6th Intl. Conference on Extending Database Technology (EDBT'98), March 23–27, 1998, Valencia, Spain. Springer LNCS 1377, pp. 404-418.

Referential Actions: From Logical Semantics to Implementation

Bertram Ludäscher Wolfgang May

Institut für Informatik, Universität Freiburg, Germany
{ludaesch,may}@informatik.uni-freiburg.de

Abstract. Referential actions (*rac*'s) are specialized triggers used to automatically maintain referential integrity. While their local effects can be grasped easily, it is far from obvious what the global semantics of a set RA of interacting *rac*'s should be. To capture the intended meaning of RA, we first present an abstract non-constructive semantics. By formalizing RA as a logic program P_{RA}, a constructive semantics is obtained. The equivalence of the logic programming semantics and the abstract semantics is proven using a game-theoretic characterization, which provides additional insight into the meaning of *rac*'s. As shown in previous work, for general *rac*'s, it may be infeasible to compute all *maximal admissible* solutions. Therefore, we focus on a tractable subset, i.e., *rac*'s without modifications. We show that in this case a unique maximal admissible solution exists, and derive a PTIME algorithm for computing this solution. In case a set $U_\triangleright$ of user requests is not admissible, a maximal admissible subset of $U_\triangleright$ is suggested.

1 Introduction

We study the following problem: Given a relational database D, a set of user-defined update requests $U_\triangleright$, and a set of referential actions RA, find those sets of updates Δ which (i) preserve referential integrity in the new database D', (ii) are maximal wrt. $U_\triangleright$, and (iii) reflect the intended meaning of RA. This notion of intended "optimal" updates will be formalized using so-called *maximal admissible* sets of updates.

The problem is important both from a practical and theoretical point of view: *Referential integrity constraints* (*ric*'s) are a central concept of the relational database model and frequently used in real world applications. *Referential actions* (*rac*'s) are specialized triggers used to automatically enforce integrity, thereby relieving the user from the burden of enumerating all induced updates which arise from an initial user request $U_\triangleright$.

Due to their practical importance, *rac*'s have been included in the SQL2 standard and SQL3 proposal [ISO92, ISO95]. In [DD94] and [Dat90], the problem of unpredictable behavior, i.e., ambiguities in determining the above Δ and D', in certain situations is addressed. In [Hor92, CPM96], a solution is presented, based on a rather ad-hoc run-time execution model. In a different approach, [Mar94] presents safeness conditions which aim at avoiding ambiguities at the schema level. However, as shown in [Rei96], it is in general undecidable whether a database schema with *rac*'s is ambiguous. Summarizing, from a theoretical point of view, the problem has not been solved in a satisfactory way.

In this paper, we continue our work on declarative semantics for referential actions. First results have been reported in [LMR96]. In [LML97a], it is shown that for *rac*'s with modifications, it may be infeasible to compute all maximal admissible solutions (intuitively, there are several equally justified ways how to propagate the combined effect of modifications, leading to an exponential blow up, both in the number of rules for integrity maintenance and in the number of solutions). Here, we therefore restrict to the tractable class of *rac*'s without modifications. This guarantees the existence of a unique optimal solution which can be efficiently computed.

In Section 2, we introduce the basics of referential integrity and illustrate the problem of ambiguity. In Section 3.1, we identify and formalize desirable abstract properties of updates which lead to a non-constructive global semantics of *rac*'s. A constructive definition providing a global semantics is obtained by formalizing a set of referential actions RA as a logic program P_{RA} (Section 3.2). The correctness of this characterization is proven via an equivalent game-theoretic characterization (Section 3.3) which allows intelligible proofs on a less technical level (Section 4). From the logic programming characterization, an algorithm for computing the maximal admissible solution is derived (Section 5).

2 Referential Integrity

Notation and Preliminaries. A relation schema consists of a relation name R and a vector of attributes $(A_1, \ldots, A_n)$. We identify attribute names A_i of R with the integers $1, \ldots, n$. By $\mathbf{A} = (i_1, \ldots, i_k)$ we denote a vector of $k \leq n$ distinct attributes (usually $\mathbf{A}$ will be some key).

Tuples of R are denoted by first-order atoms $R(\bar{X})$ with n-ary relation symbol R, and vector $\bar{X}$ of variables or constants from the underlying domain. To emphasize that such a vector is ground, i.e., comprises only constants, we write $\bar{x}$ instead of $\bar{X}$. The *projection* of tuples $\bar{X}$ to an attribute vector $\mathbf{A}$ is denoted by $\bar{X}[\mathbf{A}]$: e.g., if $\bar{x} = (a, b, c)$, $\mathbf{A} = (1, 3)$, then $\bar{x}[\mathbf{A}] = (a, c)$. Deletions are denoted by $\mathsf{del}{:}R(\bar{x})$.

For a relation schema R with attributes $\mathbf{A}$, a minimal subset $\mathbf{K}$ of $\mathbf{A}$ whose values uniquely identify each tuple in R is a *candidate key*. In general, the database schema specifies which attribute vectors are keys. A candidate key $R.\mathbf{K}$ has to satisfy the first-order sentence φ_{key} for every database instance D:

$$\forall \bar{X}_1, \bar{X}_2 \ (R(\bar{X}_1) \land R(\bar{X}_2) \land \bar{X}_1[\mathbf{K}] = \bar{X}_2[\mathbf{K}] \rightarrow \bar{X}_1 = \bar{X}_2 \ ) \ . \qquad (\varphi_{key})$$

Referential Integrity Constraints. A *referential integrity constraint* (*ric*) is an expression of the form

$$R_C.\mathbf{F} \rightarrow R_P.\mathbf{K} \ ,$$

where $\mathbf{F}$ is a *foreign key* of the *child relation* R_C, referencing a candidate key $\mathbf{K}$ of the *parent relation* R_P. A *ric* $R_C.\mathbf{F} \rightarrow R_P.\mathbf{K}$ is *satisfied* by a given database D, if for every child tuple $R_C(\bar{x})$ with foreign key values $\bar{x}[\mathbf{F}]$, there exists a tuple $R_P(\bar{y})$ with matching key value, i.e., $\bar{x}[\mathbf{F}] = \bar{y}[\mathbf{K}]$. Thus, for a database instance D, a *ric* is satisfied if $D \models \varphi_{ric}$:

$$\forall \bar{X} \ (\ R_C(\bar{X}) \rightarrow \exists \bar{Y} \ (R_P(\bar{Y}) \land \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}]) \ ) \ . \qquad (\varphi_{ric})$$

A *ric* is *violated* by D if it is not satisfied by D.

Referential Actions. Rule-based approaches to referential integrity maintenance are attractive since they describe how *ric*'s should be enforced using "local repairs": Given a *ric* $R_C.\mathbf{F} \to R_P.\mathbf{K}$ and an update operation insert, delete, or modify on R_P or R_C, a *referential action* (*rac*) defines some local operation on R_C or R_P, respectively. It is easy to see from the logical implication in (φ_{ric}) that insert into R_P and delete from R_C cannot introduce a violation of a *ric*, whereas the other updates can. For these, there are two strategies to maintain referential integrity by *local* actions:

- CASCADE: propagate the update from the parent to the child,
- REJECT: reject an update on the parent if there exists a referencing tuple.

The Problem of Ambiguity. With this *local* specification of behavior, there may be ambiguities wrt. the *global* semantics, leading to different final states. A relational database *schema* S with *rac*'s RA is ambiguous, if there is some database instance D and some set of user requests $U_\triangleright$ s.t. there are different final states D' depending on the execution order of referential actions. As shown in [Rei96], it is in general undecidable whether a schema with *rac*'s is ambiguous (given D and $U_\triangleright$, the problem becomes decidable). The following example from [Rei96] illustrates the problem:

Example 1 Consider the database with *rac*'s depicted in Fig. 1. Solid arcs represent *ric*'s and point from R_C to R_P, *rac*'s are denoted by dashed (`CASCADE`) or dotted (`REJECT`) arcs. Let $U_\triangleright = \{\mathsf{del}{:}R_1(a)\}$ be a user request to delete the tuple $R_1(a)$. Depending on the order of execution of *rac*'s, one of two different final states may be reached:

1. If execution follows the path R_1–R_3–R_4, the tuple $R_3(a,y)$ cannot be deleted: Since $R_4(a,x,y)$ references $R_3(a,y)$, the *rac* for R_4 forbids the deletion of $R_3(a,y)$. This in turn forbids the deletion of $R_1(a)$. Thus, the user request $\mathsf{del}{:}R_1(a)$ is rejected, and the database remains unchanged, i.e., $D' = D$.
2. If execution follows the path R_1–R_2–R_4, the tuples $R_2(a,x)$ and $R_4(a,x,y)$ are requested for deletion. Hence, the *rac* for $R_4.(1,3) \to R_3.(1,2)$ can assume that $R_4(a,x,y)$ is deleted, thus no referencing tuple exists in R_4. Therefore, all deletions can be executed, resulting in a new database state $D' \neq D$.

We argue that (2) is preferable to (1), since it accomplishes the desired user request without violating referential integrity. □

2.1 Disambiguating Strategies

The ambiguity in Example 1 can be eliminated by specifying that *rac*'s of type `REJECT` are always evaluated wrt. the database state either *before* starting the transaction or *after* the complete transaction, leading to the following strategies to maintain referential integrity by referential actions:

- CASCADE: propagate the update from the parent to the child,
- RESTRICT: (i) reject an update on the *parent* if there exists a child referencing it in the *original* database state, or (ii) reject an update on the *child* if there is no tuple with the respective parent key in the *original* database state,

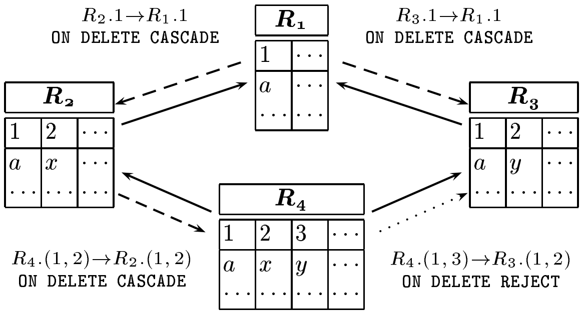


Fig. 1. Database with referential actions

- NO ACTION: similar to RESTRICT, but look at the database state *after* (hypothetically) applying all updates (in active database terminology, this corresponds to change *immediate coupling* of referential actions into *deferred coupling*).

Since the final state depends on the updates to be executed, and these may in turn depend on the final state via NO ACTION, there is a cyclic dependency. In Section 3, we show how to solve this semantical problem using different (logical and game-theoretic) characterizations of *rac*'s.

In SQL, referential actions for a given *ric* $R_C.\mathbf{F} \rightarrow R_P.\mathbf{K}$ are specified with the definition of the child relation:

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

(`RESTRICT` is not contained in SQL2, but in the SQL3 proposal.)

Due to lack of space, we do not consider insertions in the sequel. Note however, that insertions can be handled in a straightforward way by rejecting updates which aim to insert a child tuple whose corresponding parent does not exist (this is also the SQL strategy), and all results can be directly extended to incorporate insertions (cf. [LML97a, LML97b]). Moreover, as mentioned above, we deliberately exclude modifications (i.e., `ON UPDATE` triggers and `SET NULL/DEFAULT` actions, the latter being a special case of modifications), since this problem is intractable in general [LML97a].

Thus, in this work, we investigate *rics* $R_C.\mathbf{F} \rightarrow R_P.\mathbf{K}$ with corresponding *rac*'s of the form $R_C.\mathbf{F} \rightarrow R_P.\mathbf{K}$ `ON DELETE {CASCADE|RESTRICT|NO ACTION}`.

3 Semantics of Referential Actions

In order to avoid ambiguities and indeterminism like in Example 1, it is necessary to specify the intended global semantics of *rac*'s. First, we define an abstract, non-constructive semantics which serves as the basis for a notion of correctness. Next, we show how to translate a set of *rac*'s into a logic program, whose declarative semantics provides a constructive definition. An equivalent game-theoretic characterization is developed which will be used to prove the correctness of the logic programming semantics (Section 4).

3.1 Abstract Semantics

Let D be a database represented as a set of ground atoms, RA a set of *rac*'s, and $U_{\triangleright} = \{\mathsf{del}{:}R_1(\bar{x}_1), \ldots, \mathsf{del}{:}R_n(\bar{x}_n)\}$ a set of (external) *user delete requests* which are passed to the system. D and RA define three graphs $\mathcal{DC}$ (`ON DELETE CASCADE`), $\mathcal{DR}$ (`ON DELETE RESTRICT`), and $\mathcal{DN}$ (`ON DELETE NO ACTION`) corresponding to the different types of references:

$$\begin{aligned}\mathcal{DC} := \{\,& (R_C(\bar{x}), R_P(\bar{y})) \in D \times D \mid \\ & R_C.\mathbf{F} \rightarrow R_P.\mathbf{K}\ \texttt{ON DELETE CASCADE} \in RA \text{ and } \bar{x}[\mathbf{F}] = \bar{y}[\mathbf{K}]\},\end{aligned}$$

$\mathcal{DR}$ and $\mathcal{DN}$ are defined analogously. $\mathcal{DC}^*$ denotes the reflexive transitive closure of $\mathcal{DC}$. Note that the graphs describe *potential* interactions due to *rac*'s, independent of the given user requests $U_{\triangleright}$. To capture the intended semantics, $U_{\triangleright}$ has to be considered:

Definition 1 Given RA, D, and $U_{\triangleright}$, a set Δ of delete requests is called

- *founded*, if $\mathsf{del}{:}R(\bar{x}) \in \Delta$ implies $(R(\bar{x}), R'(\bar{x}')) \in \mathcal{DC}^*$ for some $\mathsf{del}{:}R'(\bar{x}') \in U_{\triangleright}$,
- *complete*, if $\mathsf{del}{:}R_P(\bar{y}) \in \Delta$ and $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DC}$ implies $\mathsf{del}{:}R_C(\bar{x}) \in \Delta$,
- *feasible*, if
  - $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DR}$ implies $\mathsf{del}{:}R_P(\bar{y}) \notin \Delta$, and
  - $\mathsf{del}{:}R_P(\bar{y}) \in \Delta$ and $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DN}$ implies $\mathsf{del}{:}R_C(\bar{x}) \in \Delta$,
- *admissible*, if it is founded, complete, and feasible. □

Foundedness guarantees that all deletions are "justified" by some user request, completeness guarantees that no cascading deletions are "forgotten", and feasibility ensures that `RESTRICT`/`NO ACTION` *rac*'s are "obeyed".

Definition 2 (Maximal Admissible Sets, Intended Semantics)
Let RA, D, and $U_{\triangleright}$ be given.

- The set of *induced updates* $\Delta(U)$ of a set of user requests $U \subseteq U_{\triangleright}$ is the least set Δ which contains U and is complete.
- A set of user requests $U \subseteq U_{\triangleright}$ is *admissible* if $\Delta(U)$ is admissible, and *maximal admissible* if there is no other admissible U', s.t. $U \subsetneq U' \subseteq U_{\triangleright}$.
- The *intended semantics* are the maximal admissible subsets of $U_{\triangleright}$. □

Proposition 1 (Correctness)
a) If $U \subseteq U_{\triangleright}$, then $\Delta(U)$ is founded and complete.
b) If Δ is complete and feasible, then $D' := D \pm \Delta(U)$ satisfies all rics. □

Proof *a)* $\Delta(U)$ is defined as the *least* complete set. It follows that $\Delta(U)$ is founded. *b)* Completeness guarantees that all *ric*'s labeled with `ON DELETE CASCADE` in RA are satisfied, feasibility guarantees that all *ric*'s labeled with `ON DELETE RESTRICT`/`NO ACTION` are satisfied. ■

Theorem 2 (Uniqueness)
Given RA, D, and $U_\triangleright$*, there is exactly one maximal admissible* $U_{\max} \subseteq U_\triangleright$*.*

PROOF Observe that $U_1 \cup U_2$ is admissible if $U_1, U_2 \subseteq U_\triangleright$ are admissible. Thus, the union of all admissible subsets of $U_\triangleright$ yields $U_{\max}$. ■

3.2 Logic Programming Characterization

We show how a set RA of *rac*'s is compiled into a logic program P_{RA} whose rules specify their local behavior. The advantage of this logical formalization is that the declarative semantics of P_{RA} defines a precise *global* semantics.

The following rule derives for every user request $\mathsf{del}{:}R(\bar{x}) \in U_\triangleright$ an *internal delete request* $\mathsf{req\_del}{:}R(\bar{x})$, provided there is no *blocking* $\mathsf{blk\_del}{:}R(\bar{x})$:

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

Referential actions are specified as follows:

- $R_C.\mathbf{F} \to R_P.\mathbf{K}$ `ON DELETE CASCADE` is encoded into two rules: the first one propagates internal delete requests downwards from the parent to the child:

  $$\mathsf{req\_del}{:}R_C(\bar{X}) \leftarrow \mathsf{req\_del}{:}R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}]. \qquad (DC_1)$$

  Additionally, blockings are propagated upwards, i.e., when the deletion of a child is blocked, the deletion of the referenced parent is also blocked:

  $$\mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow R_P(\bar{Y}),\ \mathsf{blk\_del}{:}R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}]. \qquad (DC_2)$$

- $R_C.\mathbf{F} \to R_P.\mathbf{K}$ `ON DELETE RESTRICT` blocks the deletion of a parent tuple if there is a corresponding child tuple:

  $$\mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}]. \qquad (DR)$$

- $R_C.\mathbf{F} \to R_P.\mathbf{K}$ `ON DELETE NO ACTION` blocks the deletion of a parent tuple if there is a corresponding child tuple which is not requested for deletion:

  $$\mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow R_P(\bar{Y}),\ R_C(\bar{X}),\ \neg\mathsf{req\_del}{:}R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}]. \quad (DN)$$

Due to the negative cyclic dependency $\mathsf{req\_del} \overset{\neg}{\leadsto} \mathsf{blk\_del} \overset{\neg}{\leadsto} \mathsf{req\_del}$, P_{RA} is in general not stratified.

Well-Founded Semantics. The well-founded model [VGRS91] is widely accepted as a (skeptical) declarative semantics for logic programs. The well-founded model $\mathcal{W}_{RA}$ of $P_{RA} \cup D \cup U_\triangleright$ assigns a third truth value *undefined* to atoms whose truth cannot be determined using a "well-founded" argumentation.

Often, even if not all requested updates can be accomplished, it is still possible to execute some of them while postponing the others. Thus, the information which tuple or update really causes problems is valuable for preparing a refined update that realizes the intended changes *and* is acceptable:

Example 2 Consider the database with *rac*'s in Fig. 2, and the user request $U_\triangleright = \{\mathsf{del}{:}R_1(a), \mathsf{del}{:}R_1(b)\}$. $\mathsf{del}{:}R_1(b)$ is not admissible since it is blocked by $R_5(b)$. However, the other request, $\mathsf{del}{:}R_1(a)$, can be executed without violating

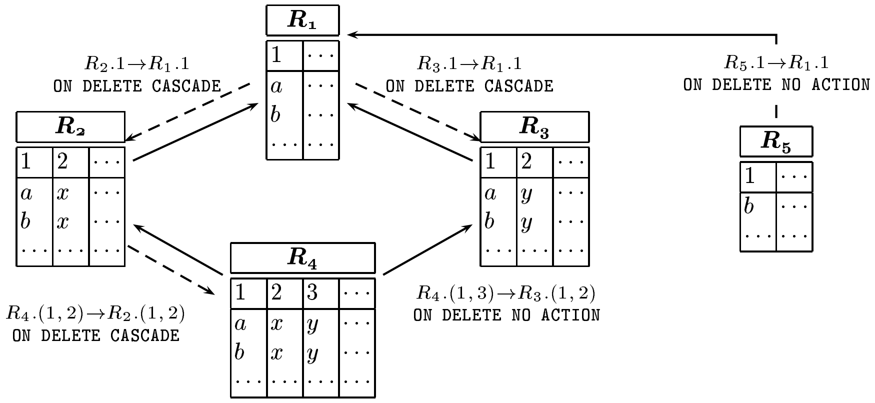


Fig. 2. Extended database with modified *rac*'s

any *ric* by deleting $R_1(a)$, $R_2(a,x)$, $R_3(a,y)$, and $R_4(a,x,y)$. Thus, the extended set $U'_{\triangleright} = \{\mathsf{del}{:}R_1(a), \mathsf{del}{:}R_1(b), \mathsf{del}{:}R_5(b)\}$ is a candidate for a refined update request which accomplishes the deletion of $R_1(a)$ and $R_1(b)$.
The well-founded semantics reflects the different status of the single updates:

Given the user request $U_{\triangleright} = \{\mathsf{del}{:}R_1(a)\}$, the delete requests $\mathsf{req\_del}$ for $R_1(a)$, $R_2(a,x)$, $R_3(a,y)$, $R_4(a,x,y)$, as well as the blockings $\mathsf{blk\_del}$ for $R_1(a)$ and $R_3(a,y)$ will be *undefined* in the well-founded model.

For the user request $U'_{\triangleright} = \{\mathsf{del}{:}R_1(b)\}$, $\mathsf{blk\_del}$ is *true* for $R_1(b)$ due to the referencing tuple $R_5(b)$. Thus, $\mathsf{req\_del}{:}R_1(b)$ is *false*, and $\mathsf{del}{:}R_1(b)$ is not admissible; hence there are no cascaded delete requests. Due to the referencing tuple $R_4(b,x,y)$ which cannot be deleted in this case, $\mathsf{blk\_del}{:}R_3(b,y)$ is also *true*. □

$\mathcal{W}_{RA}$ contains some ambiguities which can be interpreted constructively as *degrees of freedom*: The blockings and deletions induced by $U_{\triangleright} = \{\mathsf{del}{:}R_1(a)\}$ in Example 2 are undefined due to the dependency $\mathsf{req\_del} \overset{\neg}{\rightsquigarrow} \mathsf{blk\_del} \overset{\neg}{\rightsquigarrow} \mathsf{req\_del}$. This freedom may be used to define different global policies by giving priority either to deletions or blockings (cf. Theorems 10 and 11).

3.3 Triggers as Games

The following game-theoretic formalization provides an elegant characterization of *rac*'s yielding additional insight into the well-founded model of P_{RA} and the intuitive meaning of *rac*'s.

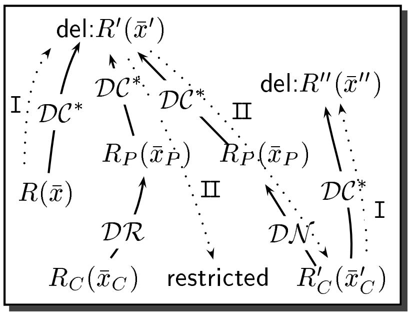
The game is played with a pebble by two players, $\mathtt{I}$ (the *"Deleter"*) and $\mathtt{II}$ (the *"Spoiler"*), who argue whether a tuple may be deleted. The players move alternately in *rounds*; each round consists of two *moves*. A player who cannot move loses. The set of *positions* of the game is $D \cup U_{\triangleright} \cup \{\mathsf{restricted}\}$. The possible moves of $\mathtt{I}$ and $\mathtt{II}$ are defined below. Note that $\mathtt{I}$ moves from D to $U_{\triangleright}$, while $\mathtt{II}$ moves from $U_{\triangleright}$ to $D \cup \{\mathsf{restricted}\}$. Initially, the pebble is placed on some tuple in D (or $U_{\triangleright}$) and $\mathtt{I}$ (or $\mathtt{II}$) starts to move. If $\mathtt{II}$ starts the game, the first round only consists of the move by $\mathtt{II}$.

By moving the pebble from $R(\bar{x}) \in D$ to some del:$R'(\bar{x}') \in U_{\triangleright}$ which cascades down to $R(\bar{x})$, I claims that the deletion of $R(\bar{x})$ is "justified" (i.e., founded) by del:$R'(\bar{x}')$. Conversely, II claims by her moves that del:$R'(\bar{x}')$ is not feasible. II can use two different arguments: Assume that the deletion of $R'(\bar{x}')$ cascades down to some tuple $R_P(\bar{x}_P)$. First, if the deletion of $R_P(\bar{x}_P)$ is restricted by a referencing child tuple $R_C(\bar{x}_C)$, then II may force I into a lost position by moving to restricted (since I cannot move from there). Second, II can move to a child tuple $R'_C(\bar{x}'_C)$ which references $R_P(\bar{x}_P)$ with a `NO ACTION` trigger. With this move, II claims that this reference to $R_P(\bar{x}_P)$ will remain in the database, so $R_P(\bar{x}_P)$ and, as a consequence, $R'(\bar{x}')$ cannot be deleted. In this case, I may start a new round of the game by finding a justification to delete the referencing child $R'_C(\bar{x}'_C)$. More precisely:

Player I can move from $R(\bar{x})$ to del:$R'(\bar{x}')$ if $(R(\bar{x}), R'(\bar{x}')) \in \mathcal{DC}^*$ and there is no $R_C(\bar{x}_C) \in D$ s.t. $(R_C(\bar{x}_C), R(\bar{x})) \in \mathcal{DR}$.

Player II can move from del:$R'(\bar{x}')$

- to restricted if there are $R_P(\bar{x}_P)$ and $R_C(\bar{x}_C)$ s.t. $(R_P(\bar{x}_P), R'(\bar{x}')) \in \mathcal{DC}^*$ and $(R_C(\bar{x}_C), R_P(\bar{x}_P)) \in \mathcal{DR}$.
- to $R'_C(\bar{x}'_C)$, if $(R_P(\bar{x}_P), R'(\bar{x}')) \in \mathcal{DC}^*$ and $(R'_C(\bar{x}'_C), R_P(\bar{x}_P)) \in \mathcal{DN}$.



Lemma 3 (Claims of I and II)

1. *If I can move from $R(\bar{x})$ to del:$R'(\bar{x}')$, then deletion of $R'(\bar{x}')$ induces the deletion of $R(\bar{x})$.*
2. *If II can move from del:$R(\bar{x})$ to restricted, then deletion of $R(\bar{x})$ is forbidden in the original database state.*
3. *If II can move from del:$R(\bar{x})$ to $R'(\bar{x}')$, then deletion of $R(\bar{x})$ is only admissible if $R'(\bar{x}')$ is also deleted.* □

PROOF 1. The move of I implies that $(R(\bar{x}), R'(\bar{x}')) \in \mathcal{DC}^*$.
The move of II means that either

2. there are $R_P(\bar{x}_P), R_C(\bar{x}_C)$ s.t. $(R_P(\bar{x}_P), R(\bar{x})) \in \mathcal{DC}^*$ and $(R_C(\bar{x}_C), R'(\bar{x}')) \in \mathcal{DR}$. Then, by (1), deletion of $R(\bar{x})$ induces the deletion of $R_P(\bar{x}_P)$, but the deletion of $R_P(\bar{x}_P)$ is restricted by $R_C(\bar{x}_C)$, or
3. $(R'(\bar{x}'), R(\bar{x})) \in \mathcal{DN} \circ \mathcal{DC}^*$, i.e., there is a $R_P(\bar{x}_P)$ s.t. $(R_P(\bar{x}_P), R(\bar{x})) \in \mathcal{DC}^*$ and $(R'(\bar{x}'), R_P(\bar{x}_P)) \in \mathcal{DN}$. Hence, by (1), deletion of $R(\bar{x})$ induces deletion of $R_P(\bar{x}_P)$, which is only allowed if $R'(\bar{x}')$ is also deleted.[1] ■

Lemma 4 *The moves are linked with the logical specification as follows:*

- *The moves of I correspond to rule (DC_1): I can move from $R(\bar{x})$ to del:$R'(\bar{x}')$ if, given the fact req_del:$R'(\bar{x}')$, req_del:$R(\bar{x})$ can be derived using (DC_1).*

[1] $\mathcal{DN} \circ \mathcal{DC}^* := \{(x, y) \mid \exists z : (x, z) \in \mathcal{DN} \text{ and } (z, y) \in \mathcal{DC}^*\}$.

- *The moves by II are reflected by the rules (DC_2) and (DR)/(DN):*
  - *II can move from del:$R(\bar{x})$ to $\mathsf{restricted}$ if $\mathsf{blk\_del}$:$R(\bar{x})$ is derivable using (DR) and (DC_2) only, or*
  - *II can move from del:$R(\bar{x})$ to $R'(\bar{x}')$ if $\mathsf{blk\_del}$:$R(\bar{x})$ is derivable using (DC_2) and an instance of (DN) if $\mathsf{req\_del}$:$R'(\bar{x}')$ is assumed to be false.*
- *The negative dependencies in (I), $\mathsf{req\_del} \rightsquigarrow \neg\mathsf{blk\_del}$, and ($DN$), $\mathsf{blk\_del} \rightsquigarrow \neg\mathsf{req\_del}$, mirror the alternation of moves between I and II, respectively.* □

Definition 3 A position $R(\bar{x}) \in D$ is *won (for I)*, if I can win the game starting from $R(\bar{x})$ no matter how II moves; del:$R(\bar{x}) \in U_{\triangleright}$ is *won for II*, if II can always win the game starting from del:$R(\bar{x})$. If $p \in D \cup U_{\triangleright}$ is won (lost) for a player, p is lost (won) for the opponent. A position which is neither lost nor won is *drawn*. In the sequel, "is won/lost" stands for "is won/lost for I". □

Drawn positions can be viewed as ambiguous situations. For the game above, this means that neither can I prove that $R(\bar{x})$ has to be deleted, nor can II prove that it is infeasible to delete $R(\bar{x})$.

Example 3 Consider again Fig. 2 with $U_{\triangleright} = \{\mathsf{del}{:}R_1(a), \mathsf{del}{:}R_1(b)\}$. From each of the "$a$"-tuples, $\{R_1(a), R_2(a,x), R_3(a,y), R_4(a,x,y)\}$, I can move to del:$R_1(a)$, while II can move from del:$R_1(a)$ to $R_4(a,x,y)$. Thus, after I has started the game moving to del:$R_1(a)$, II will answer with the move to $R_4(a,b,c)$, so I moves back to del:$R_1(a)$ again, etc. Hence the game is drawn for each of the "a"-tuples.

In contrast, for the "b"-tuples, there is an additional move from del:$R_1(b)$ to $R_5(b)$ for II, who now has a winning strategy: by moving to $R_5(b)$, there is no possible answer for I, so I loses. □

Theorem 5 (Game Semantics) *For every tuple $R(\bar{x}) \in D$:*

- *$R(\bar{x})$ is won $\Leftrightarrow$ there is a sequence of user requests from $U_{\triangleright}$ which deletes $R(\bar{x})$, and if this sequence is executed serially (independent from the order of execution of cascaded deletions), at no stage any ric is violated.*
- *$R(\bar{x})$ is won or drawn $\Leftrightarrow$ simultaneous execution of all user delete requests del:$R'(\bar{x}')$ which are won or drawn does not violate any ric and deletes $R(\bar{x})$.*
- *$R(\bar{x})$ is lost $\Leftrightarrow$ it is not possible with the given set of user delete requests to delete $R(\bar{x})$ without violating a ric.*

Proof Note that if $R(\bar{x})$ is won or drawn, then $(R_C(\bar{x}_C), R(\bar{x})) \notin \mathcal{DR}$ for any $R_C(\bar{x}_C) \in D$ (otherwise, if I moves from $R(\bar{x})$ to some $R_d(\bar{x}_d)$, II moves to $\mathsf{restricted}$ since $(R_C(\bar{x}_C), R_d(\bar{x}_d)) \in \mathcal{DR} \circ \mathcal{DC}^*$ and wins). Thus, no *ric* of the form `ON DELETE RESTRICT` is violated when deleting some won or drawn tuple.

- Let $U_{\triangleright,n} := \{u \in U_{\triangleright} \mid u \text{ is won in } n \text{ rounds}\}$. Let $R(\bar{x})$ be won in n rounds:
  - I can move from $R(\bar{x})$, thus there exists a del:$R_d(\bar{x}_d) \in U_{\triangleright,n}$ such that $(R(\bar{x}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$. Hence, executing $U_{\triangleright,n}$ also deletes $R(\bar{x})$.
  - For every $R'(\bar{x}')$: if $(R'(\bar{x}'), R(\bar{x})) \in \mathcal{DC}$, then also $(R'(\bar{x}'), R_d(\bar{x}_d)) \in \mathcal{DC}^*$ and $R'(\bar{x}')$ is won in n rounds, and will also be deleted. Thus, no *rac* `ON DELETE CASCADE` is violated when executing $U_{\triangleright,n}$.

– For every $R'(\bar{x}')$ s.t. $(R'(\bar{x}'), R(\bar{x})) \in \mathcal{DN}$, $(R'(\bar{x}'), R_d(\bar{x}_d)) \in \mathcal{DN} \circ \mathcal{DC}^*$, thus $\mathtt{II}$ can move from $\mathsf{del}{:}R_d(\bar{x}_d)$ to $R'(\bar{x}')$ which then must be won in n–1 rounds, thus it is already deleted when executing $U_{\triangleright,n-1}$. Thus, no *ric* of the form `ON DELETE NO ACTION` is violated when executing $U_{\triangleright,n}$.

– Let E_i be some enumeration of $U_{\triangleright,i}$. $(E_1, E_2, \ldots)$ can be executed sequentially and at no stage any *ric* is violated.

- Let $R(\bar{x})$ be won or drawn. Then there is a user request $\mathsf{del}{:}R_d(\bar{x}_d)$ where $\mathtt{I}$ can move to (i.e., $(R(\bar{x}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$), which is also won or drawn. Thus, when executing $\mathsf{del}{:}R_d(\bar{x}_d)$, $R(\bar{x})$ is deleted. Since all tuples $R'(\bar{x}')$ s.t. $(R'(\bar{x}'), R(\bar{x})) \in \mathcal{DC} \cup \mathcal{DN}$ are also won or drawn (since $\mathtt{II}$ can move from $R_d(\bar{x}_d)$ to $R'(\bar{x}')$), they will also be deleted. Thus, no *ric* `ON DELETE CASCADE/NO ACTION` is violated.
- A tuple $R(\bar{x})$ is lost in n rounds if either

– $(n = 0)$ there is no user request $\mathsf{del}{:}R_d(\bar{x}_d)$ s.t. $(R(\bar{x}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$, i.e., the deletion of $R(\bar{x})$ is unfounded, or

– $(n > 0)$ every user request $\mathsf{del}{:}R_d(\bar{x}_d)$ s.t. $(R(\bar{x}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$ is lost in $\leq n$ rounds, i.e., either $\mathtt{II}$ can move from $\mathsf{del}{:}R_d(\bar{x}_d)$ to $\mathsf{restricted}$ (in this case, by Lemma 3(2), $\mathsf{del}{:}R_d(\bar{x}_d)$ is forbidden), or there is some tuple $R'(\bar{x}')$ s.t. $\mathtt{II}$ can move from $\mathsf{del}{:}R_d(\bar{x}_d)$ to $R'(\bar{x}')$ and which is lost in $\leq n$–1 rounds. By induction hypothesis, $R'(\bar{x}')$ cannot be deleted, but by Lemma 3(3), it must be deleted if $R(\bar{x})$ is be deleted. Thus, $R(\bar{x})$ cannot be deleted. ■

Theorem 6 (Correctness)
The game-theoretic characterization is correct wrt. the abstract semantics:

- $U_w := \{u \in U_\triangleright \mid u \textit{ is won}\}$ *and* $U_{w,d} := \{u \in U_\triangleright \mid u \textit{ is won or drawn}\}$ *are admissible,*
- $U_{w,d} = U_{\max}$*,*
- $\Delta(U_w) = \{\mathit{del}{:}R(\bar{x}) \mid R(\bar{x}) \textit{ is won}\}$ *and*
  $\Delta(U_{\max}) = \Delta(U_{w,d}) = \{\mathit{del}{:}R(\bar{x}) \mid R(\bar{x}) \textit{ is won or drawn}\}$*.*

4 Equivalence and Correctness

We show that the logical characterization is equivalent to the game-theoretic one. Thus, the correctness of the logical characterization reduces to the correctness of the game-theoretic one proven above.

4.1 Well-Founded Semantics

The *alternating fixpoint computation* (AFP) is a method for computing the well-founded model based on successive rounds [VG93]. This characterization finally leads to an algorithm for determining the maximal admissible subset of a given set $U_\triangleright$ of user requests. We introduce AFP using
Statelog, a state-oriented extension of Datalog which allows to integrate active and deductive rules [LML96]. It can be seen as a restricted class of logic programs where every predicate contains an additional distinguished argument for *state*

terms of the form $[S{+}k]$. Here, S is the distinguished *state variable* ranging over $\mathbb{N}_0$. Statelog rules are of the form

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

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 $i \in \{1, \ldots, n\}$.

In Statelog, AFP is obtained by attaching state terms to the given non-stratified program P such that all positive literals refer to $[S{+}1]$ and all negative literals refer to $[S]$. The resulting program P_{AFP} computes the alternating fixpoint of P:

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

$\%\ R_C.\mathbf{F}{\rightarrow}R_P.\mathbf{K}$ *ON DELETE CASCADE:*

$[S{+}1]\ \mathsf{req\_del}{:}R_C(\bar{X}) \leftarrow R_C(,\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}],\ [S{+}1]\ \mathsf{req\_del}{:}R_P(\bar{Y}).$ $\quad (DC_1^A)$

$[S{+}1]\ \mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow\ R_P(\bar{Y}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}],\ [S{+}1]\ \mathsf{blk\_del}{:}R_C(\bar{X}).$ $\quad (DC_2^A)$

$\%\ R_C.\mathbf{F}{\rightarrow}R_P.\mathbf{K}$ *ON DELETE RESTRICT:*

$[S{+}1]\ \mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow\ R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}].$ $\quad (DR^A)$

$\%\ R_C.\mathbf{F}{\rightarrow}R_P.\mathbf{K}$ *ON DELETE NO ACTION:* $\quad (DN^A)$

$[S{+}1]\ \mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow\ R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}],\ [S]\ \neg\mathsf{req\_del}{:}R_C(\bar{X}).$

P_{AFP} is locally stratified, thus there is a unique *perfect model* [Prz88] $\mathcal{M}_{AFP}$ of $P_{AFP} \cup D \cup U_{\triangleright}$. $\mathcal{M}_{AFP}$ mimics the alternating fixpoint computation of $\mathcal{W}_{RA}$: 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 or undefined atoms. The *final state* n_f of the computation is reached if $\mathcal{M}[2n_f] = \mathcal{M}[2n_f{+}2]$. Then, for all relations R, the truth value of atoms $R(\bar{x})$ in $\mathcal{W}_{RA}$ can be determined from $\mathcal{M}_{AFP}$ as follows:

$$\mathcal{W}_{RA}(\mathsf{req\_del}{:}R(\bar{x})) = \begin{cases} \textit{true} & \textit{if } \mathcal{M}_{AFP} \models [2n_f]\ \mathsf{req\_del}{:}R(\bar{x})\ , \\ \textit{undef} & \textit{if } \mathcal{M}_{AFP} \models [2n_f]\ \neg\mathsf{req\_del}{:}R(\bar{x}) \wedge \\ & \qquad [2n_f{+}1]\ \mathsf{req\_del}{:}R(\bar{x})\ , \\ \textit{false} & \textit{if } \mathcal{M}_{AFP} \models [2n_f{+}1]\ \neg\mathsf{req\_del}{:}R(\bar{x})\ . \end{cases}$$

Theorem 7 (Equivalence)
The well-founded model is equivalent to the game-theoretic characterization:

- $R(\bar{x})$ *is won/lost/drawn* $\Leftrightarrow$ $\mathcal{W}_{RA}(\mathsf{req\_del}{:}R(\bar{x})) = \textit{true}/\textit{false}/\textit{undef}$.

PROOF The proof is based on a lemma which is easy to prove from Lemma 4:

Lemma 8

- *I wins at* $R(\bar{x})$ *within* $\leq n$ *rounds iff* $\mathcal{M}_{AFP} \models [2n]$ *req_del:*$R(\bar{x})$.
- *II wins at* $R(\bar{x})$ *within* $\leq n$ *rounds iff* $\mathcal{M}_{AFP} \models [2n{+}1]\ \neg$*req_del:*$R(\bar{x})$. □

From this, Theorem 7 follows directly: The n^{th} overestimate excludes deletions provably non-admissible in n rounds, whereas the n^{th} underestimate contains all deletions which can be proven in n rounds. Thus, there is an n such that $\mathcal{M}_{AFP} \models [2n]\ \mathsf{req\_del}{:}R(\bar{x})$ iff $\mathcal{W}_{RA}(\mathsf{req\_del}{:}R(\bar{x})) = \textit{true}$, and there is an n such that $\mathcal{M}_{AFP} \models [2n{+}1]\ \neg\mathsf{req\_del}{:}R(\bar{x})$ iff $\mathcal{W}_{RA}(\mathsf{req\_del}{:}R(\bar{x})) = \textit{false}$.

The game is drawn at $R(\bar{x})$ if for every tuple $R'(\bar{x}')$ which $\mathtt{II}$ chooses, $\mathtt{I}$ can find a user request which deletes it, and conversely, $\mathtt{II}$ has a witness against each of those user requests. Thus, no player has a "well-founded" proof for or against deleting those tuples. ■

With Theorem 6, the correctness of the logic programming formalization follows:

Theorem 9 (Correctness)
The logic programming characterization is correct wrt. the abstract semantics:

- $U_t := \{\mathit{del}{:}R(\bar{x}) \in U_\triangleright \mid \mathcal{W}_{RA}(\mathit{req\_del}{:}R(\bar{x})) = \mathit{true}\}$ *and* $U_{t,u} := \{\mathit{del}{:}R(\bar{x}) \in U_\triangleright \mid \mathcal{W}_{RA}(\mathit{req\_del}{:}R(\bar{x})) \in \{\mathit{true}, \mathit{undef}\}\}$ *are admissible,*
- $U_{t,u} = U_{\max}$, *and*
- $\Delta(U_{\max}) = \Delta(U_{t,u}) = \{\mathit{del}{:}R(\bar{x}) \mid \mathcal{W}_{RA}(\mathit{req\_del}{:}R(\bar{x})) \in \{\mathit{true}, \mathit{undef}\}\}$.

In the following section, it is shown that the maximal admissible subset of $U_\triangleright$, $U_{t,u}$, also corresponds to a total semantics of P.

4.2 Stable Models

The undefined atoms in the well-founded model leave some scope for further interpretation. This is carried out by *stable models*:

Definition 4 (Stable Model) [GL88] Let M_P denote the minimal model of a positive program P. Given an interpretation I, and a ground-instantiated program P, P/I denotes the reduction of P wrt. I, i.e., the program obtained by replacing every negative literal of P by its truth-value wrt. I. An interpretation I is a *stable model* if $M_{P/I} = I$. □

Every stable model $\mathcal{S}$ extends the well-founded model $\mathcal{W}$ wrt. true and false atoms: $\mathcal{S}^{true} \supseteq \mathcal{W}^{true}$, $\mathcal{S}^{false} \supseteq \mathcal{W}^{false}$. Not every program has a stable model.

Theorem 10 *Let* $\mathcal{S}_{RA}$ *be defined by*

$$\begin{aligned}\mathcal{S}_{RA} := D \cup U_\triangleright &\cup \{\mathit{req\_del}{:}R(\bar{x}) \mid \mathcal{W}_{RA}(\mathit{req\_del}{:}R(\bar{x})) \in \{\mathit{true}, \mathit{undef}\}\} \\ &\cup \{\mathit{blk\_del}{:}R(\bar{x}) \mid \mathcal{W}_{RA}(\mathit{blk\_del}{:}R(\bar{x})) = \mathit{true}\}\ .\end{aligned}$$

Then, $\mathcal{S}_{RA}$ *is a total stable model of* $P_{RA} \cup D \cup U_\triangleright$.

$\mathcal{S}_{RA}$ is the "maximal" stable model in the sense that it contains all delete requests which are true in some stable model. Consequently, deletions have priority over blockings (cf. Example 2).

Theorem 11 (Correctness) *Let* $\mathcal{S}$ *be a stable model of* $P_{RA} \cup D \cup U_\triangleright$. *Then*

- $U_{\mathcal{S}} := \{\mathit{del}{:}R(\bar{x}) \mid \mathcal{S} \models \mathit{req\_del}{:}R(\bar{x})\} \cap U_\triangleright$ *is admissible and* $\Delta(U_{\mathcal{S}}) = \{\mathit{del}{:}R(\bar{x}) \mid \mathcal{S} \models \mathit{req\_del}{:}R(\bar{x})\}$.
- $U_{\max} = U_{\mathcal{S}_{RA}}$ *and* $\Delta(U_{\max}) = \{\mathit{del}{:}R(\bar{x}) \mid \mathcal{S}_{RA} \models \mathit{req\_del}{:}R(\bar{x})\}$.

Proof *Foundedness*: follows directly from the fact that $\mathcal{S}$ is stable (unfounded $\mathsf{req\_del}{:}R(\bar{x})$ would not be stable).
Completeness: For every *ric* $R_C.\mathbf{F} \to R_P.\mathbf{K}$ `ON DELETE CASCADE`, if $\mathcal{S} \models R_C(\bar{x}) \wedge \mathsf{req\_del}{:}R_P(\bar{y}) \wedge \bar{x}[\mathbf{F}] = \bar{y}[\mathbf{K}]$, then, due to (DC_1), $\mathcal{S} = M_{P/\mathcal{S}} \models \mathsf{req\_del}{:}R_C(\bar{x})$.

Feasibility: Suppose a *ric* $R_C.\mathbf{F}{\rightarrow}R_P.\mathbf{K}$ `ON DELETE RESTRICT` or $R_C.\mathbf{F}{\rightarrow}R_P.\mathbf{K}$ `ON DELETE NO ACTION` would be violated: Then $\mathcal{S} \models \mathsf{req\_del}{:}R_P(\bar{y}) \wedge R_C(\bar{x}) \wedge \bar{x}[\mathbf{F}] = \bar{y}[\mathbf{K}]$ (for `NO ACTION` also $\mathcal{S} \models \neg\mathsf{req\_del}{:}R_C(\bar{x})$), and thus because of (DR) resp. (DN), $\mathcal{S} = M_{P/\mathcal{S}} \models \mathsf{blk\_del}{:}R_P(\bar{y})$. Thus, by (DC_2), for the founding delete request $\mathsf{del}{:}R(\bar{z})$, $\mathcal{S} \models \mathsf{blk\_del}{:}R(\bar{z})$, and by (I), $\mathcal{S} \models \neg\mathsf{req\_del}{:}R(\bar{z})$ which is a contradiction to the assumption that $\mathsf{del}{:}R(\bar{z})$ is the founding delete request.
$\Delta_{\mathcal{S}} \subseteq \Delta(U_{\mathcal{S}})$ follows from foundedness, and $\Delta_{\mathcal{S}} \supseteq \Delta(U_{\mathcal{S}})$ follows from completeness. ■

5 A Procedural Translation

Another, more "algorithmic" implementation in Statelog is obtained by "cutting" the cyclic dependency at one of the possible points, i.e., at the rules (I) and (DN).

Cutting in (DN) implements the definition of $\mathcal{S}_{RA}$, corresponding to the observation that $\mathcal{S}_{RA}$ takes exactly the blockings from the underestimate and the internal delete requests from the overestimate.
The rules (DC_1), (DC_2) and (DR) are already local rules:

$$\begin{array}{lr}
[S]\ \mathsf{req_del}{:}R_C(\bar{X}) \leftarrow\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}],\ [S]\ \mathsf{req_del}{:}R_P(\bar{Y}). & (DC_1^S)\\
[S]\ \mathsf{blk_del}{:}R_P(\bar{Y}) \leftarrow\ R_P(\bar{Y}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}],\ [S]\ \mathsf{blk_del}{:}R_C(\bar{X}). & (DC_2^S)\\
[S]\ \mathsf{blk_del}{:}R_P(\bar{Y}) \leftarrow\ R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}]. & (DR^S)
\end{array}$$

The rule (I) is also translated into a local rule:

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

(DN) incorporates the state leap and is augmented to a *progressive* rule (DN^S):

$$[S{+}1]\ \mathsf{blk\_del}{:}R_P(\bar{Y}) \leftarrow R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\mathbf{F}] = \bar{Y}[\mathbf{K}], [S]\ \neg\mathsf{req\_del}{:}R_C(\bar{X}).$$

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

P_S is *state-stratified*, which implies that it is locally stratified, so there is a unique perfect model $\mathcal{M}_S$ of $P_S \cup D \cup U_{\triangleright}$. The state-stratification $\{\mathsf{blk\_del}{:}R\} \prec \{\mathsf{req\_del}{:}R\}$, mirrors the stages of the algorithm: First, the blockings are computed by (DN^S) (the only progressive rule; for the initial state, this rule does not fire) and (DR^S), the induced blockings are derived by (DC_2^S), also determining the blocked user delete requests. The remaining user delete requests raise internal delete requests (I^S) which are cascaded by (DC_1^S). From these, the resulting blockings for the next iteration are computed.

Lemma 12 *$\mathcal{M}_{AFP}$ corresponds to $\mathcal{M}_S$ as follows:*

1. $\mathcal{M}_{AFP} \models [2n]$ *blk_del:*$R(\bar{x}) \ \Leftrightarrow\ \mathcal{M}_S \models [n]$ *blk_del:*$R(\bar{x})$.
2. $\mathcal{M}_{AFP} \models [2n{+}1]$ *req_del:*$R(\bar{x}) \ \Leftrightarrow\ \mathcal{M}_S \models [n]$ *req_del:*$R(\bar{x})$. □

PROOF P_S and P_{AFP} differ in the rules (I^S) and (I^A): In every iteration, P_S takes the blockings from the underestimate and the delete requests from the overestimates, resulting in $\mathcal{S}_{RA}$. ■

Theorem 13 (Termination) *For every database D and every set $U_{\triangleright}$ of user delete requests, the program reaches a fixpoint, i.e., there is a least $n_f \leq |U_{\triangleright}|$, s.t. $\mathcal{M}_S[n_f] = \mathcal{M}_S[n_f{+}1]$.*

PROOF A fixpoint is reached if the set of blocked user delete requests becomes stationary. Since this set is nondecreasing, there are at most $|U_{\triangleright}|$ iterations. ■

From Lemma 12 and Theorem 10, the correctness of P_S follows:

Theorem 14 (Correctness)
The final state of $\mathcal{M}_S$, $\mathcal{M}_S[n_f]$, represents $U_{\max}$ and $\Delta(U_{\max})$:

- $\mathcal{M}_S[n_f] = \mathcal{S}_{RA}$,
- $U_{\max} = \{\textsf{del:}R(\bar{x}) \mid \mathcal{M}_S[n_f] \models \textsf{req\_del:}R(\bar{x})\} \cap U_{\triangleright}$, *and*
- $\Delta(U_{\max}) = \{\textsf{del:}R(\bar{x}) \mid \mathcal{M}_S[n_f] \models \textsf{req\_del:}R(\bar{x})\}$.

5.1 Implementation in a Procedural Programming Language

The Statelog formalization P_S can be easily translated into the following algorithm $\mathbf{Alg}_S$:

Input: A consistent database D and a set $U_{\triangleright}$ of user delete requests.
$B :=$ {all blockings which result from `ON DELETE RESTRICT` triggers}.

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 cascading delete requests $U_{\triangleright}$ which are not blocked: $U^* := (U_{\triangleright} \setminus B^*)^*$.
3. Add to B all blockings which are issued by `ON DELETE NO ACTION` triggers from tuples not in U^*, i.e., which are not requested for deletion.
4. **If** $B \setminus B^* \neq \emptyset$ **then goto** 1 **else** execute requests from U^*.

Output: The new consistent database after executing $U_{\max}$ and the sets $U_{\max}$ of committed and $U_{\triangleright} \setminus U_{\max}$ of aborted user requests.

Initially, it is assumed that there are only those blockings which result directly from `ON DELETE RESTRICT` triggers. Then, blockings are propagated upwards the `ON DELETE CASCADE` chains, finally blocking the triggering user requests. 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. In the next step, all blockings are computed which are caused by `ON DELETE NO ACTION` triggers from tuples which are not reachable via cascaded deletions. These steps are repeated until a fixpoint is reached. Observe that each iteration corresponds to the evaluation of a query with PTIME data complexity. Moreover, since the fixpoint is reached after at most $|U_{\triangleright}|$ iterations (Theorem 13), the overall algorithm also has polynomial data complexity.

Theorem 15 *Algorithm $\mathbf{Alg}_S$ is correct: $U_{\max} = U^* \cap U_{\triangleright}$ and $\Delta(U_{\max}) = U^*$.*

PROOF In the n^{th} iteration, $B^* = \{\mathsf{blk\_del}{:}R(\bar{x}) \mid \mathcal{M}_S \models [n]\ \mathsf{blk\_del}{:}R(\bar{x})\}$, and $U^* = \{\mathsf{req\_del}{:}R(\bar{x}) \mid \mathcal{M}_S \models [n]\ \mathsf{req\_del}{:}R(\bar{x})\}$. ∎

For given D, $U_\triangleright$, and RA, the above algorithm computes the maximal subset $U_{\max}$ of $U_\triangleright$ which can be executed without violating any *ric*, and the set U^* of internal deletions which are induced by it. In case $U_\triangleright$ is not admissible, $U_\triangleright \setminus U_{\max}$ contains the rejected update requests, and by following the chains of blockings from them, the tuples which cause the rejection can be determined. Additionally, by investigating the stages of the algorithm, it can be determined if the blocking is due to the rejection of another request.

References

[CPM96] R. Cochrane, H. Pirahesh, and N. Mattos. Integrating Triggers and Declarative Constraints in SQL Database Sytems. In *Proc. VLDB*, pp. 567–578, Mumbai (Bombay), India, 1996.

[Dat90] C. Date. *Relational Database Writings 1985-1989*. Addison-Wesley, 1990.

[DD94] C. Date and H. Darwen. *A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL*. Addison-Wesley, 1994.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. ICLP*, pp. 1070–1080, 1988.

[Hor92] B. M. Horowitz. A Run-Time Execution Model for Referential Integrity Maintenance. In *Proc. Intl. Conf. on Data Engineering*, pp. 548–556, 1992.

[ISO92] ISO/IEC JTC1/SC21. Information Technology – Database Languages – SQL2, July 1992. ANSI, 1430 Broadway, New York, NY 10018.

[ISO95] ISO/ANSI Working draft. *Database Languages – SQL3*, October 1995.

[LML96] B. Ludäscher, W. May, and G. Lausen. Nested Transactions in a Logical Language for Active Rules. In *Proc. Intl. Workshop on Logic in Databases (LID)*, LNCS 1154, pp. 196–222, 1996. Springer.

[LML97a] B. Ludäscher, W. May, and G. Lausen. Referential Actions as Logical Rules. In *Proc. PODS'97*, pp. 217–224, 1997.

[LML97b] B. Ludäscher, W. May, and G. Lausen. Triggers, Games, and Stable Models. Technical report, Institut für Informatik, Universität Freiburg, 1997.

[LMR96] B. Ludäscher, W. May, and J. Reinert. Towards a Logical Semantics for Referential Actions in SQL. In *Proc. 6th Intl. Workshop on Foundations of Models and Languages for Data and Objects: Integrity in Databases*, Dagstuhl, Germany, 1996.

[Mar94] V. M. Markowitz. Safe Referential Integrity and Null Constraint Structures in Relational Databases. *Information Systems*, 19(4):359–378, 1994.

[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*, pp. 191–216. Morgan Kaufmann, 1988.

[Rei96] J. Reinert. Ambiguity for Referential Integrity is Undecidable. In *Constraint Databases and Applications*, LNCS 1034, pp. 132–147. Springer, 1996.

[VG93] A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

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