Semantics for Update Rule Programs and Implementation in a Relational Database Management System [1]

Louiqa Raschid
Department of Information Systems and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Jorge Lobo
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60680

Abstract

In this paper we present our research on defining a correct semantics for a class of update rule (UR) programs, and discuss implementing these programs in a DBMS environment. Update rules execute by updating relations in a database which may cause the further execution of rules. A correct semantics must guarantee that the execution of the rules will terminate and that it will produce a minimal updated database. The class of UR programs is syntactically identified, based upon a concept that is similar to stratification. We extend the strict definition of stratification and allow a relaxed criterion for partitioning of the rules in the UR program. This relaxation allows a limited degree of non-determinism in rule execution. We define an execution semantics based upon a monotonic fixpoint operator T_{UR}, resulting in a set of fixpoints for UR. The monotonicity of the operator is maintained by explicitly representing the effect of asserting and retracting tuples in the database. A declarative semantics for the update rule program is obtained by associating a normal logic program $\overline{UR}$ to represent the UR program. We use the stable model semantics which characterize a normal logic program by a set of minimal models which are called stable models. We show the equivalence between the set of fixpoints for UR and the set of stable models for $\overline{UR}$. We briefly discuss implementing the fixpoint semantics of the UR program in a DBMS environment. Relations that can be updated by the rules are *updatable* relations and they are extended with two flags. An update rule is represented by a database query, which queries the updatable relations as well as database relations, i.e., those relations which are not updated by rules. We describe an algorithm to process the queries and compute a fixpoint in the DBMS environment and obtain a final database.

[1] This research is partially sponsored by the National Science Foundation under grant IRI-9008208 and by the Institute of Advanced Computer Studies.

للاستشارات

www.manaraa.com

1. Introduction

Since the introduction of deductive databases as a research area [Bocc86a, Bocc86b, KoGM87, Mink88, MUvG86, TsZa86], there has been a corresponding interest in extending the functionality of database management systems (DBMS) to provide support for rules. Some of this research has been described in [CeWi90, DeEt88, MaSi88, SeLR88, SeLR93, SiMa88, Wido91, WiFi90]. Whereas rules in deductive database systems have been primarily perceived as supporting retrieval queries against the database, most DBMS research has focused on supporting rules whose execution can update the database and trigger the further execution of rules. This paradigm is similar to the forward-chaining, rule-based production system paradigm in artificial intelligence research [Haye75]. The motivation for choosing this paradigm is partly because triggers and integrity constraints, which are common in DBMS, also update the database, and are often implemented in a similar fashion to forward-chaining rules.

The success of research in deductive bases, where Horn logic programs are integrated with function-free first-order relational databases, can be largely attributed to the fact that there is a strong mathematical foundation for deductive databases. Horn logic programs have a declarative semantics and an equivalent fixpoint semantics [vEKo76]. However, the same is not true for database updates and rules whose execution may update the database. There has been some research in the area of precisely defining the semantics associated with updates. In [FKUV86], a framework for dealing with the general problem of database updates is presented. Since then, there has been research in updates in logic programs [MaWa88, NaKr88] where the semantics is defined using a Dynamic Logic, and there has been research in very powerful procedural languages that are equivalent to Datalog-like extensions with fixpoint semantics [AbSV90, AbVi90, AbVi91, SiMa88]. Most DBMS implementations of production rules [DeEt88, MaSi88, SeLR93, SiMa88, WiFi90] have an operational or execution semantics defined for them.

In this paper, we are interested in defining a semantics for update rules in a DBMS, where rule execution can update the database and cause further execution of rules. Our research focuses on supporting update rule programs (a collection of update rules and an extensional database of facts) in an extended DBMS environment. Although there has been previous research in the general area of providing a semantics for updates in a logic program or in a DBMS, and in defining powerful rule languages for updates, as mentioned, this problem has not been studied in the context of DBMS implementations. Our motivation is to define a class of update rule programs which can be identified in a fairly straightforward manner by a DBMS designer, and which would be useful in DBMS applications such as to capture triggering information, to maintain integrity constraints, or to make simple inferences from the information in the database. An important criterion in defining this class of update rule programs is that its semantics must be easily, and correctly, implemented in a relational DBMS with some extensions.

There are two problems that must be resolved in providing a correct execution semantics for update rules in a DBMS. One problem is to deal with non-terminating execution of update rule programs where rules insert and/or delete the same tuples indefinitely. The second problem arises when there are negative literals in the rules. With different execution schedules that execute rules in different sequences, a database produced by some schedule may not be minimal. Informally, this means a database produced by one schedule may be a subset of a database produced by another schedule. This occurs because negative literals that are evaluated by the rules may be affected by subsequent insertions and/or deletions of tuples, by other rules.

To deal with the first problem of non-terminating programs, we define a fixpoint semantics for update rule programs based on a monotonic fixpoint operator. The monotonicity of the fixpoint operator is achieved by explicitly recording the effects of inserts and deletes of tuples in the database. The retraction of a tuple (by a rule) is not affected by subsequent assertions of the same tuple (by other rules), and this ensures the termination of the execution. The second problem of how to deal with negative literals has been researched within the context of logic programs. Stratification [ApBW88] is a technique used to identify a special class of logic programs where the rules have negative literals. Stratification imposes an ordering between the rules and such an ordering supports a correct interpretation of the negative literals in the rules. In previous research [Rasc90, Rasc94], we defined a strict

www.manaraa.com

stratification criterion for the update rules. However, stratification is an overly strict criterion, and there are many update rule programs which, while not stratified, capture some very useful properties in a DBMS [RaLo94]. A further restriction of stratification is that the execution of the program must be deterministic, i.e., it must produce a single answer. However, there are many programs that are naturally non-deterministic. For example, when the database does not satisfy a given integrity constraint, then there may be several update rules which could be applied to restore consistency. These rules could be non-deterministically chosen, depending on the application, and could result in different answers.

In this paper, we relax the strict definition of stratification for update rules, which was first introduced in [Rasc90, Rasc94]. We present a more flexible definition for partitioning the update rule program UR into a set of partitions, $UR_1 \dot{\cup} .... \dot{\cup} UR_n$. Each partition may include cliques of rules which do not satisfy the criterion for strict stratification. As with strictly stratified programs, there is an implicit ordering among the update rules from one partition to the next. We define a fixpoint semantics for the UR program based upon a fixpoint operator T_{UR}. Rules in partition UR_i must be executed and a fixpoint obtained before executing rules in the partition UR_{i+1}. Since rules are non-deterministically selected for execution, there are several possible fixpoints. Each fixpoint for the UR program corresponds to a final database. The monotonicity of the fixpoint operator, the condition that once a tuple is retracted it is not affected by subsequent assertions (of the same tuple) and the stratification criterion results in an execution which is guaranteed to terminate and the execution will also produce a minimal database.

To provide a precise definition for the final (minimal) database(s), we associate a normal logic program $\overline{UR}$ with each UR program. The translation to obtain this program $\overline{UR}$, from UR, interprets rules which update the database by retracting tuples as integrity constraints. This is different from other approaches, where retractions in the heads of rules are treated as negative literals in the heads of the rules. We believe our interpretation is more natural for DBMS applications, and we discuss this in the section on related research. We use the stable model semantics for normal logic programs [GeLi88] which characterizes a normal logic program with a set of minimal models called *stable models*. A stable model for the program $\overline{UR}$ must be identical to a final database obtained when a fixpoint is reached from executing the rules of UR, and vice versa. The translation to obtain $\overline{UR}$ and the declarative semantics for UR, based on stable models is described in detail in this paper.

Implementing the UR program in a DBMS is straightforward and computing the fixpoint for the rules is similar to query processing in a DBMS. Each predicate in the UR program corresponds to a relation. The *updatable* relations that are updated by the rules are extended with two additional attributes. This extension allows us to explicitly store information on tuples which are inserted or deleted by the rules. The rules themselves are translated into queries which retrieve tuples from the updatable relations and the database relations (which are not updated by the rules). Queries corresponding to the rules in each partition are executed, in turn, until the queries in the partition can no longer update the updatable relations. We note that the complexity of query evaluation is not affected due to the additional attributes used with the updatable relations. The simplicity of the implementation motivates the choice of this criterion for relaxing the strict stratification when defining the acceptable update rule programs.

This paper is organized as follows: In section 2, we identify the syntax of the UR programs and provide some example programs to illustrate the problems that arise during execution of UR programs. In section 3, we introduce the concept of stratification and define UR programs which are strictly stratified. We then describe two simple extensions to the stratification conditions. Each extension introduces *cliques* of rules (which fail the strict stratification criterion) into each partition, and provides a more flexible criterion for partitioning the rules of the UR program. The usefulness of these extensions is motivated with example UR programs. In section 4, we define the fixpoint semantics for the UR programs defined in section 3. In section 5, we provide a translation to obtain a normal logic program $\overline{UR}$, and we present the equivalent declarative semantics based on the stable model semantics for $\overline{UR}$. In section 6, we provide a proof that the fixpoint semantics for UR is sound and complete wrt the declarative

المنارة للاستشارات
www.manaraa.com

semantics for $\overline{UR}$. In section 7, we discuss related research on providing a semantics for updates, and we compare these approaches with our research on semantics of UR programs. In section 8, we outline a method for implementing the fixpoint semantics of the UR programs (described in section 4) in a relational DBMS. Finally, in section 9, we summarize our research and we discuss future research. Appendix 1 provides a translation from an update rule program to a normal logic program in the first order case.

2. A Syntax for Update Rule Programs

In this section we introduce the syntax for specifying update rules. We borrow concepts from the OPS5 production system language [Forg81 and Forg82], as well as the Datalog language with procedural extensions [AbVi90, AbVi91].

2.1. Syntax for Update Rules

An *update rule* consists of (1) the **name** (2) the **antecedent** on the left hand side (LHS), also referred to as the body of the rule, (3) the symbol $\longrightarrow$, and (4) the **consequent actions**, on the right hand side (RHS), also known as the head of the rule.

The antecedent is a conjunction of first order positive literals of the form $P(\bar{u})$ or negative literals of the form $\neg Q(\bar{v})$ or *evaluable* predicates [Ullm89] of the form: eval(op, arg_1, arg_2). The **consequent actions** are of the form **assert** $R(\bar{u})$ or **retract** $S(\bar{v})$, where the **assert** and **retract** are special constants reserved for the assert and retract operations (to be explained later) and R and S are predicates. Let P, Q, R, S, etc., be predicates which are represented by relations in the DBMS. The relations corresponding to predicates that occur in the actions in the heads of the update rules are *updatable* relations, e.g., R or S. Relations that correspond to predicates that only occur in the body of update rules, but not in the heads of rules are database relations. They are not updated by the update rules but they can be updated by explicit insertions and deletions, just as the updatable relations. Literals in the body of a rule can refer to either database or updatable relations. $\bar{u}$ and $\bar{v}$ are vectors of terms from a non-empty finite or infinite set of constants C, and from a set of variables X. An evaluable predicate *eval* is a binary built-in predicate supported by many database systems [Ullm85]; op is one of the binary comparison operators such as $\{<, \leq, >, \geq, =, \neq\}$ and arg_1 and arg_2 are either variables or constants from the set C. We assume that there exists a total order in the elements of C [1].

We assume that all variables are *range-restricted* [Demo82, Nico82, NiDe83], i.e., any variable that occurs in a literal, or in the evaluable predicate eval, must appear in a positive literal in the body of a rule. This restriction correspond to the safety of evaluating queries [1]. It ensures that only ground atoms are inserted into the updatable relations through the execution of the actions of the update rules. The syntax restricts the literal referred to by the **retract** action, to occur positively in the antecedent of that rule, so that only ground atoms (tuples) that are actually in the database, (in the updatable relations) are retracted through the execution of the update rules.

A *function-free update rule program* UR, consists of the following [2]:

---

[1] Note that the set of constants usually corresponds to a collection of sets of different types, e.g., integer, string, etc. To simplify the discussion, we assume all the constants to be of the same type.

[1] Note that the variables are not also instantiated to safely computable values corresponding to either arithmetic functions or aggregate functions, as is common in a DBMS. It is straightforward to modify the syntax for the rules to include such safely computable functions, corresponding to the variables occurring in a predicate.

[2] Note that an action in the head of a rule which updates the value(s) of one (or more) variable(s) of a literal (i.e., attributes of a corresponding tuple), can be interpreted as a pair of corresponding **assert** and **retract** actions. Such an update rule can be rewritten as a pair of appropriate a-update and r-update rules, relevant to this literal.

www.manaraa.com

(i) A set of update rules, each of which has a *single* [3] **assert** action in its head. These are *a-update* rules and they *assert* the literal occurring in the head of the rule, or they insert tuples into the corresponding updatable relation.

(ii) A set of update rules, each of which has a *single* [3] **retract** action in its consequent. These are *r-update* rules and they *retract* the literal which occurs in the head of the rule, or they retract tuples from the corresponding updatable relation.

(iii) An initial extensional database of ground atoms EDB_{init}.

We informally describe the execution of an update rule program. The antecedent of each rule is interpreted as a query against the relations, as follows: (1) For each of the positive literals, P($\bar{u}$), relation P is queried, and a set of *instantiated* tuples of P satisfying each positive literal in the antecedent is retrieved. (2) For each of the negative literals, $\neg$ Q($\bar{v}$), the query is verified against the corresponding relation Q. Each variable x in $\bar{v}$ is range restricted to the value obtained by evaluating a query for some positive literal P in which x occurs. (3) Finally, for each occurrence of the evaluable predicate, eval, it is evaluated by the DBMS to return a value which is either true or false. Since the variables in the eval predicate are range-restricted they can be evaluated safely; this is discussed later. The antecedent of an update rule is *satisfied* if the relations contain instantiated tuples corresponding to each of the positive literals, if the relations do not contain tuples corresponding to the negative literals, and if all occurrences of the eval predicate in the body of the rule evaluate to true. Rules whose antecedents are satisfied are (non-deterministically) selected for execution and they update the database. Depending on the action in the head of the selected rule, the updatable relations are updated. Either new facts are asserted by the a-update rules or existing facts are retracted by the r-update rules. This process continues until the updatable relations can no longer be updated.

2.2. Problems in Executing Example Update Rule Programs

There are several problems that may arise when executing a set of update rules in a DBMS. If we interpret the **assert** and **retract** actions in a straightforward manner by inserting and deleting database tuples from the corresponding relations, then there is a possibility that the execution of the program may not terminate and the relations may be updated indefinitely. The second problem is that when there is more than one execution schedule, the execution may not produce a minimal database. We present some motivating example programs that highlight these problems.

Example 1

Consider the update rule program with relations *Employee, GoodWorker, Manager* and *Unfriendly*, where the initial database has the tuples, {Employee(Mike). GoodWorker(Mike).}, and with the following set of update rules:

r_1: Employee(X), GoodWorker(X) $\rightarrow$ **assert** Manager(X)

r_2: Employee(X), $\neg$ HasOffice(X) $\rightarrow$ **assert** Unfriendly(X)

r_3: Manager(X), Unfriendly(X) $\rightarrow$ **retract** Manager(X)

The first rule promotes employees who are good workers to managers, while the second rule asserts the fact that employees who do not have offices become unfriendly. Finally the third rule rescinds the promotion of managers who are unfriendly. Given this initial database and set of rules, the rules will execute until a fixpoint is reached and the database can no longer be updated. Rules r_1, and r_2 will execute (in any order) and the tuples

[3] An update rule program in which a rule can have multiple actions in the head can be replaced with an equivalent update rule program where each rule has a single action in the head. This may require the introduction of special predicates, in the new program, which are previously unused in the original program.

www.manaraa.com

Manager(Mike) and Unfriendly(Mike) will be added to the database. Next, the rule r_3 executes and the tuple Manager(Mike) will be deleted from the the database. Subsequently, r_1 and r_3 will execute, first inserting the tuple Manager(Mike) and then deleting this tuple. Processing of these two rules could continue indefinitely, alternately inserting and deleting the tuple Manager(Mike) from the Manager relation, and the UR program may not be able to terminate.

Example 2

This is an example where the update rules produce different databases, due to differences in selecting and executing the rules.

The initial database = {Employee(Mike). GoodWorker(Mike)} and the update rules are as follows:
r_1: Employee(X), GoodWorker(X) $\rightarrow$ **assert** Manager(X)
r_2: Employee(X), Manager(X) $\rightarrow$ **assert** IncreasePay(X)
r_3: Employee(X), $\neg$ Manager(X) $\rightarrow$ **assert** DecreasePay(X)

The first rule promotes employees who are good workers to managers. The second rule increases the salary of managers, while the third rule decreases the salary of employees who are not managers. The problem here is that depending on the order in which the rules are selected for execution, there will be two different answers obtained. For example, if the update rules are executed in the sequence r_1 followed by r_2, then Mike will be promoted to a manager and his salary will be increased. However, if the execution sequence was r_3 followed by r_1 and r_2, then, first, Mike's salary will be decreased, following which Mike will be promoted to a manager, and his salary will be increased. It is in this sense that we say that the second execution is not minimal or does not produce a minimal database compared to the first execution. Informally, the database resulting from the asserts and retracts of the executed rules, produced by one execution schedule, includes all the answers corresponding to a different execution. The formal definition of a minimal database will be introduced in section 6. We will define the equivalent declarative stable model semantics, and the corresponding (minimal) stable models which correspond to the minimal databases.

Example 2 also points out the difference between a deterministic and a non-deterministic execution. Suppose we were to consider a deterministic rule execution strategy that allowed all the rules to execute simultaneously. Then, the rules r_1 and r_3 would execute and Mike would be promoted to a manager and his salary would be decreased simultaneously. Then rule r_2 would execute and his salary would be increased. Such an execution is deterministic in that there can be only one final answer.

Another variation results when we consider the execution of a single rule. Consider the following rule:
r: ManagerOf(P, X), ManagerOf(P, Y), Employee(P) $\rightarrow$ **retract** ManagerOf(P,X)

Suppose Mary is an employee with two managers Jane and Martha. Then, the set of tuples {ManagerOf(Mary, Jane), ManagerOf(Mary, Martha)} will satisfy the literals ManagerOf(P,X) and ManagerOf(P,Y). A deterministic execution will then delete both the tuples of ManagerOf leaving Mary without a manager! However, a non-deterministic execution will only delete one of those tuples. These issues relating to deterministic or non-deterministic execution and tuple-oriented or set-oriented execution have been studied in different contexts. Researchers who study theoretical issues have reported on such executions in [AbVi90, AbVi91] and it has also been studied from an implementation viewpoint since the set-oriented execution can provide more efficient execution [PaGo91, WiFi91]. We will discuss this issue in a later section where we compare our research with other research.

3. Identifying Acceptable Update Rule Programs

In the previous section, we saw examples of update rule programs whose execution did not terminate, or did not produce minimal databases. To avoid this problem, we require a method to syntactically identify legal update

المنارة للاستشارات
www.manaraa.com

rule programs and define a semantics for these programs that disallows this behavior. One reason for the undesirable behavior of programs is that negative literals in the body of update rules must be correctly interpreted, as the tuples are asserted or retracted. Informally, an error occurs if a literal occurring negatively in a rule is interpreted to be false, while there is another rule which can assert a tuple which can prove this literal. Similarly, an error may occur if a literal occurring negatively in the body of a rule is interpreted to be true, while there may be another rule which can retract some tuple so that after the retraction, the literal is no longer true. In order to solve this problem, we apply the concept of stratification to partition the rules of a update rule program. The order imposed by this partitioning of the rules will order the sequence of execution of the update rules. As a result, the literals will be interpreted correctly, as defined by the corresponding fixpoint semantics for the update rule program.

Stratified logic programs are an extension of Horn programs to more general logic programs to allow negative literals in the antecedent of a rule. Since we apply the concept of stratification in our research, we informally define it in this section, as it applies to a stratified logic program [ApBW88]. We refer the reader to [ApBW88] for a discussion on stratified logic programs.

A logic program P consists of a finite set of rules of the following form:

$A \leftarrow L_1, L_2, ..., L_m$

where A is an atom and each of the L_i are literals. If m=0, then A is a fact.

A logic program P is stratified if there exists a *partition* $P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$ where $\dot{\cup}$ is a disjunctive union, such that the following conditions hold for i = 1, 2,...,n:

(1) for each predicate symbol, corresponding to a literal occurring *positively* in the body of a rule in P_i, its definition is contained within $\bigcup_{j \leq i} P_j$. The definition (of a predicate symbol) is all rules in which the predicate symbol corresponds to the literal occurring positively in the head of the rule.

(2) for each predicate symbol, corresponding to a literal occurring *negatively* in the body of a rule in P_i, then its definition is contained within $\bigcup_{j < i} P_j$.

P_1 may not contain any rules of P. We say that P is stratified by $P_1 \dot{\cup} \ldots \dot{\cup} P_n$ and each P_i is called a stratum of P.

3.1. A Stratified UR program

Definitions

An update rule is an *a-update* rule if it has a single **assert** action in its head.

An update rule is an *r-update* rule if it has a single **retract** action in its head.

A rule is *relevant* to a predicate Q if the literal Q occurs in the head of the rule.

A rule p has a *positive/negative dependency* on the predicate Q when Q occurs in a positive or negative literal in the body of the rule. The predicate Q is a distinguished predicate in the rule.

A predicate P is retractable if there is an r-update rule relevant to that predicate, otherwise the predicate is unretractable. □

A stratified update rule program UR, is a function-free stratified program and comprises the following:

(1) a set of a-update rules, as follows:

المنارة للاستشارات
www.manaraa.com

r_1: $A_1, ..., A_a, \neg B_1, ..., \neg B_b \rightarrow$ **assert** P

(2) a set of r-update rules, as follows:

r_2: $P, C_1, ..., C_c, \neg D_1, ..., \neg D_d \rightarrow$ **retract** P

(3) an initial database of facts (EDB_{init}).

Note that we have not considered the evaluable predicates in the body of the rules. This is because the presence of the evaluable predicates does not affect the stratification criterion for the program.

There must exist a partition so that UR is a stratified program. Thus, $UR = UR_0 \dot{\cup} UR_2 \ldots \dot{\cup} UR_n$. Each of the partitions UR_i comprises a set of a-update rules and a set of r-update rules; the sets of a-update rules or r-update rules in each partition may be empty. Partition UR_0 corresponds to the initial database EDB_{init} and there are no rules in UR_0. The following conditions must hold for the stratification of the UR program:

(1) $UR = UR_0 \dot{\cup} UR_1 \dot{\cup} \ldots \dot{\cup} UR_n$

(2) For every distinguished predicate, occurring in a positive literal A_k or C_k, in a rule such as r_1 or r_2 in UR_i, all relevant a-update rules (in which the predicate occurs in the **assert** action, in the head), must be contained within $\bigcup_{j \le i} UR_j$.

(3) For every distinguished predicate, occurring in a positive literal A_k or C_k, in a rule in UR_i, all relevant r-update rules (where the predicate occurs in the **retract** action) must be contained within $\bigcup_{j < i} UR_j$.

(4) For every distinguished predicate, occurring in a negative literal C_k or D_k, in a rule in UR_i, all relevant a-update rules must be contained within $\bigcup_{j < i} UR_j$.

(5) For every distinguished predicate, occurring in a negative literal C_k or D_k, in a rule in UR_i, all relevant r-update rules must be contained within $\bigcup_{j < i} UR_j$.

(6) Finally, for the distinguished predicate occurring in a positive literal P, in an r-update rule such as r_2, which has **retract** P in its head, in UR_i, all relevant rules with P in the head must be contained within $\bigcup_{j \le i} UR_j$. □

Note that condition 6, together with the syntactic restriction that the distinguished predicate P must occur in a positive literal in an r-update rule such as r_2 that has (**retract** P) in its head, will place *all* r-update rules with (**retract** P) in their heads in the same partition. In addition, it will not admit r-update rules with **retract** P in the head, and where some other distinguished predicate A_k or B_k is also P.

Based on this stratification condition, we note that in Example 2, the update rule r_3 will be placed in a higher partition w.r.t. the other two rules. The impact of this partitioning on the execution of the rules is to ensure that a minimal database is obtained. The proof of obtaining a minimal database for the strictly-stratified programs is in a paper on related research [Rasc94].

3.2. Relaxing the Conditions for Stratification

Enforcing that a UR program is stratified is a very strict condition which eliminates many useful programs. When integrity constraints are maintained by r-update rules, or when a-update rules deduce new information, these rules query the database and make conditional updates. The queries and conditional updates made by two or more

المنارة للاستشارات
www.manaraa.com

rules may violate the strict stratification criterion previously described. It would therefore be advantageous to relax the stratification criteria so as to accept more UR programs. However, the relaxed criterion must still ensure that the execution of programs terminates and produces minimal databases. If we consider the problem of combining multiple knowledge bases, and answering queries and maintaining a combined knowledge base, then it becomes very important that the strict stratification criterion be relaxed. We motivate the relaxations by presenting example UR programs not meeting the strict stratification criteria. Nonetheless, they are useful programs. We also note that the strictly-stratified programs are deterministic and compute a single answer; this can be a drawback in many instances.

3.2.1. First Relaxation

We first relax the stratification conditions to allow r-update rules that are relevant to some set of predicates to co-exist in the same partition when they all have a mutual positive dependency on all the predicates in that set. This exception allows us to include rules which are illegal under the initial strict definition of stratification. Consider a database with relations corresponding to the relations *person*, *male* and *female*. A person can be either a male or a female, but not both. This is a constraint that we want to be enforced. The following pair of r-update rules maintain this constraint.

Example 3

r_1: person(x), male(x), female(x), choice(x, "m") $\rightarrow$ **retract** female(x)

r_2: person(x), male(x), female(x), choice(x, "f") $\rightarrow$ **retract** male(x)

In the event that the database is updated so that it violates this constraint, then, either the fact that a person is a male or that she is a female must be retracted from the database, i.e., the corresponding tuple must be retracted. Since this is not an arbitrary decision which could update the database in a non-deterministic manner, in this example, we introduce an additional predicate *choice*. This predicate is used to prompt the user to select whether the person is a male or a female. Depending upon the value of (the second attribute of) choice, either the tuple satisfying female(x) or male(x), for some bound value of variable x, will be retracted from the corresponding relation.

If we apply the strict definition of stratification to these rules, condition (3) of this definition in section 3.1 requires that rule r_1 be in a lower partition than rule r_2 and that r_2 be in a lower partition than r_1. Obviously, these two requirements cannot be simultaneously satisfied, and the program would be rejected.

This situation is characterized by two or more rules querying the same subset of predicates in the body of the rule(s), or querying some mutually overlapping subsets of some set of predicates. The update actions of the rules are different, and the rules are designed so that usually all the rules are not expected to be executed. Typically, this situation occurs when an integrity constraint is being verified against the database. When the database violates the constraint, there may be many different ways to restore consistency, corresponding to each rule. One of the ways is chosen, either selected by the user as in this example, or more non-deterministically, when a particular rule is selected by the system. Thus, these rules will check the same conditions in the body of the rules, while the actions in the head of each rule will be different. As a result, the rules will fail the strict-stratification criterion.

Example 4

Consider another example where the rules do not satisfy the strict stratification criterion but are still useful. In this example, two different answers can be non-deterministically obtained by executing the rules. The database has relations corresponding to the predicates *manager*, *goodworker* and *unfriendly*, and the rules are as follows:

r_1: manager(x), unfriendly(x) $\rightarrow$ **retract** manager(x)

المنارة للاستشارات
www.manaraa.com

r_2: manager(x), unfriendly(x), goodworker(x) $\rightarrow$ **retract** unfriendly(x)

r_3: unfriendly(x), goodworker(x) $\rightarrow$ **retract** goodworker(x)

Consider Joan who is a good worker and a manager, but is unfriendly. We can either execute rules r_1 and r_3 which maintain the constraints that an unfriendly person should not be a manager, and that an unfriendly person is not a good worker, respectively. The final database is Joan is an unfriendly person. Alternately, we can execute rule r_2 which maintains the constraint that a person who is a manager and also a good worker is not an unfriendly person. The final database is that Joan is a good worker and a manager.

In these cases, instead of rejecting the program, our solution is to relax the strict stratification criteria, so that these programs are accepted. The relaxation to accept rules such as those in Examples 3 and 4 applies to condition (3) of the stratification criterion. We first identify a *clique* of r-update rules that are relevant to a set of predicates, say $\{P_1, ..., P_n\}$. These r-update rules in the clique may co-exist in the same partition when they each have mutual positive dependencies on these same distinguished predicates, $P_1, ..., P_n$.

Definition

A set of r-update rules participate in an *r-clique* if there is some *maximal* set of predicates $\{P_1, ..., P_n\}$ such that each rule in the r-clique is relevant to some predicate from this set; if there is at least one rule in the r-clique which is relevant to each predicate in this set; and if all of the predicates in the set are distinguished and occur in a positive literal in each rule of this r-clique. All predicates in this maximal set are *retractable* in this r-clique. □

The following r-update rules form an r-clique with a maximal set of predicates $\{P_1, ..., P_p\}$, where each P_i is a distinguished predicate in each r-update rule in the r-clique:

r_1: $P_1(\overline{u_{1,1}}), P_2(\overline{u_{1,2}}), ..., P_p(\overline{u_{1,p}}), A_{1,1}, ..., A_{1,a1}, \neg B_{1,1}, ..., \neg B_{1,b1} \rightarrow$ **retract** $P_1(\overline{u_{1,1}})$

r_2: $P_1(\overline{u_{2,1}}), P_2(\overline{u_{2,2}}), ..., P_p(\overline{u_{2,p}}), A_{2,1}, ..., A_{2,a2}, \neg B_{2,1}, ..., \neg B_{2,b2} \rightarrow$ **retract** $P_2(\overline{u_{2,2}})$

:

r_p: $P_1(\overline{u_{p,1}}), P_2(\overline{u_{p,2}}), ..., P_p(\overline{u_{p,p}}), A_{p,1}, ..., A_{p,ap}, \neg B_{p,1}, ..., \neg B_{p,bp} \rightarrow$ **retract** $P_p(\overline{u_{p,p}})$

Each $\overline{u_{i,j}}$ is a vector of terms (variables and constants), and for each clique, all terms $\overline{u_{1,i}}$ ... $\overline{u_{p,i}}$, associated with each predicate P_i in the maximal set must be unifiable.

We relax the original condition (3) of the strict-stratification criterion which was as follows:

(3) For every distinguished predicate occurring in a positive literal in a rule in UR_i, all r-update rules relevant to this predicate must be contained within $\bigcup_{j<i} UR_j$.

We relax this condition to allow rules such as in the previous example to be legal update rule programs. Condition 3 is modified as follows:

(3^*) For every distinguished predicate occurring in a positive literal in an r-update rule p, in UR_i, any relevant r-update rule q must be contained within $\bigcup_{j<i} UR_j$

unless the rules p and q participate in an r-clique in the partition UR_i. Then p and q may be contained within the same partition UR_i.

Definition

2 (or more) r-cliques *overlap* if the predicates in the set of the maximal set of predicates of each clique is not

www.manaraa.com

disjoint and the corresponding vectors of terms are unifiable. □

3.3. Second Relaxation

A similar situation may occur with rules that are used to infer new information. The body of each rule queries some subset of predicates. If the rules (which infer different information), query the same subset of predicates or some mutually overlapping subsets, then it is possible that the stratification conditions will be violated. Sometimes, if one rule executes and asserts some new information, then this prevents the execution of another rule. The relaxation applies to a-update rules. They are allowed to co-exist in the same partition when they have a negative dependency on each predicate in some set.

For this example, we consider the same relations as in Example 3. The rules are based on the knowledge that each person must be either a male or a female. This knowledge is again subject to the constraint that a person cannot be a male and a female simultaneously. As before, the choice predicate is used to select whether the person should be a male or a female. Rule r_1 asserts the fact that the person is a female, but it first queries the database to make sure the person is not a male as well. Similarly, rule r_2 asserts that a person is a male, after having first verified that this person is not a female as well. The rules are as follows:

Example 5

r_1: person(x) ¬ male(x), choice(x, "f") → **assert** female(x)

r_2: person(x) ¬ female(x), choice(x, "m") → **assert** male(x)

Again, if we apply the strict stratification criterion, condition (4) requires that r_1 be in a lower partition than r_2 and vice versa. Our solution is to relax the criteria for stratification.

Definition

A set of a-update rules participate in an *a-clique* if there is some *maximal* set of predicates $\{P_1, ..., P_n\}$ such that each rule in the a-clique is relevant to some predicate from this set; if there is at least one rule in the a-clique which is relevant to each predicate in this set; and if all of the predicates in the set (except the predicate in the head of each rule) are distinguished and occur in a negative literal in each rule of this a-clique. □

The following a-update rules form an a-clique with a maximal set of predicates $\{P_1, ..., P_p\}$, where each P_i is a distinguished predicate in each a-update rule in the a-clique, except in the rule relevant to P_i:

r_1: $\neg P_2(\overline{u_{1,2}}), \neg P_3(\overline{u_{1,3}}), ..., \neg P_p(\overline{u_{1,p}}), A_{1,1}, ..., A_{1,a1}, \neg B_{1,1}, ..., \neg B_{1,b1} \rightarrow$ **assert** $P_1(\overline{u_{1,1}})$

r_2: $\neg P_1(\overline{u_{2,1}}), \neg P_3(\overline{u_{2,3}}), ..., \neg P_p(\overline{u_{2,p}}), A_{2,1}, ..., A_{2,a2}, \neg B_{2,1}, ..., \neg B_{2,b2} \rightarrow$ **assert** $P_2(\overline{u_{2,2}})$

:

r_p: $\neg P_1(\overline{u_{p,1}}), \neg P_2(\overline{u_{p,2}}), ..., \neg P_{p-1}(\overline{u_{p,p-1}}), A_{p,1}, ..., A_{p,ap}, \neg B_{p,1}, ..., \neg B_{p,bp} \rightarrow$ **assert** $P_p(\overline{u_{p,p}})$

As before, we must ensure that the execution of the rule terminates and produces a minimal database. We relax condition (4) of the stratification criterion, which was stated as follows:

(4) For every distinguished predicate occurring in a negative literal in a rule in UR_i, all a-update rules relevant to this distinguished predicate must be contained within $\bigcup_{j<i} UR_j$.

We modify this condition, as follows:

(4*) For every distinguished predicate occurring in a negative literal in an a-update rule p, in UR_i, any relevant a-update rule q must be contained within $\bigcup_{j<i} UR_j$

www.manaraa.com

unless the rules p and q are a-update rules that participate in an a-clique in the partition UR_i. Then, p and q may be contained within the same partition UR_i.

Definition

2 (or more) a-cliques *overlap* if the predicates in the set of the maximal set of predicates of each clique is not disjoint and the corresponding vectors of terms are unifiable. □

We note from the definition for relaxing the stratification conditions that the same predicate may not occur in the maximal set of both an a-clique and an r-clique. However, the definition will allow programs with overlapping cliques. Thus, a predicate may occur in the maximal sets of several a-cliques, or of several r-cliques, simultaneously. The relaxed criterion admits many programs which would have been rejected by the strict stratification criterion.

4. A Fixpoint Semantics for Update Rule Programs

We define a fixpoint semantics for acceptable update rule programs which guarantees that the execution of the program terminates and each execution schedule produces a minimal database. Non-terminating behavior results when rules can indefinitely keep asserting and retracting the same tuples. A solution is to explicitly keep track of tuples which are asserted and retracted. Once a tuple is retracted it is not affected by further assertions (of the same tuple), so so that this process cannot occur indefinitely. The execution of update rules is defined based on a monotonic operator T_{UR}. A fixpoint is non-deterministically obtained for the update rule program UR. The monotonic operator T_{UR} is non-deterministically applied to all the rules in each partition, UR_i, in the order i= 1, 2, ..., n. A fixpoint EDB_i is computed for each partition. Each EDB_i is represented by the corresponding relations, after the execution of all the rules in the partition that can be executed. Thus, a sequence of fixpoints EDB_1, EDB_2,...,., corresponding to each partition UR_1, UR_2, . . . , , are computed, in turn. The final updated database is EDB_n and will contain all the answers obtained from the update rule program.

Consider a domain, D_{UR}, for an update rule program to be the set of all possible ground atoms, i.e., atoms with no variables, of the form **assert**P($\bar{t}$) and **retract**P($\bar{t}$), where P is any predicate and $\bar{t}$ is a vector of constants from the ordered set C.

Definition

EDB_0 = {**assert**P($\bar{t}$) | P($\bar{t}$) ∈ EDB_{init}}. □

EDB_0 is a set of tuples in the relations **assert**P, **assert**Q, etc., corresponding to the facts of the initial database EDB_{init} and the relations **retract**P, **retract**Q, etc. will initially be empty, since the database only contains facts, initially.

Definition

Given a subset S of D_{UR}, we have the following:

- S entails P($\bar{t}$) if **assert**P($\bar{t}$) ∈ S and **retract**P($\bar{t}$) ∉ S.
- S entails ¬ P($\bar{t}$) if and only if S does not entail P($\bar{t}$). This occurs with the following:

  assertP($\bar{t}$) ∈ S and **retract**P($\bar{t}$) ∈ S or

  assertP($\bar{t}$) ∉ S and **retract**P($\bar{t}$) ∉ S.
- The following cannot occur in S: **assert**P($\bar{t}$) ∉ S and **retract**P($\bar{t}$) ∈ S; this is due to the syntactic restriction on the language. □

Definition

Given the ordered set of constants C over which D_{UR} is defined, we have eval(op_i, $arg_{i,1}$, $arg_{i,2}$) is true if

www.manaraa.com

(1) op_i is in the set { ≤ , <, ≥ , >, =, ≠ }; and

(2) each of the arguments $arg_{i,1}$ and $arg_{i,2}$ are constants in the set C, or they are range restricted variables occurring in a positive literal, i.e., they can be instantiated to constants in the set C; and

(3) ($arg_{i,1}$ op $arg_{i,2}$) when interpreted in the usual fashion evaluates to true.

Otherwise, eval(op_i, $arg_{i,1}$, $arg_{i,2}$) evaluates to false. □

We define an operator T_{UR} which, when applied to a rule in a partition of the update rule program, will update the EDB relations. Since T_{UR} is applied non-deterministically, we first define an intermediate operator M_{UR}.

Definition

Let $M_{UR} : 2^{D_{UR}} \rightarrow 2^{D_{UR}}$ be a mapping from subsets of D_{UR} to subsets of D_{UR}, defined as follows, where S is a subset of D_{UR}:

$M_{UR}(S)$= {**assert**P($\bar{t}$) | there is a ground instance of a rule p (i.e., an instance of p where all the variables are substituted by constants from C):

$A_1, A_2, ..., A_n, \neg B_1, \neg B_2, ..., \neg B_m$, eval($op_1$, $arg_{1,1}$, $arg_{1,2}$), eval(op_2, $arg_{2,1}$, $arg_{2,2}$), ..., eval(op_k, $arg_{k,1}$, $arg_{k,2}$),

→ **assert**P($\bar{t}$)

of an update rule such that S entails $B_1, B_2, ..., B_n, \neg C_1, \neg C_2, ..., \neg C_m$,

and each eval(op_i, $arg_{i,1}$, $arg_{i,2}$) evaluates to true }.

∪

{ **retract**P($\bar{t}$) | there is a ground instance p:

$A_1, A_2, ..., A_n, \neg B_1, \neg B_2, ..., \neg B_m$, eval(op_1, $arg_{1,1}$, $arg_{1,2}$), eval(op_2, $arg_{2,1}$, $arg_{2,2}$), ..., eval(op_k, $arg_{k,1}$, $arg_{k,2}$),

→ **retract**P($\bar{t}$)

of an update rule such that S entails $B_1, B_2, ..., B_n, \neg C_1, \neg C_2, ..., \neg C_m$,

and each eval(op_i, $arg_{i,1}$, $arg_{i,2}$) evaluates to true }. □

Definition

Let $T_{UR}: 2^{D_{UR}} \rightarrow 2^{D_{UR}}$ be a mapping from a subset of D_{UR} to a subset of D_{UR} defined as follows:

$T_{UR}(S) = S \cup \{Q\}$ where $Q \in M_{UR}(S)$ and $Q \notin S$. □

Note that Q is selected non-deterministically.

4.1. Computing the Operational Fixpoint

Each EDB_i is computed iteratively, applying the operator T_{UR} to the rules in each partition UR_i until a fixpoint EDB_i is reached. T_{UR} will select a single rule for execution in each step and this will result in a possibly non-deterministic execution. Since T_{UR} is monotonic, a fixpoint is obtained for each partition, UR_i, when there are no longer any rule that can change EDB_i. Processing for the update rule program terminates when the fixpoint for rules in UR_n is reached and EDB_n is computed.

$EDB_1 = T_{UR_1} \uparrow \omega\ (EDB_0)$

$EDB_2 = T_{UR_2} \uparrow \omega\ (EDB_1)$

:

$EDB_n = T_{UR_n} \uparrow \omega\ (EDB_{n-1})$

المنارة للاستشارات
www.manaraa.com

Theorem

$EDB_i = T_{UR_i} \uparrow \omega\, (EDB_{i-1})$ is a fixpoint of T_{UR_i}, for i=1,2,..,n. EDB_n is a fixpoint for T_{UR}.

We observe from the definition of the operator T_{UR} that it is growing monotonically, i.e., $S \subseteq T_{UR}(S)$ and that $T_{UR} \uparrow n\, (S) \subseteq T_{UR}\, (\, T_{UR} \uparrow n\, (S)\,)$. Since the domain D_{UR} for the update rule program is finite, after some finite number of applications n, $n < \omega$, the operator will not add any new element of the form (**assert** $P(\bar{t})$) or (**retract** $P(\bar{t})$) to EDB_i.

Thus, $EDB_i = T_{UR_i} \uparrow \omega\, (EDB_{i-1})$ is a fixpoint of T_{UR_i} for 1 = 1, 2, ..., n. □

We note that there may be several ways to partition a program, and to identify overlapping cliques (within the same partition). However, this does not have an effect on computing the fixpoint.

4.2. Some Relevant Properties of the Update Rule Programs

We present some relevant properties of the update rule programs, UR. These properties, represented by the following lemmas, 4.1 through 4.6, follow from the criterion for strict stratification or the relaxed criterion presented in Section 3. The proofs are straightforward and are omitted for readability. The results will be applied to the proof of the equivalence between the fixpoint semantics for UR programs and the declarative semantics, which is discussed in the next few sections. The following properties (expressed in the propositional case) hold true for the UR programs:

Lemma 4.1

Let $UR = UR_1 \cup \cdots \cup UR_n$ be a UR program. Let $UR^- = UR_1^- \cup \cdots \cup UR_n^-$ be the resulting program after eliminating all the a-update rules in UR which participate in each a-clique and all the r-update rules which participate in each r-clique, in all the partitions of UR. Then, UR^- is a strictly-stratified UR program. □

Lemma 4.2

Let UR be an update rule program. Then the set of all atoms that occur in any maximal set associated with any a-clique and the set of all atoms that occur in any maximal set associated with any r-clique are mutually disjoint. □

Lemma 4.3

Let $UR = UR_1 \cup \cdots \cup UR_n$ be an update rule program. Let the retractable atom P_i occur in the maximal set associated with some r-clique in the partition UR_j. Then, P_i may not be distinguished in a rule in any partition $\bigcup_{k<j} UR_k$. Further, all rules q in UR_j in which P_i is distinguished must be r-update rules and they must participate in this r-clique. □

Lemma 4.4

Let $UR = UR_1 \cup \cdots \cup UR_n$ be an update rule program. Let the atom P_i occur in the maximal set associated with some a-clique in the partition UR_j. Then, P_i may not be distinguished in a rule in any partition $\bigcup_{k<j} UR_k$. Further, all rules q in UR_j in which this P_i is distinguished (and occurs in a negative literal) must be an a-update rule and they should participate in the a-clique. P_i may also be a distinguished atom (occurring in a positive literal) in other rules in UR_j. Finally, P_i is unretractable in the UR program. □

Lemma 4.5

Let p and q be example a-update rules or r-update rules in UR_t. Then, all distinguished atoms in p or q, which do not occur in a maximal set of atoms associated with some r-clique in UR_t, must be unretractable in UR_t. □

The proofs of these Lemmas follow directly from the relaxed stratification criterion for the update rule programs.

www.manaraa.com

Lemma 4.6

For any set of a-update rules participating in an a-clique in partition UR_k, or for any set of r-update rules participating in an r-clique, at most one of the rules in the a-clique (or one of the rules in the r-clique) may execute (for a given assignment for the variables, in the first order case).

The proof for the a-clique follows from the fact that for each a-update rule relevant to P_i, participating in an a-clique, all P_j, $j \neq t$, in the maximal set, must be distinguished and occur in a negative literal in the a-update rule. Thus, for the execution of any a-update rule in the a-clique, relevant to P_i, none of the P_j, $j \neq i$, must be entailed by EDB_k. The execution of any one of the a-update rules in the a-clique will add some **assert** P_i, to EDB_k and EDB_k will entail this P_i. From the definition of the stratification criterion, this P_i is unretractable in this partition, It follows that since this P_i is distinguished and occurs in a negative literal in each of the other a-update rules in the a-clique, none of these other rules may be executed. Thus, at most one a-update rule in an a-clique may be executed.

Similarly, for each r-update rule participating in an r-clique, all Q_j in the maximal set must be distinguished atoms in these r-update rules. Thus, for the execution of any r-update rule in the r-clique, each of the Q_j must be entailed by EDB_k. The execution of any one of the r-update rules will add some **retract** Q_j, to EDB_k and EDB_k will not entail this Q_j. It follows that none of the other rules may be executed and at most one r-update rule in an r-clique will execute. □

4.3. Set-oriented and Tuple-oriented Execution of Rules

The difference between a tuple-oriented and a set-oriented execution, when the action(s) in the head of the rules update one tuple, or a set of tuples, respectively, can be informally stated, as follows:

When the literals in the body of an update rule are evaluated against the relations of the database, and when a single tuple or a set of tuples can be retrieved, corresponding to the action(s) in the head of the update rule, do the following execution options produce identical fixpoints?

(1) a *tuple-oriented execution* -- execute the update action(s) in the head of the rule, corresponding to a single selected tuple, and then re-evaluate the literals in the body of the rule against the updated database.

(2) a *set-oriented execution* -- execute the update action(s) in the head of the rule, corresponding to a single tuple, some subset of the set of tuples or all the tuples in the set, simultaneously.

In our case, the syntactic criterion for admitting UR programs ensures that the fixpoint(s) obtained from the set-oriented execution are a subset of the set of fixpoints corresponding to a tuple-oriented execution. This may be of practical significance since the set-oriented execution is more efficient. We do not discuss this issue in detail in this paper, and refer the reader to a discussion of implementation issues in [PaRa94].

5. A Declarative Semantics for Update Rule Programs

We have described a fixpoint semantics for UR programs which is guaranteed to terminate and to produce a set of fixpoints for UR. To obtain a declarative semantics, we first associate a normal logic program $\overline{UR}$ for a given UR program. We use the *stable model* semantics of Gelfond and Lifschitz [GeLi88] to obtain a set of minimal models for $\overline{UR}$ which are called *stable models*. We show the equivalence between the set of fixpoints for UR and the set of stable models for $\overline{UR}$.

المنارة للاستشارات
www.manaraa.com

5.1. Obtaining a Normal Logic Program

To obtain a normal logic program $\overline{UR}$, the a-update rules that assert new tuples are treated like deductive rules which define the asserted tuple. The r-update rules are similar to *denial* integrity constraints proposed in [KoSa89, KoSa90]. A *denial* integrity constraint is given by a denial which is a conjunction of literals, followed by a *retractable* atom which is enclosed within [ ]. The retractable atom must occur in the conjunction of literals. An example is as follows:

$L_1, L_2, ..., L_n \rightarrow.$ $[L_i]$.

The meaning associated with this constraint is that $L_1 \wedge L_2 \wedge \cdots \wedge L_n$ cannot be true in the database, if the database is to be consistent with the denial. If the denial is violated in the database, then consistency can be restored by making one of the positive literals in the denial false. A single atom L_i which occurs in the denial is chosen as the retractable atom, and is specified as such in the constraint.

If we examine the following r-update rule:

r_i: $P, A_1, ..., A_n, \neg B_1, ..., \neg B_m \rightarrow$ **retract** P

since the atom P which is retracted by the r-update rule must occur in the conjunction in the body of the rule, the r-update rule is syntactically similar to the denial integrity constraints. We consider the conjunction in the body of the r-update rule to correspond to the denial.

If we consider the forward chaining semantics of update rule programs, the r-update rules query the database to determine if the condition in the body of the rule is true, i.e., the database must entail P and every literal A_k, but must not entail any literal B_k. The condition will include the fact, P, that is to be deleted. When the condition is true, the corresponding fact P, (which must exist), is deleted, so that the condition is no longer true in the database. Operationally, this is very close to the task of maintaining an integrity constraint. When we consider related research, we will examine different semantics where the deletion action corresponds to a negation.

There has been considerable research on the problem of maintaining consistency in databases, with respect to a set of integrity constraints. Informally, a database must satisfy its integrity constraints as it changes over time. Usually, an update to the database (more precisely an update to facts in the database) may cause the violation of an integrity constraint. Such updates must be rejected or modified. Sometimes, the database itself, i.e., the initial facts and the facts that are inferred from the rules, may be inconsistent with the constraints, and the database must be modified to maintain consistency with the constraints. This is the approach we have taken to provide a semantics for the updates.

The translation to obtain $\overline{UR}$ uses as input the update rules of the program UR and the initial database EDB_{init}, and produces the logic program $\overline{UR}$ as output. For simplicity, we show this transformation for the propositional case. The translation in the first order case is included in Appendix 1.

A Translation to obtain a Normal Logic Program

Step 1.

Each r-update rule which is relevant to the atom P transforms each a-update rule in UR which is relevant to the atom P, or the fact P if it occurs in EDB_{init}.

Suppose the a-update rule relevant to P, is of the following form:

$A_1, ..., A_a, \neg B_1, ..., \neg B_b \rightarrow$ **assert** P.

where a or b could be equal to 0, i.e., P is a fact, is transformed by each r-update rule relevant to P, which is as follows:

المنارة للاستشارات
www.manaraa.com

P, C_1, ..., C_c, $\neg D_1$, ..., $\neg D_d$ $\rightarrow$ **retract** P.

The following rules are placed in $\overline{\text{UR}}$, where P^* is a special unused atom associated with each atom P in the program:

C_1, ..., C_c, $\neg D_1$, ..., $\neg D_d$, $\rightarrow$ P^*.

$\neg P^*$, A_1, ..., A_a, $\neg B_1$, ..., $\neg B_b$, $\rightarrow$ P.

Step 2

Each fact, say P, in EDB_{init} which is not modified by any r-update rule, i.e., there are no r-update rules in UR relevant to P is placed in $\overline{\text{UR}}$.

Each a-update rule relevant to the atom P which is as follows:

A_1, ..., A_a, $\neg B_1$, ..., $\neg B_b$ $\rightarrow$ **assert** P.

and which is not modified by any r-update rule in UR will derive the following rule in $\overline{\text{UR}}$:

A_1, ..., A_a, $\neg B_1$, ..., $\neg B_b$ $\rightarrow$ P.

5.2. Stable Model Semantics

The program $\overline{\text{UR}}$ may not be a stratified logic program and may not have a unique minimal model. We use the stable model semantics to obtain a meaning for this program. A model theory for normal logic programs [GeLi88] characterizes the meaning of a normal logic program by a set of minimal models called *stable models*, which are defined using the Gelfond-Lifschitz transformation. This transformation is defined as follows.

Definition

Let P be a normal logic program and let I be an interpretation.

$P^I = \{ A \leftarrow B_1, ..., B_n : A \leftarrow B_1, ..., B_n, \neg D_1, ..., \neg D_m$ is a ground instance of a clause in P and $\{D_1, ..., D_m\} \cap I = \Phi\}$

P^I is the Gelfond-Lifschitz transformation of P with respect to I, where the A_i, B_j and D_l are atomic formulae. □

The result of the Gelfond-Lifschitz transformation is a negation-free (definite) program. Stable models for non-disjunctive logic programs may now be defined as follows:

Definition

Let P be a definite normal program. M is a stable model of P iff M is the unique minimal model of P^M. □

5.3. Equivalence of the Fixpoint and Declarative Semantics

Theorem

Let $\overline{\text{UR}}$ be the normal logic program derived from the update rule program UR corresponding to the set of update rules and the initial database EDB_{init}.

A fixpoint of the operational semantics for UR represented by a final updated database EDB_n, is identical to one of the (k) stable models for the normal logic program $\overline{\text{UR}}$, as characterized by the stable model semantics.

المنارة للاستشارات
www.manaraa.com

All of the (k) stable models for the corresponding normal logic program $\overline{UR}$, as characterized by the stable model semantics, can each be non-deterministically obtained as a final updated database EDB_n, from the UR program.

The following figure is a graphical representation of this equivalence.

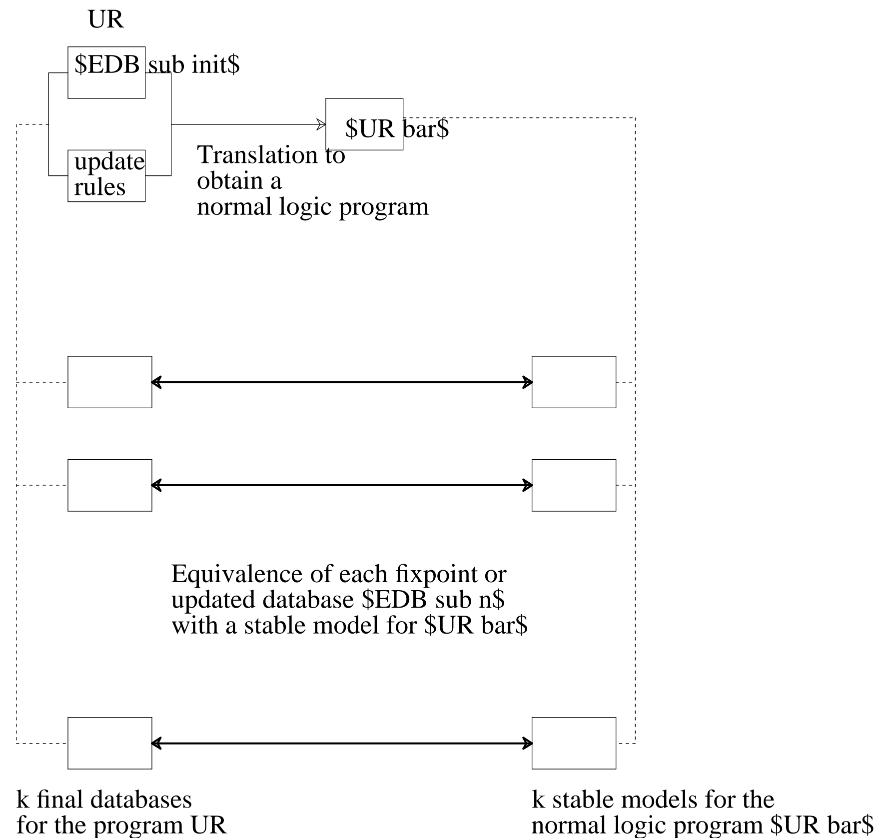

Figure 1. Equivalence Between the Fixpoint and Declarative Semantics

6. Proving the Equivalence of the Fixpoint and Declarative Semantics

In this section, we present the proof of equivalence of the fixpoint and declarative semantics.

6.1. Some Results from the Stable Model Semantics for Normal Logic Programs

We first present some results on the stable model semantics and the corresponding minimal (stable) models, for some normal logic programs $\overline{UR}$. The normal logic programs that we consider will correspond to some interesting UR program fragments. These results will be applied to the proof of equivalence of the fixpoint and declarative semantics. The results are represented by the following lemmas, (expressed in the propositional case):

Lemma 6.1

Let $\overline{UR}$ be a normal logic program derived from an update rule program fragment comprising a single (non-overlapping) r-clique with a maximal set of atoms $\{P_1,..., P_p\}$, where each P_i is a retractable atom, and the a-update rules relevant to each P_i. Let P_i^* be the special atoms associated with each atom P_i in the maximal set.

المنارة للاستشارات
www.manaraa.com

Suppose that all other distinguished literals in the rules may be ignored. Then there are p possible stable models for $\overline{UR}$. In each stable model, exactly one of the p atoms in the maximal set will be false, the corresponding special atom will be true, all other atoms in the maximal set will be true, and the corresponding special atoms will be false.

To prove this result, let $\overline{UR}$ be derived from a program fragment comprising a a set of r-update rules $s_1, ..., s_p$, which form an r-clique with a maximal set of atoms $\{P_1, ..., P_p\}$. To obtain $\overline{UR}$, we also consider any a-update rules q_k relevant to some atom P_k occurring in the maximal set of this r-clique. For simplicity, we assume that there are exactly p r-update rules and p a-update rules, each one relevant to exactly one P_i. However, this lemma also applies to the case where there may be several rules relevant to each atom. Since the r-clique is non-overlapping, none of the r-update rules s_k participate in another r-clique. For notational convenience, all other distinguished literals in the rules are represented using some conjunction *Conj-i,j*.

The normal logic program $\overline{UR}$ from the rules p_k and s_k, is as follows:

$r_{1,1}$: $P_2, ..., P_p$, *Conj-1,1* $\rightarrow P_1^*$

: each $r_{1,k}$ is from s_k: $P_1, .., P_k, ..,P_p$, *Conj-1,k* $\rightarrow$ **retract** P_k

$r_{1,p}$: $P_1, ..., P_{p-1}$, *Conj-1,p* $\rightarrow P_p^*$

$r_{2,1}$: $\neg P_1^*$, *Conj-2,1* $\rightarrow P_1$ each $r_{2,k}$ is from s_k transforming q_k: *Conj-2,k* $\rightarrow$ **assert** P_k

:

$r_{2,p}$: $\neg P_p^*$, *Conj-2,p* $\rightarrow P_p$

Assume that for all the stable models, all the distinguished atoms in each conjunction *Conj-i,j* are true and each distinguished literal that occurs negatively in each conjunction is false. We may ignore each of these conjunctions in the rules. This assumption will be clarified later.

Now consider some model M_k in which one P_k^* and all P_t, $t \neq k$, (from the maximal set of atoms) are true. P_k and all P_t^*, $t \neq k$, are false in this model. To prove this is a stable model, we first apply the Gelfond-Lifschitz transformation to obtain the negation-free program $\overline{UR}_{Mk}$. All rules $r_{1,1}$ through $r_{1,p}$ will be unchanged in $\overline{UR}_{Mk}$ (ignoring the conjunctions). All rules $r_{2,1}$ through $r_{2,p}$, except $r_{2,k}$, will be modified in $\overline{UR}_{Mk}$ so that the special distinguished literals P_t^*, $t \neq k$, occurring negatively in these rules are eliminated. These rules will be as follows:

$r_{2,t}$: *Conj-2,t* $\rightarrow P_t$

The rule $r_{2,k}$ (or set of rules) will be eliminated in $\overline{UR}_{Mk}$.

The above transformation is independent of whether there are one or more a-update rules and r-update rules relevant to each P_i and the program that is obtained with multiple rules will be similar. Now, from the rule (rules) $r_{1,k}$ (ignoring the conjunctions), we have P_k^* must be true. From each rule (rules) $r_{2,t}$, $t \neq k$, we have each P_t, $t \neq k$, must be true. Further, since rule (rules) $r_{2,k}$ is eliminated, P_k must be false. Since P_k is a distinguished atom in each of the rules $r_{1,t}$, $t \neq k$, each P_t^*, $t \neq k$, must be false. Thus, we have M_k is a stable model for the program.

Next, we prove that each such M_k is a minimal model for $\overline{UR}_{Mk}$. We prove this by considering the following two cases:

Case 1

Suppose there is a model M_{c1} which is a proper subset of M_k, in which P_k and one (or more) P_x from the maximal set are false. However, from each rule (rules) $r_{2,x}$ in $\overline{UR}_{Mk}$, we have each P_x, $x \neq k$ must be true. Thus, this model is not stable.

Case 2

Suppose there is a model M_{c2} which is a proper subset of M_k, in which the special atom P_k^* is false, i.e., none

المنارة للاستشارات
www.manaraa.com

of the special atoms are true. Then, from rule (rules) $r_{1,k}$ in $\overline{UR}_{Mk}$, since P_k^* is false, at least one P_x, $x \neq k$, must be false. However, from the rule (rules) $r_{2,x}$ in $\overline{UR}_{Mk}$, we have each P_x, $x \neq k$ must be true. Thus, this model is not stable.

Finally, we prove that the models M_k enumerated in Lemma 6.1, are the only stable models for the program, by considering these following two cases:

Case 3

Suppose there is a model M_{c3} in which all atoms in the maximal set are true. But we know that the rule (rules) $r_{2,k}$, relevant to P_k is eliminated in $\overline{UR}_{Mk}$. Thus, this model is not a stable model.

Case 4

Suppose there is a model M_{c4} in which some P_k^* and one (or more) P_y^*, $y \neq k$, is true. From rule (rules) $r_{1,y}$ in $\overline{UR}_{Mk}$, it follows that P_k must be true. However, the rule (rules) $r_{2,k}$, relevant to P_k, is eliminated in $\overline{UR}_{Mk}$. Thus, this model is not a stable model.

Thus, we have shown that there are p possible stable models, independent of whether there are several rules relevant to each P_i in the maximal set. □

Lemma 6.2

Let $\overline{UR}$ be a normal logic program derived from an update rule program fragment comprising overlapping r-cliques with maximal sets of atoms $\{P_1,..., P_p, C_1,..., C_c\}$ and $\{Q_1,..., Q_q, C_1,..., C_c\}$, where $\{C_1,..., C_c\}$ are common to both maximal sets. We also consider the relevant a-update rules. Each P_i, Q_i and C_i is a retractable atom. Let P_i^*, Q_i^* and C_i^* be the special atoms associated with the atoms in the maximal sets. Assume that all other distinguished literals in the rules in $\overline{UR}$ may be ignored. Then there will be $(p * q + c)$ possible stable models. In $(p * q)$ of these models, exactly one P_{k1} and exactly one Q_{k2}, will be false and exactly one P_{k1}^* and exactly one Q_{k2}^* will be true. In addition all other P_t, $t \neq k1$, and all other Q_t, $t \neq k2$, will be true and these P_t^* and Q_t^*, will be false. Finally, all C_s will be true and all C_s^* will be false. In c of these extensions, exactly one atom C_{k3} which is common to both the maximal sets is false, C_{k3}^* is true, all other atoms C_t, $t \neq k3$, will be true and all C_t^*, $t \neq k3$, will be false. In addition, in all these extensions, all P_s and all Q_r will also be true and all P_s^* and all Q_r^* will be false.

The proof of Lemma 6.2 is omitted. □

Lemma 6.3

Let $\overline{UR}$ be a normal logic program derived from an update rule program fragment comprising a single non-overlapping a-clique with a maximal set of atoms $\{P_1,..., P_p\}$. Assume that all other distinguished literals in the rules in $\overline{UR}$ may be ignored. Then there are p possible stable models for $\overline{UR}$, In each stable model, exactly one of the p atoms in the maximal set will be true and all other atoms will be false.

To prove this result, let $\overline{UR}$ be derived from a program fragment comprising a set of a-update rules s_1.., s_p, which form an a-clique with the maximal set of atoms $\{P_1,..., P_p\}$. For simplicity, we assume that there are exactly p such rules, each one relevant to exactly one P_k. However, the lemma also applies with several rules relevant to each P_k. Since the a-clique is non-overlapping, none of the a-update rules s_k participate in another a-clique. All other distinguished literals in the rules are represented using the conjunctions *Conj-k*. The normal logic program $\overline{UR}$ obtained from the rules is as follows:

r_1 : $\neg P_2, ..., \neg P_p$, *Conj-1* $\rightarrow P_1$

r_k is from s_k : $\neg P_1, .., \neg P_{k-1}, \neg P_{k+1}, .., \neg P_p \rightarrow$ **assert** P_k

r_p : $\neg P_1, ..., \neg P_{p-1}$, *Conj-p* $\rightarrow P_p$.

Assume that for all the stable models, all the distinguished atoms in each conjunction are true and each distinguished literal that occurs negatively in each conjunction is false. We ignore each of these conjunctions in the

المنارة للاستشارات
www.manaraa.com

rules.

Now consider some model M_k in which one P_k is true and all P_t, $t \neq k$, (from the maximal set) are false. To prove this is a stable model, we first apply the Gelfond-Lifschitz transformation to obtain the negation free program $\overline{UR}_{Mk}$. All rules r_1 through r_p except the rule r_k will be eliminated in $\overline{UR}_{Mk}$ since P_k is a distinguished literal occurring negatively in these rules. The rule r_k will be modified as follows:

r_k : *Conj-k* $\rightarrow P_k$

The transformed program will be similar if there are several rules relevant to each of the predicates P_i. From this rule (rules) r_k, ignoring the conjunction, we have P_k must be true. Thus each M_k is a stable model.

To prove that each M_k is a minimal model, we only need consider the empty interpretation since it is the only proper subset of any M_k. However, from the rule (or rules) r_k in $\overline{UR}_{Mk}$, we have P_k is true, and so this is not stable.

There can be no other possible models for $\overline{UR}_{Mk}$ which are minimal models. Thus, there are p possible models, and this independent of whether there are multiple rules relevant to each P_i. □

Lemma 6.4:

Let $\overline{UR}$ be a normal logic program derived from an update rule program fragment comprising overlapping a-cliques with maximal sets of atoms $\{P_1,..., P_p, C_1,..., C_c\}$ and $\{Q_1,..., Q_q, C_1,..., C_c\}$, where $\{C_1,..., C_c\}$ are common to both maximal sets. Assume we ignore all other literals in $\overline{UR}$. Then there will be $(c + p * q)$ possible stable models. In $(p * q)$ of these models, exactly one P_{k1} and exactly one Q_{k2}, will be true. All other P_t, $t \neq k1$, and all other Q_t, $t \neq k2$, as well as all other C_t, will be false. In c of these models, exactly one atom C_{k3} which is common to both the maximal sets is true. All other atoms C_t, $t \neq k3$, will be false as well as all P_s and all Q_r.

The proof of Lemma 6.4 is omitted. □

6.2. Soundness and Completeness of the Fixpoint Semantics

The proof that the fixpoint semantics is sound and complete wrt the declarative semantics for the corresponding normal logic program $\overline{UR}$ is presented in this section. This proof builds upon a previous result for strictly-stratified UR programs. In [Rasc94], we have shown that a stratified logic program can be obtained corresponding to each strictly-stratified UR program. We have further shown that the fixpoint semantics for the strictly-stratified UR program is sound and complete with respect to the declarative semantics for the corresponding stratified logic program. This theorem is as follows:

Theorem [Rasc94]

Suppose $UR^- = UR^-_0 \cup UR^-_1 \cup \cdots \cup UR^-_n$ is a strictly-stratified UR program. Let $\overline{UR^-(0,..,i)}$ be the stratified logic program corresponding to the initial database EDB_0 (or UR^-_0) and the rules in the partitions, $UR^-_1 \cup \cdots \cup UR^-_i$. Let $M_{\overline{UR^-(0,..,i)}}$ be the standard model for $\overline{UR^-(0,..,i)}$. Then, EDB^-_i entails $M_{\overline{UR^-(0,..,i)}}$. In other words, if P is true in $M_{\overline{UR^-(0,..,i)}}$ then EDB^-_i entails P, and if P is false in $M_{\overline{UR^-(0,..,i)}}$ then EDB^-_i does not entail P.

Recall from Lemma 4.1 that by removing all the r-update rules that participate in any r-cliques and the a-update rules that participate in any a-cliques, we can obtain a strictly-stratified UR program from the update rule program UR. Building upon this previous theorem for the strictly-stratified UR programs, we will start from a strictly-stratified UR program and then introduce into each partition in turn, the r-update rules that participate in r-cliques or the a-update rules that participate in a-cliques. We allow these rules to execute until a fixpoint is obtained for this partition. We will then show that the fixpoint semantics for the resulting UR program, after these r-update rules and the a-update rules are introduced into each partition, is sound and complete wrt the declarative semantics

المنارة للاستشارات
www.manaraa.com

for the corresponding normal logic program $\overline{UR}$. We present the proof in the propositional case. It should hold in the function free first order case where all terms are ground.

Definition

Let $\overline{UR}$ be a normal logic program. Then, a distinguished atom P in a rule has a positive dependency on an atom Q if and only if P occurs in the head of a rule p and Q is a distinguished atom in p, or if R is a distinguished atom in the rule p and R has a positive dependency on Q.

Theorem

Let $\overline{UR}$ be the normal logic program derived from an update rule program UR.

Each fixpoint from the UR program, represented by a final updated database EDB_n, will entail one of the stable models that characterize $\overline{UR}$.

Each of the stable models that characterize $\overline{UR}$ is entailed by a fixpoint or final updated database EDB_n, from the UR program. [1]

Proof

Let $UR = UR_0 \cup UR_1 \cup \cdots \cup UR_n$ be an update rule program UR, where $UR_0 = EDB_0$.

Let $\overline{UR(0,..,i)}$ be the normal logic program derived from the initial database EDB_0 and the rules in the partitions, $UR_1 \cup \cdots \cup UR_i$.

Let $UR^- = UR^-_0 \cup UR^-_1 \cup \cdots \cup UR^-_n$ be the strictly-stratified UR program that is obtained by eliminating the r-update rules in each of the r-cliques and the a-update rules in each of the a-cliques, in each of the partitions of UR.

Let $\overline{UR^-(0,..,n)}$ be the stratified logic program corresponding to the strictly-stratified UR program UR^-.

Base Case

The base case is to prove that EDB_0 entails $M_{\overline{UR-(0)}}$. By definition, there are no update rules in UR_0 and $\overline{UR-(0)}$ is identical to the initial database EDB_{init}. The proof trivially follows from the definition of EDB_0.

Inductive Case

Given EDB_i entails a stable model M_i for the corresponding normal logic program $\overline{UR(0,..,i)}$, and

$EDB^-_{i+1} = T_{UR^-_{i+1}} \uparrow\omega\,(EDB_i)$, the fixpoint for the update rules in UR^-_{i+1} evaluated over EDB_i, entails the standard model for the stratified logic program derived from $\overline{M_i \cup UR^-(i+1)}$ [Rasc94],

then, each EDB_{i+1} entails a stable model for the corresponding normal logic program $\overline{UR(0,..,i+1)}$. In addition, each stable model for $\overline{UR(0,..,i+1)}$ is entailed by some fixpoint EDB_{i+1} for the update rules in UR_{i+1}.

Suppose we introduce the update rules in UR_{i+1} which participate in either r-cliques or a-cliques. Executing these update rules will only affect the atoms that occur in some maximal set associated with an r-clique or an a-clique. By Lemma 4.2, these sets of atoms are mutually exclusive, so we can independently consider the effect of

[1] The special literals P^*, corresponding to each literal P in some maximal set only occur in the corresponding normal logic program and they do not occur in the update rule program. They are not entailed in the fixpoint of the update rule program and hence they are not considered in the proof of the equivalence between the fixpoints of the update rule program and the stable models for the corresponding normal logic program.

المنارة للاستشارات
www.manaraa.com

introducing the r-cliques or the a-cliques. Lemmas 4.3, 4.4 and 4.5 place restrictions on the atoms in the maximal sets, so that they may only be distinguished in certain rules in UR_{i+1} and they are not retractable in UR_j, $j > i+1$. Following these restrictions, the only rules of the stratified logic program $\overline{UR^{-}(0,i+1)}$ that will be modified in the normal logic program $\overline{UR(0,i+1)}$, through the introduction of the a-cliques or the r-cliques in this partition, are those rules that are relevant to the atoms in the maximal sets. The **Inductive Case** will thus be presented independently for the a-cliques and the r-cliques, respectively, and for each we will present a (nested) base case (no overlaps) and an inductive case (with overlaps).

Nested base case for a-cliques

We consider one (or more) non-overlapping a-clique, comprising a-update rules s_1 through s_p, with p atoms in the maximal set $\{P_1,..., P_p\}$, in the program. The rules are as follows:

s_k: $\neg P_1, .., \neg P_{k-1}, \neg P_{k+1}, ., \neg P_p$, *Conj-k* $\rightarrow$ **assert** P_k

The corresponding normal logic program $\overline{UR(0,1)}$ is in Lemma 6.3. Since the cliques are non-overlapping, they may each be considered separately, and we base the analysis on the following four sub-cases:

Sub-case B.a.1:

Suppose EDB^{-}_{i+1} entailed one (or more) P_k. Then from Lemma 4.4, since this atom P_k is unretractable, we have EDB_1 will also entail this atom. P_k will be true in each fixpoint. If we consider the corresponding logic program of Lemma 6.3, the rule(s) in the program $\overline{UR^{-}(0,i+1)}$ that prove P_k, or the fact P_k, and the corresponding rule(s) in any $\overline{UR(0,i+1)}_M$, transformed wrt some stable model M, will not be affected by the introduction of the rule(s) derived from the a-update rule(s) s_k. Thus, P_k will be true in each of the stable models for $\overline{UR(0,i+1)}$.

Sub-case B.a.2:

Suppose EDB_i does not entail some P_k occurring in this maximal set. Further, suppose that some distinguished atom C_k in the corresponding conjunction *Conj-k* in s_k is not entailed by EDB^{-}_{i+1} and is false in $M_{\overline{UR^{-}(0,i+1)}}$. Finally, suppose this C_k does not have a positive dependency on any atom in the maximal set of any a-clique. Alternately, suppose some distinguished literal D_k that occurs negatively in *Conj-k* is entailed by EDB^{-}_{i+1} and is true in $M_{\overline{UR^{-}(0,i+1)}}$. From Lemma 4.5, these literals in *Conj-k* are unretractable in UR_1. Now s_k will not execute and both EDB^{-}_{i+1} and EDB_{i+1} will not entail this P_k and it will be false in each fixpoint. In this case, the rule(s) r_k corresponding to P_k will either be eliminated in the corresponding program $\overline{UR(0,i+1)}_M$, transformed wrt some stable model M, (if D_k is true), or it will not be able to prove P_k (if C_k is false). It follows that P_k will be false in any stable model for $\overline{UR(0,i+1)}$.

Sub-case B.a.3:

Suppose EDB_i does not entail some P_k in this maximal set. Further, suppose that some distinguished atom C_k in the corresponding conjunction *Conj-k* in s_k is not entailed by EDB^{-}_{i+1} and is false in $M_{\overline{UR^{-}(0,i+1)}}$. Finally, suppose that C_k has a *positive dependency* on some atom which occurs in the maximal set of this a-clique, and which is not entailed by EDB^{-}_{i+1}. For simplicity, we assume that all other literals in *Conj-k* may be ignored (although this is not necessary for the proof). Suppose this atom C_k, which is distinguished in *Conj-k*, has a positive dependency on P_k, i.e., the atom in the head of the same rule. Then, s_k will not execute. EDB_1 will not entail C_k or P_k. If we examine the corresponding stable models for $\overline{UR(0,i+1)}$, C_k cannot be true in any (minimal) stable model. Any model in which P_k is true will be unstable, since C_k is false. Thus, both P_k and C_k must be false in every stable model. Alternately, suppose C_k has a positive dependency on some P_t that occurs in the maximal set, $t \neq k$. In this case, too, s_k cannot execute since some other rule(s)s_t in the a-clique must execute first so that EDB_1 entails P_t. But from Lemma 4.6, if some s_t executes, then s_k cannot execute. Thus, EDB_{i+1} will not entail P_k but may entail C_k and P_t, $t \neq k$. If we examine the corresponding stable models for $\overline{UR(0,i+1)}$, any model in which P_k is true must also include C_k and P_t, $t \neq k$, to be stable. But such

www.manaraa.com

a model in which both P_k and some P_t, $t \neq k$, are true is not a minimal model for the corresponding program (from Lemma 6.3). Thus, P_k will not be true in any stable model.

Sub-case B.a.4:

One sub-case of interest is where EDB^-_{i+1} does not entail each of the P_k. Next, suppose that for at least one conjunction *Conj-k*, all distinguished atoms are entailed by EDB^-_{i+1} and are true in $M_{\overline{UR^-(0,i+1)}}$ and all distinguished literals that occur negatively are not entailed and are false. It follows that the corresponding a-update rule s_k can execute and EDB_{i+1} will entail this P_k. Now since P_k is distinguished and occurs negatively in all other a-update rules participating in this a-clique, it follows that none of the other a-update rules may execute. The interesting case is where we can ignore all the conjunctions *Conj-k* (as was done in the proof of Lemma 6.3). If all of the a-update rules may execute, then there are p possible fixpoints for UR_{i+1}, corresponding to this a-clique. In each of the fixpoints exactly one of the P_k will be true and all other atoms in the maximal set will be false. If we consider the results of Lemma 6.3, we see that these fixpoints correspond exactly to the p possible stable models for the logic program $\overline{UR(0,i+1)}$.

Sub-case B.a.5

A final situation is to suppose that C_k, some distinguished atom in *Conj-k*, has a positive dependency on some atom that occurs in the maximal set of some other non-overlapping a-clique. This situation differs from B.a.4 in that EDB_i may not entail this C_k but it may be entailed in EDB_{i+1}, after introducing the non-overlapping a-clique. If EDB_{i+1} indeed entails C_k, then this sub-case will be similar to sub-case B.a.4, where the rule s_k may execute. If EDB_{i+1} does not entail C_k, then it is similar to sub-case B.a.2, where the rule s_k cannot execute.

Nested inductive case for a-cliques

Suppose the fixpoints for programs with n a-cliques that overlap in UR_{i+1} entail all possible stable models for the corresponding Gelfond-Lifschitz transformed programs. Now consider introducing another a-clique which overlaps with the previously overlapping n a-cliques, i,e,, there are n+1 overlapping a-cliques. Each of the overlaps of this clique with one of the n other cliques may be represented as a set of a-update rules s_1 through s_{p+c}, relevant to p+c atoms $\{P_1,..., P_p, C_1,..., C_c\}$ and a set of a-update rules u_1 through u_{q+c}, relevant to q+c atoms $\{Q_1,..., Q_q, C_1,..., C_c\}$. $\{C_1,...,C_c\}$ represent the overlapping atoms common to the maximal sets, for each of the overlaps with the n a-cliques, respectively. The rules are as follows, where s_{k1} is representative of rules s_1 through s_p, and s_{p+k3} is representative of rules s_{p+1} through s_{p+c}, etc. : [1]

s_{k1}: $\neg P_1, .., \neg P_{k1-1}, \neg P_{k1+1}, ..,\neg P_p, \neg C_1, ..., \neg C_c,$ *Conj-s,k1* $\rightarrow$ **assert** P_{k1}

s_{p+k3}: $\neg P_1, ..,\neg P_p, \neg C_1, ..., \neg C_{k3-1}, \neg C_{k3+1}, ..., \neg C_c, \neg Q_1, ..., \neg Q_q,$ *Conj-s,p+k3* $\rightarrow$ **assert** C_{k3}

u_{k2}: $\neg Q_1, .., \neg Q_{k2-1}, \neg Q_{k2+1}, ..,\neg Q_q, \neg C_1, ..., \neg C_c,$ *Conj-u,k2* $\rightarrow$ **assert** Q_{k2}

u_{q+k3}: $\neg Q_1, ..,\neg Q_q, \neg C_1, ..., \neg C_{k3-1}, \neg C_{k3+1}, ..., \neg C_c, \neg P_1, ..., \neg P_p,$ *Conj-u,q+k3* $\rightarrow$ **assert** C_{k3}

The corresponding normal logic program $\overline{UR(0,i+1)}$, for each of these n overlaps, is in Lemma 6.4. We consider several sub-cases as follows:

Sub-case I.a.1

In this sub-case, each of the a-update rules s_1 through s_{p+c} and u_1 through u_{q+c} do not execute. The conditions for these rules to not execute are already detailed in sub-cases B.a.1 through B.a.3, and B.a.5, of the base case. There is a slight variation that must be considered in sub-case B.a.5, since now there are overlaps among the

[1] Note that the a-update rules that are relevant to the overlapping atoms in the maximal sets are repeated in this program and should not be considered to be different rules.

www.manaraa.com

a-cliques. For this case, we may suppose that C_{k1}, some distinguished atom in *Conj-s,k1*, has a positive dependency on either some atom Q_{k2} (which is not an atom in the overlap) or on some atom which occurs in the maximal set of some other a-clique with which there is no overlap at all. In all of these sub-cases, the overlap with each of the n a-cliques is itself not significant, and the analysis is similar to the base case for a-cliques. We omit this discussion here.

There are several sub-cases of interest where the a-update rules may execute, and they are as follows:

Sub-case I.a.2:

Suppose EDB^{-}_{i+1} does not entail each of the P_{k1} and C_{k3} occurring in the maximal set for rules s_1 through s_{p+c}. Further, suppose that we can ignore each of the conjunctions in the a-update rules relevant to each P_{k1} and each C_{k3}, i.e., all distinguished atoms are entailed by EDB^{-}_{i+1} and are true in $M_{\overline{UR^{-}(0,i+1)}}$, and all distinguished literals occurring negatively are not entailed and are false. Thus, the rules s_1 through s_{p+c} may execute. However, the a-update rules u_1 through u_{q+c} may not execute (similar to the situation described in one of the previous sub-cases of the base case (B.a.1 through B.a.3 and B.a.5).

Sub-case I.a.3:

Suppose the rules s_1 through s_{p+c} may not execute (similar to one of the sub-cases B-a.1 through B.a.3 and B.a.5) but the rules u_1 through u_{q+c} may execute.

Each of these two sub-cases will now reduce to n overlapping a-cliques and the result follows from the statement of the inductive case for n overlapping a-cliques.

Sub-case I.a.4:

Suppose EDB^{-}_{i+1} does not entail each of the P_{k1}, Q_{k2} and C_{k3} occurring in both maximal sets. Further, suppose that we can ignore each of the conjunctions in the a-update rules relevant to each P_{k1}, Q_{k2} and each C_{k3}. We made this assumption when proving the results of Lemma 6.4. Now any of the a-update rules in the overlapping a-cliques may execute. Suppose an a-update rule s_{k1} were to execute and entail some P_{k1}. From lemma 4.6, no other a-update rule s_t, $t \neq k1$, could execute and no other P_t, $t \neq k1$, may be entailed. Similarly, no C_t may be entailed. However, exactly one a-update rule u_{k2} can execute and entail exactly one Q_{k2}. Suppose on the other hand that some a-update rule relevant to some C_{k3} were to execute and entail some C_{k3}. Then, from Lemma 4.6, no other C_t, $t \neq k3$, may be entailed nor any other Q_t or P_t. It follows that there are exactly $(p * q + c)$ possible fixpoints, exactly corresponding to the $(p * q + c)$ stable models of the program $\overline{UR(0,i+1)}$, as seen in Lemma 6.4.

Nested base case for r-cliques

We consider one (or more) non-overlapping r-clique with p retractable atoms in the maximal set $\{P_1,..., P_p\}$. The p r-update rules s_1, .., s_p participating in the r-clique are as follows:

s_k: P_1, .., P_k, ..,P_p, *Conj-s,k* → **retract** P_k

The a-update rules p_k which are relevant to the retractable atoms in the maximal set of atoms associated with the r-clique and and are modified by these r-update rules are as follows:

p_k: *Conj-p,k* → **assert** P_k

From Lemma 4.2, we know that these a-update rules may not participate in an a-clique. The corresponding normal logic program $\overline{UR(0,i+1)}$ is in Lemma 6.1. Since the cliques are non-overlapping, each can be analyzed separately, based on the following sub-cases:

Sub-case B.r.1:

Suppose that some distinguished atom C_k in *Conj-s,k* is not entailed by EDB^{-}_{i+1} and is false in $M_{\overline{UR^{-}(0,i+1)}}$. Further suppose that this atom C_k does not occur in the maximal set of any a-clique in UR_{i+1}, nor does it have

المنارة للاستشارات
www.manaraa.com

a positive dependency on any other atom that occurs in a maximal set of an a-clique. Alternately, suppose some distinguished literal D_k that occurs negatively is entailed by EDB^{-}_{i+1} and is true in $M_{\overline{UR^{-}(0,i+1)}}$. From Lemma 4.5, these literals are unretractable in UR_{i+1}. In this case, the corresponding r-update rule(s) s_k will not be able to execute. If EDB^{-}_{i+1} entailed the corresponding atom P_k, then, EDB_{i+1} will also entail this atom.

If we consider the corresponding logic program of Lemma 6.1, and the program $\overline{UR(0,i+1)}_M$ transformed wrt to any stable model M, the rule(s) $r_{1,k}$, derived from s_k, will either be eliminated (if D_k is true), or it will not be able to prove P_k^* (if C_k is false). P_k^* will be false in any stable model. Consider the rule(s) $r_{2,k}$ in $\overline{UR^{-}(0,i+1)}$, relevant to P_k and which is derived from p_k transformed by s_k. It will be unchanged in any transformed program $\overline{UR(0,i+1)}_M$, since the literal P_k^* occurring negatively in this rule will be eliminated. We do not further consider this case since the fixpoint EDB_{i+1} remains unchanged from EDB^{-}_{i+1} wrt this atom P_k, and the corresponding rules relevant to P_k, in any $\overline{UR(0,i+1)}_M$, remain unchanged from $\overline{UR^{-}(0,i+1)}$.

Sub-case B.r.2:

Similarly, suppose EDB_i does not entail some P_k occurring in this maximal set. Further, suppose that some distinguished atom C_k in the corresponding conjunction *Conj-p,k* in p_k is not entailed by EDB^{-}_{i+1} and is false in $M_{\overline{UR^{-}(0,i+1)}}$. Again, we assume the same restrictions for C_k as in B.r.1. Alternately, suppose some distinguished literal D_k that occurs negatively is entailed and is true. Now, p_k will not execute, and both EDB^{-}_{i+1} and EDB_{i+1} will not entail this P_k. The corresponding rule(s) $r_{2,k}$ will not be able to prove P_k in $\overline{UR^{-}(0,i+1)}$. Also, in any $\overline{UR(0,i+1)}_M$, transformed wrt any stable model M, rule(s) $r_{2,k}$ will either not be able to prove P_k or will be eliminated. Thus, this case is similar to the previous case B.r.1.

Sub-case B.r.3:

One case of interest is where we can first ignore all the conjunctions *Conj-p,k*. This was one of the assumptions made to prove the results of Lemma 6.1. First, suppose all distinguished atoms in all such conjunctions *Conj-p,k* are entailed by EDB^{-}_{i+1} and are true in $M_{\overline{UR^{-}(0,i+1)}}$ and all distinguished literals that occur negatively in these conjunctions are not entailed and are false. Now EDB^{-}_{i+1} will entail all P_k and they will all be true in $M_{\overline{UR^{-}(0,i+1)}}$. Next, suppose that for at least one conjunction *Conj-s,k*, all distinguished atoms are entailed by EDB^{-}_{i+1} and are true in $M_{\overline{UR^{-}(0,i+1)}}$ and all distinguished literals occurring negatively are not entailed and are false. It follows that the corresponding r-update rule s_k can execute and EDB_{i+1} will not entail P_k. From lemma 4.6, none of these other r-update rules s_t, $t \neq k$, can execute. Thus, EDB_{i+1} will not entail some P_k and will entail all other P_t, $t \neq k$.

In the most general case, suppose that for all the r-update rules participating in the r-clique, we can ignore all the conjunctions, *Conj-s,k* (as was done. in Lemma 6.1). It follows that any of the r-update rules participating in this r-clique may execute and from Lemma 4.6, only one r-update rule in the r-clique will execute. There will be p possible fixpoints represented by some EDB_{i+1}. Each fixpoint will not entail exactly one P_k and will entail all other P_t, $t \neq k$.

From Lemma 6.1, when we correspondingly ignore all the literals in the conjunctions *Conj-s,k* and *Conj-p,k*, we have that there are exactly p stable models for the corresponding normal logic program $\overline{UR(0,i+1)}$. In each of these stable models, exactly one retractable atom of the maximal set of atoms will be false and all other retractable atoms will be true, where we ignore the special literals P^* introduced into the normal logic program. Thus, each possible fixpoint exactly entails one possible stable model and vice versa, ignoring the special literals introduced into the normal logic program.

Sub-case B.r.4:

Suppose that some distinguished atom X_k in *Conj-s,k* in some rule s_k (or in *Conj-p,k* in some rule p_k) is not entailed by EDB^{-}_{i+1} and is false in $M_{\overline{UR^{-}(0,i+1)}}$. Further suppose this atom X_k either occurs in the maximal set

المنارة للاستشارات
www.manaraa.com

of some a-clique in UR_{i+1}, or it has a positive dependency on some other atom that occurs in a maximal set of an a-clique. Now, after executing the rules in the a-clique, EDB_{i+1} may entail X_k. This sub-case is different from B.r.1 and B.r.2, in that EDB^-_{i+1} will not entail X_k, but X_k may be entailed by the fixpoint EDB_{i+1}. The rules s_k (or p_k) may or may not execute, depending on the sequence of execution of the rules, when the rules in the a-cliques and the r-cliques are introduced. For example, if a rule in some a-clique entailed X_k, then p_k alone or s_k and p_k may execute. However, if some other rule s_t $t \neq k$, in the r-clique, executes first, then s_k will not execute. If some rule in the r-clique does not execute, then this sub-case will be similar to B.r.1 or B.r.2, and if some rule does execute, then this sub-case will reduce to some form of the previous sub-case B.r.3.

Nested inductive case for a-cliques

Suppose the fixpoints for a programs with n r-cliques that overlap in UR_{i+1} entail all possible stable models for the corresponding Gelfond-Lifschitz transformed programs. Now consider introducing another r-clique which overlaps with the previously overlapping n r-cliques, i.e., there are n+1 overlapping r-cliques. Each of the overlaps of this r-clique with one of the n other cliques may be represented as a set of r-update rules u_1 through u_{p+c}, relevant to p+c atoms $\{P_1,..., P_p, C_1,..., C_c\}$ and and a set of r-update rules s_1 through s_{q+c}, relevant to q+c atoms $\{Q_1,..., Q_q, C_1,..., C_c\}$. $\{C_1,...,C_c\}$ represent the overlapping atoms common to the maximal sets, for each of the overlaps with the other n r-cliques. These overlapping r-cliques are as follows: [1]

u_{k1}: $P_1, .., P_{k1}, .., P_p, C_1, ..., C_c,$ *Conj-p,k1* $\rightarrow$ **retract** P_{k1}

u_{k3}: $P_1, .., P_p, C_1, ..., C_{k3}, ..., C_c, Q_1, ..., Q_q,$ *Conj-u,k3* $\rightarrow$ **retract** C_{k3}

s_{k2}: $Q_1, .., Q_{k2}, .., Q_q, C_1, .., C_c,$ *Conj-s,k2* $\rightarrow$ **retract** Q_{k2}

s_{k3}: $Q_1, .., Q_q, C_1, ..., C_{k3}, ..., C_c, P_1, ..., P_p,$ *Conj-s,k3* $\rightarrow$ **retract** C_{k3}

The a-update rules which are relevant to these retractable atoms and are modified by the r-update rules are as follows:

p_{k1}: *Conj-p,k1* $\rightarrow$ **assert** P_{k1}

p_{k2}: *Conj-p,k2* $\rightarrow$ **assert** Q_{k2}

p_{k3}: *Conj-p,k3* $\rightarrow$ **assert** C_{k3}

From Lemma 4.2, these a-update rules do not participate in an a-clique. The corresponding normal logic program is as listed in Lemma 6.2. As in the analysis of the previous **nested base case** for r-cliques, there are several situations that are not of much interest. For example, when all of the a-update rules and r-update rules cannot execute (detailed in B.r.1 and B.r.2), the overlap with each of the n a-cliques is itself not significant. Similarly, if we can ignore all the a-update rules and r-update rules relevant to some overlapping maximal set , then this reduces to the case of n overlapping r-cliques and the result follows from the statement of the nested inductive case for n overlapping cliques. We do not list these sub-cases here for readability.

Sub-case I.r.1:

One interesting case is to suppose EDB^-_{i+1} will entail all P_k and all C_m as well as all Q_t and they will all be true in $M_{\overline{UR^-(0,i+1)}}$. Once more we suppose that we can ignore some (or all) of the conjunctions *Conj-u,k*, *Conj-u,m*, *Conj-s,t* or *Conj-s,m*. [1]

Now either exactly one r_{k1} and exactly one s_{k2} will execute and EDB_{i+1} will not entail exactly one P_{k1} and exactly one Q_{k2}. An alternative is that exactly one r-update rule r_{k3} (or s_{k3}) will execute and EDB_{i+1} will not entail exactly one C_{k3} In the most general case, suppose that any of the r-update rules participating in the

[1] Note that the r-update rules that are common to the intersecting r-cliques are repeated.

المنارة للاستشارات
www.manaraa.com

mutually overlapping r-cliques may be chosen for execution. Then, since there are (p+c) and (q+c) atoms in the maximal sets, respectively, with c mutually overlapping literals, there will be ($c + p * q$) possible fixpoints, in this most general case.

We note that we ignored the various conjunctions in the rules of the corresponding normal logic program in the proof of Lemma 6.2. We further note from Lemma 6.2, that the ($p * q + c$) possible fixpoints that we have enumerated for the most general case exactly correspond to the stable models for the normal logic program, ignoring all the special literals introduced into the normal logic program. We refer the reader to Lemma 6.2 for the details.

Sub-case I.r.2:

The final sub-case of interest is where some distinguished atom X_k in one of the a-update rules (that do not participate in any a-clique) or any of the r-update rules of any r-clique, is not entailed by EDB^{-}_{i+1} and is false in $M_{\overline{UR^{-}(0,i+1)}}$. However, suppose this atom X_k either occurs in the maximal set of some a-clique in UR_{i+1}, or it has a positive dependency on some other atom that occurs in a maximal set of an a-clique. The execution of the rules in this a-clique may entail X_k. This sub-case is similar to B.r.4 in that EDB^{-}_{i+1} will not entail X_k, but X_k may be entailed in the fixpoint EDB_{i+1}. As before the r-update rules in the r-cliques and the a-update rules (that do not participate in any a-cliques) may or may not execute, depending on the execution sequence of the rules in the a-cliques and the r-cliques. This case may be similar to I.r.1 if X_k is entailed or it will reduce to n overlapping r-cliques if X_k is not entailed. We omit the detailed discussion. □

7. Comparisons with Related Research

We now compare our research with related research in providing a semantics for update rules as well as research on implementing rules in a DBMS. There has been considerable research to define a correct semantics for update rule programs. In [FKUV86], a general framework for dealing with updates is presented. The problem they consider is updating a theory by inserting and/or deleting sentences. A theory T is defined as a consistent set of sentences, in particular, a set of first-order, well-formed closed formula. The set of logical consequences of the theory is denoted T^*. Let σ be a sentence. Then, the theory S accomplishes the update corresponding to the insertion of σ into T if σ is an element of S. Similarly, S accomplishes the update corresponding to the deletion of σ from T if $\sigma \notin S^*$. Notice that the inserted sentence is explicit in S. Now suppose that $T \cup \sigma$ is inconsistent but σ is to be inserted, or that σ is to be deleted while $\sigma \in T^*$. In both of these cases, a *minimal* new theory S must be defined. Suppose that T_1 and T_2 are both theories that accomplish the update. Then, the authors define what it means for T_1 to accomplish the update with a *smaller change* than T_2. Now, S accomplishes an update u of T *minimally* if there is no theory S' which accomplishes update u with a *smaller change* than S.

The research in [FKUV86] provides a conceptual framework for defining a semantics for updating sentences in a theory. The problem we address is a simpler problem since our database corresponds to a set of facts. The UR program comprises an initial database of facts EDB_{init} and a set of update rules. Intuitively, the a-update rules that assert new facts are similar to deductive rules since each rule asserts new information based on some condition that is satisfied by the database. The r-update rules that retract facts intuitively resemble integrity constraints. They check the database for conditions that must not hold. If the condition does hold, then they retract some information that exists in the database so the condition no longer holds. This is very similar to the process of maintaining integrity constraints in a DBMS environment. Our objective then is to obtain a final database for the initial facts and the new facts that are asserted such that this final database is minimal, and is consistent with the conditions specified by the r-update rules, representing the integrity constraints.

المنارة للاستشارات
www.manaraa.com

Our approach is different from the framework suggested in [FKUV86] since we treat the r-update rules as integrity constraints. The final database must satisfy these integrity constraints. As a result, information which is explicitly specified by the user in the initial database EDB_{init} may be later retracted by some update rule, in order to maintain an integrity constraint. The same is true for information which is explicitly deleted by the user since it may be asserted by another rule. In contrast, in [FKUV86], information which is explicitly updated or retracted by the user is not changed. Another feature of our fixpoint semantics is that when a tuple is retracted by a rule, then subsequent assertions (of the tuple) have no effect and the database will not entail this tuple.

To obtain a declarative semantics, we associate a normal logic program, $\overline{UR}$, with a UR program comprising of the initial database EDB_{init} and the update rules. The stable model semantics for normal logic programs [GeLi88] associates a set of minimal models which are called stable models with a normal logic program $\overline{UR}$. A stable model for $\overline{UR}$ must be equivalent to the database obtained through the execution of the update rules of UR. The stable model is consistent with the conditions specified by the integrity constraints. This approach to providing a declarative semantics based on the stable model semantics of a corresponding normal logic program is also different from other research in this area.

The most extensive research in update rules has been described in [AbSV90, AbVi90, AbVi91]. They extend Datalog programs together with a procedural component as a means of overcoming the limited expressive power of purely declarative semantics, for example, the inability to capture updates in pure Datalog programs. The focus is on the expressive power of the languages. Theoretical results on computational complexity are discussed. They also investigate the connections between Datalog extensions with fixpoint semantics, explicitly procedural languages and fixpoint extensions of first order logics. The Datalog extensions that are considered include negative literals in the body of the rules as well as in the heads. They define invented values which correspond to free variables occurring in the heads of rules. Negative literals in the heads of rules are interpreted as deletions. Both deterministic and non-deterministic semantics for Datalog programs extended with negation are discussed. In a deterministic execution, all the rules whose bodies are satisfied in the database will be chosen simultaneously, and their updates applied to the database, so that the order of executing rules is not significant. The issue of applying the updates of more than one rule simultaneously has also been addressed by other researchers in the context of set-oriented execution, as will be discussed.

In our research, we are interested in a much smaller class of update rules whose fixpoint semantics can be correctly implemented in a relational DBMS with extensions. Another difference is that update rules which delete facts are interpreted by us as integrity constraints. This is intuitively closer to the meaning commonly associated with rules that delete facts, from the perspective of a designer in a DBMS environment. This contrasts with their solution of equating deletions with negative conclusions in the heads of rules. Our fixpoint semantics is non-deterministic in that the fixpoint operator selects a single rule from a partition, in each step. Since the order of rule selection is significant, it could produce more than one final database.

The research most similar to our approach is described in [MaSi88, SiMa88]. They consider production rule languages that are extensions of logical query languages such as Datalog with updates in the head of the rules. They provide an operational and declarative semantics for stratified Datalog programs that have a sequence of updates in the heads of the rules. Their declarative semantics is similar in spirit to modal logics. In addition to an implicit ordering of the rules due to stratification, explicit ordering is also considered. In our research, we have extended the concept of stratification for update rules to permit us to represent non-deterministic programs, but we use no explicit control. Their execution model is based on a Predicate Compilation Network (PCN) and is derived from Petri-Net models. One drawback is that in the PCN model for rule execution, there may not be a stable state associated with the network representing the rule programs; this implies that rule execution may not terminate. They use explicit control during rule execution to deal with this problem. Our execution semantics is based upon a monotonic fixpoint operator. Asserts and retracts are explicitly stored and the fixpoint operator guarantees termination of rule

www.manaraa.com

execution. We do not require any explicit control information for the execution to terminate.

We note that although we have not considered programs with multiple updates in the head of a rule, this is a straightforward extension. Another difference from the research in [MaSi88] is that we interpret update rules which delete information as integrity constraints; we feel that this is more intuitive with the use of these rules from the viewpoint of DBMS designers. We are also interested in identifying a class of programs that can be easily implemented in a DBMS. Executing a rule must be very similar to query execution so that all the techniques developed for the efficient execution of queries is applicable to rule execution as well.

There has been some research in update semantics in conjunction with logic programming research [MaWa88, NaKr88]. In the language proposed in [MaWa88], updates can occur in the body of the rules, as well as in the heads. In the language proposed in [NaKr88], the updates occur only in the body of the rules. The declarative semantics for both proposals are based on modal logics. The focus of their research is in providing a formal semantics; implementations are based on a proof theory, which is not as applicable in a DBMS environment where the rule execution is usually implemented to be a forward chaining evaluation.

Finally, we discuss related research in [CeWi90, Wido91, WiFi90], which describes implementing production rules in a DBMS environment. A very powerful set-oriented production rule language is defined and an execution semantics which is also set-oriented is provided in [WiFi90]. Since the language is set-oriented, there is a related issue of whether a tuple-oriented and a set-oriented implementation will provide the same or different answers; this is an important issue if a precise semantics for the program is to be defined. We note that although the set-oriented execution described in [WiFi90] will apply the changes of multiple rules simultaneously, it does not necessarily produce a deterministic answer, and the order of execution of each rule is significant. Their semantics is therefore different from the deterministic semantics of [AbSV90]. If we consider our rule language, the syntactic restrictions we place on the UR programs guarantees that a set-oriented execution is a special case of the tuple-oriented execution. We see this as an advantage since the set-oriented execution is similar to query processing in a DBMS and can be efficient.

The focus of the research in [Wido91, WiFi90] is on the computational procedures required to determine when to trigger rule execution, and determining when no more rules can be executed. They have identified the concept of a *transition table* to represent all the tuples which are affected by executing a transaction; the transition table is critical in determining when the rules are to be evaluated. These are important implementation issues. The transition table also allows representation of a change in state of the database after rule execution, and thus represents a very different semantics from our research.

In [CeWi90], a very powerful constraint language is defined and a procedure for deriving constraint maintaining production rules is described. Their research focus is on the nature of constraints and the process of maintaining constraints. We do not explore these issues in our paper. They guarantee that if the correct actions are specified, corresponding to each violation of a constraint, then the rules are guaranteed to take the corrective actions, as specified. In our research, too, we treat the update rules which retract information as rules that maintain an integrity constraint. However, in our research, we use the declarative meaning given to the program comprising the initial database and the update rules to determine which rules must execute and to determine the final database. This is an advantage since we provide a declarative meaning for the fixpoint or forward-chaining semantics. An interesting issue for future research would be to consider using our update rules and the fixpoint semantics to implement the constraint language defined in [CeWi90].

8. Implementing Update Rules in a Relational DBMS

We briefly outline an implementation of the fixpoint semantics for update rule programs in an extended DBMS environment. An important requirement is that the implementation should be straightforward, and require minimal extensions to the relations or to the query processing strategy in a DBMS. We describe an incremental

المنارة للاستشارات
www.manaraa.com

evaluation scheme where each rule is represented by a query which conditionally updates the relations in the database.

8.1. Obtaining Queries from the Rules

We assume there is a relation corresponding to each predicate. The number and type of the attributes of the relations correspond to the number and type of the terms in the predicate. The *database* relations correspond to predicates which do not occur in the head of any update rules. The *updatable* relations correspond to predicates which occur in the head of the update rules. These predicates may also occur in the body of the rules. To represent tuples that are asserted and retracted, while guaranteeing that rule execution will terminate, each updatable relation is extended with two additional attributes. These two attributes are named the **a-flag** and **r-flag**, respectively. The update rules are represented by a set of queries that execute against both the database and the updatable relations.

When a tuple is inserted into an updatable relation through an explicit insert operation or when it is asserted through the execution of an a-update rule, then the corresponding tuple is placed in the updatable relation. It has its **a-flag** value set to **1**, and its **r-flag** value set to 0, to indicate it has been asserted. When this tuple is retracted from an updatable relation, by executing an r-update rule, the value of the **r-flag** is set to **1**. When a tuple is deleted from an updatable relation explicitly, then it will be actually deleted from this relation. Inserts and deletes from the database relations are as usual, i.e., these relations are not extended with the flags.

Literals that correspond to database relations are interpreted in the usual manner against the tuples in the database relations. Informally, a *unifying* tuple in a database relation *satisfies* a positive literal when the attribute values of the unifying tuple satisfy the selection predicate that is specified in the vector of terms (constants and/or variables), corresponding to that literal. The exact relationship between the vector of terms in the literal and the attributes of the unifying tuple is straightforward. A literal occurring negatively in the body of a rule, which corresponds to a database relation, is satisfied when there are no unifying tuples in the corresponding relation that satisfy the selection predicate.

However, for literals corresponding to updatable relations, there are additional requirements that must be met for the literals to be satisfied. They are as follows:

(1) A literal occurring positively in the body of a rule will be satisfied by the existence of a unifying tuple of an updatable relation which satisfies the selection predicate and has its **a-flag** value equal to 1 and its **r-flag** value equal to 0. This indicates that the unifying tuple has either been explicitly inserted into the relation or it has been asserted by an a-update rule, but it has not been retracted by an r-update rule.

(2) A literal occurring negatively in the body of a rule will be satisfied if *all* unifying tuples of an updatable relation that satisfy the selection predicate have both the **a-flag** value and the **r-flag** value set equal to 1.

(3) A literal occurring negatively in the body of a rule will also be satisfied when are no unifying tuples in the corresponding updatable relations, or all the unifying tuples do not satisfy the selection predicate.

(4) A literal occurring negatively in the body of a rule will *not* be satisfied if there exists *any* unifying tuple which matches the selection predicate and has its **a-flag** set equal to 1 but its **r-flag** set equal to 0. This tuple has been explicitly inserted or asserted by an a-update rule but it has not been retracted by an r-update rule or explicitly deleted.

The details of obtaining the corresponding queries from the update rules, based on the above criteria, is straightforward. When these queries are executed, unifying tuples are retrieved from the relations corresponding to the literals in the body of the rule, so that it can be determined if these unifying tuples satisfy the selection predicate in the literals in the body of the rules. For those rules where the literals in the body are satisfied, the queries update the updatable relation corresponding to the literal in the head of the rule.

المنارة للاستشارات
www.manaraa.com

8.2. Maintaining an Update Rule System in a DBMS

The first step is to create a database and populate the relations to reflect the state of the initial database. This is the *Initialize* step. Let EDB_{init} be a set of tuples representing the initial database. This step is as follows:

Initialize

```
For each database relation P
   For each tuple P(t̄) in EDB_init
      { INSERT INTO P
                  (P.attr_i := the corresponding constant in P(t̄) ) }
For each updatable relation P
   For each tuple P(t̄) in EDB_init
         { INSERT INTO P
                  (P.a-flag := 1) AND (P.r-flag := 0) AND
                  (P.attr_i := the corresponding constant term in P(t̄) ) }
```

Once a database has been created, there are three phases in the processing required to implement the semantics of the update rule system. We expect that the system will cycle through these phases each time the database relations are explicitly updated by a user, in order to maintain the updatable relations.

In the first phase, *ExplicitUpdate*, the contents of the relations must be updated to reflect the changes to both database and updatable relations, as specified explicitly by the user. Suppose EDB_{ins} are the tuples that are inserted and EDB_{del} are the tuples that are deleted. Clearly, as the database is updated over time, only a small portion of EDB_{init} may actually change in each cycle. In the ExplicitUpdate phase, the processing is as follows:

ExplicitUpdate

```
For each database relation P
         For each tuple P(t̄) in EDB_ins
            { INSERT INTO P
                  (P.attr_i := the corresponding constant in P(t̄) ) }
         For each tuple P(t̄) in P which is in EDB_del
            { DELETE FROM P
                  WHERE {for each attribute in P}
                  (P.attr_i = the corresponding constant in P(t̄) ) }
For each updatable relation P
         For each tuple P(t̄) in EDB_ins
            { INSERT INTO P
                  (P.a-flag := 1) AND (P.r-flag := 0) AND
                  (P.attr_i := the corresponding constant in P(t̄) ) }
         For each tuple P(t̄) in P which is in EDB_del
           { DELETE FROM P
                  WHERE
                  (P.a-flag = 1) AND (P.r-flag = 0) AND
                  {for each attribute in P}
                     (P.attr_i = the corresponding constant in P(t̄) ) }
```

The second phase is the *Evaluation*. Let $Q{-}UR_i$, be the set of database queries derived from the set of a-update rules and/or r-update rules in each partition UR_i of the update rule program UR. The database queries corresponding to each partition of UR_i are executed, in turn, starting from UR_1, until the queries in each partition, $Q{-}UR_i$ can no longer update the updatable relations, i.e., a fixpoint is reached.

المنارة للاستشارات
www.manaraa.com

Evaluation

For each partition UR_i in increasing order of i
    Non-deterministically select and execute a query from $Q-UR_i$
        and update the updatable relations until a fixpoint is reached

The final phase is the *Completion*. All the tuples which have been retracted by the execution of an r-update rule are deleted from the updatable relations. To explain, it is only during the processing of the *Evaluation* phase that we need to explicitly maintain information on all the tuples that were retracted through the execution of r-update rules so that the execution will terminate. Once the *Evaluation* completes, we no longer require this information about the retracted tuples, and these tuples can be deleted. This will reduce the size of the updatable relations. It will also allow a tuple which was retracted through the execution of an r-update rule in the current Evaluation phase to be re-asserted in the next ExplicitUpdate phase, without compromising the conditions for termination of rule execution. The *Completion* is as follows:

Completion

For each updatable relation P
  { DELETE FROM P
      WHERE
      (P.**a-flag** = 1) AND (P.**r-flag** = 1) }

9. Summary and Future Research

In this paper, we have defined a semantics for update rules, i.e., rules whose execution can update the database and can cause the further execution of rules. Our motivation is to define a class of programs which can be easily identified by a DBMS designer and which will be useful in common DBMS applications such as to capture triggering information, to maintain integrity constraints, or to make simple inferences. Our research focus is on supporting these UR programs in an extended DBMS environment. The class of UR programs is syntactically identified, based on the concept of stratification. We extend the strict definition of stratification and allow a more flexible criterion for partitioning the rules in the UR program. This relaxation allows a certain degree of non-determinism in rule execution, which can be very useful in maintaining integrity constraints in the database.

We define a semantics which is based upon a monotonic fixpoint operator T_{UR}. The monotonicity of the operator is achieved by explicitly recording the effects of inserts and deletes of the tuples in the database. The semantics guarantee the execution of the rules will terminate and will produce a minimal database.

To obtain a declarative semantics for UR programs, we associate a normal logic program $\overline{UR}$ with each UR program. We use the stable model semantics for normal logic programs [GeLi88], which associates a set of stable models with each normal logic program. The set of stable models are equivalent to the minimal databases that are (non-deterministically) obtained in the fixpoint for the UR program. We prove that the fixpoint semantics for the update rule program is sound and complete wrt the stable model semantics for the corresponding normal logic program.

We describe implementing the semantics of the UR program in a DBMS environment. Relations that can be updated by the rules or *updatable* relations are extended with two flags. An update rule can be represented by a database query, which queries the updatable relations as well as database relations (which are not updated by the rules). We describe an algorithm to compute a fixpoint in the DBMS environment and obtain the final updated database. A prototype system that recognizes the legal update rule programs, translates the rules into database queries and executes these queries against the database relations and the updatable relations is being built.

In future research, we propose extensions to the class of UR programs, so that they may be more useful in the context of DBMS applications. This includes the support of explicit control information for rule execution. We

www.manaraa.com

also wish to allow multiple updates in the head of each rule. From the viewpoint of declarative semantics, we would like to be able to extend the semantics so that the programs may be able to handle exceptions as used in AI programs, and causality [RaLo94]. We feel these extensions will be important to solve the problem of combining multiple knowledge bases.

We also propose to study the implementation issues in a DBMS more extensively. This includes the problem of incremental computation of queries, and the concurrent execution and set-oriented execution of update rules. We are also investigating magic transformations of the update rule programs to provide more efficient bottom-up computation techniques [PaRa94].

المنارة للاستشارات
www.manaraa.com

10. References

[AbSV90] Abiteboul, S., Simon, E. and Vianu, V. Non-deterministic languages to express deterministic transformations. *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 218-229, 1990.

[AbVi90] Abiteboul, S. and Vianu, V. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, Volume 41, Number 2, pages 181-229, October 1990.

[AbVi91] Abiteboul, S. and Vianu, V. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, Volume 43, Number 1, pages 62-124, 1991.

[ApBW88] Apt, K.R., Blair, H.A. and Walker, A. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, (Minker, J., editor), pages 89-148, Morgan Kaufmann Publishers, Inc., 1988.

[Bocc86a] Bocca, J. EDUCE-a marriage of convenience: Prolog and a relational DBMS. *Proceedings of the Third IEEE Symposium on Logic Programming,* pages 36-45, 1986.

[Bocc86b] Bocca, J. On the evaluation strategy of EDUCE. *Proceedings of the ACM Sigmod International Conference on the Management of Data,* pages 368-378, 1986.

[CeWi90] Ceri, S. and Widom, J. Deriving production rules for constraint maintenance. *Proceedings of the Conference on Very Large Data Bases*, Brisbane, Australia, pages 566-577, 1990.

[Demo82] Demolombe, R. Syntactical characterization of a subset of domain independent formulas. Technical Report, ONERA-CERT, Toulouse, 1982.

[DeEt88] Delcambre, L.M.L. and Etheredge, J.N. A self-controlling interpreter for the Relational Production Language. *Proceedings of the ACM Sigmod International Conference on the Management of Data,* pages 396-403, 1988.

[FKUV86] Fagin, R., Kuper, G., Ullman, J.D. and Vardi, M.Y. Updating Logical Databases. In *Advances in Computing Research*, Volume 3, pages 1-18, (Kanellakis, P., editor) JAI Press, 1986.

[Forg81] Forgy, C.L. OPS5 User's Manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[Forg82] Forgy, C.L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* (19), 1982.

[GoPa91] Gordin, D. and Pasik, A. "Set-oriented constructs for rule-based systems. *Proceedings of the ACM Sigmod International Conference on the Management of Data,* pages 60-67, 1991.

[GuFN89] Gupta, A., Forgy, C. and Newell, A. High speed implementations of rule-based systems. *ACM Transactions on Computer Systems*, Volume 7, Number 2, pages 119-1467,1989.

[Haye85] Hayes-Roth, F. Rule based systems. *Communications of the ACM* (28) 9, 1985.

[KoGM87] Kohli, M., Giuliano, M., Minker, J. An overview of the PRISM project. *Computer Architecture News*, Vol 15, Number 1, 1987.

[KoSa89] Kowalski, R. and Sadri, F. Knowledge representation without integrity constraints. Technical Report, Department of Computing, Imperial College of Science and Technology, 1989.

[KoSa90] Kowalski, R. and Sadri, F. Logic programs with exceptions. Proceedings of the International Conference on Logic Programming, pages 598-613, 1990.

[Lobo90] Lobo, J. Semantics for normal disjunctive logic programs. Ph.D. thesis, Department of Computer Science, University of Maryland, 1990.

المنارة للاستشارات
www.manaraa.com

[MaSi88] Maindreville, C. de and Simon, E. Modeling non-deterministic queries and updates in deductive databases. *Proceedings of the Conference on Very Large Data Bases,* pages 395-406 , 1988.

[MaWa88] Manchanda, S. and Warren, D. A logic-based language for database updates. In Minker. J. editor, *Foundations of deductive databases and logic programming*, Morgan Kaufmann Publishers, Inc., 1988.

[Mira87] Miranker, D.P. TREAT: A better match algorithm for AI production systems. *Proceedings of the National Conference on Artificial Intelligence*, 1987.

[Mink88] Minker. J., editor, *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publishers, Inc., 1988.

[MUvG86] Morris, K., Ullman, J. and van Gelder, A. Design overview of the NAIL system. *Proceedings of the Third International Conference on Logic Programming,* pages 554-568, 1986.

[NaKr88] Naqvi, S. and Krishnamurthy, R. Database updates in logic programming. *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 251-262, 1988.

[Nico82] Nicolas, J.M. Logic for improving integrity checking in relational data bases. *Acta Informatica,* 18, 3, pages 227-253, 1982.

[NiDe83] Nicolas, J.M., and Demolombe, R. On the stability of relational queries. Technical Report, ONERA-CERT, Toulouse, 1983.

[PaRa94] Pang, P. and Raschid, L., Bottom-Up magic evaluation techniques for stratified production rules in a DBMS. Technical Report, University of Maryland Institute for Advanced Computer Studies, 1994

[Rasc90] Raschid, L. Maintaining consistency in a stratified production system program. *Proceedings of the AAAI National Conference on Artificial Intelligence,* pages 284-289, 1990.

[Rasc94] Raschid, L. A semantics for a class of stratified production system programs. *Journal of Logic Programming*, Volume 21, Number 1, pages 31-57, 1994.

[RaLo94] Raschid, L. and Lobo, J. A Semantics for a class of non-deterministic and causal production system programs. *Journal of Automated Reasoning*, Volume 12, pages 305-349, December 1994.

[SeLR88] Sellis, T., Lin, C-C. and Raschid, L. Implementing large production systems in a DBMS environment: concepts and algorithms. *Proceedings of the ACM Sigmod International Conference on the Management of Data,* pages 34-45, 1988.

[SeLR93] Sellis, T., Lin, C-C. and Raschid, L. Coupling production systems and database systems: a homogeneous approach. *IEEE Transactions on Knowledge and Data Engineering,* Volume 5, Number 2, pages 240-256, April 1993.

[SiMa88] Simon, E. and Maindreville, C. de Deciding whether a production rule is relational computable. *Proceedings of the International Conference on Database Theory*, Bruges, Belgium, pages 205-222, 1988.

[StHP88] Stonebraker, M., Hanson, E.N., and Potamianos, S. The POSTGRES rule manager. *IEEE Transactions on Software Engineering* (14) 7 (1988)

[TsZa86] Tsur, S. and Zaniolo, C. LDL: A logic-based data language. *Proceedings of the Twelfth International Conference on Very Large Data Bases,*, pages 33-41, 1986.

[Ullm85] Ullman, J. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, Volume 10, Number 3, 1985.

المنارة للاستشارات
www.manaraa.com

[vEKo76] van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language. *Journal of the ACM*, Volume 23, Number 4, pages 733-742, 1976.

[Wido91] Widom, J. Deduction in the Starburst production rule system. IBM Research Report RJ 8135, IBM Research Division, Yorktown Heights, NY, 1991.

[WiFi90] Widom, J. and Finkelstein, S.J. Set-oriented production rules in relational database systems. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 259-270, 1990.

المنارة للاستشارات

www.manaraa.com

11. Appendix 1: The Translation with First-Order Predicates

In the case of the first-order predicates, we first specify some syntactic restrictions on the vectors of terms, (variables and/or constants), associated with predicates in the cliques[1]. These restrictions ensure that the same (set of) tuples are unifiable with each occurrence of the predicates, in all of the rules participating in a clique. We then describe the translation to obtain a normal logic program.

11.1. Specifying the Cliques for First-Order Predicates

Suppose $P_1,..., P_p$, are the predicates that occur in some maximal set associated with either an a-clique or an r-clique. Further, suppose that the vector of terms associated with predicate P_k, in some a-update rule or r-update rule r_t, in this clique, is $\overline{u_{k,t}}$. The term $\overline{u_{k,t}}$ must be unifiable with all other terms $\overline{u_{k,s}}$, for P_k, occurring in some other rule r_s, $t \neq s$. In addition, the following conditions must hold:[2]

(1) Consider any two rules r_{t1} and r_{t2} in the clique, and the terms $\overline{u_{k,t1}}$ and $\overline{u_{k,t2}}$, associated with P_k. Suppose the i-th element of $\overline{u_{k,t1}}$ is a variable (or constant). Then, the i-th element of $\overline{u_{k,t2}}$ must also be a variable (or constant).

(2) Consider any two rules r_{t1} and r_{t2} in the clique, and any terms $\overline{u_{k1,t1}}$, $\overline{u_{k1,t2}}$, $\overline{u_{k2,t1}}$, $\overline{u_{k2,t2}}$, associated with any two predicates P_{k1} and P_{k2}, in these rules, respectively. Suppose the i-th element of $\overline{u_{k1,t1}}$ and the j-th element of $\overline{u_{k1,t2}}$ are common variables, i.e., they correspond to a join condition. Then the i-th element of $\overline{u_{k2,t1}}$ and the j-th element of $\overline{u_{k2,t2}}$ must also correspond to a join condition.

(3) All the variables in the vector of terms associated with the predicates in the maximal set of the clique, $P_1,...,$ P_p, must be mutually disjoint with the all the variables that occur in the vector of terms associated with the other predicates, in the other rules of the clique.

11.2. The Translation for the First-Order Predicates

Step 1

Each r-update rule q relevant to $P(\bar{v})$ transforms each a-update rule p which is relevant to $P(\bar{u})$, or the fact $P(\bar{u})$, if it occurs in EDB_{init}. Two special unused predicates P^* and P^{**} are associated with each predicate P in the UR program which is transformed in this step.

Let the a-update rule p be of the following form:

p: $A_1, ..., A_a, \neg B_1, ..., \neg B_b \rightarrow$ **assert** $P(\bar{u})$

where a or b could be equal to 0, i.e., $P(\bar{u})$ could be a fact.

Let each r-update rule q relevant to P be as follows:

q: $P(\bar{v}), C_1, ..., C_c, \neg D_1, ..., \neg D_d \rightarrow$ **retract** $P(\bar{v})$

Let $\bar{u}$ and $\bar{v}$ unify with the most general unifier θ. Then, the following rules are placed in $\overline{UR}$:

$C_1, ..., C_c, \neg D_1, ..., \neg D_d \rightarrow P^*(\bar{v})$

$\neg P^*(\bar{v})\,\theta, A_1\,\theta, ..., A_a\,\theta, \neg B_1\,\theta, ..., \neg B_b\,\theta \rightarrow P(\bar{u})\,\theta$

---

[1] we use $\bar{v}$ to represent the vector of terms $(\overline{v_1}, ..., \overline{v_n})$.

[2] The conditions we present are very simple and may eliminate several cliques. We can define more complicated conditions that accommodate more cliques, but do not present the details here.

المنارة للاستشارات
www.manaraa.com

$\neg P^{**}(\bar{u}), A_1, ..., A_a, \neg B_1, ..., \neg B_b \rightarrow P(\bar{u})$

$(\bar{v} = \bar{u}) \rightarrow P^{**}(\bar{u})$

Step 2

Each fact in EDB_{init} which is not modified by any r-update rule, i.e., there are no r-update rules relevant to this fact is placed unchanged in $\overline{UR}$.

Each a-update rule p relevant to $P(\bar{u})$ which is as follows:

p: $A_1, ..., A_a, \neg B_1, ..., \neg B_b \rightarrow$ **assert** $P(\bar{u})$

and which is not modified by any r-update rule will derive the following rule in $\overline{UR}$:

$A_1, ..., A_a, \neg B_1, ..., \neg B_b \rightarrow P(\bar{u})$

المنارة للاستشارات

www.manaraa.com