A Formal Semantics for mOPS5

Herbert Groiss
Institut für Informatik
Universität Klagenfurt
A-9022 Klagenfurt, AUSTRIA
e-mail: herb@ifi.uni-klu.ac.at

Abstract

In this paper a formal declarative semantics of the rule-based language mOPS5 is defined. As the name of the language says, it is a simplified version of OPS5. At first we describe informally this language, the semantics of mOPS5 is then defined in terms of Datalog$^{f\neg}$, a logic programming language, which has a well-defined minimal model semantics and fixpoint semantics. The availability of a clean semantics has several advantages: it allows program analysis and optimization as well as judging the correctness of implementations.

1 Introduction

Rule-based languages are widely used in the fields of Artificial Intelligence and expert systems, because they provide a problem-near knowledge representation for many domains. Popular representatives of this language class are OPS5 [6, 2], CLIPS [4] and SOAR [8]. In the field of databases rule-based languages have gained more attention in the last years as so-called Active Databases. Generally the semantics of rule-based languages is defined by giving an informal description of the evaluation algorithm and the conflict resolution strategy. This lack of formal definition prohibits the use of formal methods for program analysis. Moreover, the descriptions of the semantics is often incomplete so that some programs are indeterministic, porting such programs to another machine or version of the language may lead to unexpected results. Namely, in OPS5 the specification of the conflict resolution strategy is incomplete [6, 2], i.e. in certain cases an instantiation is arbitrarily chosen.

We have defined the language mOPS5 (for miniOPS5), a slightly restricted dialect of the popular rule-based language OPS5, and describe the semantics of this language formally. For formalizing the state changes a rule performs in the working memory we use a kind of situation calculus (originally from McCarthy [9]) which can be expressed in Datalog$^{f\neg}$, a subset of first-order predicate logic which has a well-defined model-theoretic and fixpoint semantics. Each mOPS5 rule can then be translated to a set of Datalog$^{f\neg}$ clauses.

There has been some approaches to formalize the semantics of rule-based languages, but in general for very simple languages or for languages with execution models different from the widely used rule-based languages [7]. In [14], Jennifer Widom presents a denotational semantics of the Starburst rule language. The main difference to languages like OPS5 is the kind of evaluation: In Starburst the evaluation is set-oriented, i.e., in each cycle, an action is applied to a *set* of tuples, not a single tuple. In [16] a fixed point semantics of rule trigger systems is presented. This allows it to formulate a sufficient criteria for identifying the class of programs computing an unique least fixpoint independent from the rule evaluation order.

The situation calculus is used to formalize database updates in [13]. Zaniolo [15] also uses situation calculus in combination with XY-stratified programs (similar to acyclic programs we use) to formalize the behavior of rule-based systems with set-oriented evaluation. Raschid [11] defines a declarative and fixpoint semantics for a rule-based language, which is slightly extended in [12]. However, only a restricted class of rule-based programs - stratified ones - are considered and the rule language is far from the power of OPS5. The main contribution of this work is the consideration and declarative description of rule priorities and tuple-oriented execution.

The rest of this paper is organized as follows: In the next section we describe informally the language mOPS5. The following two sections describe the syntax and semantics of Datalog and the extensions we define as well as the situation calculus. In section 5 we present the formal semantics of mOPS5 by giving the translation algorithm to Datalog. Concluding remarks and an outlook are given in section 5.

2 The Language mOPS5

We assume the reader is familiar with the language OPS5 and refer to [6]. The main restrictions of mOPS5 are: in the LHS no disjunctions are allowed, in the RHS only the actions make, modify, and remove are possible. The first restriction is only syntactical because disjunctions in the LHS can be factored into multiple rules.

The LHS of a rule is composed of one or more negated or non-negated condition elements (CEs). A CE consists of a class name and a set of attribute-value pairs. The attribute is specified by a caret '^' followed by a declared attribute name. The value is either a constant, a variable - represented by a name enclosed in angled brackets -, an expression, or a functional expression. An expression consists of a predicate ($<, <=, >, >=, <>$) and a value, a variable or a functional expression. A set of expressions enclosed in $\{\}$ indicates a conjunction. A functional expression is indicated with the keyword `compute` and evaluates arithmetic functions with the operators +,-,*,and /.

The RHS of a rule consists of a sequence of actions. The only actions allowed in mOPS5 are make, modify and remove. The arguments for make are a class name and a sequence of patterns like in the LHS. The first argument of modify and the only argument of remove is a number n indicating that the element matching condition n should be modified or removed. The remaining arguments of modify are attribute-value pairs like in the LHS.

Example 1. In a table `teams` the names of teams and their points are stored. The points of a team are raised by two points, when a play between the team and another one took place and the team shot more goals than the other team. The following rule implements this situation:

```
(p winner
   (play ^team1 <t1> ^team1 <t2>
         ^goal1 <g1>
         ^goal2 {<g2> < <g1>})
   (team ^name <t1> ^points <p>)
  -->
   (modify 2
       ^points (compute (<p> + 2)))
   (remove 1))
```

The working memory may contain the following facts:

```
1: (team t1 4)
2: (team t2 5)
3: (play t1 t2 6 4)
```

In this rule, the attribute `points` of the fact matching the second condition (`(team ...)`) is altered. Applying the rule on the facts results in an instantiation with fact 1 matching the first condition and fact 3 matching the second condition. Fact 1 is then altered to `(team t1 6)` and fact 3 is removed. □

2.1 Conflict Resolution

The conflict resolution strategies of OPS5 are MEA and LEX. They are incomplete and therefore not suitable for defining a semantics which gives every program a unique meaning. We use a slightly modified strategy which defines a total order over all possible rule applications. We first describe this strategy informal, in section 5 it is defined formally.

The rules are ordered with natural numbers from 1 to n. The order corresponds to the position of the rules in the source file. The facts are numbered according to the time of insertion in the working memory, this number is called the time-stamp.

The evaluation of a rule program is done in a cycle (recognize-act cycle) with the following steps: 1. match, 2. select, 3. apply.

In the match phase all instantiations containing working memory elements added in the previous cycle are inserted into the conflict set (cs). An instantiation is a tuple containing a rule and n working memory elements, where the i-th element matches the i-th rule condition. In the next step one of those is selected for evaluation, where the search order is the following:

1. Select from cs the instantiations containing the elements with the lowest time-stamp.

2. Select from the set produced by step 1 the instantiations with the rule with the lowest order.

3. if there is still more than one instantiation, compare at first the elements matching the first condition, then the elements matching the second condition, and so on. When the first difference is found, select the instantiation where the element has the lowest time-stamp.

In the third step the rule is applied and the rule actions are performed. The algorithm stops when the conflict set becomes empty. Although this strategy differs from the MEA and LEX strategies of OPS5, the differences will only come up in few programs which can be reformulated easily. Miranker [10] uses a similar conflict resolution strategy for the implementation of a OPS5 version using the LEAPS algorithm.

3 Datalog

In the following we assume the reader is familiar with first order logic. However, we begin by reviewing some well-known concepts of first-order logic and logic programming. The main notations used throughout the paper are presented in this section.

For a deeper introduction into the field of logic programming and databases we refer to [3] or [5].

3.1 Syntax

The syntax of Datalog is similar to that of PROLOG. A Datalog program consists of a finite set of *rules* and *facts*. Both, rules and facts are represented as Horn-Clauses with the following syntax: L_0 :- $L_1, ..., L_n$. Each L_i is a literal of the form $p_i(t_1, ..., t_n)$, p_i is a predicate symbol and the t_i are terms. A term can be a constant or a variable. Throughout the paper we use lowercase letters from the start of the alphabet to represent constants $(a, b, c, ...)$, lowercase letters from the end of the alphabet to represent variables $(s, ..., x, y, z)$.

The left-hand side of a Datalog clause is called its *head* and the right-hand side is called its *body*. The body of a clause may be empty. Clauses with an empty body represent facts, clauses with at least one literal in the body represent rules. A literal or clause which does not contain any variables is called *ground*. The set of predicate symbols *Pred* is divided into two parts, *EPred* (*extensional* predicates) contains all predicates occurring in facts stored in the database. *IPred* (for *intensional* predicates) is the set of the predicates occurring in the program but not in the extensional database.

To guarantee the *safety* of a Datalog program P, i.e. the finiteness of the set of facts that can be derived by the program, it must satisfy the following conditions: Each fact must be ground, each variable which occurs in the head of a rule of P must also occur in the body of the same rule.

3.2 Semantics

Each Datalog Fact F can be identified with an atomic formula of First-Order-Logic. Each Datalog rule R of the form specified above represents a first order formula R^* of the form $\forall x_1 ... \forall x_m (L_1 \wedge ... \wedge L_n \rightarrow L_0)$, where $x_1, ..., x_m$ are all the variables occurring in R. A set S of Datalog clauses corresponds to the conjunction of all formulas C^* such that $C \in S$.

The semantics of a logic program is defined by means of particular models of the program. Again, we make some definitions. The Herbrand base HB is the set of all ground facts of the form $p(c_1, ..., c_n)$, where p is a predicate symbol in *Pred* and all c_i are constants. Analogous to *EPred* and *IPred* we define the extensional *HB (EHB)* and the intensional *HB (IHB)*.

A Herbrand Interpretation *HI* is a subset of the *HB*, i.e. the set of ground facts holding in the interpretation. Implicitly, ground facts not in *HI* are false for *HI*. If a clause C is true under a given interpretation I, we say that this interpretation *satisfies* C and that I is a *model* for C. A Herbrand Interpretation M is a *minimal model* of a set of clauses F iff M is a model of F and for each M' such that M' is a model of F, $M' \subseteq M$ implies $M = M'$.

A ground fact $p(c_1, ..., c_n)$ is true under the interpretation I iff $p(c_1, ..., c_n) \in I$. A Datalog rule is true under I iff for each substitution θ which replaces variables by constants, whenever $L_1 \in I \wedge ... \wedge L_n \in I$, then it also holds that $L_0 \in I$.

The declarative semantics of a Datalog program P is simply defined as the minimal Herbrand model of P. The semantics of Datalog can also be defined with fixpoint theory, see [3].

3.3 Functions

Built-in predicates (or "built-ins") are expressed by special predicate symbols such as $<, >, \leq$, etc. with a predefined meaning.

In most cases built-ins represent infinite relations, therefore the Herbrand model of a Datalog program using built-ins is not necessarily finite. Safety can be guaranteed by requiring that each variable occurring as an argument of a built-in predicate in a rule body must also occur in an ordinary predicate of the same rule body, or must be bound by an equality (or a sequence of equalities) to a variable of such a predicate or a constant.

In a similar way, functions can be used, for example arithmetic functions $(+, -, *, /)$ or user-defined functions. A predicate $plus(x, y, z)$ can be used to express the relation $x + y = z$. The "input variables", x and y must occur in an ordinary predicate of the rule body. The function can then be evaluated as soon as these variables are bound. Note, that for guaranteeing the finiteness of the Herbrand model all arguments would have to be bound in ordinary predicates.

3.4 Negation

In pure Datalog, negated literals in rules or facts are not allowed. However, we may infer negative facts by adopting the closed world assumption (CWA): If a fact F does not follow from a set of Datalog clauses, then we conclude that the negation of F, $\neg F$, is true.

The extension of pure Datalog including negated literals in the body of rules is called Datalog$^\neg$. For safety reasons we require that each variable occurring in an negative literal in the rule body also occurs in a positive literal in the same body. A set of Datalog$^\neg$ clauses may have more than one minimal model. Stratified Datalog$^\neg$ programs are a subclass of Datalog$^\neg$, where one distinguished minimal model can be selected as the model of the program. For this we require, that each negative literal in the body of a rule can be evaluated before the predicate of the head of the rule is evaluated. If a program fulfills this condition it is called stratified. Any stratified program can be partitioned into disjoint sets of clauses $P = P^1 \cup ... \cup P^n$ called strata, such that each P^i contains only clauses whose negative literals correspond to predicates defined in lower strata. The evaluation is now done stratum-by-stratum. First, P^1 is evaluated, by applying the CWA locally to the EDB. Then the other strata are evaluated in ascending order.

A refinement of stratification is local stratification, where the Herbrand Base instead of the predicates is divided into strata. If the HB is infinite (due to the use of functions), we can have an infinite number of strata. A subclass of locally stratified programs are the so-called *acyclic* programs [1]. We define a level mapping $\mathcal{L}$: HB $\rightarrow \mathbb{N}$ of ground facts to natural numbers. If in every ground instance of every rule of P, $\mathcal{L}(L_i) < \mathcal{L}(L)$, i.e., the level of all literals of the body is less than that of the head, then program P is acyclic. The level of a literal also defines its stratum, every acyclic program is also locally stratified.

Example 2. Let us consider a logic program that defines the predicate $even$ for natural numbers:

$even(0).$
$even(y)$:– $succ(x, y), \neg even(x).$

$succ$ is the successor function, defined as: $succ(i, i + 1)$ for all $i \in \mathbb{N}$. The level mapping can be defined as $\mathcal{L}(even(x)) = x, \mathcal{L}(succ(x, y)) = 0.$
It is easy to see, that: $\mathcal{L}(even(x + 1)) > \mathcal{L}(even(x))$ and $\mathcal{L}(even(y)) > \mathcal{L}(succ(x, y))$. The program is therefore acyclic and, as a consequence, locally stratified. □

Acyclic programs can be evaluated using fixpoint iteration. In the following we denote by *Datalog*$^{f\neg}$ acyclic Datalog$^\neg$ with built-in predicates.

4 Situation Calculus

For formalizing database updates we use a variation of the *situation calculus*, originally developed by McCarthy, [9]. Those relations, whose truth values may vary from state to state, are called fluents and are denoted by predicate symbols taking a state term as additional argument. This state term specifies the particular state (or situation), in which the fact is true.

The main difficulty with this formalism is the so-called *Frame problem*: "Which facts holding in an earlier situation are still valid in a later situation?" or in other words: which facts are not invalidated by an action?

Here we use a simple solution, which is, however, less general than others (for example [13]). For each fluent p we define the predicates p' and $p^\neg$, either with the state term as additional argument. p' is used to emphasize that p holds in a state s, $p^\neg$ represents that p becomes false in a situation s.

A term $p(x_1, ..., x_n)$ is then true, if p' with the same arguments was true in an earlier state and has not been invalidated by $p^\neg$ since then. More formally:

Definition 3. $p(x_1, ..., x_n)$ is true in situation s, iff
$\exists s_1 \forall s_2 (p'(x_1, ..., x_n, s_1) \wedge \neg p^\neg(x_1, ..., x_n, s_2) \wedge$
$s > s_2 > s1).$ □

By using single numbers as state term, we can formulate this calculus in Datalog$^{f\neg}$. To guarantee acyclic programs, we demand that:

1. each fact has a unique state constant,
2. for all rules and all possible substitutions: the state variable of the head of the rule is greater than all state variables of literals in the body.

The level mapping can then be defined as:
$\mathcal{L}(p^*(x_1, ..., x_n, s)) = s$, where p is a predicate of *Pred* and * is either ' or ¬.

So far, we have defined the formal basis for the formulation of the semantics of mOPS5.

5 Translation of mOPS5 to Datalog$^{f\neg}$

In this section, we define the semantics of mOPS5 by giving an algorithm for translating a mOPS5 program L into an equivalent Datalog$^{f\neg}$ program P. The result of P, the minimal model M_P, is then retranslated to a set M_L, the result of the original program L.

As mentioned above, we use the situation calculus for describing the updates in the database. Each fact initially in the database or asserted by a rule has a unique state term s, represented by a sequence.

A fact asserted by a rule has the state term:

$$min(c_1, ..., c_n) + [O_r, c_1, .., c_n]$$

where c_1 to c_n are the state terms of the facts of the instantiation, + is the concatenation operator, and O_r is the order of the applied rule. The facts initially in the database have the state terms [1] to $[|db|]$, where $|db|$ is the cardinality of the initial database. This representation guarantees a unique state constant of all elements in the working memory.

Between two state sequences s and t the relation $<_s$ is defined as follows:

$$s <_s t \text{ iff } \exists m \forall k : 0 \leq k < n, s_k = t_k, and$$
$$(s_{k+1} < t_{k+1} \text{ or } |s| = k, |t| > k)$$

A sequence s is less than a sequence t, iff the first k elements of s and t are equal and the next element of s is less than the corresponding element of t or if the sequence s has k elements and t has more than k elements. For this total ordering we can define a polynom computing a single number for each state term. In the following the relations representing the functions which compute the state sequences for a rule r and action k are denoted by $next_{rk}(s_r, c_1, ..., c_n)$, where the c_i's are the input variables and s_r is the output variable. s_r is the sequence of $max(c_i)$ appended with $[O_r, c_1, .., c_n]$ and a sequence of $k-1$ zeros, where k is the position of the action in the right-hand side of the rule.

The translation of the facts from a mOPS5 program to Datalog$^{f\neg}$ is simply done by adding the state constant to each of them.

We can now formulate the translation procedure for the rules:

INPUT: a mOPS5 rule R with n conditions and m actions
OUTPUT: one or more Datalog$^{f\neg}$ rules.

1. Replace the RHS-action modify by remove and make.

2. If there is more than one action in the RHS, split the rule R into m rules R_1 to R_m with the corresponding actions in the RHS.

3. Conditions: The variables and constants of the attribute-value pairs are ordered according to the definition of the predicate, for attributes not appearing in the condition a variable unique in the rule is inserted. In the following X stands this sequence of terms.
The condition is then translated to $p'(X, s_i), \neg p^{\neg}(X, s'_i), s_i < s'_i < s_R$, where i is the position of the literal in the LHS of the rule, and p is the predicate.

4. The body of the k-th rule is completed with:
$next_{rk}(s_r, s_1, ..., s_n)$, where k is between 1 and m.

5. The `compute` functions and comparison expressions in the RHS of the rule are replaced with variables, the predicates for the arithmetic functions are added to the rule body.

6. The actions in the RHS of the rules are handled in the following way:

   a) `(make p X)`: the head of the rule is $p'(X, s_r)$

   b) `(remove i)`: the head is $p^{\neg}(X, s_r)$, where the predicate of the i-th condition is p and X is the sequence of terms for this condition.

With this steps done for all rules of a mOPS5 program L an equivalent Datalog$^{f\neg}$ program P is generated. The program P is acyclic, because $s_R > max(s_i)$, by definition of $next_{rk}$. The result of P is the minimal model M_P. The result of the original program L is the set of facts M_L, defined in analogy to definition 3:

Definition 4.
$M_L := \{p(x_1, ..., x_n) | \exists s_1 : p'(x_1, ..., x_n, s_1) \in M_P \wedge \forall s_2 > s_1 : p^{\neg}(x_1, ..., x_n, s_2) \notin M_P\}$ □

The following example shows how the translation of a mOPS5 program is performed:

Example 5. This program computes the sum of all `x` in elements `(element <x>)`, whenever an element `(sum 0 0)` is inserted into the database:

```
(p sum
  (element <i>)
  (sum ^res <j> ^number <k>)
 -->
  (remove 1)
  (modify 2  ^res (compute (<i> + <j>))
       ^number (compute (<k> + 1))))
```

On each rule application one matching element is removed from the working memory and the two attributes of the element `sum` are modified, where the first holds the sum, the second the number of elements.
The rule must be split into three rules, all with the following body (i runs from 1 to 3):

BODY:
$element'(i, s_1), \neg element^{\neg}(i, s'_1), s_1 < s'_1 < s_{ri},$
$sum'(j, k, s_2), \neg sum^{\neg}(j, k, s'_2), s_2 < s'_2 < s_{ri},$
$next_{sum,i}(s_{ri}, s_1, s_2)$

The complete rules are:

$element^{\neg}(i, s_{r1})$:– BODY.
$sum^{\neg}(j, k, s_{r2})$:– BODY.
$sum'(m, l, s_{r3})$:– BODY, $plus(i, j, m), plus(k, 1, l)$. □

The next example shows that the behavior of mOPS5 programs can be analyzed using the corresponding Datalog$^{f\neg}$ program.

Example 6. Consider the following two rules:

```
(a 0)
(p r1 (a <x>)
  --> (make a (compute (<x> + 1)))

(p r2 (a <x>)
  --> (remove 1))
```

Intuitively it is clear that the program behaves different depending on the priority of the rules: If rule r2 is fired first, the program stops. Otherwise, it does not terminate, because rule r1 computes p(x) for all $x \in \mathbb{N}$.
Due to the formal semantics we can investigate under which conditions the program does not terminate. The translation to Datalog$^{f\neg}$ yields to the following program:

$a'(0, [1])$.
$a'(y, s_{r1})$:– $a'(x, s_1), \neg a^{\neg}(x, s'_1), s_{r1} > s'_1 > s_1, next_{r1}(s_{r1}, s_1), plus(x, 1, y)$.
$a^{\neg}(x, s_{r2})$:– $a'(x, s_1), \neg a^{\neg}(x, s'_1), s_{r2} > s'_1 > s_1, next_{r2}(s_{r2}, s_2)$.

We examine now under which conditions the first rule does not fire: This is the case when the negated literal in the body, $\neg a^{\neg}(x, s'_1)$, becomes true, this fact can be asserted from the second rule. Therefore the rule does not fire, if

$$s_{r1} > s_{r2} > s_1, next_{r1}(s_{r1}, s_1), next_{r2}(s_{r2}, s_1)$$

From the definition of the function $next$, we can follow that this expression becomes true only if $O_{r1} > O_{r2}$, i.e. the order of r1 is lower than that of r2. If the order of the priorities is reverse, the program does not terminate. □

The next example shows how the semantics defines an ordering between multiple instantiations of the same rule:

Example 7. The rule r1 fires once and removes the tuples of the instantiation from the database.

```
(b 1)
(a 1)
(a 2)
(p r1 (a <x>) (b <y>)
    -->
  (remove 1) (remove 2))
```

The corresponding Datalog$^{f\neg}$ program is:
$b(1, [1])$. $a(1, [2])$. $a(2, [3])$.
$a^{\neg}(x, s_{r11})$:– BODY.
$b^{\neg}(y, s_{r12})$:– BODY.
where BODY is: $a'(x, s_1), \neg a^{\neg}(x, s_2), s_1 > s_2 > s$, $b'(y, s_3), \neg b^{\neg}(y, s_3), s_3 > s_4 > s, next_{r1}(s_1, s_2, s_{r1i})$.

The state variable of $a^{\neg}(1, s_{r11})$ is therefore:
$[2, P_{r1}, [2], [1]]$
and of $a^{\neg}(2, s'_{r11})$ it is: $[3, P_{r1}, [3], [1]]$
s_{r11} is smaller than s'_{r11}, the $level$ of the corresponding fact is therefore lower and the fact is derived. The next fact which is derived is $b^{\neg}(1, s_{r12})$. With this a fixpoint is reached and the minimal model is found. Note that selecting the other instantiation (with (a 2)) does not lead to a model of the program.
Therefore, the resulting database contains only the fact `(a 2)`. □

6 Conclusions

A formal declarative semantics of the rule-based language mOPS5 has been presented. The advantages of having a well-defined semantics are numerous: The semantics can be used as a basis for program analysis, for example to check programs for termination, redundancy, or unreachable rules. Further research should focus on the development of such tools. Optimization techniques based on rewriting the rules - as known for Datalog - should be applicable. Moreover, the formulation of mOPS5 in terms of Datalog$^{f\neg}$ should make the unification of active and deductive databases easier, because most languages for deductive databases are based on Datalog. This combination allows the development of languages suitable for a wider application area.

References

[1] Krysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9:335–363, 1991.

[2] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Addison Wesley, 1985.

[3] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[4] C. Culbert. *CLIPS Reference Manual*. Artificial Intelligence Section, Johnson Space Center, Houston, 1989.

[5] Johann Eder. Logic and databases. In V. Marik, editor, *Advanced Topics in AI*. Springer-Verlag, 1992.

[6] C. L. Forgy. Ops5 users's manual. Technical report, Department of Computer Science, Carnegie-Mellon University, 1981.

[7] Herbert Groiss. A formal semantics for a rule-based language. In *IJCAI-93 Workshop on Production Systems and their Innovative Applications*, 1993.

[8] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, September 1987.

[9] John McCarthy. Programs with common sense. In M. Minsky, editor, *Semantic Information Processing*, pages 403 – 418. MIT Press, 1968.

[10] Daniel P. Miranker and David A. Brant. An algorithmic basis for integrating production systems an large databases. In *Proc. of the National Conference on Artificial Intelligence*, 1990.

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

[12] Louiqa Raschid and Jorge Lobo. Semantics for update rule programs and implementation in a relational database management system. *to appear in Transactions on Database Systems*, 1995.

[13] Raymond Reiter. On formalizing database updates: Preliminary report. In *Proc. of EDBT*, 1992.

[14] Jennifer Widom. A denotational semantics for the starburst rule language. *SIGMOD Record*, 21(3):4–9, 1992.

[15] Carlo Zaniolo. A unified semantics for active and deductive database systems. In Norman W. Paton and M. Howard Williams, editors, *Rules in Database Systems*. Springer, 1993.

[16] Yuli Zhou and Meichun Hsu. A theory for rule triggering systems. In *Proc. of EDBT*, 1990.