Characterizing Production Systems using Logic Programming and Situation Calculus

Chitta Baral*
Department of Computer Sc.
Univ. of Texas at El Paso
El Paso, TX, 79968, USA
chitta@cs.utep.edu

Jorge Lobo†
Department of EECS
Univ. of Illinois at Chicago
Chicago, IL 60607, USA
jorge@eecs.uic.edu

Abstract

In this paper we characterize production systems using logic programs and situation calculus notation. Our characterization is different from the earlier attempts [3, 6, 7, 2] and unlike them does not define a *new* semantics. We show that the semantics defined by our characterization is equivalent to the operational semantics of production systems. This opens up the possibility of using results and techniques of logic programming to prove results about production systems, and combining production systems with other systems based on logic programming (such as deductive database systems). Moreover, the use of situation calculus notation allows us to extend production systems to incorporate various facets of reasoning about actions, such as actions that do more than adding and deleting fluents, actions with indirect effects, etc.

1 Introduction and motivation

A production system (also sometimes referred to as a forward-chaining rule-based system) consists of (1) a set of rules, (2) a working memory, and (3) an inference engine. Rules are statements of the form:

If **Condition** then **Action**.

A simple inference engine is defined as follows: In each cycle of execution the condition part of all rules are evaluated against the working memory and from the successful rules the action part of one of the rules is executed on the working memory.

*Support was provided by the National Science Foundation, under grant Nr. IRI-9211662 and IRI-9501577.

†Support was provided by the National Science Foundation, under grant Nr. IRI-92-10220.

Production systems (first developed by Post in 1943) have been used to build several large and useful knowledge based systems. For example, XCON and XSEL, two OPS5 based production systems are in active industrial use at DEC. The popularity of these systems resides in the fact that the structure of the rules used in them arises naturally when humans describe even the most complicated techniques that they use to solve problems. Production systems have also been used [9, 4] as a psychological model of human behavior. Extensions of production systems (from condition-action rules to event-condition-action rules) have also been used in active databases [11].

But so far the only semantics of production systems that has been widely used is an *operational semantics*. This has made it difficult (in many cases) to analyze the behavior of production systems, especially when it consists of a large number of interacting rules. Some of the important behaviors that often become necessary to be analyzed are: termination of a production system, correctness of a production systems, complexity results etc. We believe that different characterizations of production systems are needed to analyze different behaviors. This is similar to the the different characterization (such as procedural, model-theoretic, fixpoint etc.) of logic programs and their usefulness in analyzing different aspects of logic programs. Moreover, different characterization of production systems will also help in analyzing the behavior of a production system obtained by integrating two production systems, and also when a productions system is integrated with a database system.

The above motivated us to search for a different characterization of production systems. But ours is not the first attempt in this direction. Raschid in [6] defines a translation of production systems to a logic program (with integrity constraints) and defines the semantics of the production system in terms of the semantics of the translated logic program. This idea of translating to a logic program is a clever idea and allows researchers to use the vast array of results [1] in logic programming semantics. *But unfortunately the semantics defined by Raschid is in many cases different from the original operational semantics.* Thus, Raschid does not give a different characterization of the original semantics of production systems; rather she gives a new semantics of production systems.

In this paper we give a different characterization of the original operational semantics of production systems. Unlike Raschid's our characterization agrees with the operational semantics of production systems. We would like to make it clear that we are not saying that one semantics is better than the other. Our intention is to give a different characterization of the original semantics and show that our characterization gives us added power and machinery to analyze production systems.

We use the clever idea of Raschid of translating production systems to logic programs. But in addition we use the situation calculus notation. This allows us to refer to different states of the working memory and avoid certain restrictions (such as stratification) that were made in [6].

The structure of the rest of the paper is as follow: In the next section we introduce the background definitions of production rule systems. In Sections 3 and 4 we present the steps to translate production systems into logic programs and relate stable models of the logic programs with the execution of the production systems. In the last section (Section 5) we briefly touch upon how this work can be extended.

2 Production systems

A production system is defined by a set of rules or productions together with a database of assertions, called the working memory (WM). For convenience in notation and the translation we will present later, we will define the syntax of the production rules and the working memory in terms of first-order logic. We will assume that there exists a function free first-order language $\mathcal{L}$ plus a finite set of production names $\mathcal{R}$.

Definition 2.1 A rule consists of a conjunction of literals from $\mathcal{L}$, called the left hand side (LHS) or antecedent of the rule, along with a conjunction of consequent actions, called the right hand side (RHS) of the rule. An action a is a term of the form $add(A)$ or $del(A)$, where A is an atom from $\mathcal{L}$. Each rule has associated a unique name $\mathbf{r}$ from a set of names $\mathcal{R}$. Rules will be denoted by terms of the following form:

$$\mathbf{r} : f_1, \ldots, f_n \rightarrow a_1, \ldots, a_m \tag{1}$$

where, $f_1, \ldots, f_n$ are literals and together referred to as the LHS of the rule, and $a_1, \ldots, a_m$ are actions and together referred to as the RHS of the rule, and $\mathbf{r}$ is the name of the rule.

The working memory of a production system is a set of ground atoms from $\mathcal{L}$.

A production system is defined by a finite set of productions and an initial set of atoms in the working memory. □

The LHS of a rule specifies a condition that must be verified in the WM. If the condition successfully matches with the data in the WM then the action in the RHS may be applied to the WM. That is, if all the positive atoms in the RHS match atoms in the WM, and none of the atoms in the negated literals matched with atoms in the WM, then atoms are added or removed from the WM according to the actions specified in the RHS. It is assumed that all the variables that appear in a rule have at least one occurrence in a positive literal in the LHS of the rule so that the matching of the positive literals ground the whole rule.[1]

The inference engine of a production systems repeatedly executes the following sequence of steps:

[1] This is a standard condition also used in databases to define safe queries.

1. For each rule, determine if the LHS matches the current WM.
2. Choose one of the matching instances of the rules according to some predefined criterion, called the conflict resolution strategy.
3. Add or remove from the WM the assertions in the RHS of the selected rule according to the actions specified in the rule.

The system stops when the application of any of the matching rules does not modify the WM.

Example 2.1 *The following rule:*

$$\mathbf{r_1}: \begin{array}{c} preference(X,C), \neg selected(X), \neg selected(C) \\ \rightarrow \\ add(selected(X)), add(selected(C)), add(teaches(X,C)). \end{array}$$

assigns teachers to courses according to the teacher's preferences. If the WM is initialized with the database:

$$\begin{array}{l} preference(bob, theory). \\ preference(bob, algorithms). \end{array}$$

and the conflict resolution strategy picks rules non-deterministically, there are two possible final states for this system:

$$\begin{array}{l} preference(bob, theory). \\ preference(bob, algorithms). \\ selected(bob). \\ selected(theory). \\ teaches(bob, theory). \end{array}$$

And

$$\begin{array}{l} preference(bob, theory). \\ preference(bob, algorithms). \\ selected(bob). \\ selected(algorithms). \\ teaches(bob, algorithms). \end{array}$$

□

Definition 2.2 We say that a production system *may terminate* if it has an execution that terminates. We say that a production system *terminates* if all its executions terminates. □

Definition 2.3 A production system is said to be *deterministic* w.r.t. a fluent f if all executions of P terminate with f as true. If a production system is deterministic w.r.t. all fluent literals then we say it to be *completely deterministic*. $\Box$

We now present the transformation of production systems in three steps. First, we will define a transformation for a *simple* set of production rule systems. We will initially assume that the system is propositional (i.e. no variables appear in the rules) and that the RHS of all rules are formed by a single action. Then, we will extend the transformation to rules with multiple actions. In the third step we explain how to introduce variables. We will use a non-deterministic selection as the conflict resolution rule. We will briefly describe in the conclusions how more elaborate conflict resolution rules can be incorporated.

3 Semantics of simple production systems

In this section we give the semantics of simple production systems in terms of a translation Π_s from production systems to logic programs with situation calculus notation.

The logic program $\Pi_s P$ where P is the production system program uses terms of four sorts. (We follow the convention of variables starting with capital letters and constants starting with small letters.)

- $fluents$: Fluent constants are denoted by f, f', and fluent variables are denoted by $F, F', \ldots$.
- $actions$: We have only two kind of actions, $add(f)$ and $del(f)$. Intuitively, the action $add(f)$ when executed makes the fluent f true in the next situation and the action $del(f)$ when executed makes the fluent f false in the next situation.
- $situations$: s_0 is a situation constant, and all other situation constants are constructed by application of a function res to a situation constant and an action. For example, if a denotes an action and s denotes a situation constant then $res(a, s)$ denotes a situation constant. Intuitively the situation constant $res(a, s)$ refers to the situation obtained by executing action a in situation s. For convenience, by $[a_1, \ldots, a_m]S$ we denote the situation $res(a_m, res(a_{m-1}, \ldots, res(a_1, S)) \ldots)$. The length of a situation is defined as the number of application of the function res in that situation.
- $rules$: Rule constants are denoted by r, r', and rule variables are denoted by $R, R', \ldots$.

The program $\Pi_s P$ consists of the translations of the individual production system rules, the initial state and certain other logic programming rules. The various predicates that are used in the translation have the following intuitive meaning.

- $actual(s)$: Intuitively, $actual(s)$ is true if s is an actual situation, i.e., the situation s is reached during an execution of the production system. The situation constants that are not actual are referred to as hypothetical. Although we can reason about these hypothetical situations these situations are not reached during the execution of the production system.

- $holds(f, s)$ and $hyp\_holds(f, s)$: Intuitively, $hyp\_holds(f, s)$ is true if f holds in the situation (hypothetical or actual) s, and $holds(f, s)$ is true if f holds in the actual situation s.

- $possibly\_occurs(r, s)$ and $occurs(r, s)$: Intuitively, $possibly\_occurs(r, s)$ is true if the rule r can be fired in the situation s, i.e., the antecedent of the production rule r are $true$ with respect to the situation s. But since there may be many such rules that can fired in the situation s, and the semantics of production systems dictates that only one such rule should be actually fired, we denote the firing of this particular rule by the atom $occurs(r, s)$. In other words, intuitively $occurs(r, s)$ is true if the rule r actually fires in situation s.

- $actual\_modification(a, s)$: Intuitively, $actual\_modification(a, s)$ is true if execution of a in situation s actually modifies the working memory.

- $ab(f, a, s)$: Intuitively, $ab(f, a, s)$ is true if the execution of action a in situation s may change the value of f.

- $last(s), terminates, finally(f)$: Intuitively, $last(s)$ is true if s is the situation where the production system program $terminates$, and $finally(f)$ is true if f is true at the end of the execution of the production system program.

We now present the translation $\Pi_s P$.

Translation 3.1 Translating Simple Production System Programs

Step (1) For a simple production rule of the form

$\mathbf{r} : f_1, \ldots, f_n \rightarrow a$

the program $\Pi_s P$ contains the following two rules:

(1.1) $possibly\_occurs(\mathbf{r}, S) \leftarrow holds(f_1, S), \ldots, holds(f_n, S), actual\_modification(a, S)$

(1.2) $actual(res(a, S)) \leftarrow actual(S), occurs(\mathbf{r}, S)$

Intuitively, the rule (1.1) means that if the antecedent of a production rule r is true with respect to a situation S, and if executing the action a in the consequent part of that rule changes the situation S, then the rule r may possibly occur in the situation

S. Intuitively, the rule (1.2) means that if a rule r gets fired in an actual situation S and the action a is in the consequent of the rule r, then $res(a, S)$ is an actual situation.

Step 2 (Initial State):

(2.1) $actual(s_0) \leftarrow$

If f is true in the initial state of the WM then $\Pi_s P$ contains the following rule.

(2.2) $hyp\_holds(f, s_0) \leftarrow$

Otherwise $\Pi_s P$ contains the rule:

(2.3) $hyp\_holds(f', s_0) \leftarrow$

Step 3 The other rules in $\Pi_s P$:

(3.1) Rules defining $actual\_modification$:

(3.1.1) $actual\_modification(add(F), S) \leftarrow holds(F', S)$

(3.1.2) $actual\_modification(del(F), S) \leftarrow holds(F, S)$

The rule (3.1.1) says that if a fluent F is false in a situation S and the action $add(F)$ is executed in S then this action actually modifies the situation S. The meaning of the rule (3.1.2) is similar.

(3.2) Rules defining $occurs$:

(3.2.1) $occurs(R, S) \leftarrow possibly\_occurs(R, S), not\ not\_occurs(R, S)$

(3.2.2) $not\_occurs(R, S) \leftarrow occurs(R1, S), neq(R, R1)$

The above two rules implement the *choice* operator which was first introduced in [10]. Intuitively, they guarantee that among all the rules that may possibly be fired in a situation S, only one gets fired in each stable model of the program and there is one stable model corresponding to each of the rules that may be possibly fired.

(3.3) Effect of actions:

(3.3.1) $hyp\_holds(F, res(add(F), S)) \leftarrow$

(3.3.2) $hyp\_holds(F', res(del(F), S)) \leftarrow$

(3.3.3) $ab(F', add(F), S) \leftarrow$

(3.3.4) $ab(F, del(F), S) \leftarrow$

(3.4) Inertia:

$hyp\_holds(F, res(A, S)) \leftarrow hyp\_holds(F, S), not\ ab(F, A, S)$

The rules in (3.3) and (3.4) do hypothetical reasoning about the actions $add(F)$ and $del(F)$ in the forward direction. The rules (3.3.1) and (3.3.2) describe the effect of these actions, the rule (3.4) encode the inertia or the frame axiom and the rules (3.3.3) and (3.3.4) define the abnormal predicate with respect to the frame axioms.

(3.5) Defining *holds*:

$holds(F, S) \leftarrow hyp\_holds(F, S), actual(S)$

The above rules make a fluent literal to hold in an actual situation S if it hypothetical holds in S.

(3.6) Defining *last*:

(3.6.1) $not\_last(S) \leftarrow occurs(R, S)$

(3.6.2) $last(S) \leftarrow not\ not\_last(S)$

The above two rules define the last situation.

(3.7) Defining *terminates*:

$terminates \leftarrow last(S)$

(3.8) Defining *finally*

$finally(F) \leftarrow last(S), holds(F, S)$

The above rule defines which fluent literals are true after the production system terminates. □

Proposition 3.1 *For any production system P, the logic program $\Pi_s P$ has at least one stable model.* □

Proof (sketch):
The instantiated version of $\Pi_s P$ can be split [LT94] into layers as follows:

- layer 3n consists of all rules with the predicates $possibly\_occurs$, $actual\_modification$, $holds$, $actual$, hyp_holds and ab in the head with the situation part in the head instantiated by a situation of length n.

- layer 3n+1 consists of all rules with the predicates *occurs* and *not_occurs* in the head with S in the head instantiated by a situation of length n
- layer 3n+2 consists of all rules with predicates *not_last* and *last* in the head with S in the head instantiated by a situation of length n, and all rules with predicates *terminates* and *finally* in the head with S in the body instantiated by a situation of length n.

It can be easily shown that the 3n and 3n-2 layers are acyclic, and hence the program obtained by partially evaluating them with respect to the layers below them has a stable model. The 3n+1 layers are not acyclic but use the choice construct and hence have stable models. Hence, using the splitting theorem in [LT94] this program has stable models. □

Theorem 1
1. A simple production System P *may* terminate iff there exists a stable model M of $\Pi_s P$ such that $terminates \in M$.
2. A simple production System P terminates iff for every stable model M of $\Pi_s P$, $terminates \in M$.
3. If the simple production system P terminates with fluent literal f true at the end then there exists a stable model M of $\Pi_s P$ such that $finally(f) \in M$.
4. If $\Pi_s P$ has a stable model M such that $finally(f) \in M$ then there exist some execution of the simple production system P which terminates with f true.
5. A simple production system is completely deterministic iff $\Pi_s P$ has a unique stable model containing *terminates*.
6. A simple production system is said to be deterministic w.r.t. a fluent literal f iff $\Pi_s P \models finally(f)$ □

Proof (idea):
The proof uses the splitting described in the proof of Proposition 3.1. □

3.1 Relation with earlier approaches

Before we present the semantics for more complex systems we would like to say a few words about Raschid's production systems [5, 6, 7, 8]. To ensure termination, Raschid modifies the execution of the actions and the matching process in the operational semantics of rule systems. Deletions do not remove elements from the WM. Instead, a marker is added to the WM indicating that a remove operation has been applied to the element. The matching is based on the following criteria:

1. A positive atom is satisfied if it matches with an atom in the WM and that atom has not been marked as deleted.
2. A negated atom is satisfied if the atom belongs to the WM but it has been marked as deleted.
3. A negated atom is satisfied if the atom does not belong to the WM (and it is assumed that has not been marked as deleted).
4. An atom that does not belong to the WM can not be marked as deleted.

The last condition is easily preserved in Raschid's systems since she assumes that when the RHS of a rule removes an atom, the atom must appear positively in the LHS of the rule.

In our transformation we only need to omit the rules $ab(F', add(F), S) \leftarrow$ and $ab(F, del(F), S) \leftarrow$ to model Raschid's semantics and replace the $finally$ rule with the rules:

$$finally(F) \leftarrow last(S), holds(F, S), not\ holds(F', S).$$

$$finally(F') \leftarrow last(S), holds(F', S).$$

A nice property of our transformation is that the final WM is not directly modeled with a stable model of the program. Instead we use the *hold* predicate and the situation variable. In Raschid and Lobo [7, 8] the models of the program were used directly. Hence, a final state of the WM could not be a proper subset of another possible final state since it will violate the minimality property of the intended models of the logic program restricting the class of production rule programs that could be modeled. Our semantics covers every program.

We now consider production systems that allow a sequence of actions in the consequent part of their productions rules. The earlier semantics of Raschid was restricted to simple systems, but most real production rule systems permit several actions in each rule.

4 Productions with multiple actions

Given a production system P we give its semantics using a translation to a logic program denoted by ΠP. The program ΠP consist of the following rules:

Translation 4.1 Translating Production Systems

Step 1: For a production rule of the form:

$$\mathbf{r} : f_1, \ldots, f_n \rightarrow a_1, \ldots, a_m \tag{2}$$

The program ΠP contains the following two rules:

$$\begin{aligned}(1.1')possibly\_occurs(\mathbf{r}, S) \quad \leftarrow \quad & holds(f_1, S), \ldots, holds(f_n, S), \\ & actual\_modification([a_1, \ldots, a_m], S)\end{aligned}$$

and

$$(1.2')actual([a_1, \ldots, a_m]S) \leftarrow actual(S), occurs(\mathbf{r}, S) \tag{3}$$

Step 2: Same as in Transformation 3.1 and consist of rules 2.1, 2.2, and 2.3.

Step3: In addition ΠP contains the rules 3.2.1-3.2.2, 3.3.1-3.3.4, 3.4, 3.5, 3.6.1-3.6.2, 3.7, 3.8 described in Transformation 3.1 and the following rules:

(3.1') Defining $actual\_modification$

(3.1.1') $actual\_modification([a_1, \ldots, a_m], S) \leftarrow holds(F', S), hyp\_holds(F, [a_1, \ldots a_m]S)$

(3.1.2') $actual\_modification([a_1, \ldots, a_m], S) \leftarrow holds(F, S), hyp\_holds(F', [a_1, \ldots a_m]S)$

□

Proposition 4.1 Every ΠP program has a stable model. □

Proof:
Similar to the proof of Proposition 3.1, except in the assignment of layers to rules with $actual\_modification$ and $actual$ in their head. □

Theorem 2

1. A production System P *may* terminate iff there exists a stable model M of ΠP such that $terminates \in M$.

2. A production System P terminates iff for every stable model M of ΠP, $terminates \in M$.

3. If the production system P terminates with fluent literal f true at the end then there exists a stable model M of ΠP such that $finally(f) \in M$.

4. If ΠP has a stable model M such that $finally(f) \in M$ then there exist some execution of the production system P which terminates with f true.

5. A production system is completely deterministic iff ΠP has a unique stable model containing *terminates*.

6. A production system is said to be deterministic w.r.t. a fluent literal f iff $\Pi P \models finally(f)$ □

We would like to remark that in the current translation the actions in the RHS of a rule that is fired are executed left to right. Hence, the rule $LHS \rightarrow add(a), del(a)$ will not produce the same effect as the rule $LHS \rightarrow del(a), add(a)$. By simple modifications of the effect actions we can model other semantics (such as the "simultaneous" execution of the actions in a rule).

4.1 Adding variables

When variables appear in a production rule, this rule can be considered to be the shorthand description of the set of rules that is obtained from all possible instantiations of the variables in the rule. In our previous translations the name of the rules were used by the predicate *occurs* to selected the rule to be fired. Since now a name is associated with a set of rules (instead of a single rule) the rule names are not descriptive enough to be used directly as a parameter of the *occurs* predicate. In the following translation we append the variables that occur in the rules to the rule names, so instantiations themselves will identify the actual instance of the rule to be selected.

Translation 4.2 Translating Production Systems with variables

Step 1: For each production rule of the form of Equation 1, the program ΠP contains the following two rules:

$$(1.1'')possibly\_occurs(\mathbf{r}(x_1,\ldots,x_l),S) \leftarrow holds(f_1,S),\ldots,holds(f_n,S), actual\_modification([a_1,\ldots,a_m],S).$$

and

$$(1.2'')actual([a_1,\ldots,a_m]S) \leftarrow actual(S), occurs(\mathbf{r}(x_1,\ldots,x_l),S)$$

The *vector of variables* $(x_1,\ldots,x_l)$ contains all the variables that appear in the RHS of the rule ordered by their textual occurrence.

Steps 2 and **3** are as in Translation 4.1. □

For example, the rule in Example 2.1 is translated into:

$$possibly\_occurs(\mathbf{r_1}(X,C),S) \leftarrow holds(preference(X,C),S),$$

$$
\begin{aligned}
& holds(\neg selected(X), S), \\
& holds(\neg selected(C), S), \\
& actual\_modification([add(selected(X), add(selected(C)), \\
& \qquad add(teaches(X, C)], S)
\end{aligned}
$$

$$
\begin{aligned}
actual(res(add(teaches(X, C)), res(add(selected(C)), res(add(selected(X)), S)))) \\
\leftarrow actual(S), occurs(\mathbf{r}(X, C), S)
\end{aligned}
$$

All properties in Theorem 2 also hold for production rules with variables.

5 Discussion and future work

In this paper we have given a different characterization of production systems that agrees with the original operational semantics. Since the characterization is based on translation to logic programs, and since databases also use the same language, the production system can be used together with (or on top of) a database system. We now briefly discuss how this work can be extended to allow conflict resolution strategies, and how several aspects of reasoning about actions can play a role in production systems.

5.1 Incorporating priorities

In the characterization of production systems in the past section we did not prioritize between the rules whose LHS was satisfied in each execution cycle. We gave equal priority to all of them. In reality, production systems allow prioritizing between these rules. Several different prioritizing strategies (also referred to as conflict resolution strategies), such as priority, specificity, recency, etc. are used. We now briefly sketch how these strategies can be incorporated in the logic programming translations.

Consider a predicate $preferable$ of arity two that defines a preference relation between two rules. For example, $preferable(R1, R2)$ means that the rule $R1$ is preferable to rule $R2$. At this point we do not worry about how the relation $preferable$ is obtained. (It may be directly given or it may be defined in terms of other conflict resolution parameters such as how recently the rules were used.) We assume that the relation given by $preferable$ is a partial order.

Using the relation $preferable$ we can now define a predicate $preferably\_occurs$ as follows:

$$not\_preferably\_occurs(R, S) \leftarrow possibly\_occurs(R', S), preferable(R', R), neq(R, R')$$

$$preferably\_occurs(R, S) \leftarrow possibly\_occurs(R, S), not\ not\_preferably\_occurs(R, S)$$

The predicate *occurs* is now defined in terms of *preferably_occurs* instead of *possibly_occurs* in the following way:

$$occurs(R, S)) \leftarrow preferably\_occurs(R, S), not\ not\_occurs(R, S)$$

$$not\_occurs(R, S) \leftarrow occurs(R', S), neq(R, R')$$

5.2 Issues that arise in 'Reasoning about Actions'

Let us now briefly consider some issues that arise in theories of actions. Let us extend Example 2.1 and add the predicate *teacher* in the database. Now if we have a rule in the production system that removes a certain teacher we would like to remove all his preferences. A simple way to deal with it would be to specify the removal of preferences of that teacher in the RHS of *all rules* whose RHS already contains $del(teacher(X))$.

In theories of actions such cases are elegantly represented using constraints between fluent that cause ramification [LR94]. This idea can be extended to production systems either by allowing constraints in the production system language or simulating them within the current framework.

Another idea from theories of actions that needs to be explored in the context of production systems is the use of more complex actions than the actions *add* and *del* that are used in production systems. The translation to logic programs described in this paper can be easily extended in the lines of [GL92] to allow more complex actions.

References

[1] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19, 20:73–148, 1994.

[GL92] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In *Joint International Conference and Symposium on Logic Programming.*, pages 559–573, 1992.

[2] H. Groiss. A formal semantics for a rule-based language. In *IJCAI Workshop on Production Systems and their Innovative Applications*, 1993.

[LT94] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. of the Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.

[LR94] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, October 1994.

[3] C. Maindreville and E. Simon. Modeling non-deterministic queries and updates in deductive databases. In *Proc. of VLDB*, 1988.

[4] A. Newell. Unified theories of Cognition. 1990.

[5] L. Raschid. Maintaining consistency in a stratified production program. In *Proc. AAAI National Conference on Artificial Intelligence*, 1990.

[6] L. Raschid. A semantics for a class of stratified production system programs. *Journal of Logic Programming*, 21(1):31–57, 1994.

[7] L. Raschid and J. Lobo. A semantics for a class of non-deterministic and causal production system programs. *Journal of Automated Reasoning*, 12:305–349, 1994.

[8] L.Raschid and J. Lobo. Semantics for update rule programs and implementation in a relational database management system. *ACM Transactions on Database Systems*. To appear.

[9] P. Rosenbloom, J. Laird, and A. Newell. The Soar papers: Readings on integrated intelligence. MIT Press, 1993.

[10] D. Sacca and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *Proc. PODS*, 205-217, 1990.

[11] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, 1995.