Dagstuhl, Germany, 1996

Towards a Logical Semantics for Referential Actions in SQL

Bertram Ludäscher* Wolfgang May* Joachim Reinert+

*Institut für Informatik, Universität Freiburg, {ludaesch,may}@informatik.uni-freiburg.de
+Fachbereich Informatik, Universität Kaiserslautern, jreinert@informatik.uni-kl.de

Abstract

We investigate a logical semantics which unambiguously specifies the meaning of SQL-like referential actions of the form `ON DELETE CASCADE` and `ON DELETE RESTRICT`. The semantics is given by a translation of referential actions into logical rules. The proposed semantics is less restrictive than the standard SQL semantics, yet preserves all referential integrity constraints. First, a preliminary set of rules is introduced which rejects a set of user requests if a single request is rejected. Subsequently, a refined translation is presented using *Statelog* [LHL95], a state-oriented Datalog extension which allows to define active and deductive rules within a unified framework. We show that our semantics yields the maximal admissible subset of a given set of user requests. Apart from the Statelog formalization, a three-valued formalization based on the well-founded semantics and an equivalent game-theoretic specification are presented, which give further insight into the problem of ambiguity of triggers.

1 Introduction

The concept of referential integrity has been present in the relational model from the beginning [Cod70]. Basically defined to guarantee the existence of referenced objects, it was refined by Date [Dat81] to a more active concept, ie the possibility to descriptively define reactions in order to compensate violations of referential integrity by so-called *referential actions*. Thus, referential actions are used to *automatically enforce* integrity. This task is more involved than integrity *checking*: e.g., it is well-known that all common constraints in the relational model (functional, join, multivalued and inclusion dependencies) can be expressed by first-order formulas, which in turn can be defined as deductive rules. A simplistic way to enforce integrity is to let the user define all updates to the database, check whether the new database is consistent, and abort the update if a constraint is violated. In order to relieve the user from the burden of defining every induced update which arises from some given user request wrt. referential integrity constraints, referential actions have been proposed. These ideas have been included in the SQL2 and SQL3 standards [JTC92, JTC94]. Unfortunately, even those restricted versions of "active rules" may lead (in a straightforward implementation) to some indeterminism caused by ambiguities during the evaluation of user requests. Clearly, this is undesired and therefore not allowed in the SQL standards.

In this paper, we propose a step towards a logical semantics for referential actions by specifying these actions as a *logic program* P. The main benefits of this approach are:

- Referential actions are precisely axiomatized by the logical semantics of P, thereby leaving no freedom of interpretation, or doubt about the meaning of a set of referential actions. In particular, ambiguities due to unspecified behavior of the operational semantics are avoided.

- Formal verification techniques become applicable, e.g. to prove that a set of referential actions *guarantees* the satisfaction of all referential integrity constraints for all instances D of the database.

- The rules of P can be executed using well-known evaluation techniques developed for deductive databases. Thus, an operational semantics for the execution of referential actions is obtained as a "by-product" of the logical specification.

The paper is structured as follows. In Section 2, the basics of referential integrity and referential actions in SQL are briefly reviewed and an example illustrating the problem of ambiguity is presented. In Section 3, we propose a logic-based specification of referential actions which provides a simple method of enforcing referential integrity. In Section 4, a more sophisticated algorithm is introduced, which determines the maximal set of user delete requests which can be executed without violating any referential integrity constraint. In Section 4.1, this algorithm is formalized in Statelog. Two alternative characterizations of the algorithm using well-founded Datalog and a game-theoretic approach are given in Sections 4.2 and 4.3 yielding additional insight into the properties of the algorithm. Section 5 contains concluding remarks; proofs are included in Appendix A.

2 Referential Integrity

Notation and Preliminaries. In order to define the concept of referential integrity, we introduce some notation. Let R be a relation name. W.l.o.g., we assume that an order (e.g. lexicographic) is given on the set A of attributes of R. Therefore, A can be written as a vector $A = (A_1, \dots, A_k)$ of attributes. Then, $R(A_1, \dots, A_k)$ denotes the relation schema of R. We further assume that all attributes range over the same underlying domain.[1] Often, some attributes of A are distinguished, especially those which form a key. For notational convenience, these distinguished attributes are grouped into a vector

$$\vec{A} = (A_{i_1}, \dots, A_{i_d}) \quad .$$

All remaining attributes are denoted as

$$\bar{A} = (A_{j_1}, \dots, A_{j_r}) \quad .$$

Since we will use first-order logic notation, $R(A_1, \dots, A_k)$ is overloaded and also denotes a logic *atom*, where R is the relation name and $A_1, \dots, A_k$ are variables for the (domain) values of the corresponding attributes.

Referential Integrity Constraints. Let $R_C(X_1, \dots, X_n)$ and $R_P(Y_1, \dots, Y_m)$ be relation schemas, $\vec{X} = (X_{i_1}, \dots, X_{i_k})$ and $\vec{Y} = (Y_{j_1}, \dots, Y_{j_k})$ be two vectors of k distinct attributes of R_C and R_P, respectively. A *referential integrity constraint* (*ric*) is an expression of the form

$$R_C.\vec{X} \rightarrow R_P.\vec{Y} \quad .$$

$\vec{X}$ is called a *foreign key* of the *child relation* R_C; it refers to the (candidate or primary) key $\vec{Y}$ of the *parent relation* R_P.

A *ric* $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ is *satisfied* by a given database D, if for every value of the foreign key $\vec{X}$ of a tuple in R_C, there exists a tuple with key $\vec{Y}$ in R_P such that $\vec{X} = \vec{Y}$ [2]. This is denoted as $D \models \varphi_{ric}$ with the first-order sentence

$$\forall \vec{X}, \bar{X} \, (R_C(\vec{X}, \bar{X}) \rightarrow \exists \vec{Y}, \bar{Y} \, (\vec{Y} = \vec{X} \wedge R_P(\vec{Y}, \bar{Y}))) \quad . \qquad (\varphi_{ric})$$

A *ric* is *violated* by D, if it is not satisfied by D.[3]

[1]The extension to the "typed version" with attributes ranging over different domains is straightforward.

[2]Here, the "overloaded meaning" as explained above is used, ie $\vec{X} = \vec{Y}$ denotes equality of *values* of the corresponding attributes (and not of the attribute *names*).

[3]If null values are allowed in foreign keys, *ric*'s should not be violated by such "null pointers". This can be achieved by the following modification of (φ_{ric}): $\forall \vec{X}, \bar{X} \, (R_C(\vec{X}, \bar{X}) \wedge \texttt{null} \notin \vec{X} \rightarrow \exists \vec{Y}, \bar{Y} \, (\vec{Y} = \vec{X} \wedge R_P(\vec{Y}, \bar{Y})))$.

Referential Actions in SQL. There are three basic manipulation operations which potentially may violate a *ric*, ie `insert into`, `update`, and `delete from` one of the relations R_P and R_C, respectively. It is easy to see from the logical implication in φ_{ric} above that `insert into` R_P and `delete from` R_C cannot introduce a violation. Furthermore, the operations `insert into` R_C and `update` R_C on the child are forbidden in SQL (and immediately backed out) if these would result in a violation. Therefore, only the two operations `update` R_P and `delete from` R_P have to be handled by referential actions.

In SQL, referential actions are specified in the declaration of the child relation. When the user issues an update request on the current state of the database D (which is assumed to be consistent), these referential actions ensure that all referential integrity constraints remain satisfied in the new database state D'. A referential action for the referential integrity constraint $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ is specified in SQL as follows:

```
{CREATE | ALTER} TABLE R_C
    ...
    FOREIGN KEY X REFERENCES R_P Y
    [ON UPDATE {CASCADE | RESTRICT | SET NULL | SET DEFAULT | NO ACTION}]
    [ON DELETE {CASCADE | RESTRICT | SET NULL | SET DEFAULT | NO ACTION}]
    ...
```

The Problem of Ambiguity. It is common to implement integrity maintenance using an independent trigger or ECA-rule for each integrity constraint (see e.g. [Day88, Esw76]). Such rules are defined like "ON delete of R_P DO delete R_C" and are executed in a recognize-act cycle [For81]. If the semantics of these triggers is only given by an informal description, some indeterminism with respect to the outcome of a user operation may occur. This is illustrated by the following example [Rei96]:

Example 1 Consider the database with referential actions as depicted in Figure 1. For this example, assume that *all dotted parts are empty.* Let $\triangleright$del:$R_A(a)$ be a user request to delete the tuple (a) from relation R_A.[4] Depending on the order of execution of referential actions, one of two different final states may be reached:

(1) If execution follows the path $R_A \rightsquigarrow R_C \rightsquigarrow R_D$, the tuple $R_C(a,c)$ cannot be deleted: Since $R_D(a,b,c)$ references $R_C(a,c)$, the referential action for R_D restricts the deletions of $R_C(a,c)$. This in turn also blocks the deletion of $R_A(a)$. Consequently, the user request $\triangleright$del:$R_A(a)$ is rejected, and the database state remains unchanged, ie $D' = D$.

(2) If execution follows the path $R_A \rightsquigarrow R_B \rightsquigarrow R_D$, the tuple $R_B(a,b)$ and – as a consequence – $R_D(a,b,c)$ are requested for deletion. Hence, the trigger for $R_D.(X,Z) \rightarrow R_C.(X,Z)$ "assumes" that $R_D(a,b,c)$ is deleted, thus no referencing tuple exists in R_D. Thus, all deletions can be executed, resulting in the new database state $D' = \emptyset$.

If there are different possible final states of a database instance D (depending on the execution order of referential actions), D is called *ambiguous* wrt. the given referential actions. Given a set of referential actions, a database *schema* is *ambiguous*, if some instance D is ambiguous. As shown in [Rei96] it is in general undecidable, whether a database schema with referential actions is ambiguous. However note that, although the above schema is ambiguous, (2) may be preferable to (1), because (1) – which is the semantics of SQL – does *not* accomplish the desired user request – indeed, nothing is done at all. In contrast, (2) leads to a new consistent state, in which the user request is accomplished. In the sequel, we present logical rules which avoid ambiguities caused by conflicting referential actions. This does not contradict the result

[4]The triangle "$\triangleright$" denotes *external* (ie, user-defined) requests.

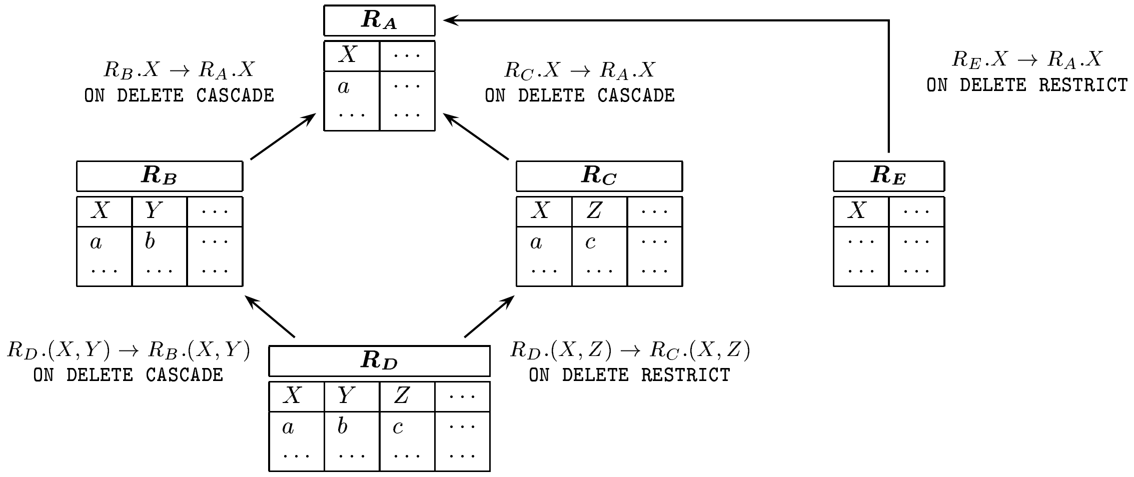

Figure 1: Database with Referential Actions

of [Rei96], since our semantics does not have to discriminate ambiguous from unambiguous schemas: instead, deletions are preferred to restrictions whenever possible. We confine ourselves to the specification of triggers of the form `ON DELETE CASCADE` and `ON DELETE RESTRICT`; a comprehensive scheme covering all SQL-triggers is beyond the scope of this paper.

3 Referential Actions as Logic Rules

We first specify the semantics of referential actions as a stratified Datalog program which can also serve as the implementation of a naive algorithm executing user requests in an all-or-nothing style. The given rules provide the basis for further refinements in subsequent sections. Let $U_{\triangleright} = \{\triangleright\mathsf{del}{:}R_1(\bar{x}_1), \ldots, \triangleright\mathsf{del}{:}R_n(\bar{x}_n)\}$ be a set of *user delete requests* which are passed to the database system.[5] From these external requests, *internal delete requests* $\mathsf{req\_del}{:}R(\bar{x})$ are derived:

$$\mathsf{req\_del}{:}R(\bar{X}) \leftarrow\ \triangleright\mathsf{del}{:}R(\bar{X}). \tag{I}$$

The referential actions are specified as follows:

- $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ `ON DELETE CASCADE`: This trigger generates two logical rules: the first one propagates internal delete requests downwards from the parent to the child:

$$\mathsf{req\_del}{:}R_C(\vec{X},\bar{X}) \leftarrow\ \mathsf{req\_del}{:}R_P(\vec{Y},\bar{Y}), R_C(\vec{X},\bar{X}), \vec{X} = \vec{Y}. \tag{DC_1}$$

  Additionally, restrictions are propagated upwards, ie when the deletion of a child is restricted, the deletion of the referenced parent is also restricted (blocked):

$$\mathsf{blk\_del}{:}R_P(\vec{Y},\bar{Y}) \leftarrow\ R_P(\vec{Y},\bar{Y}), \mathsf{blk\_del}{:}R_C(\vec{X},\bar{X}), \vec{X} = \vec{Y}. \tag{DC_2}$$

- $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ `ON DELETE RESTRICT`: The deletion of a parent tuple is blocked, if there is a corresponding child tuple which is not requested for deletion:

$$\mathsf{blk\_del}{:}R_P(\vec{Y},\bar{Y}) \leftarrow\ R_P(\vec{Y},\bar{Y}), R_C(\vec{X},\bar{X}), \neg\mathsf{req\_del}{:}R_C(\vec{X},\bar{X}), \vec{X} = \vec{Y}. \tag{DR}$$

[5] R_i are (not necessarily distinct) base relations, $\bar{x}_i$ are tuples of constants from the underlying domain.

Note, that for a given set of referential actions, the logic program P generated by (DC_1), (DC_2) and (DR) is stratified, ie does not contain negative cyclic dependencies. The strata are given by

$$\{R, \mathsf{req\_del}{:}R\} \prec \{\mathsf{blk\_del}{:}R\}$$

for all base relations R. Therefore, P has a unique stratified model.

Note further, that this logic program solves the conflicts and ambiguities between `CASCADE` and `RESTRICT` actions (Example 1) by the following strategy: First, all requested deletions are cascaded without considering restrictions. Then, all restricted deletions are computed using the delete requests from the first step. This two-phase approach is the abstract formalization to the lengthy textual descriptions in the standard documents.[6]

Example 2 Consider again the database given in Figure 1, where all dotted parts are empty. Given the user request $\triangleright\mathsf{del}{:}R_A(a)$, the above program derives delete requests $\mathsf{req\_del}{:}R(\bar{x})$ for $R_A(a)$, $R_B(a,b)$, $R_C(a,c)$, and $R_D(a,b,c)$, but no blocked requests of the form $\mathsf{blk\_del}{:}R(\bar{x})$ (because $R_D(a,b,c)$ is requested for deletion before it gets a chance to block other requests). Hence all deletions are computed correctly.

Unfounded Deletions. However, the above rule set may give rise to *unfounded delete requests*: a triggered delete request is unfounded, if its triggering request is blocked, but the triggered request itself is not blocked:

Example 3 Assume that the tuple $R_E(a)$ is added to the database in Figure 1. The trigger $R_E.X \rightarrow R_A.X$ `ON DELETE RESTRICT` blocks the deletion of $R_A(a)$, but not the deletions of $R_B(a,b)$, $R_C(a,c)$ and $R_D(a,b,c)$ which then become unfounded!

This problem is avoided if a triggered request is executed only if its triggering request is executed itself. One way to guarantee this condition is to require that *all* delete requests are admissible, otherwise the transaction aborts:

$$\begin{array}{lcl} \mathsf{del}{:}R(\bar{X}) & \leftarrow & \mathsf{req\_del}{:}R(\bar{X}), \neg\mathsf{abort}. \\ \mathsf{abort} & \leftarrow & \triangleright\mathsf{del}{:}R(\bar{X}), \mathsf{blk\_del}{:}R(\bar{X}). \end{array} \qquad (DEL)$$

Here, $\mathsf{del}{:}R$ denotes the set of final deletions to be executed by the system. The rules (DEL) guarantee that the whole set of user delete requests is executed in an *all-or-nothing* style.

4 Refined Translation

Although the preliminary translation given above is less restrictive than the standard SQL semantics, it is still more restrictive than necessary:

Example 4 Consider the database as depicted in Figure 2 and assume the user requests $\{\triangleright\mathsf{del}{:}R_A(a), \triangleright\mathsf{del}{:}R_A(b)\}$ are given. Like in Example 3, $\triangleright\mathsf{del}{:}R_A(a)$ is not admissible since $R_E(a)$ blocks $\triangleright\mathsf{del}{:}R_A(a)$. However, the other request, $\triangleright\mathsf{del}{:}R_A(b)$, could be executed without violating any *ric* by deleting $R_A(b)$, $R_B(b,b)$, $R_C(b,c)$ and $R_D(b,b,c)$.

In the following, a more flexible strategy is developed which determines the maximal subset of admissible deletions of $U_\triangleright$ which does not violate any *ric* thereby relieving the user from trying all alternatives by himself. The basic idea of the refinement is to consider only those user delete requests which are not blocked in rule (I). However, this introduces an inherent negative cyclic dependency $\mathsf{req\_del} \overset{\neg}{\leftarrow} \mathsf{blk\_del} \overset{\neg}{\leftarrow} \mathsf{req\_del}$ resulting in a non-stratified logic program P_W. The properties of P_W will be further investigated in Section 4.2.

[6]In fact the standard SQL semantics is more restrictive than our proposal, since it does not allow the existence of *any* referencing tuple (even if it is marked for deletion). This more restrictive semantics the style of SQL can be modeled by the following rule (DR): $\mathsf{blk\_del}{:}R_P(\vec{Y}, \bar{Y}) \leftarrow R_P(\vec{Y}, \bar{Y}), R_C(\vec{X}, \bar{X}), \vec{X} = \vec{Y}$.

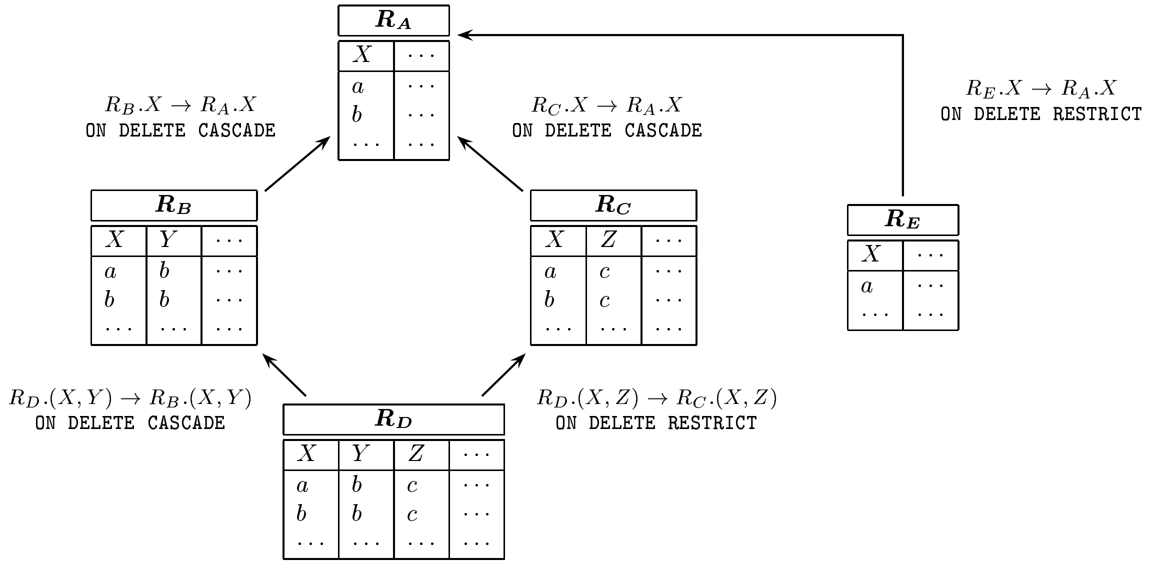

Figure 2: Extended Database (Example 4)

Informal Description of the Algorithm (cf. Figure 3): Initially, it is assumed that there are no blockings (ie, blk_del:$R(\bar{x})$ does not hold for any tuple $R(\bar{x})$). By cascading all user requests, all potential delete requests are computed. In the next step, all blockings are computed caused by tuples which are not reachable by cascaded deletions. At this point, the preliminary algorithm aborts if there is some delete request which is blocked. The refined algorithm analyzes the situation in order to abort as few user requests as possible: For all blocked requests, the triggering user request is also blocked by propagating blockings upwards the `ON DELETE CASCADE` chain to parent tuples. For the remaining unblocked user requests, the cascaded requests are recomputed. Thus some more tuples will remain in the database, which could block other requests. These steps are repeated until a fixpoint is reached.

4.1 State-Oriented Logic Formalization

The first translation from referential actions to logic rules given above resulted in a stratified Datalog program. However, the improved algorithm contains negative cyclic dependencies, since requested deletions and blocked deletions may depend negatively on each other. Therefore, a direct translation of the informally given algorithm into a logic program would result in an non-stratified program (cf. Section 4.2). The improved algorithm can easily be implemented in a state-oriented logic programming framework.

Statelog is a state-oriented extension to Datalog which allows to define active and deductive rules within a unified logical framework [LL94, LHL95]. Since in this language different states of the database can be accessed, Statelog is well-suited as a specification and implementation language for defining the behavior of referential actions.

In Statelog, different database states are accessed using *state terms* of the form $[S+k]$, where $S+k$ denotes the k-fold application of the unary function symbol "+1" to the *state variable* S. The domain of S is $\mathbb{N}_0$, ie computations in Statelog evolve over a linear state space. Statelog rules are of the form

$$[S+k_0]\ H(\bar{X}) \ \leftarrow\ [S+k_1]\ B_1(\bar{X}_1),\ldots,[S+k_n]\ B_n(\bar{X}_n) \quad ,$$

where the head $H(\bar{X})$ is an atom, $B_i(\bar{X}_i)$ are atoms or negated atoms, and $k_0 \geq k_i$, for all $i \in \{1,\ldots,n\}$. A rule is *local*, if $k_0 = k_i$, for all $i \in \{1,\ldots,n\}$. Thus, a Statelog program

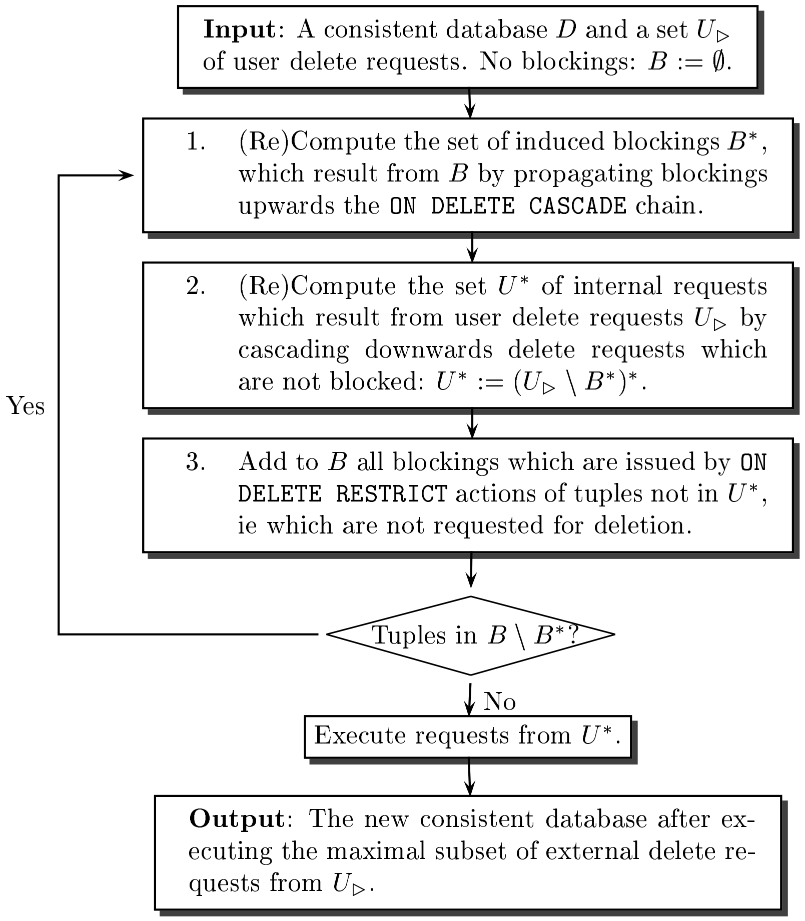
Input: A consistent database D and a set $U_\triangleright$ of user delete requests. No blockings: $B := \emptyset$.
1. (Re)Compute the set of induced blockings B^*, which result from B by propagating blockings upwards the ON DELETE CASCADE chain.
2. (Re)Compute the set U^* of internal requests which result from user delete requests $U_\triangleright$ by cascading downwards delete requests which are not blocked: $U^* := (U_\triangleright \setminus B^*)^*$.
3. Add to B all blockings which are issued by ON DELETE RESTRICT actions of tuples not in U^*, ie which are not requested for deletion.
Yes
Tuples in $B \setminus B^*$?
No
Execute requests from U^*.
Output: The new consistent database after executing the maximal subset of external delete requests from $U_\triangleright$.

Figure 3: Algorithm $\mathcal{A}$: Computing all admissible deletions

can be viewed as a syntactical variant of a logic program in which every predicate contains one additional distinguished argument for state terms. In particular, local rules refer only to the current state $[S + k]$ and not to the transition between different states. Thus local rules correspond to (stratified) Datalog rules which are applied locally in every state $[S + k]$.

Formalization in Statelog. The above algorithm is formalized in Statelog as follows:[7]
EDB relations R are propagated to subsequent states (modulo the set of final deletions $\mathsf{del}{:}R(\bar{X})$; see below) by frame rules:

$$[S{+}1]\ R(\bar{X}) \leftarrow\ [S]\ R(\bar{X}), \neg\mathsf{del}{:}R(\bar{X}).$$

User requests $\triangleright\mathsf{del}{:}R$ are propagated to the successor state as long as the computation is running:

$$[S{+}1]\ \triangleright\mathsf{del}{:}R(\bar{X}) \leftarrow\ [S]\ \triangleright\mathsf{del}{:}R(\bar{X}), \mathsf{running}.$$

From user delete requests $\triangleright\mathsf{del}{:}R$, internal requests $\mathsf{req\_del}{:}R$ are raised unless they are blocked:

$$[S]\ \mathsf{req\_del}{:}R(\bar{X}) \leftarrow\ [S]\ \triangleright\mathsf{del}{:}R(\bar{X}), \neg\mathsf{blk\_del}{:}R(\bar{X}). \qquad (I^S)$$

Referential actions are translated as follows:

- $R_C.\vec{X} \to R_P.\vec{Y}$ `ON DELETE CASCADE`: The rules (DC_1) and (DC_2) above are simply extended by state terms $[S]$ and yield the following local rules:

[7]In literals referring to the same state, only the leftmost literal is prefixed with a state term.

$$[S]\ \mathsf{req\_del}{:}R_C(\vec{X},\bar{X}) \leftarrow\ [S]\ \mathsf{req\_del}{:}R_P(\vec{Y},\bar{Y}), R_C(\vec{X},\bar{X}), \vec{X}=\vec{Y}. \qquad (DC_1^S)$$

$$[S]\ \mathsf{blk\_del}{:}R_P(\vec{Y},\bar{Y}) \leftarrow\ [S]\ R_P(\vec{Y},\bar{Y}), \mathsf{blk\_del}{:}R_C(\vec{X},\bar{X}), \vec{X}=\vec{Y}. \qquad (DC_2^S)$$

- $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ `ON DELETE RESTRICT`: The new rule (DR^S) for `ON DELETE RESTRICT` contains the crux of the refined algorithm: In the *successor state* $[S{+}1]$ only those tuples cause blockings, which are not requested for deletion in the *current state* $[S]$. This corresponds to the iteration step in Figure 3 and avoids negative cyclic dependencies within a state.

$$\begin{aligned}[S{+}1]\ \mathsf{blk\_del}{:}R_P(\vec{Y},\bar{Y}) \leftarrow [S]\ & R_P(\vec{Y},\bar{Y}), R_C(\vec{X},\bar{X}), \\ & \neg\mathsf{req\_del}{:}R_C(\vec{X},\bar{X}), \vec{X}{=}\vec{Y}, \mathsf{running}.\end{aligned} \qquad (DR^S)$$

The whole process keeps running while there are new blockings:

$$\left.\begin{array}{l}[0]\ \mathsf{running}. \\ [S{+}1]\ \mathsf{running} \leftarrow [S{+}1]\ \mathsf{blk\_del}{:}R(X), [S]\ \neg\mathsf{blk\_del}{:}R(X).\end{array}\right. \qquad (R_1^S)$$

When the iteration terminates, the final set of delete requests is derived:

$$[S{+}1]\ \mathsf{del}{:}R(\bar{X}) \leftarrow\ [S{+}1]\ \neg\mathsf{running}, [S]\ \mathsf{running}, \mathsf{req\_del}{:}R(\bar{X}). \qquad (R_2^S)$$

After termination, the sets of performed and abandoned updates can be determined:

$$\left.\begin{array}{l}[S]\ \mathsf{committed\_del}{:}R(\bar{X}) \leftarrow [S]\ \neg\mathsf{running}, \triangleright\mathsf{del}{:}R(\bar{X}), \neg\mathsf{blk\_del}{:}R(\bar{X}). \\ [S]\ \mathsf{aborted\_del}{:}R(\bar{X}) \leftarrow [S]\ \neg\mathsf{running}, \triangleright\mathsf{del}{:}R(\bar{X}), \mathsf{blk\_del}{:}R(\bar{X}).\end{array}\right. \qquad (R_3^S)$$

In the following, we refer to this program as P_S.

P_S is *state-stratified*, which implies that it is locally stratified and has a unique *perfect model* [Prz88]. The notion of state-stratification takes into account the different "time-stamps" of relations:

Definition 1 The labeled dependency graph $\mathcal{G}(P)$ of a Statelog program P is defined as follows. Its vertices are the relation names occurring in P. For every rule

$$[S_0]\ H(\bar{X}_0) \leftarrow\ [S_1]\ B_1(\bar{X}_1), \ldots, [S_n]\ B_1(\bar{X}_n)\ .$$

of P, $\mathcal{G}(P)$ contains for every $i = 1, \ldots, n$

- a negative edge $A_i \overset{l_i,\neg}{\rightarrow} H$, if B_i is a negative literal $\neg A_i(\bar{X})$
- a positive edge $B_i \overset{l_i}{\rightarrow} H$ otherwise.

Here, the label $l_i := S_0 - S_i \geq 0$ is the "gap" between states; it may be omitted for $l = 0$.

A cycle of $\mathcal{G}(P)$ involving only edges with $l = 0$ is called a *local cycle*. A program P is called *state-stratified* if no local cycle of $\mathcal{G}(P)$ contains a negative edge. □

Figure 4 shows the dependency graph for P_S. The labels of edges have been depicted as follows: Solid lines represent local edges (marked with $l = 0$), dotted lines represent edges across state transitions (ie, labeled with $l = 1$). Note that only solid edges have to be considered for state-stratification.

From the dependency graph it is clear that the program implementing the algorithm is state-stratified. Thus, for every database D and every set $U_\triangleright$ of user delete requests, it has a perfect model $\mathcal{M}(P_S, D, U_\triangleright)$.

The dependency graph also mirrors the stages of the algorithm: The main relations involved in the computation (represented by solid-lined ovals) are R, $\triangleright\mathsf{del}{:}R$, $\mathsf{req\_del}{:}R$ and $\mathsf{blk\_del}{:}R$ for EDB relations R. The relations R, $\triangleright\mathsf{del}{:}R$ remain unchanged during the iteration. Since $\mathsf{req\_del}{:}R$ depends negatively on $\mathsf{blk\_del}{:}R$, the naturally given stratification $\{\mathsf{blk\_del}{:}R\} \prec \{\mathsf{req\_del}{:}R\}$ corresponds to the steps shown in Figure 3:

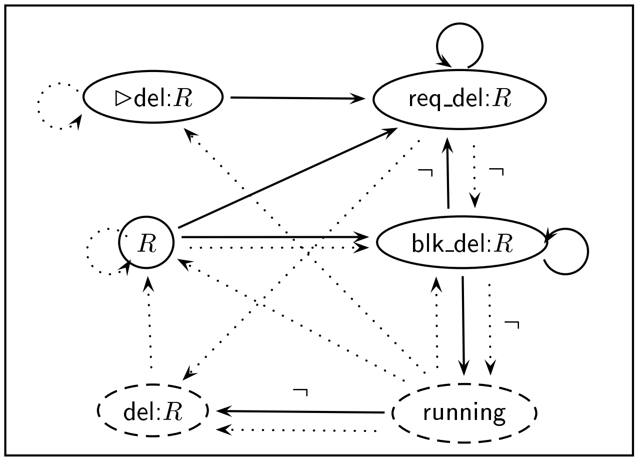

Figure 4: Dependency Graph

Every iteration of the algorithm starts with a set of blockings, which is given in the Statelog translation by (DR^S). According to the stratification, at first the induced blockings are computed (DC_2^S) also determining the blocked user delete requests. Then the remaining user delete requests issue internal delete requests (I^S) which are cascaded by (DC_1^S). From these, the resulting blockings for the next iteration are computed.

With the formal basis given by the Statelog program, the correctness of the algorithm can be proven:

Definition 2 Let D be a database, $U_\triangleright$ a set of user delete requests, and RA a set of referential actions of the form $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ `ON DELETE` { `CASCADE` | `RESTRICT` }. A set U^* of (internal) delete requests is called *admissible* if

1. every $R(\bar{x}) \in U^*$ is *founded* by some $\triangleright\mathsf{del}{:}R'(\bar{x}')$, ie there is a chain of references from $R(\bar{x})$ to $R'(\bar{x}')$ in D using `ON DELETE CASCADE` triggers from RA, and
2. all referential actions RA (and hence all *ric*'s) are satisfied in the new database $D' := D \backslash U^*$. □

Definition 3 Let $DEL^n(P_S, D, U_\triangleright)$ be the set of delete requests in state $[n]$, $DEL^n_\triangleright(P_S, D, U_\triangleright)$ is the subset of these which are given by the user, ie

$$\begin{array}{rcl} DEL^n(P_S, D, U_\triangleright) & := & \{\mathsf{del}{:}R(\bar{x}) \mid \mathcal{M}(P_S, D, U_\triangleright) \models [n]\,\mathsf{req\_del}{:}R(\bar{x})\} \\ DEL^n_\triangleright(P_S, D, U_\triangleright) & := & DEL^n(P_S, D, U_\triangleright) \cap \{\mathsf{del}{:}R(\bar{x}) \mid \triangleright\mathsf{del}{:}R(\bar{x}) \in U_\triangleright\} \quad . \end{array}$$

Finally, assuming that the algorithm terminates in a state $[n_{final}]$ (this is proven below), let

$$DEL(P_S, D, U_\triangleright) := DEL^{n_{final}}(P_S, D, U_\triangleright) \text{ and } DEL_\triangleright(P_S, D, U_\triangleright) := DEL^{n_{final}}_\triangleright(P_S, D, U_\triangleright) \;.$$ □

In the next theorem, arguments P_S, D, and $U_\triangleright$ of $\mathcal{M}$ and the DEL sets are omitted:

Theorem 1 (Correctness) *Given a database D, a set of ric's with corresponding referential actions, and a set of external delete requests $U_\triangleright$, the algorithm given by P_S determines the maximal set of admissible delete requests.*
More specifically, one can show (cf. Appendix A):

1. *In every state $[n]$, all internal delete requests are founded by some user request which is not blocked in $[n]$.*

2. *In every state $[n]$, no tuple is both blocked and requested for deletion, ie there is no n and $R(\bar{x})$ s.t. $\mathcal{M} \models [n]\,\textsf{\textit{req\_del}}{:}R(\bar{x}) \wedge \textsf{\textit{blk\_del}}{:}R(\bar{x})$.*
3. *Wrt. subsequent states, delete requests and blockings are nonincreasing and nondecreasing, respectively:*
$$\mathcal{M} \models [n{+}1]\,\textsf{\textit{req\_del}}{:}R(\bar{x}) \Rightarrow \mathcal{M} \models [n]\,\textsf{\textit{req\_del}}{:}R(\bar{x})$$
$$\mathcal{M} \models [n]\,\textsf{\textit{blk\_del}}{:}R(\bar{x}) \Rightarrow \mathcal{M} \models [n{+}1]\,\textsf{\textit{blk\_del}}{:}R(\bar{x}).$$
4. *In every state $[n]$, executing all internal delete requests of $[n]$ would not violate any* ***ON DELETE CASCADE*** *trigger.*
5. *In every state $[n]$, DEL^n is the set of internal deletions which have to be executed to accomplish the user requests $DEL^n_{\triangleright}$.*
6. *After termination, ie when $\mathcal{M} \models [n]\neg\textsf{\textit{running}} \wedge [n{-}1]\,\textsf{\textit{running}}$, executing all outstanding delete requests $\textsf{\textit{req\_del}}{:}R(\bar{x})$ would not violate any* ***ON DELETE RESTRICT*** *trigger.*
7. *In every step, all tuples $R(\bar{x})$ s.t. $\mathcal{M} \models [n]\,\textsf{\textit{blk\_del}}{:}R(\bar{x}) \vee \neg\textsf{\textit{req\_del}}{:}R(\bar{x})$ are not contained in any admissible set of deletions.*
8. *DEL is the maximal admissible set of delete requests, the subset $DEL_{\triangleright}$ is the maximal set of admissible user delete requests.*
9. *DEL (and thus $DEL_{\triangleright}$) is unique.*

Theorem 2 (Termination) *For every database D and every set $U_{\triangleright}$ of user delete requests, there is a unique final state $n_{final} \leq |U_{\triangleright}|+1$, ie for all $k < n_{final}$: $\mathcal{M}(P_S, D, U_{\triangleright}) \models [k]\,\textsf{\textit{running}}$, and for all $k \geq n_{final}$: $\mathcal{M}(P_S, D, U_{\triangleright}) \models [k]\neg\textsf{\textit{running}}$.*

Proof The algorithm stops in state $[n]$ if there are no new blockings compared to $[n{-}1]$. Let $n \geq 2$. Assume that there is some new blocking in $[n]$, ie $\mathcal{M} \models [n]\,\textsf{blk\_del}{:}R(\bar{x})$ and $\mathcal{M} \models [n{-}1]\,\neg\textsf{blk\_del}{:}R(\bar{x})$. Then there also has to be some $R'(\bar{x}')$ s.t. (i) $\mathcal{M} \models [n]\,\textsf{blk\_del}{:}R'(\bar{x}')$ and $\mathcal{M} \models [n{-}1]\,\neg\textsf{blk\_del}{:}R'(\bar{x}')$ and (ii) $\mathcal{M} \models [n]\,\textsf{blk\_del}{:}R'(\bar{x}')$ is derived by (DR^S). Thus, there is a tuple $R''(\bar{x}'')$ s.t. $\mathcal{M} \models [n{-}1]\,\neg\textsf{req\_del}{:}R''(\bar{x}'')$ and $\mathcal{M} \models [n{-}2]\,\textsf{req\_del}{:}R''(\bar{x}'')$. Furthermore, since $\mathcal{M} \models [n{-}2]\,\textsf{req\_del}{:}R''(\bar{x}'')$ has to be founded by some user delete request $\triangleright\textsf{del}{:}U(\bar{y})$ s.t. $\mathcal{M} \models [n{-}2]\,\neg\textsf{blk\_del}{:}U(\bar{y})$, this user delete request must be blocked in $[n{-}1]$. Hence, for each iteration, at least one user request is blocked which has not been blocked before. Since blk_del is nondecreasing and the algorithm terminates as soon as blk_del becomes stationary, there are at most $|U_{\triangleright}|+1$ iterations. ■

4.2 A Three-Valued Formalization

The presented Statelog formalization of algorithm $\mathcal{A}$ makes explicit use of state terms $[S{+}1]$ and $[S]$. This is the reason why it is possible to define updates within the Statelog language. On the other hand, it is desirable to have a "static" logical semantics which is defined without reference to different states. In this section, we show how such a semantics can be directly obtained from referential actions. However, due to the inherent negative cyclic dependecies between delete requests req_del and blockings blk_del, the resulting program will be non-stratified. The well-founded semantics [VGRS91], which is generally accepted as a declarative semantics for such programs, assigns a third truth value *undefined* to atoms whose truth value can neither be derived as true nor as false using a "well-founded" argumentation and the given logic rules. The Statelog formalization given in the previous section can be seen as a certain interpretation of this well-founded model where priority is given to deletions.

P_W**: A Direct Translation with Well-Founded Negation.** Recall the first direct translation of referential actions into logic rules from Section 3. Let P_W be the program consisting of rules (DC_1), (DC_2), (DR) and the modification of rule (I):

$$\mathsf{req\_del}{:}R(\bar{X}) \leftarrow\ \rhd\mathsf{del}{:}R(\bar{X}), \neg\mathsf{blk\_del}{:}R(\bar{X}). \qquad (I^W)$$

Due to the negative dependencies, the well-founded model $\mathcal{W}(P_W, D, U_\rhd)$ of P_W may contain atoms $\mathsf{blk\_del}{:}R(\bar{x})$ and $\mathsf{req\_del}{:}R(\bar{x})$ with the truth value *undefined*. The fact that the presented Statelog formalization is sceptic wrt. blockings and gives priority to deletions whenever possible is established by the following

Theorem 3

1. $\mathcal{M}(P_S, D, U_\rhd) \models [n_{final}]\, \mathit{req\_del}{:}R(\bar{x}) \ \Leftrightarrow\ \mathcal{W}(P_W, D, U_\rhd)(\mathit{req\_del}{:}R(\bar{x})) \in \{\mathit{true}, \mathit{undef}\}$.
2. $\mathcal{M}(P_S, D, U_\rhd) \models [n_{final}]\neg \mathit{blk\_del}{:}R(\bar{x}) \ \Leftrightarrow\ \mathcal{W}(P_W, D, U_\rhd)(\mathit{blk\_del}{:}R(\bar{x})) \in \{\mathit{false}, \mathit{undef}\}$.

Therefore, whenever the well-founded model yields the truth-value *true* or *undefined* for a delete request $\mathsf{req\_del}{:}R(\bar{x})$, the tuple $R(\bar{x})$ is deleted by P_S. On the other hand, *undefined* blockings $\mathsf{blk\_del}{:}R(\bar{x})$ in the well-founded model are ignored and regarded as false by P_S.

Example 5 The "diamond" in Figure 1 results in a "dispute" between blockings and deletions: Given the user request $\rhd\mathsf{del}{:}R_A(a)$, the delete requests $\mathsf{req\_del}$ for $R_A(a)$, $R_B(a,b)$, $R_C(a,c)$, $R_D(a,b,c)$, as well as the blockings $\mathsf{blk\_del}$ for $R_A(a)$, $R_C(a,c)$ will be *undefined* in the well-founded model. This can be regarded as an ambiguity which is resolved in the presented algorithm $\mathcal{A}$ by giving priority to delete requests. Thus, according to Theorem 3, the above delete requests are interpreted as true, while the blockings are interpreted as false.

Looking at the database in Figure 2 with the user requests $\{\rhd\mathsf{del}{:}R_A(a), \rhd\mathsf{del}{:}R_A(b)\}$, we find that the blockings for $R_A(a)$ and $R_C(a,c)$ are *true* in the well-founded model (due to the referencing tuple $R_E(a)$) and thus $R_A(a)$, $R_C(a,c)$ cannot be deleted. In contrast, the tuples $R_A(b), R_B(b,b), R_C(b,c)$ and $R_D(b,b,c)$ can be deleted, since there are undefined delete requests for them in the well-founded model, and – like above – priority is given to deletions.

4.3 Playing Games

In the following, we develop a very intuitive game-theoretic presentation of P_W which yields an alternative and elegant specification of referential actions. As indicated in the previous example, deletions and blockings can be viewed as a dispute whether a certain tuple can be deleted or has to remain in the database.

More precisely, the dispute is a *game* between two players I (the *"Deleter"*) and II (the *"Spoiler"*).[8] The game is played in *rounds* with a pebble which can be placed on any tuple of the given database D and on any user request in $U_\rhd$. Thus, $D \cup U_\rhd$ are the *positions* of the game. Each round consists of two *moves*.

Initially, the pebble is on an arbitrary tuple $R(\bar{x})$ in D. Then I starts to play and tries to prove that $R(\bar{x})$ can be deleted. He does so by moving the pebble from $R(\bar{x})$ to some user request $\rhd\mathsf{del}{:}R'(\bar{x}')$ such that there is a finite sequence of `ON DELETE CASCADE` triggers leading from $\rhd\mathsf{del}{:}R'(\bar{x}')$ to $R(\bar{x})$ in D (thus there is a chain of references from $R(\bar{x})$ to $\rhd\mathsf{del}{:}R'(\bar{x}')$). Player II tries to disprove the argument of I by moving the pebble to some tuple $R''(\bar{x}'')$ which cannot be deleted due to some `ON DELETE RESTRICT` trigger and a finite sequence of references using `ON DELETE CASCADE` triggers which will eventually also restrict the user request $\rhd\mathsf{del}{:}R'(\bar{x}')$. If a player cannot move, he has lost the game. In this case the opponent has successfully proved his claim and won the game. The following moves in the game are possible:

Player I can move from $R(\bar{x})$ to $\rhd\mathsf{del}{:}R'(\bar{x}')$:⇔

> *"there is a finite sequence of ric's with `ON DELETE CASCADE` triggers leading from $\rhd\mathit{del}{:}R'(\bar{x}')$ to $R(\bar{x})$ in D."*

[8] Read I and II as "one" and "two", respectively. From the point of view of player I, you can read it also as "I" (myself) and "You" ("II" resembles "U").

Player `II` **can move from $\triangleright$del:$R(\bar{x})$ to $R'(\bar{x}')$:$\Leftrightarrow$**

"$R'(\bar{x}')$ is blocked by some `ON DELETE RESTRICT` *trigger, and there is a finite (possibly empty) sequence of ric's with* `ON DELETE CASCADE` *triggers leading from $\triangleright$del:$R(\bar{x})$ to $R'(\bar{x}')$ in D."*

The moves by `I` are reflected in the logical specification: if there is a successful (top-down) derivation of req_del:$R(\bar{x})$ using (DC_1) and successfully ending in a fact $\triangleright$del:$R'(\bar{x}')$, then the move from $R(\bar{x})$ to $\triangleright$del:$R'(\bar{x}')$ is allowed. Similarly, moves by `II` are reflected in the logical specification by rules (DC_2) and (DR) (without the negated goal).

The game itself can be easily defined in well-founded Datalog using the famous rule:

$$win(\bar{X}) \leftarrow\ move(\bar{X},\bar{X}'), \neg\, win(\bar{X}').$$

Ambiguity Revisited. We say that a game is *won* (*lost*) for `I` at position $R(\bar{x})$, if `I` (`II`) can win the game starting at $R(\bar{x})$, no matter how `II` (`I`) moves. A position which is neither lost nor won for `I` is *drawn*. Drawn positions can be viewed as ambiguous situations: Using "well-founded" arguments, neither can `I` prove that $R(\bar{x})$ has to be deleted, nor can `II` prove that it must not be deleted: there are negative cycles in the arguments leading to the truth-value *undefined* for req_del:$R(\bar{x})$.

The previously given specification P_W correctly reflects the intuitive game-theoretic description:

Theorem 4

- *I wins at $R(\bar{x})$ iff $\mathcal{W}(P_W, D, U_\triangleright) \models$ req_del:$R(\bar{x})$,*
- *II wins at $R(\bar{x})$ iff $\mathcal{W}(P_W, D, U_\triangleright) \models \neg$req_del:$R(\bar{x})$, and*
- *$R(\bar{x})$ is drawn iff $\mathcal{W}(P_W, D, U_\triangleright)$(req_del:$R(\bar{x})$) $=$ undef.*

Example 6 Consider again the "diamond" in Figure 1. The positions are $R_A(a)$, $R_B(a,b)$, $R_C(a,c)$, $R_D(a,b,c)$, and $\triangleright$del:$R_A(a)$.

`I` can move from any position in $\{R_A(a), R_B(a,b), R_C(a,c), R_D(a,b,c)\}$ to $\triangleright$del:$R_A(a)$, while `II` can move from $\triangleright$del:$R_A(a)$ to $R_D(a,b,c)$. Thus, after `I` has started the game moving to $\triangleright$del:$R_A(a)$, `II` will answer with the move to $R_D(a,b,c)$ and so on. Hence the game is drawn for all start positions of `I`.

In contrast, if $R_E(a)$ is added to the database in Figure 1, there is an additional move from $\triangleright$del:$R_A(a)$ to $R_E(a)$ for `II`, who now has a winning strategy: by moving to $R_E(a)$, there is no possible answer for `I`, so `I` loses. By Theorems 4 and 3, $R_A(a)$ cannot be deleted.

5 Conclusion

Referential actions (triggers) have been included in the SQL2 and SQL3 standards [JTC92, JTC94] as a means to automatically enforce referential integrity in relational databases. However, a naive implementation of the standard trigger semantics can lead to ambiguities due to different execution orders resulting in different final database states after an update. Moreover, as was shown in [Rei96], it is undecidable whether a given database *schema* with a set of *ric*'s is ambiguous. For a given database, the problem becomes decidable and can be checked at run-time as proposed in the SQL2 standard.

In this paper, we have argued for an alternative, logic-based semantics of referential actions which results in a concise and elegant description of the precise behavior of triggers. In this paper, we have confined ourselves to `ON DELETE CASCADE` and `ON DELETE RESTRICT` triggers. The proposed semantics is less restrictive than the SQL semantics and allows to execute the maximal set of user delete requests. In particular, the problem of ambiguity is avoided since our semantics yields a unique answer for every database with user requests and a given set of referential actions. We have presented three different, but essentially equivalent characterizations of this semantics:

The first is based on Statelog [LL94, LHL95], a state-oriented Datalog extension which allows to define active and deductive rules within a unified logical language. The presented Statelog program not only assigns a precise meaning to referential actions, but can also be used as an implementation. The second characterization uses the direct translation of referential actions into logic rules. Due to inherent negative cyclic dependencies, the resulting rules are non-stratified. The widely accepted well-founded semantics assigns a unique three-valued model to such programs. In our translation, undefined atoms of the form req_del:$R(\bar{x})$ and blk_del:$R(\bar{x})$ can be viewed as ambiguous requests to delete or restrict the deletion of a tuple $R(\bar{x})$, respectively. The presented Statelog specification assigns priority to deletions; therefore undefined deletions are viewed as true, while undefined blockings are viewed as false. The final, game-theoretic characterization yields additional insight into the behavior of triggers: the question whether a given tuple $R(\bar{x})$ may be deleted is regarded as a game between to players: I pleads for deletion, II for keeping $R(\bar{x})$ in the database. We show that the game is drawn for $R(\bar{x})$ iff the delete request req_del:$R(\bar{x})$ is undefined in the well-founded model. In future work, we plan to extend our approach to include more referential actions like `ON UPDATE CASCADE/RESTRICT/SET NULL`.

References

[Cod70] E. Codd. A Relational Model For Large Shared Data Banks. *Communications of ACM*, 13(6):377–387, 1970.

[Dat81] C. J. Date. Referential Integrity. In *Proc. Intl. Conference on Very Large Data Bases*, pages 2–12, Cannes, France, March 1981. IEEE Computer Society Press.

[Day88] U. Dayal. Active Database Management Systems. In C. Beeri, J. Schmidt, and U. Dayal, editors, *Proceedings of the 3rd International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 150–169. Morgan Kaufmann Publishers, Inc., June 1988.

[Esw76] K. P. Eswaran. Specification, Implementation and Interactions of a Trigger Subsystem in an Integrated Database System. IBM Research Report RJ-1820(26414), IBM Almaden Research Center, IBM Research Laboratory, San Jose, California 95193, 1976.

[For81] C. Forgy. OPS5 Users's Manual. Technical Report CMU-CS-81-135, CMU, 1981.

[JTC92] I. JTC1/SC21. Information Technology - Database Languages – SQL2, July 1992. ANSI, 1430 Broadway, New York, NY 10018.

[JTC94] I. JTC1/SC21/WG3. ISO/ANSI working draft Database Languages – SQL3, August 1994. J. Melton, editor, ANSI, 1430 Broadway, New York, NY 10018.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A Logical Framework for Active Rules. In *Proc. 7th Intl. Conf. on Management of Data (COMAD)*, Pune, India, December 1995. Tata McGraw-Hill. ftp://ftp.informatik.uni-freiburg.de/documents/reports/report78/report78.ps.gz.

[LL94] G. Lausen and B. Ludäscher. Updates by Reasoning about States. In J. Eder and L. Kalinichenko, editors, *2nd Intl. East-West Database Workshop*, Workshops in Computing, Klagenfurt, Austria, 1994. Springer.

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 191 – 216. Morgan Kaufmann, 1988.

[Rei96] J. Reinert. Ambiguity for Referential Integrity is Undecidable. In G. Kuper and M. Wallace, editors, *Constraint Databases and Applications*, number 1034 in LNCS, pages 132–147. Springer, 1996.

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *JACM*, 38(3):620 – 650, July 1991.

A Appendix: Proofs

Proof of **Theorem 1**.

1. $\mathcal{M} \models [n]\,\mathsf{req\_del}{:}R(\bar{x})$ only if it is derivable by (DC_1^S) and (I^S). Thus, there is a chain of `ON DELETE CASCADE` triggers from $R'(\bar{x}')$ to $R(\bar{x})$ such that $\mathcal{M} \models [n]\,\rhd\mathsf{del}{:}R'(\bar{x}'), \neg\mathsf{blk\_del}{:}R'(\bar{x}')$.
2. If $\mathcal{M} \models [n]\,\mathsf{blk\_del}{:}R(\bar{x}) \wedge \mathsf{req\_del}{:}R(\bar{x})$ then by (1), $\mathsf{req\_del}{:}R(\bar{x})$ is founded by some user request $\rhd\mathsf{del}{:}R'(\bar{x}')$ s.t. $\mathcal{M} \models [n]\,\neg\mathsf{blk\_del}{:}R'(\bar{x}')$ using a chain of `ON DELETE CASCADE` triggers. However, in $[n]$, (DC_2^S) also propagates blocking upwards this chain from $\mathsf{blk\_del}{:}R(\bar{x})$ to $\mathsf{blk\_del}{:}R'(\bar{x}')$ which is a contradiction.
3. First observe that in P_S, $\mathsf{req\_del}{:}R$ and $\mathsf{blk\_del}{:}R$ depend negatively on each other, ie if one increases the other can only decrease and vice versa. Moreover, $\mathcal{M} \models [0]\,\neg\mathsf{blk\_del}{:}R(\bar{x})$ for all $R(\bar{x})$, thus $\mathcal{M}(P_S, D, U_\rhd) \models [0]\,\mathsf{req\_del}{:}R(\bar{x})$ for all $\mathsf{req\_del}{:}R(\bar{x})$ which are founded by some user delete request. Therefore, initially all $\mathsf{blk\_del}{:}R$ are minimal and all $\mathsf{req\_del}{:}R$ are maximal possible wrt. (1), so $\mathsf{blk\_del}{:}R$ can only increase while $\mathsf{req\_del}{:}R$ can only decrease.
4. All delete requests are cascaded exhaustively: If a *ric* encoded as `ON DELETE CASCADE` is violated, then there are $R_P(\vec{x}, \bar{x})$ and $R_C(\vec{y}, \bar{y})$ such that $\vec{x} = \vec{y}$ and $\mathcal{M} \models [n]\,\mathsf{req\_del}{:}R_P(\vec{x}, \bar{x})$, but not $\mathcal{M} \models [n]\,\neg\mathsf{req\_del}{:}R_C(\vec{y}, \bar{y})$. This contradicts rule (DC_1^S) in P_S for the corresponding *ric*.
5. Follows from (1) and (4): In every state all internal delete requests are founded by some non-blocked user delete request and all non-blocked user delete requests are cascaded exhaustively.
6. Because of $\mathcal{M} \models [0]\,\mathsf{running}$, $n \geq 1$.
   If a *ric* of the form `ON DELETE RESTRICT` is violated, then there are $R_P(\vec{x}, \bar{x})$ and $R_C(\vec{y}, \bar{y})$ such that $\vec{x} = \vec{y}$ and $\mathcal{M} \models [n]\,\mathsf{req\_del}{:}R_P(\vec{x}, \bar{x}) \wedge \neg\mathsf{req\_del}{:}R_C(\vec{y}, \bar{y})$. Since $\mathsf{req\_del}{:}R_P$ is nonincreasing, $\mathcal{M} \models [n{-}1]\,\mathsf{req\_del}{:}R_P(\vec{x}, \bar{x})$.

   (i) Assume that in $[n{-}1]$ this *ric* is not violated. Then $\mathcal{M} \models [n{-}1]\,\mathsf{req\_del}{:}R_C(\vec{y}, \bar{y})$. Since in every state, internal delete requests are computed by cascading deletions from all non-blocked user delete requests, all user delete requests which founded $\mathsf{req\_del}{:}R_C(\vec{y}, \bar{y})$ in $[n{-}1]$ (and thus were not blocked in $[n{-}1]$) must be blocked in $[n]$, so there is at least one $\rhd\mathsf{del}{:}R'(\bar{x}')$ s.t. $\mathcal{M} \models [n{-}1]\,\neg\mathsf{blk\_del}{:}R'(\bar{x}')$ and $\mathcal{M} \models [n]\,\mathsf{blk\_del}{:}R'(\bar{x}')$, thus by (R_1^S), $\mathcal{M} \models [n]\,\mathsf{running}$ and $[n]$ cannot be the final state.

   (ii) If $[n{-}1]$ violates the above *ric*, $\mathcal{M} \models [n{-}1]\,\mathsf{req\_del}{:}R_P(\vec{x}, \bar{x}) \wedge \neg\mathsf{req\_del}{:}R_C(\vec{y}, \bar{y})$. By rule (DR^S) we have $\mathcal{M} \models [n]\,\mathsf{blk\_del}{:}R_P(\vec{x}, \bar{x})$. Contradiction to (2).
7. $[0]$: As shown in the proof of (3), $\mathsf{req\_del}{:}R$ is overestimated to be the whole set of founded delete requests, thus every deletion of a tuple $R(\bar{x})$ s.t. $\mathcal{M} \models [0]\,\neg\mathsf{req\_del}{:}R(\bar{x})$ would be unfounded. $\mathsf{blk\_del}{:}R$ is empty in the first step.

   $[n{-}1] \rightarrow [n]$: If $\mathcal{M} \models [n]\,\mathsf{blk\_del}{:}R(\bar{x}) \wedge [n{-}1]\,\neg\mathsf{blk\_del}{:}R(\bar{x})$ then by (DC_2^S) and (DR^S), there is some $R'(\bar{x}')$ and $R''(\bar{x}'')$ s.t. there is a sequence of `ON DELETE CASCADE` triggers from $R(\bar{x})$ to $R'(\bar{x}')$, and an `ON DELETE RESTRICT` from $R'(\bar{x}')$ to $R''(\bar{x}'')$ and $\mathcal{M} \models [n{-}1]\,\neg\mathsf{req\_del}{:}R''(\bar{x}'')$. Thus, by induction hypothesis, $R''(\bar{x}'')$ is not in an admissible set of deletions. Since deletion of $R(\bar{x})$ would trigger the deletion of $R'(\bar{x}')$, but this is restricted by $R''(\bar{x}'')$, it follows that $R(\bar{x})$ can also not be deleted, ie is not in an admissible set of deletions.

   If $\mathcal{M} \models [n]\,\neg\mathsf{req\_del}{:}R(\bar{x}) \wedge [n{-}1]\,\mathsf{req\_del}{:}R(\bar{x})$ then all user delete requests which founded the deletion of $R(\bar{x})$ in $[n{-}1]$ are blocked in $[n]$ ((I^S) and (DC_1^S)), ie $\mathcal{M} \models [n]\,\mathsf{blk\_del}{:}R'(\bar{x}')$ for all those tuples. Thus, as proven above, all of them cannot be deleted, thus the deletion of $R(\bar{x})$ would be unfounded wrt. the remaining set of user delete requests.
8. At the beginning, $\mathsf{req\_del}{:}R$ is overestimated to be the whole set of founded internal delete requests. As shown in (7), only tuples are removed from $\mathsf{req\_del}{:}R$ which cannot be in any admissible set. Thus the set is maximal.

As shown in (4), no *ric* encoded as `ON DELETE CASCADE` is violated in any state $[n]$ when executing all internal delete requests of this state, and (6) gives that no *ric* encoded as `ON DELETE RESTRICT` is violated in the final state when executing all internal delete requests.

9. For any two admissible sets of deletions U_1, U_2, also $U_1 \cup U_2$ is admissible. Hence there is a unique maximal admissible set DEL.

■

PROOF of **Theorem 3**. This is shown by recasting the alternating fixpoint computation of $\mathcal{W}(P_W)$ using an equivalent Statelog program P_A. Finally, we show how P_A and P_S are related which concludes the proof. As described in [LHL95], P_A can be constructed as follows:

Attach state terms to the given non-stratified program P_W, such that all positive literals refer to $[S{+}1]$ and all negative literals refer to $[S]$. The resulting Statelog program P_A computes the alternating fixpoint of P_W:[9]

$$[S{+}1]\,\mathsf{req\_del}{:}R(\bar{X}) \leftarrow\ \triangleright\mathsf{del}{:}R(\bar{X}), [S]\ \neg\mathsf{blk\_del}{:}R(\bar{X}). \qquad (I^A)$$

% $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ ON DELETE CASCADE:

$$[S{+}1]\ \mathsf{req\_del}{:}R_C(\vec{X},\bar{X}) \leftarrow\ R_C(\vec{X},\bar{X}), \vec{X}=\vec{Y}, [S{+}1]\ \mathsf{req\_del}{:}R_P(\vec{Y},\bar{Y}), \qquad (DC_1^A)$$

$$[S{+}1]\ \mathsf{blk\_del}{:}R_P(\vec{Y},\bar{Y}) \leftarrow\ R_P(\vec{Y},\bar{Y}), \vec{X}=\vec{Y}, [S{+}1]\ \mathsf{blk\_del}{:}R_C(\vec{X},\bar{X}). \qquad (DC_1^A)$$

% $R_C.\vec{X} \rightarrow R_P.\vec{Y}$ ON DELETE RESTRICT:

$$[S{+}1]\ \mathsf{blk\_del}{:}R_P(\vec{Y},\bar{Y}) \leftarrow\ R_P(\vec{Y},\bar{Y}), R_C(\vec{X},\bar{X}), \vec{X}{=}\vec{Y}, [S]\ \neg\mathsf{req\_del}{:}R_C(\vec{X},\bar{X}). \qquad (DR^A)$$

Note that P_A is a state-stratified Statelog program. Its perfect model $\mathcal{M}(P_A, D, U_\triangleright)$ mimics the alternating fixpoint computation of $\mathcal{W}(P_W, D, U_\triangleright)$: even-numbered states $[2n]$ correspond to the increasing sequence of underestimates of true atoms, while odd-numbered states $[2n+1]$ represent the decreasing sequence of overestimates of true (and undefined) atoms. The final state of the computation is reached if $\mathcal{M}[2n_{final}] = \mathcal{M}[2n_{final}+2]$. Then for all relations R, the truth value of atoms $R(\bar{x})$ in $\mathcal{W}(P_W)$ can be determined from $\mathcal{M}(P_A)$ as follows:

$$\mathcal{W}(P_W, D, U_\triangleright)(R(\bar{x})) = \begin{cases} \textit{true} & \textit{if } \mathcal{M}(P_A, D, U_\triangleright) \models [2n_{final}]\ R(\bar{x}) \\ \textit{undef} & \textit{if } \mathcal{M}(P_A, D, U_\triangleright) \models [2n_{final}]\ \neg R(\bar{x}) \wedge [2n_{final}+1]\ R(\bar{x}) \\ \textit{false} & \textit{if } \mathcal{M}(P_A, D, U_\triangleright) \models [2n_{final}+1]\ \neg R(\bar{x}) \end{cases}$$

It remains to show how P_A and P_S are related:

Lemma 5 *The model $\mathcal{M}(P_A, D, U_\triangleright)$ corresponds to $\mathcal{M}(P_S, D, U_\triangleright)$ as follows:*

1. $\mathcal{M}(P_A, D, U_\triangleright) \models [2n]\ \mathsf{blk\_del}{:}R(\bar{x}) \ \Leftrightarrow\ \mathcal{M}(P_S, D, U_\triangleright) \models [n]\ \mathsf{blk\_del}{:}R(\bar{x})$.
2. $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{+}1]\ \mathsf{req\_del}{:}R(\bar{x}) \ \Leftrightarrow\ \mathcal{M}(P_S, D, U_\triangleright) \models [n]\ \mathsf{req\_del}{:}R(\bar{x})$.

PROOF P_S and P_A differ in the rules (I^S) and (I^A): While (I^A) derives internal delete requests in $[S{+}1]$ from unblocked user requests in $[S]$, (I^S) already establishes these in the *current* state $[S]$.

In $[0]$ neither program derives blockings $\mathsf{blk\_del}{:}R(\bar{x})$; hence we have an underestimate of the final set of blockings. From this, both programs derive an overestimate of delete requests $\mathsf{req\_del}{:}R(\bar{x})$. Due to rules (I^S) and (I^A) these overestimates are computed in $[S]$ and $[S{+}1]$ by (I^S) and (I^A), respectively. Using these overestimates, the next sets of underestimates $\mathsf{blk\_del}{:}R(\bar{x})$ are derived in $[1]$ for P_S, and in $[2]$ for P_A. Applied inductively, this argument concludes the proof. ■

PROOF of **Theorem 4**. First, we prove the following

[9] It is assumed that base relations R and user requests $\triangleright\mathsf{del}{:}R$ are propagated unchanged by frame rules, so no state terms are needed for these relations.

Lemma 6

- *I wins at $R(\bar{x})$ within n rounds iff $\mathcal{M}(P_A, D, U_\triangleright) \models [2n]$ $\textsf{req\_del}{:}R(\bar{x})$.*
- *II wins at $R(\bar{x})$ within n rounds iff $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{-}1]$ $\neg\textsf{req\_del}{:}R(\bar{x})$.*

Proof (All subproofs below can be extended to "iff", but for better readability, this is not always formulated exactly.)

II wins in one round starting at $R(\bar{x})$ iff Player I cannot move to a user request, ie if the deletion of $R(\bar{x})$ is unfounded. That is the case iff in the first overestimate of P_A, $R(\bar{x})$ is not requested for deletion: $\mathcal{M}(P_A, D, U_\triangleright) \models [1]$ $\neg\textsf{req\_del}{:}R(\bar{x})$.

I wins in one round at $R(\bar{x})$ iff the deletion of $R(\bar{x})$ is founded by some user delete request $\triangleright\textsf{del}{:}R'(\bar{x}')$, and II cannot move from $\triangleright\textsf{del}{:}R'(\bar{x}')$. This is the case, if there is no `ON DELETE CASCADE` chain from $R'(\bar{x}')$ to a tuple $R''(\bar{x}'')$ which is restricted by some other tuple. Thus, in this case, in the first overestimate of P_A, the deletions of $R''(\bar{x}'')$ and $R'(\bar{x}')$ are not blocked: $\mathcal{M}(P_A, D, U_\triangleright) \models [1]$ $\neg\textsf{blk\_del}{:}R'(\bar{x}')$. Then, since there is a user delete request $\triangleright\textsf{del}{:}R'(\bar{x}')$, $\mathcal{M}(P_A, D, U_\triangleright) \models [2]$ $\textsf{req\_del}{:}R'(\bar{x}')$ and $\mathcal{M}(P_A, D, U_\triangleright) \models [2]$ $\textsf{req\_del}{:}R(\bar{x})$.

The induction step follows the same line of argumentation:

II wins in $n{+}1$ rounds at $R(\bar{x})$ iff for all moves to some $\triangleright\textsf{del}{:}R'(\bar{x}')$ of I, he can move to some tuple $R''(\bar{x}'')$ which he wins in n rounds: $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{-}1]$ $\neg\textsf{req\_del}{:}R''(\bar{x}'')$ by induction hypothesis. Thus, since there is a move from $\triangleright\textsf{del}{:}R'(\bar{x}')$ to $R''(\bar{x}'')$, there are triggers `ON DELETE RESTRICT` and `ON DELETE CASCADE` s.t. $\mathcal{M}(P_A, D, U_\triangleright) \models [2n]$ $\textsf{blk\_del}{:}R'(\bar{x}')$. Since this is the case for all $R'(\bar{x}')$ where I can move to from $R(\bar{x})$, $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{+}1]$ $\neg\textsf{req\_del}{:}R''(\bar{x}'')$.

I wins in $n{+}1$ rounds at $R(\bar{x})$ if there is a $R'(\bar{x}')$ he can move to s.t. for all positions $R''(\bar{x}'')$ where II can move to from $R'(\bar{x}')$, II will lose in at most n rounds. By induction hypothesis, for all those $R''(\bar{x}'')$, $\mathcal{M}(P_A, D, U_\triangleright) \models [2n]$ $\textsf{req\_del}{:}R''(\bar{x}'')$. Thus, $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{+}1]$ $\neg\textsf{blk\_del}{:}R'(\bar{x}')$ and $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{+}2]$ $\textsf{req\_del}{:}R(\bar{x})$. ■

From the previous lemma, Theorem 4 follows immediately: Since even-numbered states are underestimates, there is an n such that $\mathcal{M}(P_A, D, U_\triangleright) \models [2n]$ $\textsf{req\_del}{:}R(\bar{x})$ iff $\mathcal{W}(P_W, D, U_\triangleright) \models$ $\textsf{req\_del}{:}R(\bar{x})$, and on the other hand, since odd-numbered states are overestimates, there is an n such that $\mathcal{M}(P_A, D, U_\triangleright) \models [2n{+}1]$ $\neg\textsf{req\_del}{:}R''(\bar{x}'')$ iff $\mathcal{W}(P_W, D, U_\triangleright) \models \neg\textsf{req\_del}{:}R''(\bar{x}'')$.

The game is drawn at $R(\bar{x})$ if for every tuple $R'(\bar{x}')$ which II chooses, I can find a user request which deletes it, and conversly, II has a witness against each such user request. Therefore each player has no "well-founded" proof for or against deleting those tuples. This directly corresponds to the alternating fixpoint characterization of the well-founded model: The n-th overestimate of deletions excludes those tuples which can be disproved in n rounds, whereas the n-th underestimate contains all tuples which can be proved in n rounds. ■