $\leftarrow$ *init-conditions(T)* is an instance of a clause in ***A***. *p(T+1)* may have other arguments, and *other-conditions* are any other conditions respecting range-restriction.

It is easy to satisfy this condition, and all the examples in this paper, if done in full will have this property. Note that we can plan to achieve intentional atoms by combining such clauses in $\boldsymbol{L_{plan}}$ with clauses in $\boldsymbol{L_{ram}}$.

Theorem 4: Given a range-restricted LPS framework < ***V, P, L, A,*** $\boldsymbol{W_0}$ **,** $\boldsymbol{G_0}$ >, safe selection function ***s***, and search strategy $\sum$, if the LPS cycle terminates successfully and $\boldsymbol{L_{plan}}$ is compatible with ***A***, then some clause C_0 in $\boldsymbol{G_0}$ is true in ***M*** and all the rules in ***P*** are true in ***M***.

Sketch of proof: If the cycle terminates successfully, then some $\boldsymbol{G_n}$ contains the empty clause and $\boldsymbol{G_{n-1}}^* = \boldsymbol{G_{n-1}}$. The proof of this empty clause can be traced backwards

to a sequence of clauses, starting with some C_0 in $\boldsymbol{G_0}$: C_0 ,…,C_i, ….,C_m = {}, where C_{i+1} is obtained from C_i in one of two ways:

1) C_{i+1} is C_i conjoined with *conclusion* σ for every instance *condition* $\sigma \rightarrow$ *conclusion* σ of a rule in $\boldsymbol{P}$ such that *condition* σ holds in $\boldsymbol{W^*_i} \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}}$.
2) C_{i+1} is obtained by SLD-resolution between C_i and some clause in $\boldsymbol{W^*}_i \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}} \cup \boldsymbol{L_{plan}} \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}$.

It suffices to prove the **lemma:** All the C_i are true in $\boldsymbol{M}$. The lemma implies that $\boldsymbol{C_0}$ is true in $\boldsymbol{M}$. Together with the condition $\boldsymbol{G_{n-1}}^* = \boldsymbol{G_{n-1}}$, the lemma also implies that all the rules in $\boldsymbol{P}$ are true in $\boldsymbol{M}$.

Proof of lemma: The lemma follows by induction, by showing base case C_m = {} is true in $\boldsymbol{M}$ and induction step if C_{i+1} is true in $\boldsymbol{M}$, then C_i is true in $\boldsymbol{M}$. The base case is trivial. For the induction step, there are two cases: In case 1 above, if C_{i+1} is true in $\boldsymbol{M}$, then C_i is true in $\boldsymbol{M}$, because if a conjunction is true then so are all of its conjuncts.

In case 2 above, the clauses C_{i+1} and C_i are actually the negations of clauses in ordinary resolution. So, according to the soundness of ordinary resolution, $\neg C_{i+1}$ is a logical consequence of $\neg C_i$ and C. Therefore, if both C and C_{i+1} are true in $\boldsymbol{M}$, then C_i is true in M. But any clause C in $\boldsymbol{W^*}_i \cup \{a_{i-1}\} \cup \boldsymbol{L_{stateless}} \cup \boldsymbol{L_{ram}} \cup \boldsymbol{L_{temp}} \cup \boldsymbol{L_{macro}}$ is true in $\boldsymbol{M}$ by the definition of $\boldsymbol{M}$. It suffices to show that all clauses in $\boldsymbol{L}_{\text{plan}}$ are also true in $\boldsymbol{M}$. But this follows from the compatibility of $\boldsymbol{L_{plan}}$ with $\boldsymbol{A}$.

Completeness. Following the completeness results for ALP, it might be expected that a similar completeness result would hold for LPS: Given a minimal model $\boldsymbol{M}$ of some clause C_0 in $\boldsymbol{G_0}$ and of all the rules in $\boldsymbol{P}$, it might be hoped that there would exist some search strategy $\sum$ that together with the LPS cycle could generate some related model $\boldsymbol{M}$', possibly determined by a subequence of the actions of $\boldsymbol{M}$. Unfortunately this is not always possible. The LPS cycle will not generate models that make rules true by making their conditions false. For example:

$\boldsymbol{P}$: $q \rightarrow a$ $\quad$ $\boldsymbol{A}$: *terminates(b, q)* $\quad$ $\boldsymbol{W_0}$: {q}

Here a and b are actions, and we ignore states for simplicity. There is a minimal model corresponding to the sequence of actions *b; a*, but the LPS cycle can only generate the non-terminating sequence a, a, …

4 Related and Future Work

LPS combines production rules and logic programs with a destructively updated database, and gives this combination a model-theoretic semantics. To the best of our knowledge, both of these contributions are novel. Raschid [12] considers a restricted kind of production rules and translates them into stratified logic programs. She shows that the final state of the production system cycle is identical to the minimal model of the associated logic program. However, the model is the final state of the cycle, rather than as in the LPS case the totality of all the states generated from the initial state to the final state.

Other systems more closely related to LPS have been developed mainly in other fields, such as logic programming, active databases and intelligent agents. EVOLP [1], in particular, gives a model-theoretic semantics to evolving logic programs that change state destructively over the course of their execution. Several authors, including [6] obtain a model-theoretic semantics for event-condition-action rules in active database systems, by translating rules into logic programs with their associated

model theory. Metatemp, [5] on the other hand, is a programming language with a Kripke semantics for modal logic sentences resembling production rules. The Bonner Kifer Transaction Logic [2] gives a Kripke-like semantics for macro-actions, which is similar to that of LPS.

LPS is based on ALP agents, but differs in its use of a destructively changing database. Unlike ALP agents, LPS does not generate actions to terminate conditions of production rules. However, Dung and Mancarella [3] address a related problem of allowing future actions to affect the conditions of production rules. They give the resulting production system an argumentation semantics.

Perhaps the most important direction for future work is to harmonise and unify related work in production systems, active databases, and intelligent agents, all of which employ rules in a cycle that generates a sequence of database states. LPS, based on ALP agents, is a contribution to this unification. However, further work is necessary both to extend the database to include external updates, and to investigate further the problem of completeness. We are currently working on an implementation.

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

References

1. Alferes, J., Leite, J., Pereira, L.M., Przymusinska, H. & Przymusinski, T.:Dynamic Updates of Non-Monotonic Knowledge Bases, J. of Logic Programming 45(1-3):43-70 (2000)
2. Bonner and M. Kifer.: Transaction logic programming. In Warren D. S., (ed.), Logic Programming: Proc. of the 10th International Conf., 257-279 (1993)
3. Dung, P. M. Mancarella, P.: Production Systems with Negation as Failure, IEEE Transactions on Knowledge and Data Engineering, Vol 14; 2, 336-352 (2002)
4. van Emden, M. and Kowalski, R.: The Semantics of Predicate Logic as a Programming Language, in JACM, Vol. 23, No. 4, 733-742 (1976)
5. Fisher, M.: A Survey of Concurrent METATEM - The Language and its Applications. Lecture notes in computer science, 827, Springer Verlag (1994)
6. Flesca, S. and Greco, S. Declarative semantics for active rules. Theory and Practice of Logic Programming **1** (1): 43-69, (2001)
7. Fung, T.H. and Kowalski, R. : The IFF Proof Procedure for Abductive Logic Programming. J. of Logic Programming (1997)
8. Kakas,T., Kowalski, R., Toni, F.:The Role of Logic Programming in Abduction, Handbook of Logic in Artificial Intelligence and Programming 5, Oxford University Press, 235-324 (1998)
9. Kowalski, R.: Predicate Logic as Programming Language, in Proceedings IFIP Congress, Stockholm, North Holland Publishing Co., 569-574 (1974)
10. Kowalski, R. and Sadri, F.: From Logic Programming towards Multi-agent Systems, Annals of Mathematics and Artificial Intelligence , Volume 25, 391-419 (1999)
11. Rao, A.: Agents Breaking Away, In Lecture Notes in Artificial Intelligence, Volume 1038, Springer Verlag, Amsterdam, Netherlands (1996)
12. Raschid, L.: A Semantics for a Class of Stratified Production System Programs. J. of Logic Programming. 21(1) (1994)
13. Russell, S. and Norvig, P.: Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentice Hall (2003)
14. Simon, H.: Production Systems. In Wilson, R. and Keil, F. (eds.): The MIT Encyclopedia of the Cognitive Sciences. The MIT Press. 676-677 (1999)
15. Thagard, H.: Mind: Introduction to Co*gnitive Science.* Second Edition. MIT Press (2005)