Updates in a Rule-Based Language for Objects

Michael Kramer*          Georg Lausen†          Gunter Saake‡

Abstract

The integration of object-oriented concepts into deductive databases has been investigated for a certain time now. Various approaches to incorporate updates into deduction have been proposed. The current paper presents an approach which is based on object versioning; different versions of one object may be created and referenced during an update-process. By means of such versions it becomes possible to exert explicit control on the update process during bottom-up evaluation in a rather intuitive way. The units for updates are the result sets of base methods, i.e. methods, whose results are stored in the object-base and are not defined by rules. However, the update itself may be defined by rules. Update programs have fixpoint semantics; the fixpoint can be computed by a bottom-up evaluation according to a certain stratification.

1 Introduction

The integration of object-oriented concepts into deductive databases has been discussed and investigated for a certain time now [Ban86,DOO89,Abi90,AK89,KL89,KLW90,DOO91]. Various approaches to incorporate updates into deduction have been proposed. However, only a few of these take object-orientation into account. In the current paper we present an approach which is based on object versioning; different versions of one object may be created and referenced during an update-process. By means of such versions it becomes possible to exert explicit control on the update process during bottom-up evaluation in a rather intuitive way. As units for updates we consider the result sets of base methods, i.e. methods, whose results are stored in the object-base; we do not consider derived methods, i.e. methods, whose results are defined by rules. However, the update itself may be defined by rules. Update programs have fixpoint semantics; the fixpoint can be computed by a bottom-up evaluation according to a certain stratification.

In deductive databases, depending on whether top-down or bottom-up evaluation strategies are applied, updates are done in rule-bodies or rule-heads. In top-down approaches, updates are contained in the rule-bodies and are performed as side-effects of the refutation process. Much work has been done on the topic of updating derived (intensional) predicates. These approaches typically rely on SLD-, SLDNF-Resolution or Abduction (e.g. [AT91,Dec90,KM90,Tom88]). Examples for approaches considering updates of base predicates are Prolog, LDL [NT89] and DLP [MW87]; DLP manages updates of derived predicates, too. Bottom-up approaches for updates also have been proposed. In [AV91] various extensions of Datalog including deletions are investigated, and the language RDLi [dMS88] provides a separate component for explicit control of the bottom-up evaluation. Moreover, updates in production systems (e.g. OPS5 [BFKM86]) and corresponding extensions of relational databases by rules (e.g. [SJGP90,WF92,ZH90]) are realized by applying the rules in a bottom-up way, and, finally, also some database programming languages which incorporate rules follow this way (e.g. [PDR91,HJ91]).

From those deductive languages involving object-oriented features, only a few provide update concepts, e.g. Logres [CCCR+90] and LOCO [LVVS90]. Logres is a typed extension of Datalog, supporting object-identity, classes and isa-hierarchies. Updates can be...
expressed by using rules with deletions in the head; the evaluation of the rules may be done according to stratified or inflationary semantics. In addition, the set of relevant rules may also be updated; based on this feature also derived methods can be updated. LOC0 is based on ordered logic [LSV90]: a set of Datalog-like rules (allowing negation in rule-heads) may be ordered in a isa-hierarchy to allow inheritance. Updates are done by making the new rules an instance of the to-be-updated object; applying inheritance with overriding yields the instance as updated object.

In this paper we present a different approach to the update problem. The intentions are to provide a rule-language which allows to exert explicit control on the update process during bottom-up evaluation in a rather intuitive way. Control is based on so called version-identities (VIDs), which are special object-identities, built-up by function symbols denoting types of updates (insert, delete, modify) in such a way, that they admit tracing back the history of updates performed on each object. This approach is stimulated by F-logic [KI, KI90], where general terms are used to denote objects (see also [CW89, KW89]) and to control versions; however, updates are not considered in these works. VIDs have temporal characteristics, denoting different versions of an object during its update-process. Each object-version can be considered as a single stage - corresponding to a certain time-step - of the entire process of updating the object. A set of update-rules forms an update-program. Update-programs have fixpoint semantics; the fixpoint can be computed by a bottom-up evaluation according to a certain stratification.

Object-versions are a well established concept in object-oriented databases [Kim91]. Object-versions are used to manage the (long-term) evolution of an object, e.g. to support cooperative work. In the current paper we use versions in a different context. We consider versions as a means to support single updates, several of them may give rise to introduce a new version in the usual sense. Thus our approach outlines a complementary application of the version concept in rule based object-oriented databases.

The rest of this paper is organized as follows: In Section 2, we introduce a simple rule-language to define updates, outline our ideas, give a motivating example and a discussion of related approaches. In Section 3 we introduce an immediate consequence operator, which is the basis for bottom-up evaluation. Bottom-up evaluation is discussed in Section 4. In Section 5 the construction of the updated object-base is outlined, and, finally, Section 6 suggests extensions of our language and indicates future work.

2 Updates by Versioning

2.1 An Update-Language for Objects

We are interested in a language for objects, by which we can define updates using rules. The alphabet of our update language consists of (1) a nonempty set \( O \) of object-identities (OIDs) to denote the relevant objects, (2) an infinite set \( V \) of variables to denote objects, (3) an infinite set \( M \) of method-names, and (4) a set \( F := \{ \text{ins, del, mod} \} \) of function symbols of arity one denoting certain update types. Here \( \text{ins/del/mod} \) stand for insert/delete/modify, respectively. Methods are functions to express properties of objects. The result of a method-application either is a value, or is an OID which denotes an object to describe a relationship between objects. For formal simplicity, we do not introduce types for values - we consider values as specific OIDs in \( O \).

To give a first example, in the following expression a method \( \text{salary} \) is applied on an object with OID \( \text{henry} \) and gives as result (the OID) 250:

\[
\text{henry.salary} + 250.
\]

Now we will introduce terms, atoms and rules. As usual, when one of these does not contain a variable, it will be called ground. The basic constructs of our language are object-id-terms and version-id-terms. An object-id-term either is a variable or an OID. To each object there may exist several versions. To be able to reference the different version we introduce version-id-terms. A version-id-term is defined as follows: (1) any object-id-term is also a version-id-term; (2) let \( \alpha \) be a version-id-term, then \( \alpha(V) \) with \( \alpha \in F \) is a version-id-term. The set of all ground version-id-terms

\footnote{On the result-position of a method only object-id-terms will be allowed, not version-id-terms. We choose this way because versions are only introduced for the purpose of the update-process; a relationship is considered to be a more stable concept in comparison to the concept of versions in our approach.}
An atom in our language either is a usual arithmetic built-in predicate \( (\ll, \gg, =, \text{etc.}) \) or a version-term or an update-term. We consider update- and version-terms, because it is important for our approach to distinguish between (1) whether a certain update is applied on a version to create a new version with different properties, or (2) whether a version which has been created by the application of a certain update has a certain property. For the former we introduce update-terms, for the latter version-terms.

Let \( m \) be a name of a method, \( V \) a version-id-term, and \( A_1, \ldots, A_k, R \) object-id-terms. Consider \( \@ \) to be an indicator for method arguments; it is omitted if there are no arguments. A version-term is any expression of the form \( V.m@A_1, \ldots, A_k \rightarrow R \), where \( k \geq 0 \).

A set of ground version-terms is called an object-base. An expression \( m@A_1, \ldots, A_k \rightarrow R \) is also called a method-application. The state of a version \( \text{w.r.t.} \) a certain object-base is given by the set of all ground method-applications, which can be derived from its version-terms in the respective object-base.

Update-terms are the means to express changes of the states of the versions. Let \( m \) be a name of a method, \( V \) a version-id-term, and \( A_1, \ldots, A_k, R, R' \) object-id-terms. An update-term now is any expression of one of the following: \( \text{ins}[V].m@A_1, \ldots, A_k \rightarrow R \), \( \text{del}[V].m@A_1, \ldots, A_k \rightarrow R \), or \( \text{mod}[V].m@A_1, \ldots, A_k \rightarrow (R \rightarrow R') \), where \( k \geq 0 \). Each of these updates expresses a transition from the state of a version \( V \) to the state of a version \( \alpha(V) \), where \( \alpha \in \mathcal{F} \). Syntactically, updates are indicated by the braces \( \{, \} \). Note, that these braces are replaced by \( \langle, \rangle \) when referring to the version being the result of the state transition. In case of an insert, the state of version \( \text{ins}(V) \) contains a new method-application not contained in the state of version \( V \). In case of a delete, the state of version \( \text{del}(V) \) contains a method-application, which is no longer contained in the state of version \( V \), and, finally, in case of a modify, both states of the versions \( \text{mod}(V) \) and \( V \) contain a method-application \( \text{w.r.t.} \) the same method and the same arguments, however the results are different.

For example the version-term 
\[
\text{mod}(\text{henry}) \text{ salary} \rightarrow 275
\]
states that the method \( \text{salary} \) applied to the version \( \text{mod}(\text{henry}) \) of object \( \text{henry} \) yields the result \( 275 \). Here \( \text{mod}(\text{henry}) \) is a VID; \( \text{henry} \) and \( 250 \) are OIDs. We consider \( \text{mod}(\text{henry}) \) to be the version of \( \text{henry} \) after an update of type \( \text{modify} \) has been applied to \( \text{henry} \). On the other hand, the update-term 
\[
\text{mod}(\text{henry}) \text{ salary} \rightarrow (250 \rightarrow 275)
\]
defines an update of type \( \text{modify} \) changing the result of \( \text{salary} \) applied to \( \text{henry} \) from \( 250 \) to \( 275 \). The new value will hold in the state of \( \text{mod}(\text{henry}) \).

An update-rule is written as
\[
H \iff B_1 \land \ldots \land B_k, \quad k \geq 0,
\]
where \( H \) is an update-term called the head of the rule, and \( B_1, \ldots, B_k \) are positive or negated atoms forming the rule’s body. \( H \) and the \( B_i \)’s are also called literals. If \( k = 0 \), then the rule is called an update-fact. Rules are considered to be \( V \)-quantified; the domain of quantification is the set \( \mathcal{O} \), i.e. the set of all OIDs. Let \( R \) be an update-rule and let \( \tau \) be an update-rule which is derived from \( R \) by replacing variables by OIDs. We call \( \tau \) a ground instance of \( R \). We require that rules are safe (cf. [Ull88]). A set of update-rules forms an update-program. The evaluation of an update-program is called update-process. From now on when talking about “rules”, “programs” or “processes”, we always mean “update-rules”, “update-program” or “update-processes”, respectively.

As a first example, demonstrating the power of our language, consider the following rule:
\[
\text{mod}[E].\text{sal} \rightarrow (S \rightarrow S') \iff
\ E.\text{isa} \rightarrow \text{empl} \land
\ E.\text{sal} \rightarrow S \land S' = S + 1.1
\]
To every employee a 10% salary raise has to be performed. It is worthwhile noticing that this intuitive version of the \( \text{salary-update} \) terminates, when evaluated bottom-up. In the above example each employee gets his salary raised exactly once (as intended), because the rule only applies to “initial” (i.e. non-updated) employees. (Remember, that a variable can only be instantiated by a OID, not VID.) Thus versions help to avoid non-terminating update-loops.
In the following we will always consider a scenario in which a certain update-program \( P \) is executed on a given object-base \( \text{ob} \). Note, that in this framework we do not consider derived objects, i.e. objects, for which a method is defined by a rule, which is not an update-rule; our intention is to study updates of base definitions only. However, these updates are defined by rules. Further note, that we do not introduce classes, because we are in the current paper not interested in the interaction between updates and types, respectively, inheritance.

The language introduced so far can be considered as a variant of stratified Datalog: methods correspond to predicates. Methods are mappings. Whenever an object-base contains several method-applications for a certain object (-version) \( v \), all having the same method name \( m \) and the same arguments \( a_1, \ldots, a_k \), we consider the method \( m \) to be set-valued. Proceeding this way we do not have to consider consistency questions w.r.t. functionality of methods; moreover, we have a simple set-concept in our language without any additional effort. (In fact, it corresponds to the set semantics introduced in [CW89,KW89].) Further it is worth to note, that our usage of function symbols does not enforce termination problems during bottom-up evaluation, because we quantify over the set of all OIDs \( O \), only. More precisely, for safe rules only a finite number of new versions can be derived during evaluation. Thus we do not enter the computationally more difficult world of Datalog with function symbols [UI188].

2.2 General Idea

We conceive an update-program as a mapping from an (old) object-base into a (new) object-base; update-programs are evaluated bottom-up. Our update-approach bases on the idea of object-versions at different time-steps, where the first version of an object (denoted by an OID) is the one found in the current to-be-updated object-base. Updating an object is done by carrying-out on it several groups of basic updates of the same type (insert, delete or modify). Each group is implemented by one or several update-rules. Realizing one such group “transforms” an object-version into the next (further updated) version of the respective object. Conceptually this “transformation” is understood as follows: consider version \( v \) with a certain state. Further assume that a group of updates of some type \( \alpha \in \{ \text{ins, del, mod} \} \) are to be performed on \( v \). Before performing the updates, a version \( \alpha(v) \) is created as a “copy” of \( v \), i.e. all method-applications of \( v \) are taken to hold (by default) for \( \alpha(v) \). Now the updates of type \( \alpha \) defined on version \( v \) are performed by changing the default method-applications of \( \alpha(v) \) accordingly. After all updates have been performed, \( \alpha(v) \) is the \( \alpha \)-updated version of \( v \). The “last version” of an object’s update-process represents the final updated object. Moreover, during an evaluation of an update-program all versions created during that evaluation can be used to derive the desired method values.

Assume we want to update an object-base \( \text{ob} \) yielding a new object-base \( \text{ob}' \) using an update-program \( P \). Let us focus on one object in \( \text{ob} \), denoted by its OID \( o \). Assume that the update-rules in \( P \) define (and perform) some modify-updates on the not-yet-updated object \( o \), followed by some delete-updates based on the “modified version of \( o \)”, concluding with some insert-updates following the delete-updates. Consequently we here have 3 groups of basic updates of the same type. At the time before evaluation of \( P \) has started, the object is denoted by \( o \). After the modify-updates, it is denoted by \( \text{mod}(o) \); here from the OID \( o \) we have derived by the respective modify a VID \( \text{mod}(o) \). Conceptually, \( \text{mod}(o) \) can be read as “the denotation of the version of object \( o \), after updates of type modify have been performed on \( o \)”, which we consider tantamount to saying, that “the updated object-version is referenced by \( \text{mod}(o) \)”. Thus VIDs have temporal characteristics. Performing delete-updates on the version \( \text{mod}(o) \), results in a new version denoted by the VID \( \text{del}(\text{mod}(o)) \), which again can be read as “the denotation of the version of object \( o \), after updates of type modify, followed by updates of type delete, have been performed on \( o \)”. In analogy, performing the insert-updates yields the version \( \text{ins}(\text{del}(\text{mod}(o))) \), which — if no further updates follow — is taken over into the new object-base \( \text{ob}' \) (where the object then will be denoted by \( o \) again). The general case of \( k \) consecutive groups of basic updates (of types \( \alpha_1, \ldots, \alpha_k \) resp.) performed on an object \( o \), is illustrated in figure 1.

Review the salary-update example in Section 2.1. Talking in the jargon of versions we have the follow-
2.3 Illustrative Examples

Assume an enterprise-object-base holding information about employees and let a first intended update be as follows: "Each employee gets a 10% salary-raise and those in a managerial position an extra $200. Afterwards all those employees are fired, who make more than any of their superiors, and finally those of the remaining ones, who make more than $4500, are grouped into a class called hpe (high-paid-employees)." The following update program realizes the update:

$$\text{mod}(E).\text{sal}(S \cdot S')$$  \hspace{1cm} (rule1)

$$E.\text{isa} \rightarrow \text{empl}/\text{pos} \rightarrow \text{mgr}/\text{sal} \rightarrow S \wedge$$
$$S' = (S \cdot 1.1) + 200$$

$$\text{mod}(E).\text{sal}(S \cdot S')$$  \hspace{1cm} (rule2)

$$E.\text{isa} \rightarrow \text{empl}/\text{sal} \rightarrow S \wedge$$
$$E.\text{pos} \rightarrow \text{mgr} \wedge S' = (S \cdot 1.1)$$

$$\text{del}[\text{mod}(E)].*$$  \hspace{1cm} (rule3)

$$\text{mod}(E).\text{isa} \rightarrow \text{empl}/\text{boss} \rightarrow B/\text{sal} \rightarrow SE \wedge$$

$$\text{mod}(B).\text{isa} \rightarrow \text{empl}/\text{sal} \rightarrow SB \wedge SE > SB$$

$$\text{ins}[\text{mod}(E)].\text{isa} \rightarrow \text{hpe} \iff$$  \hspace{1cm} (rule4)

$$\text{mod}(E).\text{isa} \rightarrow \text{empl}/\text{sal} \rightarrow S \wedge$$
$$S > 4500 \wedge \neg \text{del}[\text{mod}(E)].\text{isa} \rightarrow \text{empl}$$

Note that a construct v.m₁ → r₁/m₂ → r₂/... is used as an obvious short notation for a conjunction of the respective method-applications w.r.t. version v; similarly, we write del[..]* to express the deletion of all method-applications of the respective version. With these explanations on hand let us explain the effect of the four update-rules, assuming a bottom-up evaluation. The first rule takes an employee in a managerial position (isa → empl/pos → mgr), who had not yet been updated (E) and initiates a modify of his salary method (mod(E).sal → (S → S')). The second rule modifies the salary of all employees who are no managers. Assume in our to-be-updated object-base a manager phil who makes $4000 and has no superior, and an employee bob who makes $4200 and phil being one of his superiors. Surely we expect that the update (as a whole) leaves phil in the class hpe with a salary of $4600 and bob fired (i.e. no more an employee). This is indeed the case (cf. figure 2). The first rule initiates a modify-update on phil resulting in a version mod(phil), which — compared to the version phil — has the salary method result modified to $4600. An analogous reasoning applies to bob together with the second rule. The third rule only deals with employees after a modify had been carried out on them (mod(...)),

Figure 1 An update of an object o, in general.
i.e. in our example only the object-versions \textit{mod}(\textit{phil}) and \textit{mod}(\textit{bob}) are considered. This rule performs a delete-update on \textit{mod}(\textit{bob}) yielding the object-version \textit{del}(\textit{mod}(\textit{bob})) with the method-applications deleted as specified in the rule-head. Note that the third rule does not apply to \textit{phil}, because in our example-object-base he has no superior. The last rule shows that in our approach update-terms are allowed to appear in rule-bodies. This rule fires, if a modified employee (\textit{mod}(\textit{E})) with salary greater $4500 exists and no delete-update, deleting his \textit{isa}-result \textit{empl}, had been performed on the \textit{mod}(\textit{E})-version. The rule applies to \textit{E} = \textit{phil} (but not to \textit{E} = \textit{bob}), initiating an insert-update of \textit{mod}(\textit{phil}), yielding the object-version \textit{ins}(\textit{mod}(\textit{phil})), for which \textit{isa} \texttt{\rightarrow} \textit{empl} and \textit{isa} \texttt{\rightarrow} \textit{hpe} hold.

The next example shows that our approach can also be used to perform some sort of "hypothetical reasoning", as the usage of versions-identities allows to revise "hypothetical" updates. In the example below we intend to determine if after a hypothetical salary-raise (non-linear) to all employees, the employee \textit{peter} would be the richest employee of the enterprise:

$$\textit{mod}(\textit{E}).\textit{sal} \rightarrow (S \rightarrow S') \iff \textit{E}.\textit{sal} \rightarrow S/\text{factor} \rightarrow F \land S' = S \times F$$ (rule1)

$$\textit{mod}(\textit{[mod}(\textit{E}))).(S' \rightarrow S) \iff \textit{mod}(\textit{E}).\textit{sal} \rightarrow S' \land \textit{E}.\textit{sal} \rightarrow S$$ (rule2)

$$\textit{ins}(\textit{[mod}(\textit{[mod}(\textit{peter}))))].\textit{richest} \rightarrow \text{no} \iff \textit{mod}(\textit{E}).\textit{sal} \rightarrow SE \land \textit{mod}(\textit{peter}).\textit{sal} \rightarrow SP \land SE > SP$$ (rule3)

$$\textit{ins}(\textit{[ins}(\textit{[mod}(\textit{[mod}(\textit{peter}))))]).\textit{richest} \rightarrow \text{yes} \iff -\textit{ins}(\textit{[mod}(\textit{[mod}(\textit{peter}))))].\textit{richest} \rightarrow \text{no}$$ (rule4)

Here the first two rules realize the hypothetical salary-raise by performing and revising it right away. For each employee \textit{e} the \textit{mod}(\textit{[mod}(\textit{e})))-version is identical to the \textit{e}-version and the \textit{mod}(\textit{e})-version contains the raised salary. The third and fourth rule determine - by using the version after the first modify - whether
peter would be the richest employee of the enterprise.3

The final example shows that also recursive rules can be used for updates. By the two rules the ancestors of some given persons are computed. Note, that in this example methods anc and parents are considered to be set-valued. The example is as follows:

\[
\text{ins}[X].\text{anc} \rightarrow P \iff \\
X.\text{isa} \rightarrow \text{person/parents} \rightarrow P
\]
\[
\text{ins}[X].\text{anc} \rightarrow P \IFF \\
\text{ins}(X).\text{isa} \rightarrow \text{person/anc} \rightarrow A \land \\
A.\text{isa} \rightarrow \text{person/parents} \rightarrow P
\]

2.4 Discussion and Comparison

The concept of object-versions integrates in a nice and easy-to-understand way procedurality into our rule update language. If, in our first example, bob would only gain $4100, then without imposing control by the structure of the VIDs, firing employees before raising salaries could have led to a different unintended updated object-base. In fact, there is a large consensus that "procedurality" or some kind of "control" is required for updates [Abi88] (update = logic + control). Not surprisingly, the introduction of control leads to an increase of computational power. In rule-based update-languages based on top-down reasoning, different control mechanisms are encountered: [Tom88,Dec90,KM90,MW87] use the implicit control strategies offered by different variants of resolution. The update language proposed by [NT89] provides in addition explicit control by allowing sequential-, conditional- and iterative- operators in rule-bodies.

A comprehensive study of various extensions of Datalog with fixpoint semantics can be found in [AV91]; deterministic and nondeterministic extensions are studied w.r.t. their expressive power and complexity. Connections to procedural languages are given which also exhibit many interesting forms of programmed control. A different way to control evaluation is pointed out in RDL1 [dMS88]: here explicit (user defined) control is achieved by adding so-called Production Compilation Networks to the rule-programs, which allow similar control patterns as Petri-Nets.

In Logres [CCCR+00] update-rules are grouped in modules, which have either inflationary or stratified semantics, and can be used to define updates of base and derived methods. By specifying orders on the execution of the modules, the user has a flexible, however "manual" means for control. An interesting approach for control is chosen in LOGO [LVVS90]: here updates are controlled by the inheritance mechanism of the language. However updates cannot be defined by rules; instead again in a "manual" way new rules have to be introduced into the isa-hierarchy to achieve the desired effects.

Our approach will provide different types of control: in addition to a rule-ordering entailed by stratified negation, an implicit control resulting from a "stratification by object-versions". We "move from version to version" by explicitly naming them: VIDs allow to refer to objects at different stages of their update-process. This version aspect gives our approach a greater functionality compared to having the whole update-process performed at the same "time-step", or breaking the process into fixed modules as it is done in Logres. There seems to be an interesting relationship to the internal event calculus in [Oli89]. Here different versions can be distinguished by certain time-points. However no notion of object is considered and our VIDs also contain information about the history of the updates. Finally, we allow update-terms in rule-heads as well as rule-bodies. In the rule-head an update-term explicitly initiates an update (as in all bottom-up approaches), while in the rule-body it requests that a certain update of a certain object-version has (or has not) already been performed.

Versioning in object-oriented databases is a well-established concept (the textbook [Kim91] contains many references to relevant work.) High sophisticated techniques have been proposed to organize the versions of a certain object. We are more restrictive in this aspect and will require, that the versions of an object must reflect a linear order, while usually a hierarchy is allowed. The motivation for this restriction is that we must choose for each object a version out of a possible set of versions to build the new object-base; requiring a linear order makes this simple. There exists an interesting relationship between our update approach

---

3 An appropriate stratification technique will be presented in section 4.
and schema evolution. The way we consider inserts and deletions would require changes of corresponding class-definitions in a strongly typed environment, because methods become undefined, respectively defined w.r.t. some objects according to the type of the update. The techniques proposed in [SZ87] seem to be a good starting point for an integration of our method into a more general environment.

3 An Immediate Consequence Operator

Let $P$ be a given program, and $ob$ an object-base. As we are interested in the bottom-up evaluation of $P$, we now introduce an operator $T_P$, which maps object-bases into object-bases. $T_P$ is an adaptation of the usual immediate consequence operator in deductive databases. Let $I$ be an object-base. Intuitively, $T_P(I)$ derives a new object-base $I'$, such that each element in $I'$ follows from an application of a rule in $P$ w.r.t. $I$.

The definition of $T_P$ needs some further prerequisites.

First we define truth of ground version- and update-terms w.r.t. an object-base $I$. Version-terms do not perform any updates, they simply refer to a certain object-version asking for a certain property. Update-terms behave differently, depending whether they occur in the head or the body of a rule. An update-term in a rule-head only then is true, if its effect has not already occurred before. For example, an insert of new information is only then allowed, if the to be inserted information does not already exist. In a rule-body, an update-term only then is true, if the stated version-transition really has occurred. For example, for an insert it is required, that the respective information did not hold w.r.t. the state of the version, on which the insert has been performed, but does hold w.r.t. the state of the version of the update-term. Similar holds for a modify-operation; however for delete-operations the situation is a bit more subtle, as we will explain next. In the sequel, by $m$ we mean a method denoted by $m$ applied to a sequence of $k \geq 0$ arguments, i.e. $m@a_1, ..., a_k$.

Consider an update-term $a[v].m \rightarrow r$. The difference between insert, respectively modify, and delete is, that in the former cases we can be sure, that there will exist a version $ins(v)$, respectively $mod(v)$ in $I'$. For a delete this is not necessarily the case, because by a delete we shrink the state of a version, such that by deleting the last method-application, also the information about existence of the version has been deleted. To avoid such loss of information we assume, that for each object $o$ in the given object base $ob$ there is defined a method $exists$ as follows: $o.exists \rightarrow o$. In addition we require, that for all programs $P$, this "system-method" $exists$ does not occur in any update-term. Proceeding this way we will achieve the desired effect, that we cannot lose all information about a version $del(v)$ of an object $o$; at least a note about its existence expressed by $del(v).exists \rightarrow o$ will survive.

1. **Version-Term**
   - A ground version-term $v.m \rightarrow r$ is true w.r.t. $I$ iff $v.m \rightarrow r \in I$.

2. **Update-Term in a Rule-Head**
   - A ground update-term $ins[v].m \rightarrow r$, which occurs in a rule-head, is true w.r.t. $I$ iff $v.m \rightarrow r \notin I$.
   - A ground update-term $del[v].m \rightarrow r$, which occurs in a rule-head, is true w.r.t. $I$ iff $v.m \rightarrow r \in I$.
   - A ground update-term $mod[v].m \rightarrow (r \sim r')$, which occurs in a rule-head, is true w.r.t. $I$ iff $v.m \rightarrow r \in I$.

3. **Update-Term in a Rule-Body**
   - A ground update-term $ins[v].m \rightarrow r$, which occurs in a rule-body, is true w.r.t. $I$ iff $v.m \rightarrow r \notin I$ and $ins[v].m \rightarrow r \in I$.
   - A ground update-term $del[v].m \rightarrow r$, which occurs in a rule-body, is true w.r.t. $I$ iff $v.m \rightarrow r \in I$ and $del(v).exists \rightarrow o \in I$ and $del(v).m \rightarrow r \notin I$ where $o$ is the object of which $del(v)$ is a version.
   - A ground update-term $mod[v].m \rightarrow (r \sim r')$, where $r \neq r'$, which occurs in a rule-body, is true w.r.t. $I$ iff $v.m \rightarrow r \in I$ and $mod(v).m \rightarrow r \notin I$ and $mod(v).m \rightarrow r' \in I$.
A ground update-term $\text{mod}[v].\overline{m} \rightarrow (r \sim r')$, where $r = r'$, which occurs in a rule-body, is true w.r.t. $I$ iff $v.\overline{m} \rightarrow r \in I$ and $\text{mod}(v).\overline{m} \rightarrow r \in I$.

Negation in rule-bodies is treated as follows. A negated ground version-term $\neg v.\overline{m} \rightarrow r$ is true w.r.t. $I$, if $v.\overline{m} \rightarrow r$ is not true w.r.t. $I$. Negation of update-terms in rule-bodies is defined analogously.

After having introduced all the prerequisites, the immediate consequence operator $T_p(I)$ now can be defined by the following 3-step procedure:

**Step 1**

Compute the set:

$$T^1_p(I) = \{ h \mid \text{there exists a ground instance of a rule in } P \text{ such that its head } h \text{ and every literal in its body is true w.r.t. } I \}$$

In this step we derive the set of updates, which have to be performed on $I$.

**Step 2**

Let $\alpha[v].\overline{m} \rightarrow r \in T^1_p(I)$, respectively, $\alpha[v].\overline{m} \rightarrow (r \sim r') \in T^1_p(I)$. Any such VID $\alpha(v)$ is called relevant; it is called active, if in addition $I$ already contains a method-application of $\alpha(v)$. Compute then the set:

$$T^2_p(I) = \{ \alpha(v).\overline{m} \rightarrow r \mid \alpha(v) \text{ is active and } \alpha(v).\overline{m} \rightarrow r \in I \} \cup$$

$$\{ \alpha(v).\overline{m} \rightarrow r \mid \alpha(v) \text{ is relevant, however not active and } v.\overline{m} \rightarrow r \in I \}$$

Now we have prepared, by copying from $I$, for each object, on which an update has to be performed, a state of a version on which the update can take place. Note, in case of an active VID, we can simply copy the state from $I$, while in case the VID is relevant, but not active, we create a new version by taking the method-applications of the previous version as default.\footnote{At this point it may be interesting to reflect on the well-known frame-problem. All knowledge true for an old version also has to be true for the new one, if it has not explicitly stated otherwise by the update. By copying old states only for the objects being updated (and not the whole object-base), we keep the unavoidable overhead low.}

**Step 3**

It remains to do the required updates. To this end, finally compute the result of applying $T_p$ on $I$:

$$T_p(I) =$$

$$\{ \text{ins}(v).\overline{m} \rightarrow r \mid \text{ins}[v].\overline{m} \rightarrow r \in T^2_p(I) \text{ or } \text{ins}(v).\overline{m} \rightarrow r \in T^2_p(I) \} \cup$$

$$\{ \text{del}(v).\overline{m} \rightarrow r \mid \text{del}(v).\overline{m} \rightarrow r \in T^2_p(I) \text{ and } \text{del}(v).\overline{m} \rightarrow r \notin T^2_p(I) \} \cup$$

$$\{ \text{mod}(v).\overline{m} \rightarrow r \mid \text{mod}(v).\overline{m} \rightarrow r \in T^2_p(I) \text{ and } \text{mod}[v].\overline{m} \rightarrow (r \sim r') \notin T^2_p(I) \} \cup$$

$$\{ \text{mod}(v).\overline{m} \rightarrow r' \mid \text{mod}[v].\overline{m} \rightarrow (r \sim r') \in T^2_p(I) \}$$

4 Bottom-Up Evaluation

Bottom-up evaluation is complicated by several reasons. First, we have nonmonotonicity because of negation in rule-bodies; second, another source of nonmonotonicity are delete- and modify-operations. Insert-operations do not impose problems here, because inserts correspond to the usual derivation of new (positive) facts. Finally, during application of the immediate consequence operator, a copy of a state of a version to get a basis for the state of a new version may occur. Once such a copy has occurred, the state being copied should not be changed further, because these changes will not be implemented in the new version's state. A solution to these problems can be achieved by a stratification of the rules in $P$. The aim of such a stratification is to partition the rules into so called strata; bottom-up evaluation then is done stratum by stratum. The results of the lower strata are the input to the respective next higher stratum. In case that for a given program $P$ there exists a stratification, after having processed all strata, a fixpoint of $P$ is reached. This follows in analogy to results for stratified Datalog [UL88].

For technical simplicity of the derivation of the required stratification, we replace in the given program $P$ each construct $a[V]$ by $a(V)$, $a \in \mathcal{F}$. First, we guarantee that once a state is copied, this state is not changed any further. This gives our first condition for stratification:

(a) If there exists a rule $r$ with a version-id-term $a(V)$
in its head, then each rule, which has a version-id-term \( V' \) in its head such that \( V \) and \( V' \) unify, is in a lower stratum than \( r \).

Consider the first example stated in Section 2.3. The following stratification fulfills condition (a):

\[
\{ \text{rule1, rule2} \}, \{ \text{rule3, rule4} \}.
\]

The condition for stratification with respect to negation can be adapted from [UllSS]. However, in our framework the role of predicate names in Datalog now has to be taken by version-id-terms. The resulting conditions for stratification can be stated as follows:

(b) If there exists a rule \( r \) with a version-id-term \( V \) of a not negated atom in the body, then each rule, which has a version-id-term in its head unifying with \( V \) is in a stratum which is at most as high as the stratum of \( r \).

(c) If there exists a rule \( r \) with a version-id-term \( V \) of a negated atom in the body, then each rule, which has a version-id-term in its head unifying with \( V \), is in a lower stratum than \( r \).

To continue our example, the following stratification fulfills conditions (a) - (c):

\[
\{ \text{rule1, rule2} \}, \{ \text{rule3} \}, \{ \text{rule4} \}.
\]

The remaining task now is to consider nonmonotonicity due to delete- and modify-operations. A further stratification is necessary because of the following reasons. Assume during bottom-up evaluation we have to delete method-applications of a version \( v \). (The case of modify is analogous.) Then, first a new version, say \( del(v) \), is created, whose method-applications are the same as for \( v \). On this version the delete operations will take place. This follows from our definition of the \( T_p \)-operator. Now assume, that the delete operations do not all take place during one application of \( T_p \). Thus, there is the possibility, that a method-application of \( del(v) \) will be used to infer some operations w.r.t. other objects, and this method-application will be deleted afterwards, as well. To avoid such counterintuitive behaviour we require, that rules which perform a delete or a modify are assigned to a lower stratum than those rules, which refer to versions on which the corresponding delete- or modify-actions take place:

(d) If there exists a rule \( r \) with a version-id-term \( del(V) \), respectively \( mod(V) \), of an atom in its body, then each rule, whose head contains a version-id-term \( del(V') \), respectively \( mod(V') \), such that \( V \) and \( V' \) unify, is in a lower stratum than \( r \).

In our example, no further partitioning of the rules is implied by condition (d).

Let \( P \) be a program and \( ob \) a respective object base. If \( P \) has a stratification such that (a) - (d) is fulfilled, then the bottom-up evaluation is realized by iterating the operator \( T_P \) stratum by stratum, starting from a given object-base \( ob \), in an analogous way as it is described in detail in [Ull88]. The result of this computation process is denoted by \( result(P) \). Note, as we are only considering safe rules, the iteration is guaranteed to terminate with respect to each stratum.

5 Building the New Object Base

Let \( P \) be a program, and \( ob \) the object base on which \( P \) is performed. Assume \( P \) is stratified and we have computed \( result(P) \). Even though during the computation a stratification has been observed, it is still possible, that \( result(P) \) contains versions, which make it impossible to derive the new updated object base \( ob' \). This is the case, if there exist two versions of the same object \( o \), with VIDs \( v, v' \), for which we cannot decide, which of the both is the one whose method-applications are to be copied into \( ob' \). For example, such a situation could occur, if \( P \) contains the rules:

\[
\text{mod}[o].m \rightarrow (a \sim b) \leftarrow \ldots \text{ a rulebody } \ldots \\
\text{del}[o].m \rightarrow a \leftarrow \ldots \text{ another rulebody } \
\]

and both rules fire during the evaluation of \( P \). In general, it is undecidable to predict whether such a situation may occur during evaluation. To exclude such programs, for the purposes of the current paper, we believe that a runtime check during the computation of \( result(P) \) is appropriate, because its realization seems to be not expensive.

We call \( result(P) \) \textit{version-linear}, if for any two VIDs \( v, v' \) of the same object \( o \) it holds, that either \( v \) is a subterm of \( v' \), or vice versa. For an object \( o \), that ver-
sion of $o$ is called the final version of $o$, whose VID contains all VIDs of the other versions of $o$ as a subterm. Version-linearity can be easily checked during evaluation: At any point of time, keep the VID of the most recent version of each object and check whether the VID of any new version of the same object contains the previous VID as subterm.

Finally, if $\text{result}(P)$ is version-linear, the updated object base $ob'$ is derived from $\text{result}(P)$ by copying for each object $o \in ob'$ the method-applications of its final version. Note, that it may be the case that for an object all method-applications are deleted in its final version, i.e. the only method defined for this version is the method $\text{exists}$. In this case no information about such an object will be present in $ob'$.

6 Conclusion

The primary intention of the current paper is to present a technique for defining updates using rules based on object-versions. To keep the framework simple, we restricted our language more than necessary. More expressive power can be gained by allowing to quantify over VIDs in addition to OIDs. However, such an extension must be done carefully not to destroy the termination properties of the evaluation process. Our investigations can be continued in several directions. First, it seems to be worth to try to develop stratification or related criteria which allow to accept a broader class of programs for evaluation. Also, alternatives to version-linearity may be interesting. Second, we did not consider derived objects. We do not see any principal problems to generalize our approach in this direction. Finally, our version-based approach has temporal characteristics. The investigation of the relationship to temporal logics seems to be an interesting field for further research.

Acknowledgement

We would like to thank the referees for their helpful comments and for pointing out many relationships to other work.

References


