Understanding the Global Semantics of Referential Actions using Logic Rules

WOLFGANG MAY
Institut für Informatik, Universität Freiburg, Germany
and
BERTRAM LUDÄSCHER
San Diego Supercomputer Center, University of California San Diego, USA

Referential actions are specialized triggers for automatically maintaining referential integrity in databases. While the *local effects* of referential actions can be grasped easily, it is far from obvious what the *global semantics* of a set of interacting referential actions should be. In particular, when using procedural execution models, ambiguities due to the execution ordering can occur. No *global, declarative* semantics of referential actions has yet been defined.

We show that the well-known logic programming semantics provide a natural *global* semantics of referential actions that is based on their *local* characterization: To capture the global meaning of a set *RA* of referential actions, we first define their abstract (but non-constructive) *intended semantics*. Next, we *formalize RA* as a *logic program* P_{RA}. The declarative, logic programming semantics of P_{RA} then provide the constructive, *global* semantics of the referential actions. So, we do not *define* a semantics for referential actions, but we show that there *exists* a unique *natural* semantics if one is ready to accept (i) the intuitive local semantics of local referential actions, (ii) the formalization of those and of the local "effect-propagating" rules, and (iii) the well-founded or stable model semantics from logic programming as "reasonable" global semantics for local rules.

We first focus on the subset of referential actions for deletions only. We prove the equivalence of the logic programming semantics and the abstract semantics via a game-theoretic characterization, which provides additional insight into the meaning of interacting referential actions. In this case a *unique maximal admissible solution exists*, computable by a PTIME algorithm.

Second, we investigate the general case—including modifications. We show that in this case there can be *multiple maximal admissible subsets* and that all maximal admissible subsets can be characterized as *3-valued stable models* of P_{RA}. We show that for a given set of user requests, in the presence of referential actions of the form `ON UPDATE CASCADE`, the admissibility check and the computation of the subsequent database state, and (for non-admissible updates) the derivation of debugging hints all are in PTIME. Thus, full referential actions can be implemented efficiently.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*Data Models*; H.2.4 [**Database Management**]: Systems—*Relational Databases*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and Constraint Programming*

Authors' addresses: W. May, Institut für Informatik, Universität Freiburg, D-79110 Freiburg, Germany; email: may@informatik.uni-freiburg.de; B. Ludäscher, San Diego Supercomputer Center, UCSD, La Jolla, CA 92093-0505; email: ludaesch@sdsc.edu.

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Database theory, game theory, logic programming, referential integrity, referential actions, relational databases

1. INTRODUCTION

The notion of integrity constraints and their automated maintenance has been an important research issue since the early days of relational databases [Codd 1970; Hammer and McLeod 1975; Eswaran 1976]. Integrity constraints in general, and *referential integrity constraints* (*ric*s) in particular, are central concepts of database models and they are frequently used in real world applications. Many approaches use *ECA (event-condition-action)-rules* for monitoring and enforcing integrity constraints: if some event (here, an update) occurs, a set of actions is executed internally. Rules for integrity maintenance [Ceri and Widom 1990] have been a starting point for the area of *active databases* and their impact is documented by a recent 10-Year Paper Award [Ceri et al. 2000]. Triggers, a special kind of ECA-rules, have been part of database systems from the beginning [Eswaran 1976] and are included in the SQL2 and SQL3 standards [ANSI/ISO 1992a, 1999]. However, even today, researchers still complain about the difficulties in understanding the "subtle behavior" of multiple triggers acting together [Ceri et al. 2000] and the limited progress that has been made [Cochrane et al. 1996].

For enforcing referential integrity, each referential integrity constraint can be associated with *referential actions* (*rac*s) that provide a *declarative*, *local* specification of how to automatically enforce referential integrity, thereby relieving the user from the burden of enumerating all induced updates that arise from an initial user request $U_{\triangleright}$. While ECA-rules and triggers are *procedural* means to enforce integrity by locally reacting on an event, the idea behind referential integrity and referential actions is a global one: the semantics of referential actions is given *declaratively* in terms of *local* actions, but with a *global* notion of "original state" (before the update) and "final state" (after the update and all induced changes) in mind.

Date and Darwen [1994] and Date [1990] report the problem of unpredictable behavior when realizing *rac*s based on SQL triggers, that is, ambiguities in determining the set of updates on the database and the final database state in certain situations. The solution of the SQL2 standard [ANSI/ISO 1992a] (for a more complete overview of related work, see Section 6) described a *procedural* semantics that was subject to anomalies: since referential actions were executed at the same time as the parent was updated, the outcome depends on the order in which rows are modified or constraints are applied [Cochrane et al. 1996]. Markowitz [1994] presents safeness conditions that aim at avoiding ambiguities at the schema level. However, as shown in Reinert [1996], it is in general undecidable whether a database schema with *rac*s is ambiguous.

Horowitz [1992] proposes a *marking algorithm* in the style of a fixpoint computation to define a *global* semantics that avoids these anomalies. An extension of this semantics was later incorporated into the SQL3 standard

[ANSI/ISO 1999]. Nevertheless, the *global* semantics is still—about 30 years after the definition of the idea of referential integrity—given by a complex, not very intuitive *procedural* algorithm, and only few commercial database systems support referential actions in their full extent.

In contrast to the majority of the work on this topic, we present a framework for maintenance of referential integrity based on *logic rules*. The user's original updates, together with the induced updates, yield a set of updates to the database that must be applied *instead* of (only) the original updates. Thus, the notions of "before" or "after" should be understood in a global, all-or-nothing manner without considering intermediate states for *defining* the meaning of updates.

The logic programming characterization given here demonstrates that the problem of *rac*s can be solved by specifying *local* behavior in manageable parts, and exploiting the fact that the well-known logic programming semantics define an unambiguous, reasonable[1] *global* semantics of the "puzzle" that results from the interaction of multiple *ric*s and *rac*s. Moreover, this semantics can be computed efficiently and thus can be implemented in actual database systems. In contrast, the process of developing a procedural characterization as has been done in ANSI/ISO [1992a]; Horowitz [1992]; Markowitz [1994]; Cochrane et al. [1996] and finally ANSI/ISO [1999] required about 20 years, including intermediate solutions that have been proven to be incomplete and/or incorrect.

The Problem. We consider the following problem: Given a database instance D, a set of user-defined update requests $U_{\triangleright}$, and a set RA of *rac*s, find the set of updates Δ that (i) is complete with respect to $U_{\triangleright}$, (ii) preserves referential integrity in the new database state D', and (iii) reflects the intended meaning of RA, that is, *how* referential integrity should be enforced.

$U_{\triangleright}$ can be given as a single (set-oriented) statement, or as a sequence of statements (including activated triggers) if such behavior is supported by the underlying transaction model. In some examples we construct specific sets $U_{\triangleright}$ in order to illustrate certain interferences. Sometimes, the updates in $U_{\triangleright}$ could be induced via cascading from a single statement, sometimes not. Our investigations not only provide the solution to a practically relevant problem, but also address the basic research problem of interacting *ric*s and *rac*s—the search for the above set Δ.

In case that no such Δ exists and $U_{\triangleright}$ is rejected, we investigate *maximal admissible* subsets of $U_{\triangleright}$, and derive hints as to where the problems are located and how they can possibly be solved. Assume that $U_{\triangleright}$ has been collected by several statements (e.g., a subtransaction) that are intended to do a certain amount of work. In case that it is rejected, something in the database (or its specification) is obviously inconsistent with the intended behavior. This points to problems in the design either of the database schema with its *ric*s and *rac*s, or in the programming of the subtransaction, or the contents of the database in the current situation is not as intended. Here, the additional information from

[1]Dix [1995] formally defines this notion using very general principles.

the analysis of the *ric*s and *rac*s can be helpful for identifying the problem:

—the schema may be flawed, that is, there are missing *rac*s (e.g., a forgotten `CASCADE`, or a `RESTRICT` where a `NO ACTION` would have been correct),
—the definition of the subtransaction is incomplete (e.g., it should generate some more update requests),
—the schema and the subtransaction are correct, but the database state is incorrect due to an incomplete definition of an earlier transaction.

Our semantics can give useful hints about where *exactly* the problem is located, that is, which *ric*s are violated, and which tuples cause the problem.

Contributions. From a theoretical perspective, we aim at providing a better understanding of referential actions: We formalize the semantics of referential integrity constraints and referential actions as a logic program P_{RA} where

(1) the *local behavior* of an individual *rac* $ra \in RA$ is precisely specified, and can be understood by solely looking at the corresponding rules $P_{ra} \subseteq P_{RA}$,
(2) the (local) *interaction* between different update requests is precisely defined by certain other rules,
(3) the *global behavior* is precisely specified and understandable from the declarative logic programming semantics. So, we do not *define* a semantics for *rac*s, but we show that there *exists* a unique *natural* semantics if one is ready to accept the local semantics (1) and (2), and the logic programming semantics, that is, the well-founded model and the stable model as "reasonable" semantics.

The logic-based characterization not only provides a natural semantics for referential actions, but also leads to efficient procedures for handling referential actions in actual database systems.

From a practical perspective, we give polynomial time constructive characterizations for the following tasks:

—checking if the set $U_{\triangleright}$ is admissible, and
—in case that it is, computing the set of updates to be accomplished (implying that `ON UPDATE CASCADE` which is currently not supported in most commercial database systems can be implemented efficiently), and
—in case that it is *not*, giving hints what updates, *ric*s, *rac*s, and tuples caused the problem.

The (complex) rule systems are *not* intended to be used by the designers of the *ric*s and *rac*s; they encode the *local* semantic conditions that are naturally induced by the application domain, and that the application designer has (correctly) in mind during the development process—so he implicitly relies on a "correct" global semantics that is ensured by our characterization. The use of the logic programming characterization in our approach is (i) in case of deletions for deriving a procedural algorithm and proving its correctness, and (ii) in case of modifications it can serve as a declarative *internal* implementation of referential actions (we do not derive a procedural algorithm for this case

as it would be excessively complex without providing any insight beyond the logic programming semantics). In both cases, if $U_{\triangleright}$ is admissible, the database silently executes it. Otherwise the problems are presented to the application developer or the user in terms of tuples and foreign key constraints. Thus, the user is not bothered with the actual formalization, presented here as a "black box"—in the same way as he is not required to know about the details of the algorithm given in Horowitz [1992] and ANSI/ISO [1999].

Audience. Thus, the audience is not the typical SQL application programmer from whose point of view the *local* effects of *rac*s should be enough to design an application, provided that the underlying DBMS assigns the correct *global* semantics to his specification. The relevance of our results for him is the formal characterization of the "built-in" correctness of lifting his local specification to the global behavior of the database system: A database that acts according to the described global semantics implements the application programmer's intensions to the largest extent possible, based on his database schema and referential actions. Then, the programmer can rely on the correct interaction of referential actions. In case that an application raises nonadmissible updates, something in this specification must be wrong, and the database system—supported by the semantics—can give hints as to where the problems come from. Thus, from the point of view of application programmers, the possible features of an implementation (especially, the conclusions that are drawn in case of rejected updates) based on our approach can be useful for improving their database design.

For implementors of database systems, the presented algorithms and methods for computing the induced set Δ of internal updates could be of interest. Moreover, the possibility of reacting on rejected updates by deriving hints for the application designer as to how to cure the problems can be useful for providing enhanced error reporting messages to the user (see Sections 3.7 and 5). The logical basis provided by three-valued logic and stable models facilitates more flexible investigations than the procedural fixed-point algorithms that are given in the SQL standards (and still only incompletely implemented by database systems).

The theoretical body of the paper—the details of the logic-based specifications and the game-theoretic analysis that lead to a *declarative, model-theoretic* characterization of the *global* semantics of referential actions—is directed at researchers studying database fundamentals and theory. For this audience, the paper provides (i) an elegant formal and "natural" (i.e., declarative) semantics of referential actions, and (ii) an application of theoretical concepts to a practically relevant problem that exhibits several levels of complexity that have to be handled by appropriate theoretical means.

Scope. Although our models and terminology are based on the relational model, the underlying issues of a "justification-based," declarative semantics as proposed in this paper, are independent of the particular database model chosen. For example, extensions to the (very limited) notion of referential integrity in XML (ID/IDREF) have been proposed [Fan and Siméon 2000], or are

included as integral parts of new XML standards, like XML SCHEMA [2000]. It should be clear that rule-based maintenance of referential integrity in XML will face the same fundamental issues that are explained and resolved by our global semantics.

Relationship with earlier publications. We provide a comprehensive and uniform treatment of our previous work on declarative semantics for referential actions [Ludäscher et al. 1997; Ludäscher and May 1998]. First preliminary results have been reported in Ludäscher et al. [1996b]. In Ludäscher et al. [1997], it is shown that for referential actions (*rac*s) with modifications, it may be intractable to compute *all* maximal admissible solutions (since the interactions may lead to an exponential blow up in the number of solutions). In Ludäscher and May [1998], we restricted the investigation to *rac*s *without modifications*—deletions only. This guarantees the existence of a unique optimal solution, which can be efficiently computed. The present paper not only provides a complete and uniform treatment of our previous results, but extends them in various ways: We present a novel game-theoretic characterization of *rac*s that gives a more abstract account of referential actions for modifications and shows that important aspects of this problem (admissibility of $U_{\triangleright}$, computation the actual set of update operations, and deriving debugging hints) are also in PTIME. Additionally, we explore the *practical* implications for actual relational DBMS that result from the theoretical investigations.

Structure of the paper. The paper is organized as follows: In Section 2, we introduce the basics of referential integrity. Then, we illustrate the problem of ambiguity that arises from the *local* specification of referential actions, and describe the disambiguation strategies of the SQL standard.

In Section 3, we investigate the class of *rac*s without modifications (i.e., deletions only). In Section 3.1, we identify and formalize desirable abstract properties of updates which lead to the intended (albeit non-constructive) global semantics of *rac*s. A constructive definition of this 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 3.4). An algorithm for computing the maximal admissible solution is derived from the logic programming characterization (Section 3.5). So far, Section 3 is based on and extends the previous work [Ludäscher and May 1998]. The correctness of our characterization(s) and of the derived algorithm with respect to the "intended" ECA-style semantics, and the relationship with the SQL3 semantics is shown in Section 3.6. There, we can completely rely on the correctness of the logic programming semantics. The practical consequences of how to debug an application in case a set of updates is rejected are described in Section 3.7.

In Section 4, we extend the investigations to include modifications. We again start by giving an abstract characterization (Section 4.1). In Section 4.2 we associate with every set *RA* of *rac*s a logic program P_{RA} whose rules capture the *local* semantics of modifications with referential actions, and show that the *global* declarative semantics of P_{RA} captures the abstract semantics, and thus

solves the problem in an unambiguous and comprehensive way. In contrast to the restricted deletions-only case, the characterization cannot be reformulated in an efficient algorithm since *stable model* semantics is required for the logical characterization. Sections 4.1 and 4.2 provide a comprehensive treatment of the results of the extended abstract [Ludäscher et al. 1997]. An equivalent game-theoretic characterization that abstracts from some details of the logical characterization is described in Section 4.3. Its details and the proof of the equivalence of all three characterizations can be found in Section B of the *electronic appendix*.

Further results showing the practicability of our approach are developed in Section 5: we show that the following tasks are computable in PTIME and derivable from the well-founded model: (i) a check on whether a user request is admissible, (ii) if so, the computation of the subsequent database state, and (iii) for non-admissible user requests, an approximation of a maximal admissible subset, together with debugging hints. Section 6 reviews related work in the area and concluding remarks can be found in Section 7.

2. REFERENTIAL INTEGRITY

2.1 Notation and Preliminaries

In the following, we introduce the necessary notions of the relational model and calculus, which provide the basic formalism of our paper. We use a *positional* and *unnamed* relational calculus/Datalog-style notation—unlike the relational model with named attributes [Abiteboul et al. 1995]. In the unnamed Datalog-style notation, each argument position of a predicate is (implicitely) associated with an attribute. We follow this convention and regard attributes to be ordered according to their argument positions. This is used when correlating foreign keys with candidate keys.

Definition 2.1 *Relational Schema, Keys.* A relation schema $R(\vec{A})$ consists of a relation name R and a sequence of attributes $\vec{A} = (A_1, \ldots, A_n)$. We identify attribute names A_i of R with the integers $1, \ldots, n$. Given $\vec{A}$, a (possibly reordered) subsequence of $\vec{A}$ (e.g., a key) is a vector $\vec{K} = (A_{i_1}, \ldots, A_{i_k})$ such that $k \leq n$ and $i_{j_1} \neq i_{j_2}$ for $j_1 \neq j_2$. Note that we have to allow that the attributes in $\vec{K}$ may have a different order than in $\vec{A}$.

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

For a relation schema R with attributes $\vec{A}$, a minimal subset $\vec{K}$ of $\vec{A}$ 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 $R.\vec{K}$ has to satisfy the first-order formula φ_{key} for every database instance D:

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

Usually, in database design, for every relation one candidate key is selected to be *the primary key* of the relation. Since the key values uniquely identify a tuple of the corresponding *parent* relation, they can be used in other *child* relations for referring to the parent tuple. The respective attributes of the child relation are then called a *foreign key* of the *child relation*.

In this work, we assume that candidate and foreign keys do not contain null values (considering null values would add much technical effort and problems that are specific to null values, without giving additional insight).

Example 1 *Primary Keys and Foreign Keys.* Consider a database that describes countries and cities as depicted below. There, Name and Code are candidate keys of country (we chose Code to be the primary key). The attribute tuple City(Name, Country) is the primary key of City; the attribute City.Country references the key Country.Code and thus is a foreign key in City. Similar, Country(Capital, Code) references a city (identified by City(Name, Country)), thus the attribute tuple (Capital, Code) is a foreign key of Country (note the change of the order in the foreign key with respect to the original attribute list of Country).

Country			
1:Name	2:Code	3:Capital	4:Area
Germany	D	Berlin	356910
Austria	A	Vienna	83850
Utd. Kingdom	GB	London	244820
⋮	⋮	⋮	⋮

City		
1:Name	2:Country	3:Pop.
Berlin	D	3472009
Munich	D	1244676
Vienna	A	1583000
London	GB	6967500
⋮	⋮	⋮

Definition 2.2 *Referential Integrity Constraints.* A *referential integrity constraint* (*ric*) is an expression of the form

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

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

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

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

Example 2 *Primary Keys and Foreign Keys* (*Cont'd*). Consider again Example 1. There, we have the *ric*s City.Country → Country.Code and Country.(Capital,Code) → City.(Name,Country) or, in the numerical encoding, City.2 → Country.2 and Country.(3,2) → City.(1,2).

Table I. Operations and Possible Repairs

	R_P			R_C		
	ins	del	mod	ins	del	mod
propagate	ok	•	•	–	ok	–
restrict	ok	•	•	•	ok	•
wait	ok	•	•	•	ok	•

ok = *ric* remains satisfied
• = *ric* may be violated, *rac* applicable
– = *ric* may be violated, *rac* not applicable

For example, changing the code GB into UK in Country would violate both *ric*s; this can be remedied by *propagating* the change, that is, applying the same renaming in City. Changing the capital of Germany to Munich would be allowed; changing it to Hamburg would violate the second *ric* (assuming that Hamburg is not stored in the City table). This could be fixed by either inserting a tuple for (Hamburg, Germany) into City, or by renaming, for example, Berlin to Hamburg. This shows that propagation of a modification is not always desired.

Definition 2.3 *Updates.* *Update requests* (*updates*) to a relation R are represented by auxiliary relations $\mathsf{ins}\_R(\bar{X})$, $\mathsf{del}\_R(\bar{X})$, and $\mathsf{mod}\_R(M, \bar{X})$. Here, M is a set of pairs i/c meaning that the i-th attribute of $R(\bar{X})$ should be set to the constant c. We say that a modification $\mathsf{mod}\_R(M_1, \bar{X})$ *subsumes* a modification $\mathsf{mod}\_R(M_2, \bar{X})$ if $M_1 \supseteq M_2$. As a shorthand for $\mathsf{mod}\_R([1/d, 3/e], (a, b, c))$, we sometimes write $\mathsf{mod}\_R(a/d, b, c/e)$.

2.2 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* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ and an update operation on R_P or R_C, a *referential action* (*rac*) defines a *local* operation to be applied to R_C or R_P, respectively. We call this the *locality principle*. The problem with the locality principle is that the intuitive local repairs can lead to complex, "subtle behavior" [Ceri et al. 2000] with different, more or less "reasonable" outcome. Thus, an unambiguous *global* semantics for these local specifications is needed. As mentioned in the introduction, such semantics have been developed in the history of SQL [ANSI/ISO 1992a; Horowitz 1992; Markowitz 1994; Cochrane et al. 1996] over the years, including incomplete and incorrect intermediate solutions, now being specified by a procedural, fixpoint-style algorithm in ANSI/ISO [1999]. In the sequel, we start with a generic, abstract version of *rac*s which is then related to the SQL version in Section 2.4.

The updates insert, delete, and modify can be applied to R_P or R_C, leading to six basic cases. It is easy to see from the logical implication in (φ_{ric}) above that insert R_P and delete R_C cannot introduce a referential integrity violation, while the other four operations can. There are in general three possible strategies as to how problems may be resolved; not all of them are applicable for all operations (cf. Table I):

—propagate: propagate ("cascade") the update along the current *ric* by executing actions on the *other* tuple. This means, to propagate an update at the parent to all children (with respect to the *current* database state), or to propagate an update at the child to the parent.
—restrict: reject an update if it may cause a problem: (i) reject an update on the parent if there exists a child referencing the parent in the *current* (that means, at the moment when the update is executed) database state, and (ii) reject an update on the child if the referenced parent does not exist in the current state.
—wait: no local action is executed. Instead, the referential integrity constraints are checked—together with the other integrity constraints—after the end of a certain unit of work (note that in contrast to propagate and restrict, this can be seen as a kind of *deferred restrict* which is already a *global* strategy).

Each *rac* consists of the *ric* which should be maintained, the triggering update on either the parent R_P or the child R_C, and the "local repair." We use the following notation, which should be self-explanatory:[2]

$R_C.\vec{F} \rightarrow R_P.\vec{K}$ on {del | ins | mod} {parent | child} {propagate | restrict | wait}

2.3 The Problem of Ambiguity

With this *local* specification of behavior where each *rac* triggers an action on every child tuple (with respect to the respective *ric*) when an update to a parent tuple is executed (see, e.g., Dayal [1988]; Eswaran [1976]), some nondeterminism with respect to the outcome of a user operation may occur. If there are different possible final states of a database instance D (depending on the execution order of referential actions), D is called *ambiguous* with respect to the given referential actions.

There are several types of ambiguities leading to potentially different final states that are described in the following. In Section 4.2.6, we show that these ambiguities have a very natural and elegant representation in our framework: "controversial" updates are *undefined* in the well-founded model; the different possible results are characterized by certain stable models.

Example 3 *Diamond.* Consider the database with *rac*s as depicted in Figure 1. Solid arcs point from R_C to R_P, *rac*s are denoted by dashed (propagate) or double (restrict) arcs. Let $U_{\triangleright} = \{\triangleright\text{del\_}R_1(a)\}$ 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 $R_1 \rightsquigarrow R_3 \rightsquigarrow R_4$, the tuple $R_3(a, c)$ cannot be deleted: Since $R_4(a, b, c)$ references $R_3(a, c)$, the *rac* for R_4 restricts the deletion of $R_3(a, c)$. This in turn also blocks the deletion of $R_1(a)$. The user request $\triangleright$del_$R_1(a)$ is rejected, and the database state remains unchanged, that is, $D' = D$.

[2]As can be seen from Table I, not all combinations are meaningful: e.g., it is perfectly reasonable to propagate (cascade) a modification from the parent to the referencing child, but not vice versa.

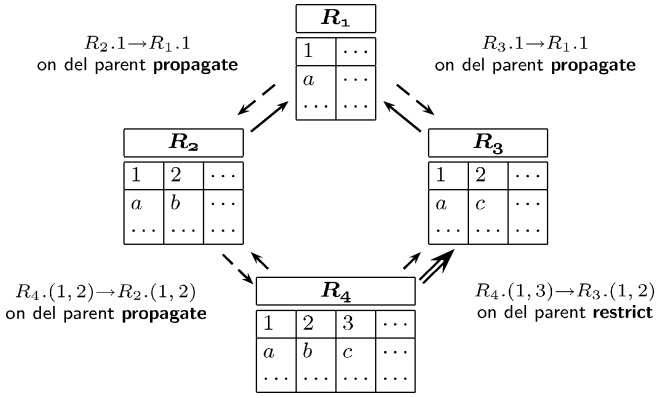

Fig. 1. Database with referential actions.

(2) If execution follows the path $R_1 \rightsquigarrow R_2 \rightsquigarrow R_4$, the tuples $R_2(a, b)$ and $R_4(a, b, c)$ are requested for deletion. Hence, the *rac* for $R_4.(1, 3) \rightarrow R_3.(1, 2)$ can assume that $R_4(a, b, c)$ is deleted, thus no referencing tuple exists in R_4. Therefore, all deletions can be executed, resulting in a new database state $D' \neq D$.

In the above "diamond", when choosing the "right" order of execution, the update is possible, whereas when going the "wrong" way, it is impossible. The reason is that the restrict action looks at the "current" database, and this depends on the order of execution.

This type of ambiguity can be eliminated by specifying that restrictions are always evaluated with respect to the *original* database state instead of the current one (as it is done in SQL, see the following section). However, the situation is more complex for *rac*s of the type wait which have to look at the *final* database state. As it turns out, in the presence of modifications, in general, there are still several "equally justified" final states, each of which has to be considered:

Example 4 *Mutex.* Consider modifications $\triangleright\mathsf{mod}\_R(a/b)$ and $\triangleright\mathsf{mod}\_R(a/c)$.[3] They are mutually exclusive, since they cannot be executed simultaneously. In our logical formalization, both will be *undefined* in the well-founded model. Moreover, there will be *two* stable models, each of which makes one modify request true, and the other false.

Another type of ambiguity may arise due to "self-attacking" requests:

Example 5 *Self-Attack.* Assume a database with *rac*s (again a "diamond" as in Example 3) such that $\triangleright\mathsf{mod}\_R([1/b, 2/c](a, a))$ triggers $\mathsf{mod}\_R_1([1/b], (a))$ and $\mathsf{mod}\_R_2([1/c], (a))$; $\mathsf{mod}\_R_1([1/b], (a))$ triggers $\mathsf{mod}\_R_3([1/b], (a))$, and $\mathsf{mod}\_R_2([1/c], (a))$ triggers $\mathsf{mod}\_R_3([1/c], (a))$. Since the original request $\triangleright\mathsf{mod}\_R([1/b, 2/c], (a, a))$ causes a conflict at R_3, it cannot be executed. On the other hand, no *other* request is in conflict with it, so there is no independent

[3]Here, (as the "$\triangleright$" shows) the modifications come directly from an (already contradictory) user request. However, the Mutex scenario can also occur *indirectly* from a non-contradictory user request.

justification *not* to execute it. Thus, the original request "attacks" itself. In our formalization, there is *no total stable model*.

Example 6 *Virgin Birth.* This example shows that not every update which does *not* violate any *ric* is also reasonable with respect to the intended semantics:

Consider the following database schema: $R_P(1)$ and $R_C(1,2)$ with a *ric* $R_C.1 \rightarrow R_P.1$ on del parent wait and a database instance $R_P(a,b)$, $R_P(d,e)$, $R_C(a,h)$. Suppose the user requests $U_{\triangleright} = \{\triangleright\mathsf{del}\_R_P(a,b), \triangleright\mathsf{mod}\_R_P([1/a], (d,e))\}$. Then, deletion of $R_P(a,b)$ is blocked due to the tuple $R_C(a,h)$.

On the other hand, one can argue that deleting $R_P(a,b)$ is possible, since after modifying $R_P(d,e)$ to $R_P(a,e)$, the child tuple $R_C(a,h)$ gets a new parent, so the *ric* $R_C.1 \rightarrow R_P.1$ remains satisfied.

Here, the *semantical* connection which is encoded in $R_C.1 \rightarrow R_P.1$ would be broken: The child tuple $R_C(a,h)$ gets a new parent *although* it is not modified, and the new parent tuple $R_P(a,e)$ "finds" a new child.

In this situation, $U_{\triangleright} = \{\triangleright\mathsf{del}\_R_P(a,b), \triangleright\mathsf{mod}\_R_P([1/a], (d,e))\}$ is not *feasible* according to our abstract semantics.

The underlying idea for treating this (and related) cases is based on the intended semantics of a database and referential integrity: Each database can be seen as a reference network (akin to the network database model), inducing a reference graph. Thus, each set of updates also defines a mapping between reference graphs. From the semantical point of view, references should only be created when a *child* tuple is inserted or modified. Changes on parent tuples are intended to either preserve references (by updating the child tuple accordingly) or delete references.

The above examples showed that the *local* ECA-style characterization considered in Section 2.2 is ambiguous. This ambiguity is caused by considering the *current* database state for applying referential actions. In the following section, we describe the SQL specification of referential actions that solves this problem on the specification level, but whose implementation aspects still suffered from these problems as long as SQL's trigger functionality was used.

2.4 Referential Actions in SQL and Global Disambiguation Strategies

In SQL [ANSI/ISO 1992a, 1999], referential actions for a referential integrity constraint $R_C.\vec{F} \rightarrow R_P.\vec{K}$ are specified with the definition of the child table. SQL allows referential actions only for modifications at the parent tuple:

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

Insertions and modifications on child tuples are handled in a straightforward way by rejecting updates which aim to generate a child tuple whose

corresponding parent does not exist. In our work, we deliberately exclude SET NULL/DEFAULT actions, since they are a special case of modifications.

In their *abstract* specification in SQL, these strategies correspond to the abstract local strategies described in Section 2.2, solving the problems that are caused by considering the *current* database state in our localized ECA-like characterization:

—CASCADE is the same as our propagate and propagates the update from the parent to the referencing tuples (evaluation is with respect to the *original* database state).

—RESTRICT is similar to restrict, but refers to the database state *before* the beginning of evaluation (instead of the current database state at the time when the update actually occurs): reject an update on the parent if there exists a child referencing it in the *original* database state,

—NO ACTION is the same as wait: no local action is executed. Instead, the referential integrity constraints are checked—together with the other integrity constraints—after the end of a certain unit of work, thus, referring to the database state *after* completing the updates.

Although this semantics is easy to understand on first sight, it is not a direct, *local* semantics that can be implemented straightforwardly by ECA-rules or triggers in the style of Dayal [1988] and Eswaran [1976]. Thus, its detailed specification in the SQL standard and even more, its realization in actual database systems has proven to be problematic.

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 (negative) cyclic dependency in the *global* strategy. Thus, any straightforward implementation via ECA-rules/triggers is bound to fail. On the other hand, logic programming semantics provide a natural solution to this kind of problem and will be used in Sections 3 and 4 for an unambiguous, elegant characterization of the *global* semantics of *rac*s.

The SQL2 standard [ANSI/ISO 1992a]—where RESTRICT did not yet exist—used the "localized," immediate specification for cascading updates: the foreign key values in all referencing tuples were immediately updated when the parent was updated. Date and Darwen [1994] and Date [1990] already report the problem of unpredictable behavior, that is, that the outcome depends on the order in which tuples are updated. The same characterization was given in the upcoming SQL3 drafts (e.g., the 1991 version [ANSI/ISO 1991] cited by Horowitz [1992] and ANSI/ISO [1994]). Concerning these specifications, Horowitz [1992] and Cochrane et al. [1996] complain about unpredictable behavior since the outcome depends on the order in which rows are modified or constraints are applied [Cochrane et al. 1996]. In Horowitz [1992], a marking algorithm for a runtime execution model for referential integrity maintenance is presented for unary keys that does not exhibit these problems.

Markowitz [1994] presents safeness conditions which aim at avoiding ambiguities at the schema level. However, as shown in Reinert [1996], it is in general

undecidable whether a database schema with *rac*s is ambiguous (if D and $U_{\triangleright}$ are given, the problem becomes decidable).

This was the state-of-the-art when we started to investigate a logic-based specification of referential actions in order to provide an unambiguous, natural, and "correct" semantics that mirrors the SQL intension described above for `CASCADE`, `RESTRICT`, and `NO ACTION`.

In the meantime, the SQL3 standard [ANSI/ISO 1999] solved the problem of ambiguous semantics of *rac*s by fixing an operational semantics using a marking algorithm based on Horowitz [1992]:

—effectively perform integrity checking at the end of the statement [includes `NO ACTION`],
—effectively determine and fix matching rows [concerning keys/foreign keys] at the beginning of the statement [("static matching")],
—effectively determine update values at the beginning of the statement (includes marking tuples that will be deleted),
—rollback any attempt to update the same data item [in Horowitz [1992], single attributes] to representationally different values in the same statement,
—perform an update referential action iff the referenced column [Horowitz [1992] was restricted to single-column keys] is updated to a representationally different value,
—effectively perform all deletes at the end of the statement,
—perform a delete referential action if and only if the referenced row has not already been marked for deletion.

Here, "effectively" means that this description specifies the intended semantics; nevertheless, actual implementations can perform actions "on the fly"—if it is guaranteed that this does not violate the above semantics. The actual specification is given in terms of a complex encoding into `BEFORE` triggers whose interactions are hard to understand. The algorithm associates a *global* semantics with referential actions that will be shown to be equivalent to ours. Horowitz [1992] proves correctness and termination of the algorithm.

Such a procedural specification in form of an algorithm deviates far from the localized, ECA-style specification "`ON` *event action*" above, and gives no declarative, easily accessible, semantics of referential actions. Our work shows that the ECA-style specification has an immediate, declarative, "natural" global semantics that is given by the logic programming meta-semantics of rule-based specifications—and that the procedural semantics (developed over about 20 years) coincides with that semantics.

Moreover, in case a set of updates causes referential problems, the transaction is simply aborted. Often in these cases, most of the requested updates are unproblematic, and only one or two are not allowed. Thus, it can be useful to return hints on how to prepare a revised request that realizes the intended changes *and* is accepted by the system, or for debugging the application. We show how to derive such hints from our semantics, and we also sketch how it can be used for deriving suggestions how to correct the behavior of an application in that case.

In Sections 3 (Deletions) and 4 (Modifications), we show how to characterize and qualify the induced semantic problems using different (logical and game-theoretic) characterizations of *rac*s.

3. SEMANTICS OF REFERENTIAL ACTIONS WITH DELETIONS

We have shown that in order to avoid ambiguities and nondeterminism as in Example 3, it is necessary to specify the intended *global* semantics of *rac*s. In this section, we investigate *ric*s $R_C.\vec{F} \to R_P.\vec{K}$ with corresponding *rac*s of the form

$$R_C.\vec{F} \to R_P.\vec{K} \quad \texttt{ON DELETE} \quad \{\texttt{CASCADE} \mid \texttt{RESTRICT} \mid \texttt{NO ACTION}\}$$

according to the above *global* strategies in the SQL sense. For these we present an efficient (PTIME) algorithm that computes the unique solution.

First, we define an abstract, non-constructive semantics that formalizes the SQL notions described in Section 2.4. This semantics then 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 then provides a constructive definition. An equivalent game-theoretic characterization is developed which will be used to prove the correctness of the logic programming semantics with respect to the abstract semantics.

3.1 Abstract Semantics

Let D be a database represented as a set of ground atoms, RA a set of *rac*s, and $U_{\triangleright} = \{\triangleright\mathsf{del}\_R_1(\bar{x}_1), \ldots, \triangleright\mathsf{del}\_R_n(\bar{x}_n)\}$ 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:

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

$\mathcal{DR}$ 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 $U_{\triangleright}$.

Definition 3.1 *Abstract Properties.* Given RA, D, and $U_{\triangleright}$ as above, a set $\Delta = \{\mathsf{del}\_R_1(\bar{x}_1), \ldots, \mathsf{del}\_R_n(\bar{x}_n)\}$ of delete requests is called

—*founded*, if for all $\mathsf{del}\_R(\bar{x}) \in \Delta$, there is a $\triangleright\mathsf{del}\_R'(\bar{x}') \in U_{\triangleright}$ s.t. $(R(\bar{x}), R'(\bar{x}')) \in \mathcal{DC}^*$ (note that here, we need reflexivity for covering $R'(\bar{x}')$ itself),

—*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 (i) $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DR}$ implies $\mathsf{del}\_R_P(\bar{y}) \notin \Delta$, and (ii) $\mathsf{del}\_R_P(\bar{y}) \in \Delta$ 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.

(for individual updates, we also write "$\mathsf{del}\_R(\bar{x})$ is admissible" instead of "$\{\mathsf{del}\_R(\bar{x})\}$ is admissible"; analogous for "founded.")

Foundedness guarantees that all deletions are "justified" by some user request, completeness guarantees that no cascading deletions are "forgotten" (see the next lemma), and feasibility ensures that RESTRICT/NO ACTION *rac*s are "obeyed."

For given external updates, the induced internal updates are characterized by the transitive closure of $\mathcal{DC}$:

Definition 3.2 *Induced Updates.* For given RA, D, $U_{\triangleright}$, and $U \subseteq U_{\triangleright}$

$$\Delta(U) := \{\mathsf{del}\_R(\bar{x}) \mid \text{ there is a } \triangleright\mathsf{del}\_R'(\bar{y}) \in U \text{ s.t. } (R(\bar{x}), R'(\bar{y})) \in \mathcal{DC}^*\}$$

is called the set of *induced updates of* U.

LEMMA 3.3 INDUCED UPDATES. *For given RA, D,* $U_{\triangleright}$*, and* $U \subseteq U_{\triangleright}$, $\Delta(U)$ *is the least set*[4] Δ *which contains the updates given by U and is complete.*

PROOF. $\mathcal{DC}^*$ is the reflexive, transitive closure of $\mathcal{DC}$. Hence $\Delta(U)$ contains all user requested updates and all cascaded updates (completeness) and nothing else (least set). □

Definition 3.4 *Admissibility and Application of* $U_{\triangleright}$. Let RA, D, and $U_{\triangleright}$ be given. $U \subseteq U_{\triangleright}$ is *admissible* if $\Delta(U)$ is admissible, and *maximal admissible* if there is no other admissible U', such that $U \subsetneq U' \subseteq U_{\triangleright}$. For a set Δ of user requests, $D' = D \pm \Delta$ denotes the database obtained by applying Δ to D.

This definition provides a precise and elegant characterization of the intended semantics. However, it is non-constructive in the sense that it does not lend itself to a computation of the intended semantics.
From the above fundamental definitions, we derive the following:

PROPOSITION 3.5 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. Since $U \subseteq U_{\triangleright}$, $\Delta(U)$ 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. □

PROPOSITION 3.6 UNIQUENESS. *For given RA, D, and* $U_{\triangleright}$,

(i) *if* $U_1, U_2 \subseteq U_{\triangleright}$ *are admissible, then* $U_1 \cup U_2$ *is also admissible,*
(ii) *thus, there is exactly one maximal admissible* $U_{\max} \subseteq U_{\triangleright}$.

PROOF. (i) is obvious. (ii) follows from (i) together with the fact that $\emptyset$ is always admissible. Thus, the union of all admissible subsets of $U_{\triangleright}$ yields $U_{\max}$. □

Note that for an admissible set U, not necessarily each subset is also admissible (there can be updates that "need" each other to be feasible).

[4]i.e., there is no proper subset that satiesfies the required properties, and it is the only minimal set.

3.2 Logic Programming Characterization

We show how a set RA of *rac*s can be translated 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. Moreover, by choosing an appropriate evaluation strategy, this logical specification can be *executed* as well, yielding the desired constructive semantics.

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

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

Referential actions. Each referential action is specified by an appropriate rule:

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

$$\mathsf{req\_del}\_R_C(\bar{X}) \leftarrow \mathsf{req\_del}\_R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. \qquad (DC_1)$$

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

$$\mathsf{blk\_del}\_R_P(\bar{Y}) \leftarrow R_P(\bar{Y}),\ \ \mathsf{blk\_del}\_R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. \qquad (DC_2)$$

Note that the atom $R_P(\bar{Y})$ can be added to the body of (DC_1), and $R_C(\bar{X})$ can be added to the body of (DC_2), but are redundant since delete requests and blockings are only derived for tuples that actually exist.

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

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

—$R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON DELETE NO ACTION` blocks the deletion of a parent tuple if there is a corresponding child tuple that is not requested for deletion:

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

Note that (i) the *local semantics* of each individual *rac* is precisely specified by one or two logic rules, and (ii) P_{RA} is in general not stratified due to the negative cyclic dependency $\mathsf{req\_del} \overset{-}{\rightsquigarrow} \mathsf{blk\_del} \overset{-}{\rightsquigarrow} \mathsf{req\_del}$. Therefore, the *global semantics* is not necessarily unique. We first consider a "skeptical" global semantics: the unique well-founded model of the generated logic program. The more "brave" stable models are considered in Section 3.4.2.

First, we add two rules that define an auxiliary relation $\mathsf{pot\_del}$, which contains all tuples that are potentially deleted when executing $U_{\triangleright}$ and cascading deletions. This relation is not used for checking the admissibility of $U_{\triangleright}$, but, as shown in Section 3.7, is useful to locate the problems in case $U_{\triangleright}$ is *not* admissible:

$$\begin{array}{l} \mathsf{pot\_del}\_R(\bar{X}) \leftarrow \triangleright\mathsf{del}\_R(\bar{X}), R(\bar{X}). \\ \text{for each } R_C.\vec{F} \rightarrow R_P.\vec{K} \text{ ON DELETE CASCADE (analogous to } (DC_1)\text{):} \\ \mathsf{pot\_del}\_R_C(\bar{X}) \leftarrow \mathsf{pot\_del}\_R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. \end{array} \qquad (P)$$

(note that the atom $\mathsf{pot\_del}\_R_P(\bar{Y})$ can be added to the body of (DC_2), (DR), and (DN) as an optimization.)
We have the following simple lemma:

LEMMA 3.7. *Given a database D and a set of user requests $U_\triangleright$, for the minimal model $\mathcal{M} := \mathcal{M}(\{P\}, D, U_\triangleright)$ of rule (P) alone,*

$$U_{pot} := \{\mathsf{del}\_R(\bar{x}) \mid \mathcal{M}(\mathsf{pot\_del}\_R(\bar{x})) = true\}$$
$$= \{\mathsf{del}\_R(\bar{x}) \mid \text{there is a } \triangleright\mathsf{del}\_R'(\bar{x}') \in U_\triangleright \text{ and } (R(\bar{x}), R'(\bar{x}')) \in \mathcal{DC}^*\} = \Delta(U_\triangleright)$$

contains exactly the deletions that are obtained when cascading all deletions in $U_\triangleright$.

Well-Founded Semantics. The well-founded model [Van Gelder et al. 1991] is widely accepted as a (skeptical) declarative semantics for logic programs containing negation. Given a database D and a set of user requests $U_\triangleright$, the well-founded model $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_\triangleright)$ assigns truth-values true and false to all uncontroversial update requests—those that are true or false under any *reasonable semantics* of P_{RA} [Dix 1995]. $\mathcal{W}$ assigns a third truth value *undefined* to atoms whose truth cannot be determined using a "well-founded" argumentation. The atoms that are undefined in $\mathcal{W}$ are controversial due to some kind of ambiguity (cf. Section 2.3). In Section 3.4.1, we will prove the following:

THEOREM 3.8 CORRECTNESS.
The logic programming characterization is correct with respect to the abstract semantics (for an atom at, Let $\mathcal{W}(at)$ denote the value of at in the model $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_\triangleright)$):

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

COROLLARY 3.9. *In case that $U_\triangleright$ is admissible, we have $U_\triangleright = U_{\max}$ and $U_{pot} = \Delta(U_\triangleright)$.*

Often, even if not all requested updates can be accomplished, a subset of them is admissible. Thus, the information as to which tuple or update really causes problems is valuable for preparing a refined update that realizes the intended changes *and* is acceptable. In Section 3.7, we will systematically investigate the information that is available in case $U_\triangleright$ is not admissible.

Example 7. Consider the database depicted in Figure 2 (ignoring R_0 for now) and the user request $U_\triangleright = \{\triangleright\mathsf{del}\_R_1(a), \triangleright\mathsf{del}\_R_1(b)\}$. Here, $\mathsf{del}\_R_1(b)$ is not admissible since it is blocked by $R_5(b)$. The other request, $\mathsf{del}\_R_1(a)$, can be executed without violating any *ric* by deleting $R_1(a)$, $R_2(a, x)$, $R_3(a, y)$, and $R_4(a, x, y)$.

The well-founded semantics reflects the different status of the single updates:

Given the user request $U_{\triangleright_a} = \{\triangleright\mathsf{del}\_R_1(a)\}$, 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.

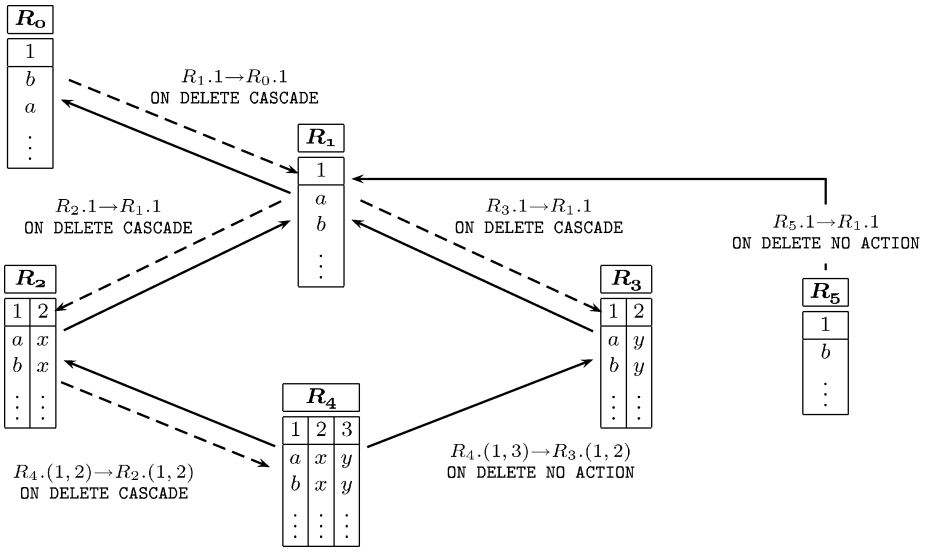

Fig. 2. Extended database with modified *rac*s.

For the user request $U_{\triangleright_b} = \{\triangleright\mathsf{del}\_R_1(b)\}$, blk_del is *true* for $R_1(b)$ due to the referencing tuple $R_5(b)$. Thus, req_del_$R_1(b)$ is *false*, and 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, blk_del_$R_3(b, y)$ is also *true*.

Note that the extended set $U'_{\triangleright} = \{\triangleright\mathsf{del}\_R_1(a), \triangleright\mathsf{del}\_R_1(b), \triangleright\mathsf{del}\_R_5(b)\}$ is a candidate for a refined request which accomplishes the deletion of $R_1(a)$ and $R_1(b)$.

$\mathcal{W}$ contains some ambiguities that can be interpreted constructively as *degrees of freedom*: The blockings and deletions induced by $U_{\triangleright} = \{\triangleright\mathsf{del}\_R_1(a)\}$ in Example 7 are undefined due to the dependency req_del $\vec{\rightsquigarrow}$ blk_del $\vec{\rightsquigarrow}$ req_del. This may be used to define different global policies by giving priority either to deletions or blockings, as will be done in Section 3.4.2.

3.3 Game-Theoretic Characterization

The following game-theoretic formalization provides an elegant characterization of *rac*s which yields 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, I (the *"Deleter"*) and 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 I and II are defined below. Note that I moves from D to $U_{\triangleright}$, while II moves from $U_{\triangleright}$ to $D \cup \{\mathsf{restricted}\}$. Initially, the pebble is placed on some tuple in D (or $U_{\triangleright}$) and I (or II) starts to move. If II begins, the first round only consists of the move by II.

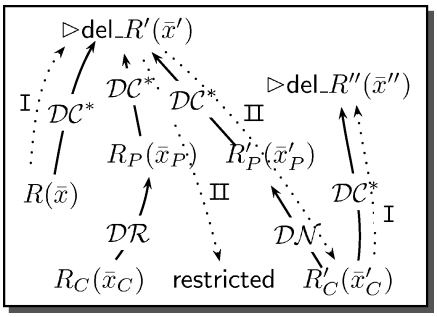

Fig. 3. Possible moves.

The possible moves are illustrated in Figure 3: By moving the pebble from $R(\bar{x}) \in D$ to some $\triangleright\mathsf{del}\_R'(\bar{x}') \in U_{\triangleright}$, which cascades down to $R(\bar{x})$, I claims that the deletion of $R(\bar{x})$ is "justified" (i.e., founded) by $\triangleright\mathsf{del}\_R'(\bar{x}')$. Conversely, II claims by her moves that $\mathsf{del}\_R'(\bar{x}')$ 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, assume that the deletion of $R'(\bar{x}')$ cascades down to some other tuple $R'_P(\bar{x}'_P)$. Then, II can move to a child tuple $R'_C(\bar{x}'_C)$, which references $R'_P(\bar{x}'_P)$ with a `NO ACTION` *rac*. 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 $\triangleright\mathsf{del}\_R'(\bar{x}') \in U_{\triangleright}$ if $(R(\bar{x}), R'(\bar{x}')) \in \mathcal{DC}^*$.

Player II can move from $\triangleright\mathsf{del}\_R'(\bar{x}')$

- to restricted if there are $R_P(\bar{x}_P)$ and $R_C(\bar{x}_C)$ such that $(R_P(\bar{x}_P), R'(\bar{x}')) \in \mathcal{DC}^*$ 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 3.10 CLAIMS OF I AND II.

(1) *If I can move from $R(\bar{x})$ to $\triangleright\mathsf{del}\_R'(\bar{x}')$, then deletion of $R'(\bar{x}')$ is founded by $U_{\triangleright}$ and induces the deletion of $R(\bar{x})$.*
(2) *If II can move from $\triangleright\mathsf{del}\_R(\bar{x})$ to restricted, then deletion of $R(\bar{x})$ is not feasible due to the existence of a referencing tuple.*
(3) *If II can move from $\triangleright\mathsf{del}\_R(\bar{x})$ to $R'(\bar{x}')$, then deletion of $R(\bar{x})$ is admissible only 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 $R_P(\bar{x}_P)$, $R_C(\bar{x}_C)$ such that $(R_P(\bar{x}_P), R(\bar{x})) \in \mathcal{DC}^*$ and $(R_C(\bar{x}_C), R_P(\bar{x}_P)) \in \mathcal{DR}$. Then, 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) $(R'_C(\bar{x}'_C), R(\bar{x})) \in \mathcal{DN} \circ \mathcal{DC}^*$, that is, there is a $R'_P(\bar{x}'_P)$ such that $(R'_P(\bar{x}'_P), R(\bar{x})) \in \mathcal{DC}^*$ and $(R'_C(\bar{x}'_C), R'_P(\bar{x}'_P)) \in \mathcal{DN}$. Hence, by (1), deletion of $R(\bar{x})$ induces deletion of $R'_P(\bar{x}'_P)$, which is only allowed if $R'_C(\bar{x}'_C)$ is also deleted.[5] □

LEMMA 3.11. *The moves are correlated with the logical specification as follows:*

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

—*The moves by* II *are reflected by the rules* (DC_2) *and (DR) / (DN):*

—II *can move from* $\triangleright$*del_*$R(\bar{x})$ *to* restricted *if* *blk_del_*$R(\bar{x})$ *is derivable using (DR) and* (DC_2) *only, or*

—II *can move from* $\triangleright$*del_*$R(\bar{x})$ *to* $R'_C(\bar{x}'_C)$ *if* *blk_del_*$R(\bar{x})$ *is derivable using* (DC_2) *and an instance of (DN) if* *req_del_*$R'_C(\bar{x}'_C)$ *is assumed to be false.*

—*The negative dependencies in (I),* *req_del* $\rightsquigarrow \neg$*blk_del, and (DN),* *blk_del* $\rightsquigarrow \neg$*req_del, mirror the alternation of moves between* I *and* II*, respectively.*

Definition 3.12. 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. If p 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." An update $\triangleright$del_$R(\bar{x}) \in U_\triangleright$ is won if $R(\bar{x}) \in D$ is won.

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

Example 8. Consider again Figure 2 with $U_\triangleright = \{\triangleright$del_$R_1(a), \triangleright$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 $\triangleright$del_$R_1(a)$, while II can move from $\triangleright$del_$R_1(a)$ to $R_4(a, x, y)$. Thus, after I has started the game moving to $\triangleright$del_$R_1(a)$, II will answer with the move to $R_4(a, x, y)$, so I moves back to $\triangleright$del_$R_1(a)$ again, and so forth. Hence the game is drawn for each of the "a"-tuples.

In contrast, for the "b"-tuples, there is an additional move from $\triangleright$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 3.13 GAME SEMANTICS. *For every tuple* $R(\bar{x}) \in D$*:*

—$R(\bar{x})$ *is lost* $\Leftrightarrow$ *it is not possible with the given set of user delete requests to delete* $R(\bar{x})$ *without violating a ric.*

—$R(\bar{x})$ *is won or drawn* $\Leftrightarrow$ *simultaneous execution of all user delete requests* $\triangleright$*del_*$R'(\bar{x}')$ *that are won or drawn does not violate any ric and deletes* $R(\bar{x})$.

PROOF. Note that if $R(\bar{x})$ is won or drawn, then there is no $R_C(\bar{x}_C) \in D$ such that $(R_C(\bar{x}_C), R(\bar{x})) \in \mathcal{DR}$ (otherwise, if I moves from $R(\bar{x})$ to some $\triangleright$del_$R_d(\bar{x}_d)$, II moves to 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 a won or drawn tuple.

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

"⇒": A tuple $R(\bar{x})$ is lost in n rounds if either

—$(n=0)$ there is no user request $\triangleright\mathsf{del\_}R_d(\bar{x}_d)$ such that $(R(\bar{x}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$, i.e., the deletion of $R(\bar{x})$ is unfounded, or

—$(n>0)$ for every user request $\triangleright\mathsf{del\_}R_d(\bar{x}_d)$ such that $(R(\bar{x}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$, $\triangleright\mathsf{del\_}R_d(\bar{x}_d)$ is lost in $\leq n$ rounds, that is, either II can move from $\triangleright\mathsf{del\_}R_d(\bar{x}_d)$ to restricted (in this case, by Lemma 3.10(2), $\triangleright\mathsf{del\_}R_d(\bar{x}_d)$ is not feasible), or there is some tuple $R'(\bar{x}')$ such as II can move from $\triangleright\mathsf{del\_}R_d(\bar{x}_d)$ 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 3.10(3), it must be deleted if $R(\bar{x})$ is deleted. Thus, $R(\bar{x})$ cannot be deleted.

"⇐": If $R(\bar{x})$ cannot be deleted without violating a *ric*, then either,

—deletion of $R(\bar{x})$ is unfounded—therefore it is lost immediately since I cannot move,

—or deletion of $R(\bar{x})$ is founded, but none of its founding user delete requests $\triangleright\mathsf{del\_}R'(\bar{x}')$ is executable. This can be either due to a $\mathcal{DC}^* \circ \mathcal{DR}$ chain to a tuple $R'_C(\bar{x}'_C)$—then $\triangleright\mathsf{del\_}R'(\bar{x}')$ is lost in one round since II moves to restricted—or due to a $\mathcal{DC}^* \circ \mathcal{DN}$ chain to a tuple $R'_C(\bar{x}'_C)$ that must be deleted, but cannot. Then, II can move there and will win (the detailed proof would argue with induction over the length of the proof *why* $\triangleright\mathsf{del\_}R'(\bar{x}')$ is not executable, analogous to the proof of "⇒").

—The second statement follows from the first by contraposition. □

The correspondence between the game semantics and the abstract semantics yields the following:

COROLLARY 3.14 CORRECTNESS. *The game-theoretic characterization is correct with respect to the abstract semantics:*

—$U_w := \{u \in U_\triangleright \mid u \text{ is won}\}$ *and* $U_{w,d} := \{u \in U_\triangleright \mid u \text{ is won or drawn}\}$ *are admissible,*

—$U_{w,d} = U_{\max}$,

—$\Delta(U_w) = \{\mathit{del\_}R(\bar{x}) \mid R(\bar{x}) \text{ is won}\}$ *and* $\Delta(U_{\max}) = \Delta(U_{w,d}) = \{\mathit{del\_}R(\bar{x}) \mid R(\bar{x}) \text{ is won or drawn}\}$.

3.4 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.

3.4.1 *Well-Founded Semantics.* The *alternating fixpoint computation* (AFP) is a method for computing the well-founded model based on successive rounds [Van Gelder 1993]. This characterization finally leads to an algorithm for determining the maximal admissible subset of a given set $U_\triangleright$ of user requests. We introduce the AFP by using **Statelog**, a state-oriented extension of Datalog which allows the integration of active and deductive rules [Ludäscher et al. 1996a; Ludäscher 1998]. It can be seen as a restricted class of logic programs

where every intensional predicate contains an additional distinguished argument for *state terms* of the form $[S+k]$. EDB predicates and built-in predicates are state-independent. 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, the AFP is obtained by attaching state terms to the program P such that all positive IDB literals refer to $[S+1]$ and all negative IDB literals refer to $[S]$. The resulting program P_{AFP} computes the alternating fixpoint of P:

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

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

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

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

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

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

% $R_C.\vec{F} \rightarrow R_P.\vec{K}$ *ON DELETE NOACTION*: (DN^A)

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

P_{AFP} is *locally* stratified, thus there is a *unique perfect model* [Przymusinski 1988] $\mathcal{M}_{AFP}$ of $P_{AFP} \cup D \cup U_{\triangleright}$. $\mathcal{M}_{AFP}$ mimics the alternating fixpoint computation of $\mathcal{W}$: 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, the truth value of atoms A in $\mathcal{W}$ can be determined from $\mathcal{M}_{AFP}$ as follows:

$$\mathcal{W}(A) = \begin{cases} \mathit{true} & \mathit{if}\ \mathcal{M}_{AFP} \models [2n_f]\ A, \\ \mathit{undef} & \mathit{if}\ \mathcal{M}_{AFP} \models [2n_f]\ \neg A\ \wedge\ [2n_f+1]\ A, \\ \mathit{false} & \mathit{if}\ \mathcal{M}_{AFP} \models [2n_f+1]\ \neg A\ . \end{cases}$$

THEOREM 3.15 EQUIVALENCE. *The well-founded model is equivalent to the game-theoretic characterization:*

—$R(\bar{x})$ *is won* $\Leftrightarrow$ $\mathcal{W}(\ \mathsf{req\_del}\_R(\bar{x})) = \mathit{true}$.
—$R(\bar{x})$ *is lost* $\Leftrightarrow$ $\mathcal{W}(\ \mathsf{req\_del}\_R(\bar{x})) = \mathit{false}$.
—$R(\bar{x})$ *is drawn* $\Leftrightarrow$ $\mathcal{W}(\ \mathsf{req\_del}\_R(\bar{x})) = \mathit{undef}$.

PROOF. The proof is based on a lemma which follows from the correspondence between moves and reference chains that has been established in Lemma 3.11 (using the same argumentation as in the proof of Theorem 3.13):

LEMMA 3.16.

—$R(\bar{x})$ *is won (for* I*) within* $\leq n$ *rounds iff* $\mathcal{M}_{AFP} \models [2n]\ \ \mathsf{req\_del}\_R(\bar{x})$.
—$R(\bar{x})$ *is lost within* $\leq n$ *rounds iff* $\mathcal{M}_{AFP} \models [2n+1]\ \neg\ \mathsf{req\_del}\_R(\bar{x})$.

From this, Theorem 3.15 follows immediately: The nth overestimate excludes deletions provably non-admissible in n rounds, whereas the nth underestimate contains all deletions which can be proven in n rounds. Thus, there is an n such that $\mathcal{M}_{AFP} \models [2n]$ req_del_$R(\bar{x})$ iff $\mathcal{W}_{RA}($ req_del_$R(\bar{x})) = true$, and there is an n such that $\mathcal{M}_{AFP} \models [2n+1]\ \neg$req_del_$R(\bar{x})$ iff $\mathcal{W}($ req_del_$R(\bar{x})) = false$.

A position $R(\bar{x})$ is drawn if for every user request $\triangleright$del_$R'(\bar{x}')$ that I uses for deleting it, II can find a witness against $\triangleright$del_$R'(\bar{x}')$, and conversely, I claims to be able to delete the witness. Thus, no player has a "well-founded" proof for or against deleting those tuples (caused by NO ACTION links that introduce cycles into the argumentation). □

From Corollary 3.14 and Theorem 3.15, the correctness of the logic programming formalization (and thus the proof of Theorem 3.8) follows. In the following section, it is shown that the maximal admissible subset $U_{\max} \subseteq U_{\triangleright}$ (by Theorem 3.8, $U_{\max} = U_{t,u}$) also corresponds to a *total* (i.e., not involving atoms with an undefined truth value) semantics of P.

3.4.2 *Stable Models.* The undefined atoms in the well-founded model leave some scope for further interpretation. This "freedom of choice" can often be used to obtain alternative solutions, given by *stable models*:

Definition 3.17 *Stable Model* [*Gelfond and Lifschitz 1988*]. Let M_P denote the minimal model of a positive program P. Given a ground-instantiated program P and an interpretation I of the atoms of P, P/I denotes the *reduction* of P with respect to I—the program obtained by replacing every negative literal of P by its truth-value with respect to I. An interpretation I is a *stable model* if $M_{P/I} = I$.

Every stable model $\mathcal{S}$ extends the well-founded model $\mathcal{W}$ with respect to true and false atoms: $\mathcal{S}^{true} \supseteq \mathcal{W}^{true}$, $\mathcal{S}^{false} \supseteq \mathcal{W}^{false}$. However, not every program has a two-valued stable model (e.g., the "self attack" in Example 5).

THEOREM 3.18. *Let $\mathcal{S}_{RA}$ be defined by*

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

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 that are true in some stable model. Consequently, deletions have priority over blockings (cf. Example 7). For the diamond example, there are two stable models:

Example 9 *Diamond—Stable Models.* Consider Example 3 and the "diamond" in Figure 1. Assume the *rac* $R_4.(1,3) \to R_3.(1,2)$ ON DELETE RESTRICT to be replaced by $R_4.(1,3) \to R_3.(1,2)$ ON DELETE NO ACTION. From the rules of P_{RA} it can be derived that the deletion of $R_1(a)$ is blocked (via $R_4 \rightsquigarrow R_3 \rightsquigarrow R_1$) if $R_4(a,b,c)$ cannot be deleted. $R_4(a,b,c)$ can be deleted (via $R_1 \rightsquigarrow R_2 \rightsquigarrow R_4$) if the deletion of $R_1(a)$ is not blocked. Hence there is a negative cycle of the form

$\{block \leftarrow \neg exec,\ exec \leftarrow \neg block\}$. Setting all requests in the diamond to true (as done in $\mathcal{S}_{RA}$) or all to false results each in a stable model.

THEOREM 3.19 CORRECTNESS.

—*Let* $\mathcal{S}$ *be a stable model of* $P_{RA} \cup D \cup U_{\triangleright}$. *Then* $U_{\mathcal{S}} := \{\triangleright del\_R(\bar{x}) \in U_{\triangleright} \mid \mathcal{S} \models req\_del\_R(\bar{x})\}$ *is admissible and* $\Delta(U_{\mathcal{S}}) = \Delta_{\mathcal{S}} := \{del\_R(\bar{x}) \mid \mathcal{S} \models req\_del\_R(\bar{x})\}$.

—$U_{\max} = U_{\mathcal{S}_{RA}}$ *and* $\Delta(U_{\max}) = \Delta_{\mathcal{S}_{RA}}$.

PROOF. *Foundedness*: follows directly from the fact that $\mathcal{S}$ is stable (an unfounded req_del_$R(\bar{x})$ would not be stable).

Completeness: For every *ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON DELETE CASCADE`, if $\mathcal{S} \models R_C(\bar{x}) \wedge$ req_del_$R_P(\bar{y}) \wedge \bar{x}[\vec{F}] = \bar{y}[\vec{K}]$, then, due to (DC_1), $\mathcal{S} = M_{P/\mathcal{S}} \models$ req_del_$R_C(\bar{x})$.

Feasibility: Suppose a *ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON DELETE RESTRICT` or $R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON DELETE NO ACTION` would be violated. Then, $\mathcal{S} \models$ req_del_$R_P(\bar{y}) \wedge R_C(\bar{x}) \wedge \bar{x}[\vec{F}] = \bar{y}[\vec{K}]$ (for `NO ACTION` also $\mathcal{S} \models \neg$req_del_$R_C(\bar{x})$), and thus because of (DR) or (DN), respectively, $\mathcal{S} = M_{P/\mathcal{S}} \models$ blk_del_$R_P(\bar{y})$. Thus, by (DC_2), for the founding delete request del_$R(\bar{z})$, $\mathcal{S} \models$ blk_del_$R(\bar{z})$, and by (I), $\mathcal{S} \models \neg$req_del_$R(\bar{z})$, which is a contradiction to the assumption that 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. □

3.5 A Procedural Translation

The declarative semantics of the well-founded model is translated into a more "algorithmic" implementation in Statelog by "cutting" the cyclic dependency at one of the possible points—at the rules (I) and (DN) (cf. the AFP characterization). From Theorem 3.15 and Corollary 3.14, the undefined deletions (which are drawn in the game-theoretic characterization) are also admissible (Theorem 3.19). Cutting in (DN) implements the definition of $\mathcal{S}_{RA}$ (giving priority to deletions over blockings), 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:

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

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

$$[S]\ \text{blk\_del\_}R_P(\bar{Y}) \leftarrow R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. \qquad (DR^S)$$

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

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

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

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

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

{blk_del_R} $\prec$ {req_del_R}, mirrors the stages of the algorithm: First, only blockings resulting from ON DELETE RESTRICT *rac*s are considered (local rules (DR^S) and (DC_2^S)). Based on these, the first maximal overestimate of internal delete requests is computed by (I^S) and (DC_1^S). Then, the blockings resulting from ON DELETE NO ACTION *rac*s whose child nodes are not deleted (by (DN^S), the only progressive rule; it does not derive anything for the initial state) are derived in the step to the subsequent state. (DR^S) contributes the blockings resulting from ON DELETE RESTRICT *rac*s. Again, the induced blockings are derived by (DC_2^S). The second stratum, consisting of (I^S) and (DC_1^S) determines the remaining non-blocked user delete requests and its induced delete requests. Then, the next iteration is started, calculating a decreasing sequence of overestimates which leads to $\mathcal{S}_{RA}$.

LEMMA 3.20. *$\mathcal{M}_{AFP}$ 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 latest underestimate, and the delete request from the latest overestimate, resulting in $\mathcal{S}_{RA}$. □

THEOREM 3.21 TERMINATION. *For every database D and every set $U_\triangleright$ of user delete requests, the program reaches a fixpoint, that is, there is a least $n_f \leq |U_\triangleright|$, such that $\mathcal{M}_S[n_f] = \mathcal{M}_S[n_f + 1]$.*

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. □

The correctness of P_S follows from Lemma 3.20 and Theorem 3.18:

THEOREM 3.22 CORRECTNESS. *The final state of $\mathcal{M}_S$, $\mathcal{M}_S[n_f]$, represents $U_{\max}$ and $\Delta(U_{\max})$:*

— $\mathcal{M}_S[n_f] = \mathcal{S}_{RA}$,
— $U_{\max} = \{\triangleright$*del_$R$*$(\bar{x}) \in U_\triangleright \mid \mathcal{M}_S[n_f] \models$ *req_del_R*$(\bar{x})\}$, *and*
— $\Delta(U_{\max}) = \{$*del_R*$(\bar{x}) \mid \mathcal{M}_S[n_f] \models$ *req_del_R*$(\bar{x})\}$.

3.5.1 *Implementation in a Procedural Programming Language.* The Statelog formalization P_S above is translated into the algorithm given in Figure 4. Initially, it is assumed that there are only those blockings which result directly from ON DELETE RESTRICT *rac*s. Then, blockings are propagated upwards along 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 can block other requests. In the next step, all blockings are computed that are caused by ON DELETE NO ACTION *rac*s from tuples that 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

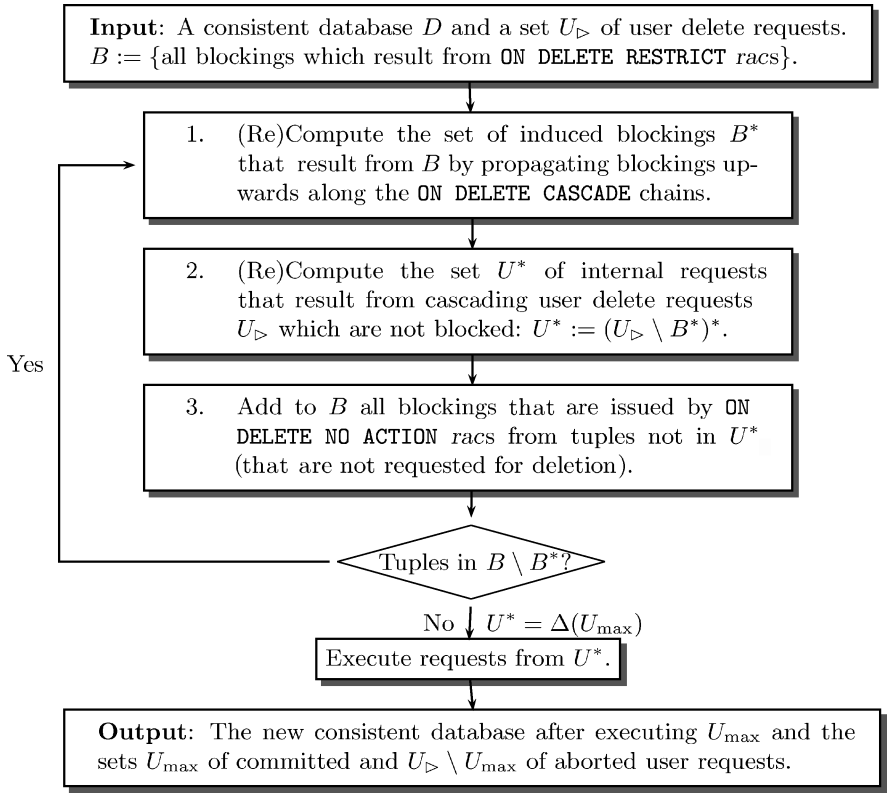

Fig. 4. Algorithm for executing all admissible deletions.

complexity. Since the fixpoint is reached after at most $|U_\triangleright|$ iterations (Theorem 3.21), the overall algorithm also has polynomial data complexity.

THEOREM 3.23. *The algorithm in Figure 4 is correct:*

—$U_{\max} = \{\triangleright del\_R(\bar{x}) \in U_\triangleright \mid req\_del\_R(\bar{x}) \in U^*\}$, *and* $\Delta(U_{\max}) = U^*$.

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

For given D, $U_\triangleright$, and RA, the algorithm in Figure 4 computes the maximal subset $U_{\max}$ of $U_\triangleright$ that can be executed without violating any *ric*, and the set $\Delta(U_{\max})$ of "internal deletions" that are induced by it. For cases when $U_\triangleright$ is not admissible, troubleshooting is described in Section 3.7.

3.6 Relationship with the SQL Semantics

The SQL semantics as presented in Horowitz [1992] and specified in the SQL3 standard [ANSI/ISO 1999] (see also Section 2.4) coincides with ours:

—matching rows (i.e., $R_C.\vec{F}(\bar{x})$ and $R_P.\vec{K}(\bar{y})$) are selected with respect to the original database state,

—the tuples to be deleted are fixed with respect to the original state, RESTRICT is also evaluated with respect to the original state,

—NOACTION is evaluated against the (prospective) result.

—an update referential action is executed whenever the parent is updated.

—effectively, Δ is applied after the computation is completed.

In the logic programming characterization, these "correctness" requirements are directly encoded into the rules (each individual rule corresponding to a single *ric* with *rac*), thus the application of the logic programming semantics cannot destroy them (provided one accepts the declarative logic programming semantics). Thus, a correctness proof with respect to the intended semantics—except that the specification of the individual rules is correct—is redundant.

Moreover, if a set of updates causes referential problems, the SQL semantics simply rejects the updates. Roughly speaking, it corresponds to the (P) rules and a simple RESTRICT test, and then checks the NO ACTION *ric*s. In case of a rejection, it is hard to find which update caused the problem. Since the SQL semantics has no notion of the semantics of the *rac*s, it cannot return useful details to the user as to how to prepare a revised request, or for debugging the application. Next, we show how the logic-based semantics directly incorporates this information, and how it can be used.

3.7 Troubleshooting in Case of Rejected Deletions

If $\triangleright\mathsf{del}\_R_d(\bar{x}_d) \in U_\triangleright$ is not admissible, there are local "situations" that cause the problem. Each such situation consists of a parent tuple that should be deleted, and a child tuple that references it. Thus, the problem is caused by

—a parent tuple $R_P(\bar{y})$ that is reachable by a chain of ON DELETE CASCADE references from $R_d(\bar{x}_d)$. Correctly, $R_P(\bar{y})$ is potentially deleted (and must be deleted when $R_d(\bar{x}_d)$ is deleted),

—a child tuple $R_C(\bar{x})$ that references $R_P(\bar{y})$,

—a *ric* that concerns the reference from $R_C(\bar{x})$ to $R_P(\bar{y})$, associated with a *rac* (either RESTRICT or NO ACTION, and the child is not requested for deletion).

In a correct application—where the database schema, containing the *ric*s and *rac*s is correct with respect to the semantics of the application domain, and where the program itself that computes $U_\triangleright$ are correct—such a situation must not occur. Thus, there is a design problem that shows up in this situation:

—the definition of the *rac* is wrong: if it were ON DELETE CASCADE, the child would be deleted. Why did the programmer not specify ON DELETE CASCADE (did he forget it, since the default is ON DELETE NO ACTION)?

—the program is wrong: the blocked $\triangleright\mathsf{del}\_R_d(\bar{x}_d)$ should not be derived. Some object of the application must not be deleted as long as there are objects that need it, and the program did not check this correctly. Possibly, the whole transaction should not be executed.

—In case that the *rac* from $R_P(\bar{y})$ to $R_C(\bar{x})$ is ON DELETE NO ACTION, it is implicitly assumed that $R_C(\bar{x})$ is deleted by some other delete request in $U_\triangleright$ (possibly via another cascading chain). Thus, there can be two reasons:
  1. the program is wrong: it should derive $\triangleright\mathsf{del}\_R_C(\bar{x})$ or it should derive a $\triangleright\mathsf{del}\_R'(\bar{x}')$ such that deleting $R'(\bar{x}')$ cascades to deletion of $R_C(\bar{x})$, or
  2. another *ric* or *rac* is wrong: The program derives such an $\triangleright\mathsf{del}\_R'(\bar{x}')$, but it does not cascade to $R_C(\bar{x})$.

—the original database state is wrong: $R_C(\bar{x})$ should not be there. A previous transaction did not execute in the intended way.

In all cases, the detailed information about the problem can provide useful hints as to how to fix the application program. For this, two conclusions must be drawn:

—where the above situation occurs, and

—for what reason(s).

Locating the Problem. The location of the problem can be found by analyzing the well-founded model: starting with the blocked deletion $\triangleright\mathsf{del}\_R_d(\bar{x}_d) \in U_\triangleright$, there is a chain of blocked deletions downwards through a series of cascading references. The "lowest" blocked deletion concerns the above tuple $R_P(\bar{y})$, and the blocking is caused by a child $R_C(\bar{x})$ via a *ric* that is associated to the above *rac*:

Definition 3.24. Given a database D and a set $U_\triangleright$ and $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_\triangleright)$ as above, a quadruple $(\triangleright\mathsf{del}\_R_d(\bar{x}_d),\ R_P(\bar{y}),\ rac,\ R_C(\bar{x}))$ is a *problem situation* if

—there is an $\triangleright\mathsf{del}\_R_d(\bar{x}_d) \in U_\triangleright$ such that $(R_P(\bar{y}), R_d(\bar{x}_d)) \in \mathcal{DC}^*$ and $\mathcal{W}(\mathsf{blk\_del}\_R_d(\bar{x}_d)) = true$ and for all $R'(\bar{x}')$ on this chain, $\mathcal{W}(\mathsf{blk\_del}\_R'(\bar{x}')) = true$ and
  (i) $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DR}$ (then, $rac =$ RESTRICT) or
  (ii) $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DN}$ (then, $rac =$ NO ACTION) and $\mathcal{W}(\mathsf{req\_del}\_R_C(\bar{x})) = false$ (i.e., $R_C(\bar{x})$ is not deleted), and $\mathcal{W}(\mathsf{pot\_del}\_R_C(\bar{x})) = false$.

Note that in case (ii), if $\mathcal{W}(\mathsf{pot\_del}\_R_C(\bar{x})) = true$, this situation is not a problem since $R_C(\bar{x})$ would be deleted by $U_\triangleright$, but the update $\triangleright\mathsf{del}\_R'(\bar{x}') \in U_\triangleright$ that should do this is itself blocked (it can be found by checking pot_del upwards, following the references).

Example 10. Consider again Example 7 where $U_\triangleright = \{\triangleright\mathsf{del}\_R_1(a), \triangleright\mathsf{del}\_R_1(b)\}$ has been rejected. In a real database environment, the rejection can be augmented by a problem description: Here, $(\triangleright\mathsf{del}\_R_1(b), R_1(b),$ NO ACTION, $R_5(b))$ is the problem situation since the deletion of $R_1(b)$ must be rejected due to the existence of the tuple $R_5(b)$ (even not potentially deleted) and the corresponding *rac*. Possible interpretations are:

—the definition of the *rac* is wrong (should be ON DELETE CASCADE to align the requested actions for $R_1(b)$ and $R_5(b)$), or

—the definition of the transaction is wrong. It should yield $U'_{\triangleright}$ (delete $R_5(b)$ too, explicitly). Thus add a program line `DELETE ... FROM` R_5.

—both are correct, but $R_5(b)$ should not be there. Another transaction had a wrong effect (e.g., by not deleting it).

Consider now the complete database in Figure 2 with $U''_{\triangleright} = \{\triangleright\mathsf{del}\_R_0(a), \triangleright\mathsf{del}\_R_0(b)\}$. It induces the update $U_{\triangleright}$ above and is again rejected. A possible conclusion from the detailed description of which tuples cause the problems could be

—there should be a *rac* $R_5.(1) \rightarrow R_0.(1)$ `ON DELETE CASCADE` that would then induce the updates given in $U'_{\triangleright}$.

Detecting problems during computation. The above problem analysis is based on the *outcome* of computing the well-founded model. However, the problem could easily be located *during* the computation of the well-founded model: the *first* blocking $\mathsf{blk\_del}\_R_P(\bar{y})$ that is derived for a requested (internal) update in an *underestimate* (even-numbered states) locates the problem. The rule that is applied for deriving it identifies the referencing child tuple $R_C(\bar{x})$ and the reference. Then, the above scenario of "indirect" blockings can be avoided: the detected problem *is* a problem.

If the procedural algorithm given in Section 3.5 is applied, the same applies: the first blocking that is found identifies the problem situation.

Concerning the game, the problem situations correspond directly to the "final moves" of Π, either, to restricted, or via $\mathcal{DC}^* \circ \mathcal{DN}$ to a tuple whose deletion is not founded (if its deletion is founded, but also "lost," the above indirect situation applies, and the game goes on, leading to another final move of Π).

Solving the Problem. Still, if a problem is located, it is not clear how it must be solved. As described above, there can be several reasons and solutions. For example, a sophisticated reasoning component could be added that investigates the well-founded model in order to analyze the possible causes of the problem:

—try the same $U_{\triangleright}$ for different arguments (e.g., checking what happens for $\triangleright\mathsf{del}\_R_1(c)$ in the above example) to check if there is a general problem, or if the problem exists only for a special value (output: a set of problematic values).

—try extensions of $U_{\triangleright}$, and check how they are related to the original update, (output: a set of additional external updates; cf. Example 7).

—try possible modifications of the *rac*s (e.g., turn `NO ACTION` into `CASCADE`), and

—try to extend P_{RA} with additional *ric*s and *rac*s.

In all cases, this information should be returned to the user. Due to the nature of the well-founded model as a stable (and reproducing) model, it is sufficient to adapt the rules or facts (immediately when a problem is detected during the computation) and to continue the evaluation starting with $\mathcal{W}$. The same holds for the presented procedural algorithm (which is a "hard-coded" version of the computation of the well-founded model for the given rule schemata).

4. SEMANTICS OF REFERENTIAL ACTIONS WITH MODIFICATIONS

In this section, we extend our approach to include modifications of tuples, resulting in a more involved translation and semantics. Interferences between cascaded modifications can occur especially in the presence of *overlapping* foreign keys. In contrast to deletions, modifications can create new foreign key values where the existence of an appropriate parent tuple has to be checked. Thus, instead of the SQL syntax, we use the more detailed syntax

```
ON UPDATE OF {PARENT|CHILD} {CASCADE|RESTRICT|NO ACTION} .
```

Recall that all updates to a relation R are represented by auxiliary relations $\mathsf{ins}\_R(\bar{X})$, $\mathsf{del}\_R(\bar{X})$, and $\mathsf{mod}\_R(M, \bar{X})$. M is a list of pairs i/c meaning that the i-th attribute of $R(\bar{X})$ should be set to the constant c. As a shorthand for $\mathsf{mod}\_R([1/d, 3/e], (a, b, c))$, we may write $\mathsf{mod}\_R(a/d, b, c/e)$. The *restriction of a modification* M to a key $\vec{A}$ is denoted by $M[\vec{A}]$; the result of *applying a modification* M to $\bar{X}$ is denoted by $M(\bar{X})$. For example if $M = [1/d, 3/e]$, $\bar{X} = (a, b, c)$, then $M[(2, 3)] = [3/e]$, and $M(\bar{X}) = (d, b, e)$. The union of two modifications is the union of the lists: $[1/a, 2/b, 3/d] \cup [1/e, 2/b, 4/f] = [1/a, 1/b, 2/b, 3/d, 4/f]$. A modification is *consistent* if is does not assign two different values to a position (the above union is inconsistent).

4.1 Abstract Semantics

Let D be a database instance, $U_\triangleright$ a set of user requests, and RA a set of *rac*s. We define abstract properties that a set of updates Δ may have with respect to D, $U_\triangleright$, and RA. These allow us to define the intended meaning of a set of *rac*s in an abstract (and non-constructive) way. Recall that $D' = D \pm \Delta$ denotes the database obtained by applying Δ to D. In contrast to deletions, we cannot simply base our considerations on the transitive closure of references. Every step has to be analyzed individually, taking into consideration possible interferences due to overlapping keys.

Definition 4.1 *Abstract Properties.* An individual update instruction is called *founded* with respect to $U_\triangleright$, Δ, D, and RA if it can be justified by the user requests and propagations:

—A delete instruction $\mathsf{del}\_R(\bar{x})$ is *founded in n steps* with respect to $U_\triangleright$, Δ, D, and RA if either $n = 0$ and $\triangleright\mathsf{del}\_R(\bar{x}) \in U_\triangleright$, or there is a delete instruction $\mathsf{del}\_R_i(\bar{x}_i) \in \Delta$ that is founded in $< n$ steps, and a *rac* $R.\vec{F}_i, \rightarrow R_i.\vec{K}_i$ `ON DELETE OF PARENT CASCADE` such that $\bar{x}[\vec{F}_i] = \bar{x}_i[\vec{K}_i]$.

—A modify instruction $\mathsf{mod}\_R(M, \bar{x})$ is *founded in n steps* with respect to $U_\triangleright$, Δ, D, and RA if there is a set $M_1, \ldots, M_n$ of modifications s.t. $M = \bigcup_{i=1..n} M_i$ (not necessarily disjoint) and for every i either $\triangleright\mathsf{mod}\_R(M_i, \bar{x}) \in U_\triangleright$ or there is a modify instruction $\mathsf{mod}\_R_i(M_i', \bar{x}_i) \in \Delta$ that is founded in $< n$ steps, and a *rac* $R.\vec{F}_i, \rightarrow R_i.\vec{K}_i$ `ON UPDATE OF PARENT CASCADE` such that $\bar{x}_i[\vec{K}_i] \neq M_i'(\bar{x}_i)[\vec{K}_i]$ and $\bar{x}[\vec{F}_i] = \bar{x}_i[\vec{K}_i]$ and $M_i = \vec{F}_i / M_i'(\bar{x}_i)[\vec{K}_i]$.

—An insert instruction $\mathsf{ins}\_R(\bar{x})$ is *founded* with respect to $U_\triangleright$, if $\triangleright\mathsf{ins}\_R(\bar{x}) \in U_\triangleright$.

For given D, $U_{\triangleright}$, and RA, a set Δ of updates is called:

—*founded with respect to $U_{\triangleright}$, D, and RA* if every update instruction $\mathsf{del}\_R(\bar{x})$ or $\mathsf{mod}\_R(M, \bar{x}) \in \Delta$ is founded.

—*complete with respect to D and RA* if it is closed with respect to propagations, that is, satisfies the following conditions:

1. if $\mathsf{del}\_R_P(\bar{y}) \in \Delta$, $R_C(\bar{x}) \in D$, $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON DELETE OF PARENT CASCADE` $\in RA$, and $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$ then $\mathsf{del}\_R_C(\bar{x}) \in \Delta$.
2. if $\mathsf{mod}\_R_P(M, \bar{y}) \in \Delta$, $R_C(\bar{x}) \in D$, $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON UPDATE OF PARENT CASCADE` $\in RA$, $\bar{y}[\vec{K}] \neq M(\bar{y})[\vec{K}]$ and $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$ then there is an M' such that $\mathsf{mod}\_R_C(M', \bar{x}) \in \Delta$ and $M' \supseteq \vec{F}/M(\bar{y})[\vec{K}]$.

—*feasible with respect to D and RA* if all *rac*s specified as no action and restrict are satisfied:

1. a) Δ contains no delete instruction $\mathsf{del}\_R_P(\bar{y})$ such that there is a *rac* $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON DELETE OF PARENT RESTRICT` $\in RA$ and a tuple $R_C(\bar{x}) \in D$ with $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$.
   b) Δ contains no modify instruction $\mathsf{mod}\_R_P(M, \bar{y})$ such that there is a *rac* $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON UPDATE OF PARENT RESTRICT` $\in RA$, $\bar{y}[\vec{K}] \neq M(\bar{y})[\vec{K}]$ and a tuple $R_C(\bar{x}) \in D$ with $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$.
2. for every *rac* $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON DELETE OF PARENT NO ACTION` or $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON UPDATE OF PARENT NO ACTION` $\in RA$, every delete instruction $\mathsf{del}\_R_P(\bar{y})$ or modify instruction $\mathsf{mod}\_R_P(M, \bar{y})$ such that $\bar{y}[\vec{K}] \neq M(\bar{y})[\vec{K}]$ that is contained in Δ, and every tuple $R_C(\bar{x}) \in D$ such that $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$, Δ contains either a delete instruction $\mathsf{del}\_R_C(\bar{x})$ or a modify instruction $\mathsf{mod}\_R_C(M, \bar{x})$ succh that $M(\bar{x})[\vec{F}] \neq \bar{y}[\vec{K}]$.
3. for every *rac* $R_C.\vec{F}, , \rightarrow R_P.\vec{K}$ `ON ... OF CHILD RESTRICT` $\in RA$ and every tuple $R_C(\bar{x}) \in D$ that either results from an insertion $\mathsf{ins}\_R_C(\bar{x}) \in \Delta$ or a modification $\mathsf{mod}\_R_C(M, \bar{x}') \in \Delta$ (i.e., $\bar{x} = M(\bar{x}')$) such that $\bar{x}'[\vec{F}] \neq M(\bar{x}')[\vec{F}]$, D contains a tuple $R_P(\bar{y})$ such that $\bar{y}[\vec{K}] = \bar{x}[\vec{F}]$ and Δ contains neither $\mathsf{del}\_R_P(\bar{y})$ nor any $\mathsf{mod}\_R_P(M, \bar{y})$ such that $M(\bar{y})[\vec{K}] \neq \bar{x}[\vec{F}]$.
4. for every *rac* $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON ... OF CHILD NO ACTION` $\in RA$ and every tuple $R_C(\bar{x}) \in D$ that either results from an insertion $\mathsf{ins}\_R_C(\bar{x}) \in \Delta$ or a modification $\mathsf{mod}\_R_C(M, \bar{x}') \in \Delta$ (i.e., $\bar{x} = M(\bar{x}')$) such that $\bar{x}'[\vec{F}] \neq M(\bar{x}')[\vec{F}]$, one of the following conditions hold:
   a) D contains a tuple $R_P(\bar{y})$ such that $\bar{y}[\vec{K}] = \bar{x}[\vec{F}]$ and Δ contains neither $\mathsf{del}\_R_P(\bar{y})$ nor any $\mathsf{mod}\_R_P(M, \bar{y})$ such that $M(\bar{y})[\vec{K}] \neq \bar{x}[\vec{F}]$, or
   b) there is an $\mathsf{ins}\_R_P(\bar{y}) \in \Delta$ such that $\bar{y}[\vec{K}] = \bar{x}[\vec{F}]$, or
   c) there is a $\mathsf{mod}\_R_P(M, \bar{y}) \in \Delta$ such that $R_P(\bar{y}) \in D$ and $M(\bar{y})[\vec{K}] = \bar{x}[\vec{F}]$.

—*coherent* if no contradicting updates are issued on the same tuple—if $upd = \mathsf{del}\_R(\bar{x}) \in \Delta$, then $\mathsf{ins}\_R(\bar{x}) \notin \Delta$ and there is no M such that $\mathsf{mod}\_R(M, \bar{x}) \in \Delta$; similar for other updates upd. Note that if Δ is coherent, $D' = D \pm \Delta$ is well-defined.

—*key-preserving* if in $D' = D \pm \Delta$ all key dependencies are satisfied.

—*admissible* if Δ is founded, complete, feasible, coherent, and key-preserving.

Again, these abstract properties are used to formalize our *intended semantics*:

Definition 4.2 *Induced Updates, Application of* $U_{\triangleright}$. Let $U_{\triangleright}$, RA, and D 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.
(analogously to the case of deletion, $\Delta(U)$ there is a unique least such set by construction).

—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', such that $U \subsetneq U' \subseteq U_{\triangleright}$.

—For a set Δ of user requests, $D' = D \pm \Delta$ denotes the database obtained by applying Δ to D.

Note that $\Delta(U)$ does not contain any subsumed updates. This semantics reflects the intended behavior of the database system—it neither "invents" nor "forgets" updates and guarantees referential integrity:

THEOREM 4.3 ADEQUACY.

(1) *If* $U \subseteq U_{\triangleright}$, *then* $\Delta(U)$ *is founded and complete.*
(2) *If a coherent* Δ *is complete and feasible, then* $D' = D \pm \Delta(U)$ *satisfies all rac*s.

PROOF.

(1) $\Delta(U)$ is defined as the *least* complete set, thus it is founded.
(2) Since Δ is complete, all updates propagated by RA are contained in Δ. Feasibility of Δ guarantees that no $upd \in \Delta$ is restricted, and all `NO ACTION` *rac*s are satisfied in D'. □

The abstract semantics specifies the notions of maximal admissible sets U and induced updates $\Delta(U)$, but provides no method as to how to compute them. In contrast to the case of deletions, where the union of admissible sets of updates was again admissible (cf. Proposition 3.6), this does not hold in general due to conflicting updates. Given a set of n user requests, there can be an exponential (in n) number of maximal admissible subsets. Not suprisingly, it is not sufficient to consider the well-founded model, but stable models will be required for the general case.

Moreover, even if it is known that U is admissible, computing $\Delta(U)$ is not straightforward: In contrast to deletions that can be propagated in a "greedy" way without considering parallel updates, in the presence of modifications, parallel updates have to be taken into account:

Example 11. Consider the database schema depicted in Figure 5. Among others there are *rac*s of type `ON UPDATE OF PARENT CASCADE` for the *ric*s $T.(1,2) \to R.(1,2)$, $T.(3,4) \to S.(1,2)$, $U.(1,2) \to T.(2,3)$, and a *rac* `ON UPDATE OF CHILD RESTRICT` for $T.(1,4) \to V.(1,2)$. Assume the database contains $R(a,b)$, $S(c,d)$, $T(a,b,c,d,\ldots)$, $U(b,c,\ldots)$, and $V(a,d,\ldots)$, $V(a',d,\ldots)$, $V(a,d',\ldots)$.

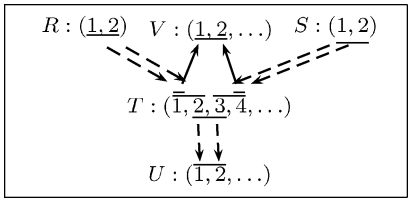

Fig. 5. Example scenario for modifications (see Example 11).

Given mod_$R(a/a', b/b')$ and mod_$S(c/c', d/d')$, the *rac*s trigger the modifications mod_$T(a/a', b/b', c, d, \ldots)$ and mod_$T(a, b, c/c', d/d', \ldots)$. Since these updates to T are coherent, they can be merged into mod_$T(a/a', b/b', c/c', d/d', \ldots)$, which triggers mod_$U(b/b', c/c', \ldots)$.

On the other hand, the *rac* $T.(1, 4) \rightarrow V.(1, 2)$ `ON UPDATE OF CHILD RESTRICT` restricts this modification since there is no tuple $V(a', d', \ldots)$. So each of the updates is admissible, but they are not admissible together, even though they do not directly contradict each other. Note that it is completely irrelevant, what modifications would be raised on the dotted parts of the tuples.

Example 11 illustrates some of the problems that may arise due to overlapping foreign keys and candidate keys, and gives an idea of the inherent complexity of rule-based referential integrity maintenance in the presence of modifications.

As indicated by Example 11, the propagation of updates cannot be seen tuple-oriented (too coarse: cf. the dotted parts of T, U, V) or attribute-oriented (too fine: mod_$T(a/a', b/b', c, d, \ldots)$ and mod_$T(a, b, c/c', d/d', \ldots)$ are both allowed in isolation, but their combination is forbidden), but must be handled *reference-oriented*. Thus, parent keys, foreign keys, references, and overlapping keys play an important role in our logical formalization.

4.2 Logic Programming Characterization

Problems and Solutions. The handling of updates adds several problems in contrast to deletions since tuples (and references) are not only removed, but also new tuples are generated that possibly contain new foreign key values. From the point of view of updating *referenced* relations, updates are not more complicated than deletions: propagate, wait, or restrict, as before. But from the point of view of updating the *referencing* relation (where deletions were trivial), we have to check whether there is an appropriate parent. If not, the update must be blocked. For deletions, we could list all blocked deletions—all tuples of the database that must not be deleted. In contrast, it is not possible to simply list all updates that are not allowed due to the absence of an appropriate parent (there are infinitely many). The considerations must be restricted to those updates that *potentially* arise from $U_{\triangleright}$. Additionally, there are *overlapping keys* as illustrated above where *interfering* modifications must be considered. As explained above, the solution is based on the analysis of *changes of key values*. Thus, the relations chg_$R.\vec{A}(M, \vec{X})$ (defined below in (CH_1)) and prp_$R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C, \vec{X})$

(defined below in (MPC_1)) are the "keys" to the solution:

—chg_$R.\vec{A}(M, \bar{X})$ denotes that the (candidate or foreign) key value $\bar{X}[\vec{A}]$ in relation R is modified by M, that is, changes to $(M(\bar{X}))[\vec{A}]$.

For candidate keys $R.\vec{A}$, this modification is propagated to all referencing foreign keys:

—prp_$R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C, \bar{x})$ denotes that a modification M of $R_P.\vec{K}$ is propagated along $R_P.\vec{K} \rightarrow R_C.\vec{F}$ to $R_C(\bar{x})$ resulting in the modification $M_C = \vec{F}/M_P(\bar{y})[\vec{K}]$ of the foreign key value.

Changes of overlapping candidate keys are computed from the incoming foreign key changes (see rule (CH_1) below).

In the logic programming characterization given for deletions in Section 3.2, the logical rules represented the *local* semantics of referential actions, and all global aspects were covered by the logic programming semantics (i.e., by the definition of the well-founded model and the stable model). When modifications are concerned, there are many more local parts in the global puzzle of the semantics of referential actions: Logical rules do not only represent the *local* semantics of referential actions, but also represent local *interferences* of updates. Again, all global aspects are provided by the chosen logic programming semantics.

The meaning of a set *RA* of *rac*s is formalized as a logic program P_{RA}, consisting of the sets P_{ra} that specify the local behavior of every *rac* ra, and a set of rules that specify the meaning of interacting update requests (instantiations for a given set of *ric*s can be found in Example 12 in the *electronic appendix*).

4.2.1 *Initalization and Bookkeeping*

User Requests. The handling of user requests incorporates the selection of admissible updates. User requests that are not blocked, raise an update to the database:

$$\begin{array}{lcl} \mathsf{del}\_R(\bar{X}) & \leftarrow & \triangleright\mathsf{del}\_R(\bar{X}),\ \neg\mathsf{blk\_del}\_R(\bar{X}). \\ \mathsf{ins}\_R(\bar{X}) & \leftarrow & \triangleright\mathsf{ins}\_R(\bar{X}),\ \neg\mathsf{blk\_ins}\_R(\bar{X}). \\ \mathsf{mod}\_R(M, \bar{X}) & \leftarrow & \triangleright\mathsf{mod}\_R(M, \bar{X}),\ \neg\mathsf{blk\_mod}\_R(M, \bar{X}). \end{array} \qquad (EXT_1)$$

Auxiliary Relations. As stated above, the approach is key-based. Thus, several auxiliary relations are maintained that contain information about referenced and referenceable candidate key values:

—is_refable_$R.\vec{K}(\bar{x})$: the (key) value $R.\vec{K}(\bar{x})$ is referenceable in the original state.

—rem_refable_$R.\vec{K}(\bar{x})$: the (key) value $R.\vec{K}(\bar{x})$ remains referenceable.

—new_refable_$R.\vec{K}(\bar{x})$: the (key) value $R.\vec{K}(\bar{x})$ becomes referenceable.

—is_refd_$R_P.\vec{K}$ _by_$R_C.\vec{F}(\bar{v})$: in the current database, the key value $R.\vec{K}(\bar{v})$ appears as foreign key value of $\vec{F}$ in some tuple $R_C(\bar{x})$.

—rem_refd_$R_P.\vec{K}$ _by_$R_C.\vec{F}(\bar{v})$: there is a reference to the key value $R.\vec{K}(\bar{v})$ as foreign key value of $\vec{F}$ in some tuple $R_C(\bar{x})$ such that $\bar{x}[\vec{F}]$ does not change.

— new_refd_$R_P.\vec{K}$ _by_$R_C.\vec{F}(\bar{v})$: a reference to the key value $R.\vec{K}(\bar{v})$ as foreign key $\vec{F}$ in some tuple $R_C(\bar{x})$ is introduced by some update.

For every candidate key $\vec{K}$ mentioned in some *ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$:

$$
\begin{array}{l}
\mathsf{is\_refable}\_R_P.\vec{K}(\bar{V}) \quad \leftarrow R_P(\bar{X}),\ \bar{V} = \bar{X}[\vec{K}]. \\
\mathsf{rem\_refable}\_R_P.\vec{K}(\bar{V}) \leftarrow R_P(\bar{X}), \bar{V} = \bar{X}[\vec{K}],\ \neg\mathsf{del}\_R_P(\bar{X}),\ \neg\,\exists M: \\
\qquad\qquad\qquad\qquad \mathsf{chg}\_R_P.\vec{K}(M, \bar{X}). \qquad\qquad (RCK) \\
\mathsf{new\_refable}\_R_P.\vec{K}(\bar{V}) \leftarrow \mathsf{ins}\_R_P(\bar{X}),\ \bar{V} = \bar{X}[\vec{K}]. \\
\mathsf{new\_refable}\_R_P.\vec{K}(\bar{V}) \leftarrow \mathsf{chg}\_R_P.\vec{K}(M, \bar{X}), M(\bar{X})[\vec{K}] = \bar{V}.
\end{array}
$$

For every *ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$:

$$
\begin{array}{l}
\mathsf{is\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{V}) \quad \leftarrow R_C(\bar{X}),\ V = \bar{X}[\vec{F}]. \\
\mathsf{rem\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{V}) \leftarrow R_C(\bar{X}),\ V = \bar{X}[\vec{F}], \neg\ \mathsf{del}\_R_C(\bar{X}), \\
\qquad\qquad\qquad\qquad \neg\exists M:\ \mathsf{chg}\_R_C.\vec{F}(M, \bar{X}). \qquad\qquad (Refd) \\
\mathsf{new\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{V}) \leftarrow R_C(\bar{X}),\ \ \mathsf{chg}\_R_C.\vec{F}(M, \bar{X}), M(\bar{X})[\vec{F}] = \bar{V}. \\
\mathsf{new\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{V}) \leftarrow\ \mathsf{ins}\_R_C(\bar{X}),\ \bar{V} = \bar{X}[\vec{F}].
\end{array}
$$

4.2.2 *Deletions.* We only have to consider *rac*s of the form $R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON DELETE OF PARENT` (cf. Table I). Logic rules that again describe their *local* behavior are generated for these *rac*s as given below:

—`ON DELETE OF PARENT CASCADE`: Deletions of parent tuples are propagated downwards to every child tuple by rule (DC_1). Additionally, blockings are propagated upwards: if the deletion of a child tuple is blocked, the deletion of the parent tuple is also blocked (DC_2).

—`ON DELETE OF PARENT RESTRICT`: The deletion of a parent tuple is blocked, if there is a referencing child tuple (DR).

—`ON DELETE OF PARENT NOACTION`: The deletion of a parent tuple is blocked, if there is a corresponding child tuple which is neither requested for deletion nor *modified away* (i.e., modified such that it references another parent) (DN).

$$
\begin{array}{|ll|}
\hline
\mathsf{del}\_R_C(\bar{X}) \quad \leftarrow \mathsf{del}\_R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. & (DC_1) \\
\mathsf{blk\_del}\_R_P(\bar{Y}) \leftarrow \mathsf{blk\_del}\_R_C(\bar{X}), \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. & (DC_2) \\
\mathsf{blk\_del}\_R_P(\bar{Y}) \leftarrow \mathsf{is\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{Y}[\vec{K}]). & (DR) \\
\mathsf{blk\_del}\_R_P(\bar{Y}) \leftarrow \mathsf{rem\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{Y}[\vec{K}]). & (DN) \\
\hline
\end{array}
$$

Again, we add the rules for tracing potential deletions (that are only used for analyzing problem situations if an update is rejected):

$$
\begin{array}{l}
\mathsf{pot\_del}\_R(\bar{X}) \leftarrow \triangleright\mathsf{del}\_R(\bar{X}), R(\bar{X}). \\
\text{for each } R_C.\vec{F}, \rightarrow R_P.\vec{K} \text{ `ON DELETE OF PARENT CASCADE`} \qquad (P) \\
\text{(analogous to } (DC_1)\text{):} \\
\mathsf{pot\_del}\_R_C(\bar{X}) \leftarrow\ \mathsf{pot\_del}\_R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}].
\end{array}
$$

4.2.3 *Modifications.* The handling of modifications follows the same principle as presented for deletions, but since the propagation of modifications is handled key-oriented, the details are more involved. Here, the predicates

chg_$R.\vec{A}(M, \bar{X})$ and prp_$R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C, \bar{x})$ are used as described above for describing changing key values and their propagation. In case of a partially modified parent key, the referencing foreign key in the child is regarded as *atomic*: no other update may change parts of it (which would mean to cut the reference to the original parent although it is used for cascading). Thus, with a modification, the whole key value is propagated, even if not all parts of it change. On the other hand, modifications on a tuple trigger a *rac* only if the key referred to in the *rac* is actually changed.

User requests for modifications are decomposed into their effects on keys. An external request is blocked if it causes a forbidden change to a key. For every key (candidate and foreign keys) $R.\vec{A}$:

$$\begin{aligned}
\text{pot_prp_}\triangleright R \rightsquigarrow R.\vec{A}(M, \bar{X}) &\leftarrow \triangleright\text{mod_}R(M', \bar{X}),\ \bar{X}[\vec{A}] \neq M'(\bar{X})[\vec{A}],\\
&\quad\ M = M'[\vec{A}].\\
\text{blk_mod_}R(M, \bar{X}) &\leftarrow \triangleright\text{mod_}R(M, \bar{X}),\ \text{blk_chg_}R.\vec{A}(M', \bar{X}), \qquad (EXT_2)\\
&\quad\ M' = M[\vec{A}].\\
\text{prp_}\triangleright R \rightsquigarrow R.\vec{A}(M, \bar{X}) &\leftarrow \text{mod_}R(M', \bar{X}),\ \bar{X}[\vec{A}] \neq M'(\bar{X})[\vec{A}],\\
&\quad\ M = M'[\vec{A}]\ .
\end{aligned}$$

Interaction of Modifications. Assume a modification mod_$R_P(M_P, \bar{y})$ and a *rac* $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ `ON UPDATE OF PARENT CASCADE` such that the *key value* $R_P.\vec{K}$ of $R_P(\bar{y})$ changes, which is denoted by chg_$R_P.\vec{K}(M_P, \bar{y})$ (analogously, there are relations pot_chg and blk_chg). Then, for every referencing child $R_C(\bar{x})$, this modification is propagated to the corresponding foreign key—$M_C = \vec{F}/M_P(\bar{y})[\vec{K}]$. This is stored in the propagation relation prop: prp_$R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C, \bar{x})$.

The changes of candidate and foreign key values in a (child) tuple are collected: modifications can be founded either by external requests or by propagating modifications from parent relations (e.g., consider a tuple $T(a, b, c, d)$ in Figure 5 where modifications come in to the primary key $T[2, 3]$ from $R[2]$ and $S[1]$).

For a given database schema, (*CH*) defines a finite set of rules for computing all the possible ways a key can change by collecting the elementary modification requests that are propagated along references.

The *only* restriction in this presentation is, that for every foreign or candidate key, only one user modify request is raised that changes the key (which is satisfied if no parallel modifications of the same tuple are allowed; overcoming this restriction requires no conceptual, but some technical expense).

For a given foreign or candidate key $R.\vec{A}$, define the set of sources of influences on $R.\vec{A}$, $S_{R.\vec{A}} \subseteq \mathit{Keys}(R) \times (\mathit{ForeignKeys}(DB) \cup \{\triangleright R\})$ as follows:

—$S_{R.\vec{A}}$ contains all pairs $(R_{P_i}.\vec{K}_i, R.\vec{F}_i)$ such that there is a *rac* $R_{P_i}.\vec{K}_i, \rightarrow R.\vec{F}_i$ `ON UPDATE OF PARENT CASCADE` and F_i overlaps $\vec{A}$ (that is, key references whose propagation influences the value of $R.\vec{A}$), and

—$(R.\vec{A}, \triangleright R) \in S_{R.\vec{A}}$ since external modifications of R also influence the value of $R.\vec{A}$.

For the following rule (CH_1), let S range over all subsets of $S_{R.\vec{A}}$[6]:

$$\begin{aligned}
\mathsf{pot_chg}_R.\vec{A}(M,\bar{X}) \leftarrow\ & (\textstyle\bigwedge_{i\in S}\ \mathsf{pot_prp}_R_{P_i}.\vec{K}_i \rightsquigarrow R.\vec{F}_i(M_i,\bar{X})),\\
& M = \textstyle\bigcup_{i\in S} M_i[\vec{A}] \text{ consistent},\ M(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}].\\
\mathsf{chg}_R.\vec{A}(M,\bar{X}) \leftarrow\ & (\textstyle\bigwedge_{i\in S}\ \mathsf{prp}_R_{P_i}.\vec{K}_i \rightsquigarrow R.\vec{F}_i(M_i,\bar{X})),\\
& (\textstyle\bigwedge_{i\in S_{R.\vec{A}}\setminus S} \neg\exists M_i :\ \mathsf{prp}_R_{P_i}.\vec{K}_i \rightsquigarrow R.\vec{F}_i(M_i,\bar{X})),\\
& M = \textstyle\bigcup_{i\in S} M_i[\vec{A}] \text{ consistent},\ M(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}].
\end{aligned} \qquad (CH_1)$$

Additionally, the interferences between blockings of changes of overlapping keys must be considered: A change on the intersection of two overlapping keys is allowed, if both changes coincide on the intersection. Furthermore, a change of a key is forbidden, if its effect on the intersection with another key is not allowed:
For every foreign key $\vec{F}$ and foreign or candidate key $\vec{A}$ such that $\vec{F}$ and $\vec{A}$ overlap:

$$\begin{aligned}
\mathsf{allow_chg}_R.\vec{F}\cap\vec{A}(M,\bar{X}) \leftarrow\ & \mathsf{chg}_R.\vec{F}(M_1,\bar{X}),\ \neg\ \mathsf{blk_chg}_R.\vec{F}(M_1,\bar{X}),\\
& M = M_1[\vec{F}\cap\vec{A}],\ \mathsf{chg}_R.\vec{A}(M_2,\bar{X}),\\
& \neg\ \mathsf{blk_chg}_R.\vec{A}(M_2,\bar{X}),\ M = M_2[\vec{F}\cap\vec{A}]\,. \qquad (CH_2)\\
\mathsf{blk_chg}_R.\vec{F}(M,\bar{X}) \leftarrow\ & \mathsf{pot_chg}_R.\vec{F}(M,\bar{X}),\ \neg\ \mathsf{allow_chg}_R.\vec{F}\cap\vec{A}(M',\bar{X}),\\
& M' = M[\vec{F}\cap\vec{A}].
\end{aligned}$$

Remark 4.4. Consider a foreign or candidate key $R.\vec{A}$ such that there is only a single constraint $R_P.\vec{K}, \rightarrow R.\vec{F}$ `ON UPDATE OF PARENT CASCADE` such that $\vec{F}$ and $\vec{A}$ overlap. Then, (CH_1) maps the incoming change on $\vec{F}$ to a change of $\vec{A}$:

$$\begin{aligned}
\mathsf{pot_chg}_R.\vec{A}(M,\bar{X}) \leftarrow\ & \mathsf{pot_prp}_R_P.\vec{K} \rightsquigarrow R.\vec{F}(M',\bar{X}),\ M{=}M'[\vec{A}],\\
& M(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}].\\
\mathsf{chg}_R.\vec{A}(M,\bar{X}) \leftarrow\ & \mathsf{prp}_R_P.\vec{K} \rightsquigarrow R.\vec{F}(M',\bar{X}),\ M{=}M'[\vec{A}],\\
& M(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}].
\end{aligned} \qquad (CH_1^0)$$

Modifications of Parent Tuples. When a candidate key in a parent tuple is modified, the usual *local* referential actions apply, yielding the rules given below:

—`ON UPDATE OF PARENT CASCADE`: Changes of parent keys are propagated downwards to foreign keys (MPC_1). If a propagated modification would change a foreign key in a forbidden way, the propagation of the modification and the change of the parent key are also blocked (MPC_2).
—`ON UPDATE OF PARENT RESTRICT`: The change of the parent key $R_P.\vec{K}$ is blocked, if there is a referencing child (MPR).

[6]if $\vec{A}$ is a foreign key, the possibilities can further be restricted to either (i) a propagation along its "own" parent-reference, or (ii) only influences from other sources, see Example 12 in the *electronic appendix*.

—ON UPDATE OF PARENT NOACTION: The change of the parent key $R_P.\vec{K}$ is blocked, if there is a referencing child which is neither requested for deletion nor modified away (*MPN*).

$$\begin{array}{lll}
\mathsf{prp}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C,\bar{X}) & \leftarrow & \mathsf{chg}_R_P.\vec{K}(M_P,\bar{Y}), R_C(\bar{X}),\ \bar{X}[\vec{F}]=\bar{Y}[\vec{K}], \\
 & & M_C=\vec{F}/M_P(\bar{Y})[\vec{K}]. \\
\mathsf{pot_prp}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C,\bar{X}) & \leftarrow & \mathsf{pot_chg}_R_P.\vec{K}(M_P,\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}]=\bar{Y}[\vec{K}], \\
 & & M_C=\vec{F}/M_P(\bar{Y})[\vec{K}]. \\
\mathsf{blk_chg}_R_P.\vec{K}(M_P,\bar{Y}) & \leftarrow & \mathsf{pot_chg}_R_P.\vec{K}(M_P,\bar{Y}),\ \mathsf{blk_prop}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M_C,\bar{X}), \\
 & & \bar{X}[\vec{F}]=\bar{Y}[\vec{K}],\ M_C=\vec{F}/M_P(\bar{Y})[\vec{K}]. \\
\mathsf{blk_prop}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M,\bar{X}) & \leftarrow & \mathsf{pot_prp}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M,\bar{X}),\ \mathsf{blk_chg}_R_C.\vec{F}(M,\bar{X}).
\end{array} \quad (MPC_1),\ (MPC_2)$$

$$\mathsf{blk\_chg}\_R_P.\vec{K}(M_P,\bar{Y}) \leftarrow \mathsf{pot\_chg}\_R_P.\vec{K}(M_P,\bar{Y}),\ \mathsf{is\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{Y}[\vec{K}]). \quad (MPR)$$

$$\mathsf{blk\_chg}\_R_P.\vec{K}(M_P,\bar{Y}) \leftarrow \mathsf{pot\_chg}\_R_P.\vec{K}(M_P,\bar{Y}),\ \mathsf{rem\_refd}\_R_P.\vec{K}\ \mathsf{\_by}\_R_C.\vec{F}(\bar{Y}[\vec{K}]). \quad (MPN)$$

Modifications on Child Tuples. When a foreign key in a child tuple is changed, it must be checked whether there is a suitable parent tuple—note that the check against the reference along which the update has been cascaded is redundant. Thus, we have only to check foreign keys where an incoming cascaded update overlaps the foreign key of *another* reference. A change on a foreign key value $R_C.\vec{F}'$ of a child tuple with respect to a ric $R_C.\vec{F}' \rightarrow R'_P.\vec{K}'$ is blocked if the change is influenced from a propagation along *another ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ (i.e., $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ ON UPDATE OF PARENT CASCADE and $R_C.\vec{F}$ and $R_C.\vec{F}'$ overlap) or from an external modification and the resulting tuple violates $R_C.\vec{F}', \rightarrow R'_P.\vec{K}'$ ON UPDATE OF CHILD.... By considering only changes which are propagated along *another ric*, the inherent negative cycle of "propagation allowed if result's reference exists," "result's reference exists if parent is modified," and "parent is modified if propagation is allowed" is avoided.[7]

For every pair $R_C.\vec{F}', \rightarrow R'_P.\vec{K}'$ ON UPDATE OF PARENT CASCADE and $R_C.\vec{F}, \rightarrow R_P.\vec{K}$ ON UPDATE OF CHILD RESTRICT such that $R_C.\vec{F} \neq R_C.\vec{F}'$ or $R_P.\vec{K} \neq R'_P.\vec{K}'$ and $R_C.\vec{F}$ and $R_C.\vec{F}'$ overlap:

$$\begin{array}{lll}
\mathsf{blk_chg}_R_C.\vec{F}(M,\bar{X}) & \leftarrow & \mathsf{pot_chg}_R_C.\vec{F}(M,\bar{X}),\ \mathsf{prp}_R'_P.\vec{K}' \rightsquigarrow R_C.\vec{F}'(M',\bar{X}), \\
 & & M[\vec{F}\cap\vec{F}'] = M'[\vec{F}\cap\vec{F}'], \\
 & & \neg\ \mathsf{is_refable}_R_P.\vec{K}(\vec{F}[M(\bar{X})]).
\end{array} \quad (MCR_1)$$

$$\begin{array}{lll}
\mathsf{blk_chg}_R_C.\vec{F}(M,\bar{X}) & \leftarrow & \mathsf{pot_chg}_R_C.\vec{F}(M,\bar{X}),\ \mathsf{prp}_\triangleright \rightsquigarrow R_C.\vec{F}(M',\bar{X}), \\
 & & M' \subseteq M,\ \neg\ \mathsf{is_refable}_R_P.\vec{K}(M(\bar{X})[\vec{F}]).
\end{array}$$

Additionally to (MCR_1) which guarantees the existence of a (unique) parent tuple to be referenced in the *current* database state, any modification of the attributes $R_P.\vec{K}$ or deletion of this tuple is blocked: For every *rac* $R_C.\vec{F}, \rightarrow R_P.\vec{K}$

[7]Note that it still occurs with every "diamond" (Figure 1).

ON UPDATE OF CHILD RESTRICT:

$$\begin{aligned}\text{blk\_chg\_}R_P.\vec{K}(M_P,\bar{Y}) \leftarrow\ & \text{pot\_chg\_}R_P.\vec{K}(M_P,\bar{Y}),\ R_C(\bar{X}),\\ & \text{chg\_}R_C.\vec{F}(M_C,\bar{X}), M(\bar{X})[\vec{F}]=\bar{Y}[\vec{K}]. \qquad (MCR_2)\\ \text{blk\_del\_}R_P(\bar{Y}) \leftarrow\ & R_C(\bar{X}),\ \ \text{chg\_}R_C.\vec{F}(M_C,\bar{X}), M(\bar{X})[\vec{F}]=\bar{Y}[\vec{K}].\end{aligned}$$

Analogous to (MCR_1) there is a rule for maintaining $R_C.\vec{F},\rightarrow R_P.\vec{K}$ ON UPDATE OF CHILD NO ACTION which checks if the parent is available *after* execution of the updates. For every pair $R_C.\vec{F}',\rightarrow R'_P.\vec{K}'$ ON UPDATE OF PARENT CASCADE and $R_C.\vec{F},\rightarrow R_P.\vec{K}$ ON UPDATE OF CHILD NO ACTION such that $R_C.\vec{F}\neq R_C.\vec{F}'$ or $R_P.\vec{K}\neq R'_P.\vec{K}'$ and $R_C.\vec{F}$ and $R_C.\vec{F}'$ overlap:

$$\begin{aligned}\text{blk\_chg\_}R_C.\vec{F}(M,\bar{X}) \leftarrow\ & \text{pot\_chg\_}R_C.\vec{F}(M,\bar{X}),\ \ \text{prp\_}R'_P.\vec{K}' \rightsquigarrow R_C.\vec{F}'(M',\bar{X}),\\ & M[\vec{F}\cap\vec{F}'] = M'[\vec{F}\cap\vec{F}'],\\ & \neg\text{rem\_refable\_}R_P.\vec{K}(M(\bar{X})[\vec{F}]),\\ & \neg\text{new\_refable\_}R_P.\vec{K}(M(\bar{X})[\vec{F}]).\\ \text{blk\_chg\_}R_C.\vec{F}(M,\bar{X}) \leftarrow\ & \text{pot\_chg\_}R_C.\vec{F}(M,\bar{X}),\ \ \text{prp\_}\triangleright \rightsquigarrow R_C.\vec{F}(M',\bar{X}),\\ & M'\subseteq M,\ \neg\text{rem\_refable\_}R_P.\vec{K}(M(\bar{X})[\vec{F}]),\\ & \neg\text{new\_refable\_}R_P.\vec{K}(M(\bar{X})[\vec{F}]).\end{aligned} \qquad (MCN)$$

Resulting Modifications. Since modifications are handled key-oriented by (CH_1), the incoming modifications must be collected for every tuple. For a given n-ary relation R, let $S_R = \text{ForeignKeys}(R)\cup\{(1,\ldots,n)\}$.

$$\text{chg\_}R.(1,\ldots,n)(M,\bar{X}) \leftarrow \triangleright\text{mod\_}R(M,\bar{X}). \qquad (CH_{\triangleright})$$

For the following rule (MOD), let S range over all subsets of S_R:

$$\begin{aligned}\text{mod\_}R(M,\bar{X}) \leftarrow\ & (\textstyle\bigwedge_{\vec{F}\in S}\ \text{chg\_}R.\vec{F}(M_{\vec{F}},\bar{X})),\ M=\bigcup_{\vec{F}\in S} M_{\vec{F}}\ \text{consistent},\\ & (\textstyle\bigwedge_{\vec{F}\in S_R\setminus S}\neg\exists M':\ \text{chg\_}R.\vec{F}(M',\bar{X})).\end{aligned} \qquad (MOD)$$

4.2.4 *Insertions.* Since insertions on parent tuples are not critical, only insertions of child tuples have to be handled analogously to (MCR) and (MCN):

—For every *ric* $R_C.\vec{F},\rightarrow R_P.\vec{K}$ ON INSERT OF CHILD RESTRICT:

$$\begin{aligned}\text{blk\_ins\_}R_C(\bar{X}) \leftarrow\ & \triangleright\text{ins\_}R_C(\bar{C}),\ \neg\text{is\_refable\_}R_P.\vec{K}(\bar{X}[\vec{F}]).\\ \text{blk\_chg\_}R_P.\vec{K}(M,\bar{Y}) \leftarrow\ & \text{pot\_chg\_}R_P.\vec{K}(M,\bar{Y}),\ \bar{Y}[\vec{K}]\neq M_P(\bar{Y})[\vec{K}],\\ & \text{ins\_}R_C(\bar{X}),\ \bar{X}[\vec{K}]=\bar{Y}[\vec{K}].\\ \text{blk\_del\_}R_P(\bar{Y}) \leftarrow\ & \text{ins\_}R_C(\bar{X}),\ \bar{X}[\vec{K}]=\bar{Y}[\vec{K}].\end{aligned} \qquad (ICR)$$

—For every *ric* $R_C.\vec{F},\rightarrow R_P.\vec{K}$ ON INSERT OF CHILD NO ACTION:

$$\begin{aligned}\text{blk\_ins\_}R_C(\bar{X}) \leftarrow\ & \triangleright\text{ins\_}R_C(\bar{X}),\ \neg\text{rem\_refable\_}R_P.\vec{K}(\bar{X}[\vec{F}]),\\ & \neg\text{new\_refable\_}R_P.\vec{K}(\bar{X}[\vec{F}]).\end{aligned} \qquad (ICN)$$

4.2.5 *Coherence and Key-Preservation.* The following rule prevents requests which are directly incoherent:

$$\begin{array}{ll}
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow \triangleright\mathsf{ins}_R(\bar{X}),\ \ \mathsf{del}_R(\bar{X}).\\
\mathsf{blk_del}_R(\bar{X}) & \leftarrow \mathsf{ins}_R(\bar{X}).\\
\mathsf{blk_mod}_R(M,\bar{X}) & \leftarrow \triangleright\mathsf{mod}_R(M,\bar{X}),\ \ \mathsf{del}_R(\bar{X}).\\
\mathsf{blk_del}_R(\bar{X}) & \leftarrow \mathsf{mod}_R(M,\bar{X}). \qquad (C)
\end{array}$$

For every *rac* $R_P.\vec{K}, \rightarrow R_C.\vec{F}$ `ON UPDATE OF PARENT CASCADE`:

$$\begin{array}{ll}
\mathsf{blk_prop}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M,\bar{X}) & \leftarrow \mathsf{pot_prp}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M,\bar{X}),\ \ \mathsf{del}_R(\bar{X}).\\
\mathsf{blk_del}_R(\bar{X}) & \leftarrow \mathsf{prp}_R_P.\vec{K} \rightsquigarrow R.\vec{F}(M,\bar{X}).
\end{array}$$

Since propagated modifications are handled key-oriented as foreign-key-modifications, it is sufficient to handle contradicting modifications at this granularity: For every pair of *rics* $R_{P_1}.\vec{K}_1, \rightarrow R.\vec{F}_1$ `ON UPDATE OF PARENT CASCADE` and $R_{P_2}.\vec{K}_2, \rightarrow R.\vec{F}_2$ `ON UPDATE OF PARENT CASCADE` such that $R.\vec{F}_1$ and $R.\vec{F}_2$ overlap:

$$\begin{array}{ll}
\mathsf{blk_prop}_R_{P_1}.\vec{K}_1 \rightsquigarrow R.\vec{F}_1(M_1,\bar{X}) & \leftarrow\ \mathsf{pot_prp}_R_{P_1}.\vec{K}_1 \rightsquigarrow R.\vec{F}_1(M_1,\bar{X}),\\
& \quad\ \mathsf{prp}_R_{P_2}.\vec{K}_2 \rightsquigarrow R.\vec{F}_2(M_2,\bar{X}),\ M_1 \cup M_2 \text{ inconsistent}.
\end{array} \qquad (C)$$

The uniqueness of a candidate key $R.\vec{K}$ is guaranteed by the following rules:

$$\begin{array}{ll}
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow \triangleright\mathsf{ins}_R(\bar{X}),\ \ \mathsf{rem_refable}_R.\vec{K}(\bar{X}[\vec{K}]).\\
\mathsf{blk_chg}_R.\vec{K}(M,\bar{X}) & \leftarrow \mathsf{pot_chg}_R.\vec{K}(M,\bar{X}),\ \ \mathsf{rem_refable}_R.\vec{K}(M(\bar{X})[\vec{K}]).\\
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow \triangleright\mathsf{ins}_R(\bar{X}),\ \ \mathsf{ins}_R(\bar{Y}),\ \bar{X}[\vec{K}] = \bar{Y}[\vec{K}].\\
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow \triangleright\mathsf{ins}_R(\bar{X}),\ \ \mathsf{chg}_R.\vec{K}(M,\bar{Y}), \bar{X}[\vec{K}] = M(\bar{Y})[\vec{K}]. \qquad (K)\\
\mathsf{blk_chg}_R.\vec{K}(M,\bar{Y}) & \leftarrow \mathsf{pot_chg}_R.\vec{K}(M,\bar{Y}),\ \ \mathsf{ins}_R(\bar{X}),\ \bar{X}[\vec{K}] = M(\bar{Y})[\vec{K}].\\
\mathsf{blk_chg}_R.\vec{K}(M,\bar{X}) & \leftarrow \mathsf{pot_chg}_R.\vec{K}(M,\bar{X}),\ \ \mathsf{chg}_R.\vec{K}(M',\bar{Y}),\\
& \quad\ M(\bar{X})[\vec{K}] = M'(\bar{Y})[\vec{K}].
\end{array}$$

On Delete / Update Set Default / Set Null. The additional *rac*s $R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON UPDATE/DELETE SET DEFAULT` and $R_C.\vec{F} \rightarrow R_P.\vec{K}$ `ON UPDATE/DELETE SET NULL` can be handled by variants of the rules (*MPC*), (*MPR*), and (*MPR*). Analogously, *NOT NULL* conditions can be integrated into the framework.

4.2.6 *Declarative Semantics and Results.* The examples in Section 2.3 illustrate different types of ambiguities which can occur for a set *RA* of *rac*s. These ambiguities become apparent by the declarative semantics of our logical formalization P_{RA}. Again, we consider well-founded and stable models:

Mutex. For two mutually exclusive operations (cf. Example 4), if one of them is rejected, the other can be executed: setting some undefined requests to false admits stable models where other updates are true, and the false ones are blocked. This situation is analogous to $\{block_1 \leftarrow exec_2,\ block_2 \leftarrow exec_1\} \cup \{exec_i \leftarrow \neg block_i \mid i = 1, 2\}$.

Self-Attack. For a self-attacking request (cf. Example 5), there is no other support for rejecting it than its "internal contradiction," thus there is no total

(i.e., two-valued) stable model making it true or false. This situation corresponds to $\{exec \leftarrow \neg block,\ block \leftarrow exec\}$, where no total stable model exists.

Definition 4.5. Every 3-valued model $\mathcal{M} := \mathcal{M}(P_{RA}, D, U_{\triangleright})$ defines sets of updates $\Delta_{\mathcal{M}}$ and user requests $U_{\mathcal{M}} \subseteq U_{\triangleright}$ which are true, false, or undefined in $\mathcal{M}$. Let *upd* be any of $\mathsf{ins}\_R(\bar{x})$, $\mathsf{del}\_R(\bar{x})$, $\mathsf{mod}\_R(M, \bar{x})$, then:

$\Delta^{true}_{\mathcal{M}} := \{upd \mid \mathcal{M}(upd) = true\}$, and $U^{true}_{\mathcal{M}} := \{\triangleright upd \in U_{\triangleright} \mid \mathcal{M}(upd) = true\}$ (*analogous for false and undef*).

Again, we first examine the *well-founded model* $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_{\triangleright})$, which provides a safe, *skeptical* semantics that is computable in *polynomial* time, and the the *stable semantics* of P_{RA}. Here, overlapping and subsuming modifications must be taken into account:

Definition 4.6. For any 3-valued model $\mathcal{M} := \mathcal{M}(P_{RA}, D, U_{\triangleright})$, let $\Delta^{true+}_{\mathcal{M}} \subseteq \Delta^{true}_{\mathcal{M}}$ denote the set of *non-subsumed updates*: the set of all $upd \in \Delta^{true}_{\mathcal{M}}$ such that there is no M' which subsumes M and $\mathsf{mod}\_R(M', \bar{x}) \in \Delta^{true}_{\mathcal{M}}$.

The result of applying a set of updates does not change when restricting to non-subsumed updates:

LEMMA 4.7. $D \pm \Delta^{true}_{\mathcal{W}} = D \pm \Delta^{true+}_{\mathcal{W}}$.

THEOREM 4.8 CORRECTNESS: WELL-FOUNDED SEMANTICS.

(1) $\Delta^{true+}_{\mathcal{W}}$ *is admissible,*
(2) $\Delta^{true+}_{\mathcal{W}} = \Delta(U^{true}_{\mathcal{W}})$ *is the set of updates that are induced by* $U^{true}_{\mathcal{W}}$.
(3) $U^{true}_{\mathcal{W}}$ *is admissible*; *the new database after submitting* $U^{true}_{\mathcal{W}}$ *is* $D' = D \pm \Delta^{true}_{\mathcal{W}}$.

PROOF.

(1) Foundedness, completeness, and feasibility are proven using the rules of all *rac*s $ra \in RA$; coherence and key-preservation is guaranteed by the rules specifying the interaction of updates.
(2) $\Delta^{true}_{\mathcal{W}} \subseteq \Delta(U^{true}_{\mathcal{W}})$ follows from foundedness, $\Delta^{true}_{\mathcal{W}} \supseteq \Delta(U^{true}_{\mathcal{W}})$ from completeness.
(3) follows from (1) and (2). □

We see that already $\Delta^{true+}_{\mathcal{W}}$ abstracts from some intermediate results by considering only non-subsumed updates. The game-theoretic characterization—that corresponds to stable models—given in Section 4.3 uses the same abstraction.

The following corollary states that $\mathcal{W}$ is a skeptical approximation: (i) every maximal admissible $U \subseteq U_{\triangleright}$ extends $U^{true}_{\mathcal{W}}$, and (ii) updates classified as false by $\mathcal{W}$ are definitely not admissible:

COROLLARY 4.9. *For every maximal admissible* $U \subseteq U_{\triangleright}$*:*
(i) $U^{true}_{\mathcal{W}} \subseteq U$, (ii) $U^{false}_{\mathcal{W}} \cap U = \emptyset$, (iii) $\Delta^{false}_{\mathcal{W}} \cap \Delta(U) = \emptyset$.

PROOF. Follows from Theorem 4.8 and model-theoretic properties of the well-founded semantics. □

The different types of undefined updates $upd \in U_{\mathcal{W}}^{undef}$ can be characterized according to the different types of controversial atoms:

—$upd \in U$ for every maximal admissible $U \subseteq U_{\triangleright}$ ("diamond"), or
—there are maximal admissible sets $U, U' \subseteq U_{\triangleright}$ such that $upd \in U$ and $upd \notin U'$ ("mutex"), or
—$upd \notin U$ for any admissible $U \subseteq U_{\triangleright}$ ("self-attack").

As already mentioned, "diamonds"—even if they do not cause a problem—cause a negative cycle that is undefined in the well-founded model. Such a case is described in Example 12 in Section A of the *electronic appendix*: there, all actual modifications and changes, as well as all blockings are undefined in the well-founded model. Nevertheless, the modification is admissible.

In such cases, similar to the case of deletions only, there are two stable models: the one that executes the user request, and the one that rejects it. In this case, taking the overestimate of an alternating fixpoint as the intended result—similar to the case where only deletions were considered—is correct. On the other hand, this is *not* correct in cases such as *Mutex* (cf. Example 4), or for self-attacking requests (cf. Example 5).

Stable Models. For further investigation of these cases, we use stable models, which provide a more detailed logical semantics for normal logic programs. Since self-attacking updates exclude the possiblity of total stable models, we have to consider *P-stable* (partial stable) models:

Definition 4.10 *P-, M-Stable Models.* [Eiter et al. 1996] Let $I = \langle I^{true}, I^{false} \rangle$ be a 3-valued interpretation (I^{true} the set ground of atoms that are true, I^{false} the set ground of atoms that are false, $I^{true} \cap I^{false} = \emptyset$). The reduction P/I of a ground-instantiated logic program P is obtained by replacing every negative literal in P by its truth-value with respect to I. Thus, P/I is positive and has a unique minimal (with respect to the truth-order $false <_t undef <_t true$) 3-valued model $\mathcal{M}_{P/I}$.

I is a *P-stable* model, if $\mathcal{M}_{P/I} = I$. A P-stable model I is *M-stable* (*maximal stable*) if there is no P-stable model $J \neq I$ such that $J^{true} \supseteq I^{true}$ and $J^{false} \supseteq I^{false}$.

In contrast to the well-founded model, which is the "most skeptical" P-stable model, M-stable models are "more brave" and handle mutually exclusive requests as expected; in particular, *all* admissible solutions are represented by P-stable models. This fact, and the generalization of Theorem 4.8 is expressed by the following theorem (proven analogously to Theorem 4.8).

THEOREM 4.11 CORRECTNESS, COMPLETENESS: STABLE SEMANTICS.

—*For every P-stable model* $\mathcal{S}$*:*
(i) $\Delta_{\mathcal{S}}^{true+}$ *is admissible,* (ii) $\Delta_{\mathcal{S}}^{true+} = \Delta(U_{\mathcal{S}}^{true})$, (iii) $U_{\mathcal{S}}^{true}$ *is admissible.*
—*For every maximal admissible* $U \subseteq U_{\triangleright}$*, there is an M-stable model* $\mathcal{MS}$ *such that* $U = U_{\mathcal{MS}}^{true}$ *and* $\Delta(U) = \Delta_{\mathcal{MS}}^{true+}$.

M-stable models of P_{RA} almost capture the notion of "optimal" (maximal admissible) solutions. Note that in case of mutual exclusion, there can be

several M-stable models which describe maximal admissible solutions. In case of a "diamond" $\{block \leftarrow \neg exec,\ exec \leftarrow \neg block\}$ (cf. Example 9) there are two M-stable models. However, executing an update should be preferred to blocking it in order to capture the notion of maximal admissibility. Therefore, we define a semantic ordering $<_a$ on P-stable models according to our intended application:

$$\mathcal{S}_1 <_a \mathcal{S}_2 \ :\Leftrightarrow\ U^{true}_{\mathcal{S}_1} \subset U^{true}_{\mathcal{S}_2}.$$

Among M-stable models, $<_a$ prefers those that make more updates true. This holds as well for the user requests as for the resulting updates:

LEMMA 4.12. *For two M-stable models $\mathcal{S}_1$ and $\mathcal{S}_2$, the following is equivalent*:

(1) $\mathcal{S}_1 <_a \mathcal{S}_2$,
(2) *for every* $\triangleright upd \in U_\triangleright$, $\mathcal{S}_1(\triangleright upd) \leq_t \mathcal{S}_2(\triangleright upd)$,
(3) *for every (internal) update* upd, $\mathcal{S}_1(upd) \leq_t \mathcal{S}_2(upd)$, *or* $upd = \mathit{mod\_R}(M, \bar{x})$ *for some* $R, M, \bar{x}$, *and there is an* M' *which subsumes* M *and* $\mathcal{S}_1(upd) \leq_t \mathcal{S}_2(\mathit{mod\_R}(M', \bar{x}))$ (*cf. Definition* 4.6 and Theorem 4.8).

The maximal stable models with respect to $<_a$ represent exactly the maximal admissible sets:

THEOREM 4.13 MAXIMALITY. *Let D and $U_\triangleright$ be as usual. Then, the following sets coincide:*

—*the set of all maximal admissible sets* $U \subseteq U_\triangleright$,
—*the set of all* $U^{true}_{\mathcal{AS}}$ *such that* $\mathcal{AS}$ *is a* $<_a$*-maximal M-stable model of* P_{RA}, D, *and* $U_\triangleright$.

4.2.7 *An Upper Bound.* Above, we have shown that the well-founded model provides a lower bound of maximal admissible subsets. Analogously, an upper bound can be derived. For deletions, Theorems 3.18 and 3.19 stated that the set of all true and undefined user requests (i) is admissible, and (ii) corresponds to a stable model. In case of modifications, this, in general, does not hold:

LEMMA 4.14. *The well-founded model induces an* upper bound *for the set of admissible updates:*
Let $\mathcal{W} = \mathcal{W}(P_{RA}, D, U_\triangleright)$ *be the* well-founded model *of* P_{RA}, D, *and* $U_\triangleright$, *and*

$$U^{ub} := U^{true \cup undef}_{\mathcal{W}} = \{\triangleright upd \in U_\triangleright \mid \mathcal{W}(upd) \in \{true, undef\}\}.$$

—*If* U^{ub} *is admissible, then there is a unique* $<_a$*-maximal stable model* $\mathcal{ASU}$ *of RA*, D, *and* $U_\triangleright$ *such that* $U^{true}_{\mathcal{ASU}} = U^{ub}$. *Then* $\Delta(U^{ub}) = \Delta^{true+}_{\mathcal{ASU}}$, *and the new database is* $D \pm \Delta^{true+}_{\mathcal{ASU}}$.
—*for every* $upd \in U_\triangleright$ *which is in* any *admissible subset of* $U_\triangleright$, $upd \in U^{ub}$.

Thus, the following necessary condition for admissiblility $U_\triangleright$ can be decided in PTIME using the well-founded model:

COROLLARY 4.15. *If $U^{ub} \subsetneq U_{\triangleright}$, then $U_{\triangleright}$ is not admissible.*

Due to the various predicates, the above $\mathcal{ASU}$ is not as easy to describe based on the well-founded model as it has been for deletions (cf. Theorem 3.18). Nevertheless, for the "positive" statements (updates, propagations, and changes) in P_{RA}, we have the following "monotonicity result":

LEMMA 4.16. *Let $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_{\triangleright})$. If U^{ub} is admissible, then for every $upd \in \Delta_{\mathcal{W}}^{true \cup undef}$, $\mathcal{ASU}(upd) = true$, or $upd = \mathsf{mod\_}R(M, \bar{x})$ for some $R, M, \bar{x}$, and there is an M' which subsumes M and $\mathcal{ASU}(\mathsf{mod\_}R(M', \bar{x})) = true$ (cf. Lemma 4.12). (analogously for* prop *and* chg *predicates.)*

Thus, in case that U^{ub} is admissible, it is "safe" to do all changes in the database which are true or undefined in $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_{\triangleright})$. We come back to this issue in Section 5. First, the game theoretic characterization gives more insight into the correctness of the stable model characterization and subsumed updates.

4.3 Game-Theoretic Semantics

The maximal admissible sets of updates can also be characterized in a game-theoretic way (the details of the game can be found in Section B of the *electronic appendix*). This "update game" needs to also consider a *history* of the game that was not required for the simpler game-theoretic characterization in Section 3.3 with deletions only (that was in the famous *win-move*-style).

For given $U_{\triangleright}$, Player I claims a subset $U \subseteq U_{\triangleright}$ to be maximal and admissible. In her first move, Player II chooses to falsify either the maximality or the admissibility. If Player II challenges the maximality, she chooses a proper superset U' such that $U \subsetneq U' \subseteq U_{\triangleright}$ which she claims to be maximal and admissible, then the roles are changed. Thus, after finitely many moves, a player challenges the admissibility of a set U suggested by the other player by examining this set with respect to its coherence and feasibility by questions. The other player has to defend U to be admissible by stepwise showing what updates are actually executed. By doing this, he constructs $\Delta(U)$ (both players are assumed to play optimally). The game is an abstraction of the logic programming characterization in the sense that I uses only non-subsumed updates (anticipating the overall result), thus the details of interfering updates can be ignored. This abstraction step is similar to that in Section 5 for deriving the practical results from the construction of the well-founded and stable models.

Setting and Initialization. The positions of the game are all tuples of the database D, and a sufficient number of empty positions for representing insertions. Actually, the "board" is practically a graphical representation of the database. The game is then played by putting *plates* that represent the update operations performed on the database: Each plate consists of a source tuple ($\in D$), an update (with additional detail information concerning the foundedness), and a result tuple, for example, $\boxed{R(a,b,c)|\mathsf{mod\_}R(1/x, 2/y], (a,b,c))\langle\ldots\rangle|R(x,y,c)}$. At the beginning, Player I puts all plates that describe a set U of updates that he claims to be maximal admissible with respect to given D, RA, and $U_{\triangleright}$. For

example, for $\mathsf{del\_}R(x) \in U$, he puts the plate $\boxed{R(\bar{x})|\mathsf{del\_}R(\bar{x})\langle 0\rangle|\varepsilon}$ on the tuple $R(\bar{x})$.

The Moves. During the game, the admissibility (that is, foundedness, completeness, and feasibility; cf. Definition 4.1) of U in the given situation is challenged by II, and defended by I. II asks a question by pointing to an instance of one of the above aspects, and I has to show how to guarantee the respective property—if he has no answer, he loses (that is, II showed that U is actually not admissible). In most answers, I puts a new plate to show that the database changes in a consistent way.

Child tuples. For any plate that describes an update to a tuple $R_P(\bar{y})$, II can point to a referencing tuple, asking I what happens to it. I answers by putting a plate on the tuple that describes an update (depending whether the reference is maintained via CASCADE, NO ACTION or RESTRICT, different restrictions apply).

Parent tuples. For any plate that describes an update to a tuple $R_C(\bar{x})$ such that a foreign key is changed, II can ask I what parent node is referenced. I has either to show an unchanged tuple, or to show a modification or insertion that generates a suitable parent.

Foundedness. In case of children whose references are maintained by the NO ACTION policy, I claims in his answer that the tuples are accordingly modified—but this update must be propagated from somewhere else (cf. diamonds). Similar to the "delete game" II can ask I to prove that these updates are founded. In contrast to the "delete game," updates can be collected from several parent tuples (cf. the (CH) rules of the logic programming characterization). Thus, the problem is much more involved than in the case of deletions. For each atomic component of the update, II asks a separate question, and I has to show a parent tuple that cascades a suitable modification (by putting a plate whose foundedness is then again attacked by II).

By putting new plates, I constructs $\Delta(U)$ (since II asks for all CASCADE references), and U is admissible if and only if I wins the game; see Theorems B.2 B.4 in the *electronic appendix*. For challenging maximality, II is also allowed to add another plate for $upd \in U_{\triangleright}$, $upd \notin U$ and to change the roles to show that $U \cup \{upd\}$ is also admissible.

Model-theoretic aspects. Note that for given D, RA and $U_{\triangleright}$—in contrast to the case of deletions only—the question is not to win *individual* positions, but to win a game for an *initial setting* $U \subseteq U_{\triangleright}$. For a given initial setting, I either wins or loses, there is no draw. In case that I wins, the whole $\Delta(U)$ is created explicitly on the board—retaining the whole history of the game. For given D, RA and $U_{\triangleright}$, there can be several Us such that I wins the game. Each of them is a maximal admissible subset of $U_{\triangleright}$. In contrast, in the "delete game," the set of all won and drawn positions characterizes the *unique* maximal admissible subset.

5. PRACTICAL ASPECTS

5.1 Admissibility and Execution is Polynomial

In Corollary 4.15, we have given a necessary condition for admissibility of $U_{\triangleright}$ that can be checked in PTIME by using only the well-founded model. In case that this condition is satisfied, we have found in Lemma 4.14 that a certain $<_a$-maximal stable model $\mathcal{AS}$ has to be considered for the final check, and for computing the actual set Δ of updates to the database. In case $U_{\triangleright}$ is not admissible, information is needed about maximal admissible subsets and about the problem situations. In the general case, there is no unique maximal admissible solution for a set of user requests, and an exponential number of stable models may have to be considered. Thus, the computation of *all* maximal admissible subsets is not feasible in practice. But, it is also not needed. For practical use, the following tasks are relevant:

—check if $U_{\triangleright}$ is admissible, and

—if $U_{\triangleright}$ is not admissible, locate the problem situations.

These tasks are now addressed based on the well-founded model (that can be computed in PTIME), building on the results of Section 4.2.7 where an upper bound for the admissible updates has been characterized by the well-founded model. Recall also that for deriving a procedural algorithm for handling deletions in Section 3.5, the negative cycle in the logic programming characterization has been "cut" by taking some ("positive"—the requested deletions) predicates from the overestimate, and the other ("negative"—the blockings) predicates from the underestimate.

—The following predicates are *fixed* predicates: pot_del, pot_prop, pot_chg.

—The following are *positive* predicates: del, prop, mod, ins, chg, allow_chg.

—blockings are *negative* predicates: blk_del, blk_prop, blk_mod, blk_ins, blk_chg.

Positive predicates represent the knowledge of Player I of what plates to play to win the game, including that he only plays non-subsumed updates. We will consider a special stable model that is induced by the fixed and positive predicates. In contrast to guessing a stable model, or to the game-theoretic characterization (that relies on guessing the correct plates), the following "model," $\mathcal{W}^+$ can be generated in polynomial time:

Definition 5.1. Given $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_{\triangleright})$, let $\mathcal{W}^+$ be defined as follows (note that $\mathcal{W}^+$ is in general not a model of P_{RA}):

—for fixed predicates (evaluate to *true* or to *false* in $\mathcal{W}$), $\mathcal{W}^+(atom) := \mathcal{W}(atom)$,

—for positive predicates, $\mathcal{W}^+(atom) = true$ if $\mathcal{W}(atom) \in \{true, undef\}$ and $atom$ is not subsumed by another one.

—$\mathcal{W}^+(atom) := false$ for all other atoms, including those over negative predicates.

Then, $\mathcal{W}^+$ has to be checked as to whether there is a stable model that coincides with $\mathcal{W}^+$ on *positive* and *fixed* predicates. We call $\mathcal{W}^+$ *positive-stable* if this

is the case, which corresponds to a game where Player II has no successful attacks.

First, we give the relationship to stable models, especially to the $<_a$-maximal M-stable models that characterize *maximal* admissible sets:

THEOREM 5.2 POSITIVE-STABLE VS. STABLE MODELS. *If $\mathcal{W}^+$ is positive-stable, then there is a (unique) $<_a$-maximal M-stable model $\mathcal{AS}$ that coincides in all atoms over* positive *and* fixed *predicates with $\mathcal{W}^+$. $\mathcal{AS}$ is computed by applying the computation of the well-founded model starting with $\mathcal{W}^+$ (instead of $\emptyset$).*

PROOF. As described above, $\mathcal{W}^+$ corresponds to cutting negative cycles in the rules with priority to executing updates. In a stable or positive-stable model, these atoms reproduce themselves. The other atoms (blockings etc.) only "fill" the gaps in the stable model. If no blockings are derived that "kill" updates in $\mathcal{W}^+$, it is stable (it cannot be attacked by Player II).

$\mathcal{W}^+$ corresponds to the *unique* $<_a$-maximal M-stable model since in Corollary 4.9 and Lemma 4.14, it has been shown that updates $upd \in U_{\triangleright}$ that are false in $\mathcal{W}$ are not contained in any admissible $U \subset U_{\triangleright}$. □

In the following, we call the above $\mathcal{AS}$ the *positive-stable model to $\mathcal{W}$*. Note again that such a model does not always exist (e.g., if $\mathcal{W}^+$ contains mutually exclusive or "self-killing" updates). With respect to the upper bound that has been discussed in Section 4.2.7, the above theorem checks whether U^{ub} is admissible. Especially, concerning the admissiblity check for $U_{\triangleright}$, the following result shows that this task can be done in PTIME since computing $\mathcal{W}$ and $\mathcal{AS}$ is polynomial:

COROLLARY 5.3 ADMISSIBILITY OF $U_{\triangleright}$. *If $U^{ub} = U_{\triangleright}$ and $\mathcal{W}^+$ is positive-stable (that is, there is an $\mathcal{AS}$ which is the positive-stable model to $\mathcal{W}$), then $U_{\triangleright}$ is admissible.*

For computing the set of induced updates, we again consider the game-theoretic characterization with its set of "winning plates":

LEMMA 5.4 INTERNAL UPDATES IN $\mathcal{AS}$ VS. GAME. *If $\mathcal{AS}$ is the positive-stable model to $\mathcal{W}$, then $\Delta^{true+}_{\mathcal{AS}} = \Delta^{true+}_{\mathcal{W}^+} = \Lambda$ where Λ is as defined in Section B.3 based on the plates that are played by I for winning the game.*

Proof Sketch. The non-subsumed updates are exactly those that are represented by the plates in the corresponding game for $U_{\triangleright}$ (that is won by Player I). Positive-stability means that there are no blockings and conflicts that interfere with the assumed updates (regarding Example 11, there can be blocked updates that are subsumed by others that are not blocked). This is equivalent to the fact that Player II does not find any argument to win the game.

The following corollary states that the admissibility check, and if it is successful, the transition to the subsequent database state, can be computed in polynomial time:

COROLLARY 5.5 MAIN RESULT. *If $\mathcal{AS}$ is the positive-stable model to $\mathcal{W}$ and $U^{true}_{\mathcal{AS}} = U_{\triangleright}$, then $U_{\triangleright}$ is admissible and $\Delta^{true+}_{\mathcal{AS}} = \Delta(U_{\triangleright})$. The whole problem can be solved in polynomial time.*

Thus, the steps for executing a set $U_{\triangleright}$ of updates on a database D are as follows:

(1) compute $\mathcal{W} = \mathcal{W}(P_{RA}, D, U_{\triangleright})$,
(2) check whether $U^{true,undef}_{\mathcal{W}} = U_{\triangleright}$,
(3) if not, reject, otherwise check if there is a positive-stable model $\mathcal{AS}$ to $\mathcal{W}$,
(4) if not, reject, otherwise compute $\Delta^{true+}_{\mathcal{AS}}$ and apply it to the database.

Note that if $U_{\triangleright}$ is accepted by the SQL semantics as presented in Horowitz [1992] and specified in the SQL3 standard [ANSI/ISO 1999] (see also Section 2.4), the semantics coincides with ours. Similarly to the considerations in Section 3.6, the global correctness is implied by the correct specification of the individual rules and the meta-correctness of the logic programming semantics. Again, in contrast to the procedural semantics for SQL, the logic-based characterization also provides information if $U_{\triangleright}$ is rejected.

5.2 Troubleshooting in Case of Rejected Updates

Similarly to Section 3.7, in case $U_{\triangleright}$ is not admissible, the information contained in the well-founded model $\mathcal{W}$ and the corresponding stable model $\mathcal{AS}$ can be used for deriving debugging hints. Problem situations are defined analogously as in Definition 3.24; additionally there can be problems due to conflicts between updates.

The above considerations also apply for the admissibility of $U^{ub} = U^{true,undef}_{\mathcal{W}}$ that provides an upper bound to which subset of $U_{\triangleright}$ can possibly be admissible. Thus, first problems are already identified during the computation of $\mathcal{W}$. The first time that a blocking for an $upd \in U_{\triangleright}$ or for a propagation along a cascading reference is derived in an underestimate, a problem situation is identified that can be reported.

Later, if $U^{ub} = U_{\triangleright}$, but it is still not admissible, the check as to whether $\mathcal{W}^+$ is positive-stable immediately shows where the problems are located: the first rule in the subsequent computation of the well-founded model that derives a blocking against an update in $\Delta^{true+}_{\mathcal{W}^+}$ identifies the problem situation. The conclusions as to how to cure the problem are the same as described in Section 3.7.

5.3 Refined Analysis

In the above well-founded model $\mathcal{W}$, in general most updates will be undefined, thus, the subsequent investigation of $\mathcal{W}^+$ is generally necessary. There are several typical situations, where even single, admissible updates yield only undefined updates in $\mathcal{W}$, for example, diamonds or cyclic dependencies. These can be detected by schema analysis, and then can be handled by refining the first (*MCN*) rule.

Diamonds. For a given schema, it is possible to detect diamonds a-priori by checking *transitive* dependencies and appropriately modifying the program

(similar to the use of the *referential action graph* in Gallagher [1986] and Horowitz [1989]). For every diamond, the key on top of the diamond, and its children are considered when reasoning about changes. Changes inside the diamond are only checked if they are due to references that come into the diamond from outside.

Cyclic Dependencies in a Single Update. Cyclic dependencies also lead to undefined modifications and blockings. In fact, cyclic dependencies are a special case of the diamond where the top relation is the same as the bottom relation.

Cyclic Dependencies between Updates. `NO ACTION` references can lead to cyclic dependencies between a set of updates. In this case, the well-founded model yields undefined updates although often all of them as a group are admissible. The "second try," based on $\mathcal{W}^+$ is then successful in deriving a total model.

In the above cases, undefined atoms in the well-founded model were caused by problems of the logic programming characterization and could be eliminated by either changing to stable models or rewriting the program according to a given schema. Additionally, undefined atoms in the well-founded model can be caused by mutual exclusion or "self-killing" updates.

Self-Killing Updates. An update is self-killing if it causes conflicting cascaded updates. In most cases this points to a severe design error. Again, potential problems can be detected by regarding the transitive closure of referential actions.

Mutual Exclusion: Multiple Admissible Sets. Mutual exclusion is the only case where the expressive power of the well-founded model is not sufficient (it cannot represent the choice between alternatives in the result). It also results in undefined updates and blockings wherever atoms are involved in mutual exclusion. Here, stable models provide the cure to the problem.

6. RELATED WORK

Referential integrity for relational databases was first considered in Codd [1970] with an implicit *or-semantics* between a foreign key and *several* parent keys (in different tables). The definition has been reformulated in Date [1981]; specifying whether a parent key must be present in *all*, *some*, or *exactly one* of the parent tables. Later [Date 1990], the *or*-semantics was cancelled and *referential actions* were defined for enforcing these referential integrity constraints after the execution of updates.

In the SQL2 [ANSI/ISO 1992a] standard, referential integrity constraints and referential actions are specified using the syntax given in Section 2.2, according to the above *all*-semantics. While the syntactic specification of referential actions in SQL is in a declarative style, the given procedural semantics causes ambiguities in some cases of network-like referential structures [Hammer and McLeod 1975; Markowitz 1990, 1991b] due to different execution orderings of referential actions (see also Date [1990] and Date and Darwen [1994]). The same characterization was also used in the early SQL3 working drafts, for example, ANSI/ISO [1991, 1992b, 1994].

Motivated by the problem of ambiguities, several research directions have been followed. *Schema-based* analysis provides conditions that are sufficient, but more restrictive than necessary. In Markowitz [1991b], such a *schema-based* strategy is presented to exclude ambiguous situations. The approach considers only delete operations with referential actions `CASCADE`, `SET NULL`, and `RESTRICT`; neither `DELETE NO ACTION` nor `UPDATE CASCADE` are allowed. Thus, already situations similar to the "diamond" from Example 3 (which has an intuitively clear semantics) are not considered. The approach is tuple-oriented, and there is no "natural" extension of the solution for updates. The approach is refined in Markowitz [1994], dropping several simplifying assumptions.

Another schema-based approach using a relation-column-based (instead of tuple-based) *referential action graph* was presented in Gallagher [1986] and Horowitz [1989].

A "*semantic*" approach is followed by Markowitz [1990]: Starting from an object-oriented or EER model of an application, criteria are given as to how to map this model to a relational model with *ric*s and *rac*s, depending on the semantics of a reference (of "blocking" or "cascading" nature). Here, the problem of cascading modifications is solved by the introduction of *surrogate attributes*, which simulate a kind of object-identity. Then, it is shown that the result does not have the negative properties identified in Date [1990]—there cannot occur any ambiguities.

At that time, in commercial database systems `ON DELETE/UPDATE NO ACTION` was used by default, only `ON DELETE CASCADE` could be specified optionally. Thus, especially cascading modifications were not supported. For an overview of the support for referential integrity in commercial RDBMSs at that time, see Markowitz [1991a].

Compile-time approaches that are based on triggers and schema information only, are too restrictive, since their criteria are sufficient but not necessary to avoid ambiguities (cf. the abovementioned undecidability of the problem). In other words, they prohibit a schema if there is the *possibility* of an anomaly in a "wrong-use-worst-case." Reinert [1996] shows that it is undecidable whether a given *schema* with referential actions can, for some database instances, lead to ambiguous update situations under the SQL2 semantics. In contrast, the problem becomes decidable for a *given* situation: when a database instance and a set of updates is given. Such considerations have led to the investigation of *run-time* approaches:

Horowitz [1992] presented a procedural execution model using a marking algorithm based on bookkeeping about deletions and modifications under *the restriction that keys consist only of a single column*. Thus, this (in practice unrealistic) assumption avoids the problems of overlapping keys and foreign keys. The evaluation model has been extended and accepted for the SQL3 [ANSI/ISO 1999] standard (see also Section 2.4); also commercial DBMS implementations conform to it (as far as they support referential actions). An integration of the semantics of SQL triggers and declarative constraints (including referential integrity constraints and referential actions) is investigated in Cochrane et al. [1996]. Their model is fully compatible with the SQL2 (and also the later SQL3)

requirements. The internal problems of referential actions are not considered. From the aspect of *active databases*, production rules have been considered for investigating triggers and referential actions, for example, Abiteboul and Vianu [1991], Picouet and Vianu [1995]; Ceri and Widom [1990] and Widom et al. [1991].

SQL3 [ANSI/ISO 1999] extends the above-mentioned marking strategy given in Horowitz [1992] for cascading referential actions (dropping the unary-key-constraint). The actual SQL3 specification is then given in terms of a complex characterization via `BEFORE` triggers. SQL3 also specifies several "levels": For the *intermediate SQL* specification (which is currently supported by commercial systems), update rules are not allowed (i.e., the only `ON UPDATE` action is the default `ON UPDATE NO ACTION`). For *full SQL*, all referential actions are allowed. Note that the approach is *not key-oriented* but tuple-oriented: already *non-interfering* updates on a tuple (concerning disjoint foreign keys) are explicitly forbidden. For updates that are allowed by full SQL3, our semantics coincides with the one specified for SQL3—note that our semantics does not use longwinded procedural or trigger-based algorithms, but only specifies the intuitive *local* behavior whereas the *global* semantics is naturally given by the well-known logic programming semantics.

Commercial DBMSs. For a long time, in commercial DBMSs, `ON DELETE/UPDATE NO ACTION` was used by default, only `ON DELETE CASCADE` could be specified optionally. Only recently, Microsoft SQL Server (since 2000) and PostgreSQL support `ON UPDATE CASCADE`.

7. CONCLUSIONS

We have investigated the semantics of arbitrary sets of user-requested updates to a database in the presence of referential integrity constraints and referential actions. Our results contribute both to database theory, and to the practical implementation of referential actions.

Theoretical Aspects. We have shown how a logic-based specification of referential actions can yield a better understanding of ambiguities and conflicts: the logic programming characterization shows how to obtain a declarative *global* semantics that is computable in polynomial time from a concise and intuitive *local* definition of *rac*s accessing only two tuples at a time. The game-theoretic formalizations abstract from details and provide additional insight into the behavior of *rac*s and are used to establish the correctness of the logic programming formalization.

In contrast to the latest SQL3 standard [ANSI/ISO 1999], where the semantics is specified by a complicated, procedural computation (after previous versions that suffered from ambiguities), our results show that the well-known semantics for general logic programs already unambiguously specify a semantics of *rac*s (that coincides with that of ANSI/ISO [1999]).

Provided one accepts the appropriateness of the well-established Logic Programming semantics, our semantics is the "natural" semantics of referential integrity constraints and referential actions specified in SQL's ECA-style syntax.

Practical Aspects. For the restricted case of deletions only, we have shown how this specification can be transformed into an efficient algorithm and implementation in an arbitrary procedural language. For modifications *in the presence of referential actions of the form* `ON UPDATE CASCADE` (which is still only partly supported in commercial database systems), we have shown how the logical characterization can be used for efficiently checking admissibility of a set of updates, and for computing the subsequent database state. In both cases, if the initial set of updates is not admissible, the computation can also be used for detecting the exact location of the problems, and for giving hints as to how to change the application specification.

ELECTRONIC APPENDIX

Attached to the Citation Page for this article in the ACM Digital Library.

See: http://doi.acm.org/10.1145/582410.582411 Appendicies and Supplements.

It contains an example for the logic-programming characterization of referential actions in the case of update operations that has been given in Section 4.2. Additionally, it describes the *detailed* game-theoretic characterization of that case that has been sketched in Section 4.3, illustrates it by an example, and shows its equivalence with the logic programming characterization.

ACKNOWLEDGMENTS

The authors thank Joachim Reinert and Georg Lausen for fruitful discussions especially in the early stages of this work, and Jörg Flum for interesting discussions about logic programming and game theory. We also want to thank Richard Snodgrass and three anonymous reviewers for their constructive critique and suggestions as to how to improve the presentation of the different aspects of the paper.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison Wesley.

ABITEBOUL, S. AND VIANU, V. 1991. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci. 43*, 1, 62–124.

ANSI/ISO. 1991. (ISO/ANSI working draft) Database Languages—SQL3 ANSI X3H2-91-254.

ANSI/ISO. 1992a. Information Technology—Database Languages—SQL2 Standard.

ANSI/ISO. 1992b. (ISO/ANSI working draft) Database Languages—SQL3 ANSI X3H2-92-154.

ANSI/ISO. 1994. (ISO/ANSI working draft) Database Languages—SQL3 ANSI X3H2-94-080.

ANSI/ISO. 1999. Information Technology—Database Languages—SQL3 Standard.

CERI, S., COCHRANE, R., AND WIDOM, J. 2000. Practical applications of triggers and constraints: Success and lingering issues (10-year award for Ceri and Widom [1990]). In *International Conference on Very Large Data Bases (VLDB)*. 254–262.

CERI, S. AND WIDOM, J. 1990. Deriving production rules for constraint maintenance. In *International Conference on Very Large Data Bases (VLDB)*. 566–577.

COCHRANE, R., PIRAHESH, H., AND MATTOS, N. 1996. Integrating triggers and declarative constraints in SQL database sytems. In *International Conference on Very Large Data Bases (VLDB)*. 567–578.

CODD, E. 1970. A relational model for large shared data banks. *Comm. ACM 13*, 6, 377–387.

DATE, C. 1981. Referential Integrity. In *International Conference on Very Large Data Bases (VLDB)*. 2–12.

DATE, C. 1990. *Relational Database Writings 1985–1989*. Addison-Wesley.

DATE, C. AND DARWEN, H. 1994. *A Guide to the SQL standard: A User's Guide to the Standard Relational Language SQL*. Addison-Wesley.

DAYAL, U. 1988. Active database management systems. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*. Morgan Kaufmann, 150–169.

DIX, J. 1995. Semantics of logic programs: Their intuitions and formal properties. In *Logic, Action and Information*, A. Fuhrmann and H. Rott, Eds. de Gruyter.

EITER, T., LEONE, N., AND SACCÀ, D. 1996. The expressive power of partial models for disjunctive deductive databases. In *International Workshop on Logic in Databases (LID)*. Springer LNCS 1154.

ESWARAN, K. P. 1976. Specification, Implementation and Interactions of a Trigger Subsystem in an Integrated Database System. IBM Research Report RJ-1820(26414).

FAN, W. AND SIMÉON, J. 2000. Integrity constraints for XML. In *ACM Symposium on Principles of Database Systems (PODS)*. 23–34.

GALLAGHER, L. 1986. Referential integrity. ANSI X3H2-86-164.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *International Conference on Logic Programming (ICLP)*. 1070–1080.

HAMMER, M. AND MCLEOD, D. 1975. Semantic Integrity in a Relational Data Base System. In *International Conference on Very Large Data Bases (VLDB)*. 25–47.

HOROWITZ, B. 1989. Relaxing referential integrity syntax rule restrictions. ANSI X3H2-89-260.

HOROWITZ, B. M. 1992. A run-time execution model for referential integrity maintenance. In *International Conference on Data Engineering (ICDE)*. 548–556.

LUDÄSCHER, B. 1998. Integration of active and deductive database rules. Ph.D. thesis, Institut für Informatik, Universität Freiburg.

LUDÄSCHER, B. AND MAY, W. 1998. Referential actions: From logical semantics to implementation. In *6th International Conference on Extending Database Technology (EDBT)*. Springer LNCS 1377, 404–418.

LUDÄSCHER, B., MAY, W., AND LAUSEN, G. 1996a. Nested transactions in a logical language for active rules. In *International Workshop on Logic in Databases (LID)*. Springer LNCS 1154, 196–222.

LUDÄSCHER, B., MAY, W., AND LAUSEN, G. 1997. Referential actions as logical rules. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS)*. 217–227.

LUDÄSCHER, B., MAY, W., AND REINERT, J. 1996b. Towards a logical semantics for referential actions in sql. In *6th Intl. Workshop on Foundations of Models and Languages for Data and Objects (FMLDO)*. 57–72.

MARKOWITZ, V. M. 1990. Referential integrity revisited: An object-oriented perspective. In *International Conference on Very Large Data Bases (VLDB)*. 578–589.

MARKOWITZ, V. M. 1991a. Problems underlying the use of referential integrity in relational database management systems. In *International Conference on Data Engineering (ICDE)*.

MARKOWITZ, V. M. 1991b. Safe referential integrity structures in relational databases. In *International Conference on Very Large Data Bases (VLDB)*. 123–132.

MARKOWITZ, V. M. 1994. Safe referential integrity and null constraint structures in relational databases. *Information Systems 19*, 4, 359–378.

PICOUET, P. AND VIANU, V. 1995. Semantics and expressiveness issues in active databases. In *ACM Symposium on Principles of Database Systems (PODS)*.

PRZYMUSINSKI, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 191–216.

REINERT, J. 1996. Ambiguity for referential integrity is undecidable. In *Constraint Databases and Applications*, G. Kuper and M. Wallace, Eds. Springer LNCS 1034, 132–147.

VAN GELDER, A. 1993. The alternating fixpoint of logic programs with negation. *J. Comput. Syst. Sci. 47*, 1, 185–221.

VAN GELDER, A., ROSS, K., AND SCHLIPF, J. 1991. The well-founded semantics for general logic programs. *J. ACM 38*, 3 (July), 620–650.

WIDOM, J., COCHRANE, R., AND LINDSAY, B. 1991. Implementing set-oriented production rules as an extension to Starburst. In *International Conference on Very Large Data Bases (VLDB)*. 275–285.

XML Schema 2000. XML Schema Parts 1/2: Structures/Datatypes. W3C Candidate Recommendation, `www.w3.org/TR/xmlschema-1,2`.

Received January 2001; revised April 2002, July 2002; accepted July 2002