Understanding the Global Semantics of Referential Actions using Logic Rules

(This is a preliminary release of an article accepted by ACM TODS. The definitive version is currently in production at ACM and, when released, will supersede this version.)

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 been defined yet.

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 PRA. The declarative, logic programming semantics of PRA 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, i.e. 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 PRA. We show that for a given set of user requests, in 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

General Terms: Theory, Algorithms

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

Authors’ addresses: W. May, Institut für Informatik, Universität Freiburg, D-79110 Freiburg, Germany; may@informatik.uni-freiburg.de
B. Ludäscher, San Diego Supercomputer Center, UCSD, La Jolla, CA 92033-0505, USA; ludasch@ucsd.edu

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires specific permission and/or a fee.

© 20YY ACM 0362-5952/20YY/0300-0001 $5.00

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 (sic'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, until today, researchers 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 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 how 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.

Already [Date and Darwen 1994] and [Date 1990] report the problem of unpredictable behavior when realizing rac’s based on SQL triggers, i.e., 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 when 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 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.

[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 has later been 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 \(U_o\), and a set RA of rac's, find the set of updates \(\Delta\) that (i) is complete wrt. \(U_o\), (ii) preserves referential integrity in the new database state \(D'\), and (iii) reflects the intended meaning of RA, i.e., how referential integrity should be enforced.

\(U_o\) 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 \(U_o\) in order to illustrate certain interferences. Sometimes, the updates in \(U_o\) 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 \(\Delta\).

In case that no such \(\Delta\) exists and \(U_o\) is rejected, we investigate maximal admissible subsets of \(U_o\), and derive hints where the problems are located and how they can possibly be solved. Assume that \(U_o\) 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 the analysis of the ric's and rac's can be helpful for identifying the problem:

— the schema may be flawed, i.e., 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, i.e., which ric's are violated, and which tuples cause the problem.

\(^1\)Dix [Dix 1996] formally defines this notion using very general principles.
Contributions. From a theoretical perspective, we aim to provide 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, i.e., the well-founded model and the stable model as "reasonable" semantics.

The logic-based characterization does not only provide 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 $U_{\beta}$ is admissible, and
- in case that it is, computing the set of updates to be accomplished (implying that UN 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 semantical 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 $U_{\beta}$ 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; 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 intentions 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 non-admissible updates, something in this specification must be wrong, and the database system – supported by the semantics – can give hints 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 $\Delta$ of internal updates could be of interest. Moreover, the possibility to react on rejected updates by deriving hints for the application designer 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 [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. 1996]. 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, i.e., 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 also 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 $U_D$, 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). From the logic programming characterization, an algorithm for computing the maximal admissible solution is derived (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 wrt. 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 how to debug an application in case a set of updates is rejected is described in Section 3.7.

In Section 4, we extend the investigations to include modifications. We start again 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) the check 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, i.e., unlike the relational model with named attributes [Abiteboul et al. 1995], in the unnamed Datalog-style notation, each argument position of a predicate is (implicitly) 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 $R(\tilde{A})$ consists of a relation name $R$ and a sequence of attributes $\tilde{A} = (A_1, \ldots, A_n)$. We identify attribute names $A_i$ of $R$ with the integers $1, \ldots, n$. Given $\tilde{A}$, a (possibly reordered) subsequence of $\tilde{A}$ (e.g., a key) is a vector $\tilde{K} = (A_{i_1}, \ldots, A_{i_k})$ 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 $\tilde{K}$ may have a different order than in $\tilde{A}$.

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

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

$$\forall \tilde{x}_1, \tilde{x}_2 (R(\tilde{x}_1) \land R(\tilde{x}_2) \land \tilde{x}_1[\tilde{K}] = \tilde{x}_2[\tilde{K}] \rightarrow \tilde{x}_1 = \tilde{x}_2).$$

(\varphi_{\text{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 wrt. the original attribute list of Country).

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
### Definition 2.2 Referential Integrity Constraints.

A referential integrity constraint (ric) is an expression of the form

\[ R_C: \tilde{F} \rightarrow R_P. \tilde{K}, \]

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

\[ \forall \tilde{x} \left( R_C(\tilde{x}) \rightarrow \exists \tilde{y} \left( R_P(\tilde{y}) \land \tilde{x}[\tilde{F}] = \tilde{y}[\tilde{K}] \right) \right). \]  

(\( \varphi_{ric} \))

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).

For example, changing the code GB into UK in Country would violate both ric’s; this can be remedied by propagating the change, i.e., applying the same renaming also 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, e.g., 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 \( \text{ins}_R(\tilde{x}) \), \( \text{del}_R(\tilde{x}) \), and \( \text{mod}_R(M, \tilde{x}) \). Here, \( M \) is a set of pairs \( i/c \) meaning that the \( i \)-th attribute of \( R(\tilde{x}) \) should be set to the constant \( c \). We say that a modification \( \text{mod}_R(M_1, \tilde{x}) \) subsumes a modification \( \text{mod}_R(M_2, \tilde{x}) \) if \( M_1 \supseteq M_2 \). As a shorthand for \( \text{mod}_R([1/d, 3/e], (a, b, c)) \), we sometimes write \( \text{mod}_R(a/d, b, c/e) \).

### 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 \( R_C: \tilde{F} \rightarrow R_P. \tilde{K} \) 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 ($\varphi_{ric}$) 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 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 (wrt. 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:

\[
R_C \cdot F \rightarrow R_P \cdot K \quad \text{on \{del | ins | mod\} \{parent | child\} \{propagate | restrict | wait\}}
\]

2.3 The Problem of Ambiguity

With this local specification of behavior where each rac triggers an action on every child tuple (wrt. 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

<table>
<thead>
<tr>
<th></th>
<th>$R_P$</th>
<th>$R_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>propagate</td>
<td>ins</td>
<td>del</td>
</tr>
<tr>
<td>ok</td>
<td>•</td>
<td>—</td>
</tr>
<tr>
<td>restrict</td>
<td>ok</td>
<td>•</td>
</tr>
<tr>
<td>wait</td>
<td>ok</td>
<td>•</td>
</tr>
</tbody>
</table>

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

Table I. Operations and Possible Repairs
of a database instance $D$ (depending on the execution order of referential actions), $D$ is called ambiguous wrt. 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 $R_U$ to $R_P$, rac's are denoted by dashed (propagate) or double (restrict) arcs. Let $U_D = \{\text{del} R_1(a)\}$ 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 $R_1 \sim R_3 \sim R_4$, 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 $\text{del}R_1(a)$ is rejected, and the database state remains unchanged, i.e., $D' = D$.

2. If execution follows the path $R_1 \sim R_2 \sim R_4$, the tuples $R_2(a,b)$ and $R_4(a,b,c)$ are requested for deletion. Hence, the rac for $R_4.(1,3) \rightarrow R_5.(1,2)$ can assume that $R_3(a,b,c)$ 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 wrt. 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:

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
EXAMPLE 4 MUTEX. Consider modifications $\triangleright \text{mod}_R(a/b)$ and $\triangleright \text{mod}_R(a/c)$.\footnote{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.} 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 $\triangleright \text{mod}_R(1/b, 2/c]], (a, a))$ triggers $\text{mod}_R_1(1/b], (a))$ and $\text{mod}_R_2([1/c], (a))$; $\text{mod}_R_1(1/b], (a))$ triggers $\text{mod}_R_3([1/b], (a))$, and $\text{mod}_R_2([1/c], (a))$ triggers $\text{mod}_R_3([1/c], (a))$.

Since the original request $\triangleright \text{mod}_R(1/b, 2/c]], (a, a))$ 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 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 wrt. 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 $U_\triangleright = \{\triangleright \text{del}_{R_P}(a, b), \triangleright \text{mod}_{R_P}(1/a], (d, e))\}$. 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, $U_\triangleright = \{\triangleright \text{del}_{R_P}(a, b), \triangleright \text{mod}_{R_P}(1/a], (d, e))\}$ 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 \( R_C : \overrightarrow{F} \rightarrow R_P, \overrightarrow{K} \) are specified with the definition of the child table. SQL allows referential actions only for modifications at the parent tuple:

\[
\text{CREATE | ALTER} \quad \text{TABLE} \quad R_C
\]

...\[
\text{FOREIGN KEY} \quad \overrightarrow{F} \quad \text{REFERENCES} \quad R_P, \overrightarrow{K}
\]

[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 \textit{SET NULL/DEFAULT} actions, since they are a special case of modifications.

In their \textit{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 \textit{current} database state in our localized ECA-like characterization:

— \textit{CASCADE} is the same as our \textit{propagate} and propagates the update from the parent to the referencing tuples (evaluation is wrt. the \textit{original} database state).

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

— \textit{NO ACTION} is the same as \textit{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 \textit{after} completing the updates.

Although this semantics is easy to understand on first sight, it is not a direct, \textit{local} semantics that can be implemented straightforwardly by ECA-rules or triggers in the style of [Dayal 1988; 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 \textit{NO ACTION}, there is a (negative) cyclic dependency in the \textit{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 \textit{global} semantics of rac’s.

The SQL2 standard [ANSI/ISO 1992a] – where \textit{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, i.e., that the outcome depends on the order how tuples are updated. The same characterization was given in the upcoming SQL3 drafts (e.g.,

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
Understanding the Global Semantics of Referential Actions using Logic Rules

the 1991 version [ANSI/ISO 1991] cited by [Horowitz 1992], and [ANSI/ISO 1994]). Concerning these specifications, [Horowitz 1992; 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, 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 to mark 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 iff 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 how to prepare a revised request which 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 like in Example 3, it is necessary to specify the intended global semantics of rac's. In this section, we investigate ric's $R_C.\vec{F}\rightarrow R_P.\vec{K}$ with corresponding rac's of the form

$$R_C.\vec{F}\rightarrow R_P.\vec{K} \text{ on delete } \{\text{CASCADE|RESTRICT|NO ACTION}\}$$

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 2A. 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 wrt. 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 $U_\triangledown = \{\triangledown\text{del}_R(\bar{x}_1), \ldots, \triangledown\text{del}_R(\bar{x}_n)\}$ 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:

$$\mathcal{DC} := \left\{ (R_C(\bar{x}), R_P(\bar{y})) \in D \times D \mid R_C.\vec{F}\rightarrow R_P.\vec{K} \text{ on delete CASCADE} \in RA \text{ and } \pi[\vec{F}] = \vec{g}(\vec{K}) \right\},$$

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

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

- **founded**, if for all $\text{del}_R(\bar{x}) \in \Delta$, there is a $\triangledown \text{del}_R(\bar{x}') \in U_\triangledown$ 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 $\text{del}_R(\bar{y}) \in \Delta$ and $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DC}$ implies $\text{del}_R(\bar{y}) \in \Delta$,
- **feasible**, if (i) $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DR}$ implies $\text{del}_R(\bar{y}) \notin \Delta$, and
  - (ii) $\text{del}_R(\bar{y}) \in \Delta$ and $(R_C(\bar{x}), R_P(\bar{y})) \in \mathcal{DN}$ implies $\text{del}_R(\bar{y}) \notin \Delta$,
- **admissible**, if it is founded, complete, and feasible.

ACM Transactions on Database Systems, Vol. V, No. N. Month 20YY.
(for individual updates, we also write “$\text{del}_R(\bar{x})$ is admissible” instead of “$\{\text{del}_R(\bar{x})\}$ 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 $\text{RESTRICT/NO ACTION}$ rac’s are “obeyed”.

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

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

$$\Delta(U) := \{\text{del}_R(\bar{x}) \mid \text{there is a } \triangledown \text{del}_R(\bar{y}) \in U \text{ s.t. } (R(\bar{x}), R(\bar{y})) \in \text{DC}^*\}$$

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

**Lemma 3.3 Induced Updates.** For given $\text{RA}$, $D$, $U_\triangledown$, and $U \subseteq U_\triangledown$, $\Delta(U)$ is the least set* $\Delta$ which contains the updates given by $U$ and is complete.

**Proof.** $\text{DC}^*$ is the reflexive, transitive closure of $\text{DC}$. 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 $U_\triangledown$.** Let $\text{RA}$, $D$, and $U_\triangledown$ be given. $U \subseteq U_\triangledown$ is admissible if $\Delta(U)$ is admissible, and maximal admissible if there is no other admissible $U'$, s.t. $U \subsetneq U' \subseteq U_\triangledown$. For a set $\Delta$ of user requests, $D' = D \pm \Delta$ denotes the database obtained by applying $\Delta$ 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 $U \subseteq U_\triangledown$, then $\Delta(U)$ is founded and complete.

b) If $\Delta$ is complete and feasible, then $D' := D \pm \Delta(U)$ satisfies all ric’s.

**Proof.** a) $\Delta(U)$ is defined as the least complete set. Since $U \subseteq U_\triangledown$, $\Delta(U)$ is founded.

b) Completeness guarantees that all ric’s labeled with $\text{ON DELETE CASCADE}$ in $\text{RA}$ are satisfied, feasibility guarantees that all ric’s labeled with $\text{ON DELETE RESTRICT/NO ACTION}$ are satisfied. □

**Proposition 3.6 Uniqueness.** For given $\text{RA}$, $D$, and $U_\triangledown$,

(i) if $U_1, U_2 \subseteq U_\triangledown$ are admissible, then $U_1 \cup U_2$ is also admissible,

(ii) thus, there is exactly one maximal admissible $U_{\text{max}} \subseteq U_\triangledown$.

**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 $U_\triangledown$ yields $U_{\text{max}}$. □

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).

---

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

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
3.2 Logic Programming Characterization

We show how a set RA of rac’s can be translated into a logic program PR_A whose rules specify their local behavior. The advantage of this logical formalization is that the declarative semantics of PR_A 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 \( \triangleright \text{del}_{\text{R}}(\bar{x}) \in U_{\triangleright} \), an internal delete request \( \text{req}_{\text{del}_{\text{R}}}(\bar{x}) \), provided there is no blocking \( \text{blk}_{\text{del}_{\text{R}}}(\bar{x}) \).

\[
\text{req}_{\text{del}_{\text{R}}}(\bar{x}) \leftarrow \triangleright \text{del}_{\text{R}}(\bar{x}), R(\bar{x}), \neg \text{blk}_{\text{del}_{\text{R}}}(\bar{x}).
\]

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

\(-R_C. \vec{F} \rightarrow R_P. \vec{K} \text{ on DELETE CASCADE} \) is encoded into two rules: the first one propagates internal delete requests downwards from the parent to the child:

\[
\text{req}_{\text{del}_{R_C}}(\bar{x}) \leftarrow \text{req}_{\text{del}_{R_P}}(\bar{y}), R_C(\bar{x}), \bar{x}[\vec{F}] = \bar{y}[\vec{K}].
\]

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

\[
\text{blk}_{\text{del}_{R_P}}(\bar{y}) \leftarrow R_P(\bar{y}), \text{blk}_{\text{del}_{R_C}}(\bar{x}), \bar{x}[\vec{F}] = \bar{y}[\vec{K}].
\]

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.

\(-R_C. \vec{F} \rightarrow R_P. \vec{K} \text{ on DELETE RESTRICT} \) blocks the deletion of a parent tuple if there is a corresponding child tuple:

\[
\text{blk}_{\text{del}_{R_P}}(\bar{y}) \leftarrow R_P(\bar{y}), R_C(\bar{x}), \text{blk}_{\text{del}_{R_C}}(\bar{x}), \bar{x}[\vec{F}] = \bar{y}[\vec{K}].
\]

\(-R_C. \vec{F} \rightarrow R_P. \vec{K} \text{ on DELETE NO ACTION} \) blocks the deletion of a parent tuple if there is a corresponding child tuple which is not requested for deletion:

\[
\text{blk}_{\text{del}_{R_P}}(\bar{y}) \leftarrow R_P(\bar{y}), R_C(\bar{x}), \text{req}_{\text{del}_{R_C}}(\bar{x}), \bar{x}[\vec{F}] = \bar{y}[\vec{K}].
\]

Note that (i) the local semantics of each individual rac is precisely specified by one or two logic rules, and (ii) \( PR_A \) is in general not stratified due to the negative cyclic dependency \( \text{req}_{\text{del}} \nLeftarrow \text{blk}_{\text{del}} \nLeftarrow \text{req}_{\text{del}} \). Therefore, the global semantics is not necessarily unique. We first consider a “skeptical” global semantics, i.e., the unique well-founded model of the generated logic program. In Section 3.4.2 the more “brave” stable models are considered.

First, we add two rules that define an auxiliary relation \( \text{pot}_{\text{del}} \) that 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 is useful to locate the problems in case that \( U_{\triangleright} \) is not admissible in Section 3.7:

\[
\text{pot}_{\text{del}_{\text{R}}}(\bar{x}) \leftarrow \triangleright \text{del}_{\text{R}}(\bar{x}), R(\bar{x}).
\]

for each \( R_C. \vec{F} \rightarrow R_P. \vec{K} \text{ on DELETE CASCADE} \) (analogous to \( (DC_1) \)):

\[
\text{pot}_{\text{del}_{R_P}}(\bar{y}) \leftarrow \text{pot}_{\text{del}_{R_P}}(\bar{y}), R_C(\bar{x}), \bar{x}[\vec{F}] = \bar{y}[\vec{K}].
\]

(note that the atom \( \text{pot}_{\text{del}_{R_P}}(\bar{y}) \) 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 $U_{D}$, for the minimal model $M := M\{P\}, D, U_{D}\}$ of rule (P) alone,

$$U_{\text{pot}} := \{ \text{del} R(\overline{x}) \mid M(\text{pot,del} R(\overline{x})) = \text{true} \}$$

$$= \{ \text{del} R(\overline{x}) \mid \text{there is a } \text{del} R'(\overline{x}') \in U_{D} \text{ and } (R(\overline{x}), R'(\overline{x}')) \in \mathcal{D}^* \} = \Delta(U_{D})$$

contains exactly the deletions that are obtained when cascading all deletions in $U_{D}$.

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 $U_{D}$, the well-founded model $W := W(P_{RA}, D, U_{D})$ assigns truth-values true and false to all uncontroversial update requests, i.e. which are true or false under any reasonable semantics of $P_{RA}$ [Dix 1995]. $W$ assigns a third truth value undefined to atoms whose truth cannot be determined using a “well-founded” argumentation. The atoms which are undefined in $W$ 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 wrt. the abstract semantics (let $W(\text{atom})$ denote the value of the atom atom in the model $W := W(P_{RA}, D, U_{D})$):

$$-U_{D} := \{ \text{del} R(\overline{x}) \in U_{D} \mid W(\text{req,del} R(\overline{x})) = \text{true} \} \text{ and }$$

$$U_{D,u} := \{ \text{del} R(\overline{x}) \in U_{D} \mid W(\text{req,del} R(\overline{x})) \in \{ \text{true, undef} \} \text{ are admissible, }$$

$$-U_{D,u} = U_{\text{max}}, \text{ and }$$

$$\Delta(U_{\text{max}}) = \Delta(U_{D,u}) = \{ \text{del} R(\overline{x}) \mid W(\text{req,del} R(\overline{x})) \in \{ \text{true, undef} \} \}.$$ 

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

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

**Example 7.** Consider the database depicted in Figure 2 (ignoring $R_{0}$ for now) and the user request $U_{D} := \{ \text{del} R_{1}(a), \text{del} R_{1}(b) \}$. Here, $\text{del} R_{1}(b)$ is not admissible since it is blocked by $R_{5}(b)$. The other request, $\text{del} R_{1}(a)$, can be executed without violating any rule 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 $U_{D,u} := \{ \text{del} R_{1}(a) \}$, the delete requests $\text{req,del} R_{1}(a)$, $R_{2}(a,x)$, $R_{3}(a,y)$, $R_{4}(a,x,y)$, as well as the blocking $\text{blk,del} R_{1}(a)$ and $R_{5}(a,y)$ will be undefined in the well-founded model.

For the user request $U_{D,u} := \{ \text{del} R_{1}(b) \}$, $\text{blk,del} R_{1}(b)$ is true for $R_{1}(b)$ due to the referencing tuple $R_{5}(b)$. Thus, $\text{req,del} R_{1}(b)$ is false, and $\text{del} R_{1}(b)$ 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, $\text{blk,del} R_{5}(b,y)$ is also true.

Note that the extended set $U_{D}^{e} = \{ \text{del} R_{1}(a), \text{del} R_{1}(b), \text{del} R_{5}(b) \}$ is a candidate for a refined request which accomplishes the deletion of $R_{1}(a)$ and $R_{1}(b)$.  

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
Fig. 2. Extended database with modified rac's

$\mathcal{W}$ contains some ambiguities that can be interpreted constructively as degrees of freedom; The blockings and deletions induced by $U_\rho = \{\triangleright \text{del}_R (a)\}$ in Example 7 are undefined due to the dependency $\text{req} \sim \text{del} \sim \text{blk} \sim \text{del}$. 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 that 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, $\text{I}$ (the "Deleter") and $\text{II}$ (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 $D \cup U_\rho \cup \{\text{restricted}\}$. The possible moves of $\text{I}$ and $\text{II}$ are defined below. Note that $\text{I}$ moves from $D$ to $U_\rho$, while $\text{II}$ moves from $U_\rho$ to $D \cup \{\text{restricted}\}$. Initially, the pebble is placed on some tuple in $D$ (or $U_\rho$) and $\text{I}$ (or $\text{II}$) starts to move. If $\text{I}$ begins, the first round only consists of the move by $\text{II}$.

The possible moves are illustrated in Figure 3: By moving the pebble from $R(\bar{x}) \in D$ to some $\triangleright \text{del}_R(\bar{x}') \in U_\rho$ which cascades down to $R(\bar{x})$, $\text{I}$ claims that the deletion of $R(\bar{x})$ is "justified" (i.e., founded) by $\triangleright \text{del}_R(\bar{x}')$. Conversely, $\text{II}$ claims by her moves that $\text{del}_R(\bar{x}')$ is not feasible. $\text{II}$ can use two different arguments: Assume that the deletion of $R(\bar{x}')$ cascades down to some tuple $R_\rho(\bar{x}_p)$. First, if the deletion of $R_\rho(\bar{x}_p)$ is restricted by a referencing child tuple $R_C(\bar{x}_C)$, then $\text{II}$ may force $\text{I}$ into a lost position by moving to restricted (since $\text{I}$ cannot move from there). Second, assume that the deletion of $R(\bar{x}')$ cascades down to some other tuple $R_\rho(\bar{x}_p')$. Then, $\text{II}$ can move to a child tuple $R_C(\bar{x}_C')$ which references $R_\rho(\bar{x}_p')$.
with a NO ACTION rac. With this move, \( \Pi \) claims that this reference to \( R'_p(x'_p) \) will remain in the database, so \( R'_p(x'_p) \) and, as a consequence, \( R'(x') \) cannot be deleted. In this case, I may start a new round of the game by finding a justification to delete the referencing child \( R'_C(x'_C) \). More precisely:

Player I can move from \( R(\bar{x}) \) to \( \triangleright \text{del}_R R'(\bar{x}') \) in \( U_\triangleright \) if \( (R(\bar{x}), R'(\bar{x}')) \in DC^* \).

Player \( \Pi \) can move from \( \triangleright \text{del}_R R'(\bar{x}') \)
- to restricted if there are \( R_p(x'_p) \) and \( R_C(x'_C) \) s.t. \( (R_p(x'_p), R(\bar{x})) \in DC^* \) and \( (R_C(x'_C), R_p(x'_p)) \in DR \).
- to \( R'_p(x'_p) \), if \( (R'_p(x'_p), R'(x')) \in DC^* \) and \( (R'_C(x'_C), R'_p(x'_p)) \in DN \).

**Lemma 3.10 Claims of I and \( \Pi \).**

1. If I can move from \( R(\bar{x}) \) to \( \triangleright \text{del}_R R'(\bar{x}') \), then deletion of \( R'(\bar{x}') \) is found by \( U_\triangleright \) and induces the deletion of \( R(\bar{x}) \).
2. If \( \Pi \) can move from \( \triangleright \text{del}_R R(\bar{x}) \) to restricted, then deletion of \( R(\bar{x}) \) is not feasible due to the existence of a referencing tuple.
3. If \( \Pi \) can move from \( \triangleright \text{del}_R R(\bar{x}) \) to \( \triangleright \text{del}_R R'(\bar{x}') \), then deletion of \( R(\bar{x}) \) is admissible only if \( R'(\bar{x}') \) is also deleted.

**Proof.** (1) The move of I implies that \( (R(\bar{x}), R'(\bar{x}')) \in DC^* \).

The move of \( \Pi \) means that either

2. there are \( R_p(\bar{x}_p), R_C(\bar{x}_C) \) s.t. \( (R_p(\bar{x}_p), R(\bar{x})) \in DC^* \) and \( (R_C(\bar{x}_C), R_p(\bar{x}_p)) \in DR \). 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. \( (R'_C(\bar{x}'_C), R(\bar{x})) \in DN \cap DC^* \), i.e., there is a \( R'_p(\bar{x}'_p) \) such that \( (R'_p(\bar{x}'_p), R(\bar{x})) \in DC^* \) and \( (R'_C(\bar{x}'_C), R'_p(\bar{x}'_p)) \in DN \). Hence, by (1), deletion of \( R(\bar{x}) \) induces deletion of \( R'_p(\bar{x}'_p) \), which is only allowed if \( R'_C(x'_C) \) is also deleted.\(^5\) \( \square \)

**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 \( \triangleright \text{del}_R R'(\bar{x}') \) if, given the fact \( \text{req}\_\text{del}_R R'(\bar{x}') \), \( \text{req}\_\text{del}_R R(\bar{x}) \) can be derived using \((DC_1)\).
- The moves by \( \Pi \) are reflected by the rules \((DC_2)\) and \((DR)\)\/{\( (DN) \)}:
  - \( \Pi \) can move from \( \triangleright \text{del}_R R(\bar{x}) \) to restricted if \( \text{blk}\_\text{del}_R R(\bar{x}) \) is derivable using \((DR)\) and \((DC_2)\) only, or
  - \( \Pi \) can move from \( \triangleright \text{del}_R R(\bar{x}) \) to \( R'_C(x'_C) \) if \( \text{blk}\_\text{del}_R R(\bar{x}) \) is derivable using \((DC_2)\) and an instance of \((DN)\) if \( \text{req}\_\text{del}_R R'_C(x'_C) \) is assumed to be false.
- The negative dependencies in (I), \( \text{req}\_\text{del} \sim \neg \text{blk}\_\text{del} \), and \((DN)\), \( \text{blk}\_\text{del} \sim \neg \text{req}\_\text{del} \), mirror the alternation of moves between I and \( \Pi \), respectively.

\(^5\)\( DN \cap DC^* := \{(x,y) \mid \exists z : (x,z) \in DN \text{ and } (z,y) \in DC^* \} \).

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
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 $\Pi$ 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 $\triangleright \text{del}_\Pi R(\bar{x}) \in U_p$ 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 $\Pi$ prove that it is infeasible to delete $R(\bar{x})$.

Example 8. Consider again Figure 2 with $U_p = \{\triangleright \text{del}_\Pi R_1(a), \triangleright \text{del}_\Pi R_1(b)\}$. 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 $\triangleright \text{del}_\Pi R_1(a)$, while $\Pi$ can move from $\triangleright \text{del}_\Pi R_1(a)$ to $R_4(a, x, y)$. Thus, after $I$ has started the game moving to $\triangleright \text{del}_\Pi R_1(a)$, $\Pi$ will answer with the move to $R_4(a, x, y)$, so $I$ moves back to $\triangleright \text{del}_\Pi R_1(a)$ again, etc. Hence the game is drawn for each of the “a”-tuples.

In contrast, for the “b”-tuples, there is an additional move from $\triangleright \text{del}_\Pi R_1(b)$ to $R_5(b)$ for $\Pi$, 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 $\iff$ it is with the given set of user delete requests to delete $R(\bar{x})$ without violating a ric.
- $R(\bar{x})$ is won or drawn $\iff$ simultaneous execution of all user delete requests $\triangleright \text{del}_\Pi R'(\bar{x}')$ which 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 $(R_C(\bar{x}_C), R(\bar{x})) \in DR$ (otherwise, if $I$ moves from $R(\bar{x})$ to some $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$, $\Pi$ moves to restricted since $(R_C(\bar{x}_C), R_\Pi(\bar{x}_d)) \in DR \circ DC^*$ and wins). Thus, no ric of the form ON DELETE RESTRICT is violated when deleting a won or drawn tuple.

$\Rightarrow$: A tuple $R(\bar{x})$ is lost in $n$ rounds if either
- $(n = 0)$ there is no user request $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$ s.t. $(R(\bar{x}), R_\Pi(\bar{x}_d)) \in DC^*$, i.e., the deletion of $R(\bar{x})$ is unfounded, or
- $(n > 0)$ for every user request $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$ s.t. $(R(\bar{x}), R_\Pi(\bar{x}_d)) \in DC^*$, $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$ is lost in $\leq n - 1$ rounds, i.e., either $\Pi$ can move from $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$ to restricted (in this case, by Lemma 3.10(2), $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$ is not feasible), or there is some tuple $R'(\bar{x}')$ s.t. $\Pi$ can move from $\triangleright \text{del}_\Pi R_\Pi(\bar{x}_d)$ 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.

$\Leftarrow$: If $R(\bar{x})$ cannot be deleted without violating a ric, then either,
- deletion of $R(\bar{x})$ is unfounded – then it is lost immediately since $I$ cannot move,
- or deletion of $R(\bar{x})$ is founded, but none of its founding user delete requests $\triangleright \text{del}_\Pi R'(\bar{x}')$ is executable. This can be either due to a $DC^* \circ DR$ chain to a tuple $R_C'(\bar{x}_C')$ – then $\triangleright \text{del}_\Pi R'(\bar{x}')$ is lost in one round since $\Pi$ moves to restricted – or due to a $DC^* \circ DN$ chain to a tuple $R_C'(\bar{x}_C')$ that must be deleted, but cannot. Then, $\Pi$ can move there and will win (the detailed proof would argue with induction over the length of the proof why $\triangleright \text{del}_\Pi R'(\bar{x}')$ is not executable, analogous to the proof of $\Rightarrow$).
The correspondence between the game semantics and the abstract semantics yields the following:

**Corollary 3.14 Correctness.**
The game-theoretic characterization is correct wrt. the abstract semantics:

- \( U_w := \{ u \in U_\triangledown \mid u \text{ is won} \} \) and \( U_{w,d} := \{ u \in U_\triangledown \mid u \text{ is won or drawn} \} \) are admissible,
- \( U_{w,d} = U_{\text{max}} \),
- \( \Delta(U_w) = \{ \text{del}_R(\bar{x}) \mid R(\bar{x}) \text{ is won} \} \) and \( \Delta(U_{\text{max}}) = \Delta(U_{w,d}) = \{ \text{del}_R(\bar{x}) \mid R(\bar{x}) \text{ is won or drawn} \} \).

### 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 \( U_\triangledown \) of user requests. We introduce the AFP by using Statelog, a state-oriented extension of Datalog which allows to integrate active and deductive rules [Ludächer et al. 1996; Ludächer 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

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

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 \( P_{\text{AFP}} \) computes the alternating fixpoint of \( P \):

\[
[S+1] \ \text{req}_{\text{del}}_R(\bar{x}) \leftarrow \triangledown \text{del}_R(\bar{x}), \ R(\bar{x}), \ [S] \neg \text{blk}_{\text{del}}_R(\bar{x}). \quad (I^A)
\]

% \( R_C \bar{F} \rightarrow R_P \bar{K} \text{ ON DELETE CASCADE:} \)

\[
[S+1] \ \text{req}_{\text{del}}_R C(\bar{x}) \leftarrow R_C(\bar{x}), \ \bar{x}[\bar{F}] = \bar{Y}[\bar{K}], \ [S+1] \ \text{req}_{\text{del}}_R P(\bar{Y}), \quad (DC^1_1)
\]

\[
[S+1] \ \text{blk}_{\text{del}}_R P(\bar{Y}) \leftarrow R_P(\bar{Y}), \ \bar{x}[\bar{F}] = \bar{Y}[\bar{K}], \ [S+1] \ \text{blk}_{\text{del}}_R C(\bar{x}). \quad (DC^1_2)
\]

% \( R_C \bar{F} \rightarrow R_P \bar{K} \text{ ON DELETE RESTRICT:} \)

\[
[S+1] \ \text{blk}_{\text{del}}_R P(\bar{Y}) \leftarrow R_P(\bar{Y}), \ R_C(\bar{x}), \ \bar{x}[\bar{F}] = \bar{Y}[\bar{K}]. \quad (DR^1)
\]

% \( R_C \bar{F} \rightarrow R_P \bar{K} \text{ ON DELETE NO ACTION:} \)

\[
[S+1] \ \text{blk}_{\text{del}}_R P(\bar{Y}) \leftarrow R_P(\bar{Y}), \ R_C(\bar{x}), \ \bar{x}[\bar{F}] = \bar{Y}[\bar{K}], \ [S] \neg \text{req}_{\text{del}}_R C(\bar{x}). \quad (DN^1)
\]

\( P_{\text{AFP}} \) is locally stratified, thus there is a unique perfect model [Przymusinski 1988] \( M_{\text{AFP}} \) of \( P_{\text{AFP}} \cup D_\triangledown U_\triangledown \). \( M_{\text{AFP}} \) mimics the alternating fixpoint computation of \( \mathcal{W} \).

\[ 
\text{ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.}
\]
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 \( \eta_f \) of the computation is reached if \( M[2n_f] = M[2n_f + 2] \). Then, the truth value of atoms \( A \) in \( W \) can be determined from \( M_{AFP} \) as follows:

\[
W(A) = \begin{cases} 
  \text{true} & \text{if } M_{AFP} = [2n_f] A, \\
  \text{undefined} & \text{if } M_{AFP} = [2n_f] \neg A \land [2n_f + 1] A, \\
  \text{false} & \text{if } M_{AFP} = [2n_f + 1] \neg A.
\end{cases}
\]

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

- \( R(\bar{x}) \) is won \( \iff \ W(\text{req.del}_R(\bar{x})) = \text{true} \).
- \( R(\bar{x}) \) is lost \( \iff \ W(\text{req.del}_R(\bar{x})) = \text{false} \).
- \( R(\bar{x}) \) is drawn \( \iff \ W(\text{req.del}_R(\bar{x})) = \text{undefined} \).

**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 \( \iff \ M_{AFP} \models [2n] \text{req.del}_R(\bar{x}) \).
- \( R(\bar{x}) \) is lost within \( \leq n \) rounds \( \iff \ M_{AFP} \models [2n + 1] \neg \text{req.del}_R(\bar{x}) \).

From this, Theorem 3.15 is immediate: The \( n^{th} \) overestimate excludes deletions provably non-admissible in \( n \) rounds, whereas the \( n^{th} \) underestimate contains all deletions which can be proven in \( n \) rounds. Thus, there is an \( n \) such that \( M_{AFP} \models [2n] \text{req.del}_R(\bar{x}) \) iff \( W_R^A(\text{req.del}_R(\bar{x})) = \text{true} \), and there is an \( n \) such that \( M_{AFP} \models [2n+1] \neg \text{req.del}_R(\bar{x}) \) iff \( W(\text{req.del}_R(\bar{x})) = \text{false} \).

A position \( R(\bar{x}) \) is drawn if for every user request \( \triangleright \text{del}_R'(\bar{x}') \) which I uses for deleting it, II can find a witness against \( \triangleright \text{del}_R(\bar{x}') \), 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 \( \text{NO ACTION} \) links that introduce cycles into the argumentation). \( \square \)

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 \( U_{\text{max}} \subseteq U_{\bowtie} \) (by Theorem 3.8, \( U_{\text{max}} = U_{\text{tr}} \)) 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 \( P \) wrt. \( I \), i.e., the program obtained by replacing every negative literal of \( P \) by its truth-value wrt. \( I \). An interpretation \( I \) is a **stable model** if \( M_{P/I} = I \).
Every stable model \( S \) extends the well-founded model \( \mathcal{W} \) wrt. true and false atoms; \( S_{\text{true}} \supseteq \mathcal{W}_{\text{true}}, S_{\text{false}} \supseteq \mathcal{W}_{\text{false}} \). However, not every program has a two-valued stable model (e.g., the “self attack” in Example 5).

**Theorem 3.18.** Let \( S_{RA} \) be defined by

\[
S_{RA} := \text{defn} \cup \text{req.del.R}(\vec{x}) \cup \{ \text{req.del.R}(\vec{x}) \mid \mathcal{W}(\text{req.del.R}(\vec{x})) \in \{ \text{true}, \text{undef} \} \\
\cup \{ \text{blk.del.R}(\vec{x}) \mid \mathcal{W}(\text{blk.del.R}(\vec{x})) = \text{true} \}.
\]

Then, \( S_{RA} \) is a total stable model of \( P_{RA} \cup D \cup U_{D} \).

\( S_{RA} \) is the “maximal” stable model in the sense that it contains all delete requests which are true in some stable model. Consequently, deletions have priority over blockings (cf. Example 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 \( R_{4}(1,3) \rightarrow R_{3}(1,2) \) ON DELETE RESTRICT to be replaced by \( R_{4}(1,3) \rightarrow R_{2}(1,2) \) ON DELETE NO ACTION. Then the rules of \( P_{RA} \) derive that the deletion of \( R_{1}(a) \) is blocked (via \( R_{4} \sim R_{3} \sim R_{1} \)) if \( R_{4}(a,b,c) \) cannot be deleted. \( R_{4}(a,b,c) \) can be deleted (via \( R_{1} \sim R_{2} \sim R_{4} \)) if the deletion of \( R_{1}(a) \) is not blocked. Hence there is a negative cycle of the form \{block \( \rightarrow \neg \text{exec}, \neg \text{exec} \rightarrow \neg \text{block}\}. Setting all requests in the diamond to true (as done in \( S_{RA} \)) or all to false results each in a stable model.

**Theorem 3.19 Correctness.**

Let \( S \) be a stable model of \( P_{RA} \cup D \cup U_{D} \). Then \( U_{S} := \{ \text{defn.del.R}(\vec{x}) \in U_{D} \mid S \models \text{req.del.R}(\vec{x}) \} \) is admissible and \( \Delta(U_{S}) = \Delta_{S} := \{ \text{defn.del.R}(\vec{x}) \mid S \models \text{req.del.R}(\vec{x}) \} \).

\(-U_{\text{max}} = U_{S_{RA}} \) and \( \Delta(U_{\text{max}}) = \Delta_{S_{RA}} \).

**Proof.** Foundedness: follows directly from the fact that \( S \) is stable (an undefined \( \text{req.del.R}(\vec{x}) \) would not be stable).

Completeness: For every \( \text{ric} R_{C} \cdot \vec{F} \rightarrow R_{P} \cdot \vec{K} \) ON DELETE CASCADE, if \( S \models R_{C}(\vec{x}) \land \text{req.del.R}(\vec{y}) \land \vec{z} \cdot [\vec{F}] = \vec{g} \cdot [\vec{K}] \), then, due to \( (DC_{1}) \), \( S = M_{P/S} \models \text{req.del.R}(\vec{z}) \).

Feasibility: Suppose a \( \text{ric} R_{C} \cdot \vec{F} \rightarrow R_{P} \cdot \vec{K} \) ON DELETE RESTRICT or \( \text{ric} \cdot \vec{F} \rightarrow R_{P} \cdot \vec{K} \) ON DELETE NO ACTION would be violated: Then \( S \models \text{req.del.R}(\vec{y}) \land R_{C}(\vec{x}) \land \vec{z} \cdot [\vec{F}] = \vec{g} \cdot [\vec{K}] \) (for NO ACTION also \( S \models \neg \text{req.del.R}(\vec{z}) \)), and thus because of \( (DR) \) or \( (DN) \), respectively, \( S = M_{P/S} \models \text{blk.del.R}(\vec{y}) \).

Thus, by \( (DC_{2}) \), for the founding request \( \text{del.R}(\vec{z}), S \models \text{blk.del.R}(\vec{z}) \), and by \( (I) \), \( S \models \neg \text{req.del.R}(\vec{z}) \) which is a contradiction to the assumption that \( \text{del.R}(\vec{z}) \) is the founding delete request. \( \Delta_{S} \subseteq \Delta(U_{S}) \) follows from foundedness, and \( \Delta_{S} \leq \Delta(U_{S}) \) follows from completeness.

**3.5 A Procedural Translation**

The declarative semantics of the well-founded model is translated into a more “algorithmic” implementation in Statatalog by “cutting” the cyclic dependency at one of the possible points, i.e., 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 \( S_{RA} \) (giving priority to deletions over blockings), corresponding to the observation that \( S_{RA} \) takes exactly the blockings from the underestimate and the internal delete requests from the overestimate.
The rules \((DC_1), (DC_2),\) and \((DR)\) are already local rules:

\[
\begin{align*}
[S] \text{ req\_del\_RC}(\bar{X}) &\leftarrow R_C(\bar{X}), \quad \bar{X}[\bar{F}] = \bar{Y}[\bar{K}], \quad [S] \text{ req\_del\_RP}(\bar{Y}). \quad (DC^S) \\
[S] \text{ blk\_del\_RP}(\bar{Y}) &\leftarrow R_P(\bar{Y}), \quad \bar{X}[\bar{F}] = \bar{Y}[\bar{K}], \quad [S] \text{ blk\_del\_RC}(\bar{X}). \quad (DC^S_2) \\
[S] \text{ blk\_del\_RP}(\bar{Y}) &\leftarrow R_P(\bar{Y}), \quad R_C(\bar{X}), \quad \bar{X}[\bar{F}] = \bar{Y}[\bar{K}]. \quad (DR^S)
\end{align*}
\]

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

\[
[S] \text{ req\_del\_R}(\bar{X}) \leftarrow \text{ del\_R}(\bar{X}), \quad R(\bar{X}), \quad [S] \neg \text{ blk\_del\_R}(\bar{X}). \quad (I^S)
\]

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

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

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 \(M_S\) of \(P_S \cup D \cup U_b\). The state-stratification \(\{\text{blk\_del\_R}\} \prec \{\text{req\_del\_R}\}\), mirrors the stages of the algorithm: First, only blockings resulting from \texttt{ON DELETE RESTRICT} \(\texttt{racs}\) are considered (local rules \((DR^S)\) and \((DC^S_2)\)). Based on these, the first maximal overestimate of internal delete requests is computed by \((I^S)\) and \((DC^S)\). Then, in the step to the subsequent state, the blockings resulting from \texttt{ON DELETE NO ACTION} \(\texttt{racs}\) whose child nodes are not deleted (by \((DN^S)\), the only progressive rule; it does not derive anything for the initial state) are derived. \((DR^S)\) contributes the blockings resulting from \texttt{ON DELETE RESTRICT} \(\texttt{racs}\). Again, the induced blockings are derived by \((DC^S_2)\). The second stratum, consisting of \((I^S)\) and \((DC^S)\) 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 \(S_{RA}\).

**Lemma 3.20.** \(M_{AFP}\) corresponds to \(M_S\) as follows:

1. \(M_{AFP} \models [2n] \text{ blk\_del\_R}(\bar{X}) \iff M_S \models [n] \text{ blk\_del\_R}(\bar{X})\).
2. \(M_{AFP} \models [2n+1] \text{ req\_del\_R}(\bar{X}) \iff M_S \models [n] \text{ req\_del\_R}(\bar{X})\).

**Proof.** \(P_S\) and \(P_{AFP}\) differ in the rules \((I^S)\) and \((IA)\): In every iteration, \(P_S\) takes the blockings from the underestimate and the delete requests from the overestimates, resulting in \(S_{RA}\). 

**Theorem 3.21** **Termination.** For every database \(D\) and every set \(U_b\) of user delete requests, the program reaches a fixpoint, i.e., there is a least \(n_f \leq |U_b|\), s.t. \(M_S[n_f] = M_S[n_f + 1]\).

**Proof.** A fixpoint is reached if the set of blocked user delete requests becomes stationary. Since this set is non-decreasing, there are at most \(|U_b|\) iterations.

From Lemma 3.20 and Theorem 3.18, the correctness of \(P_S\) follows:

**Theorem 3.22** **Correctness.**

The final state of \(M_S, M_S[n_f]\), represents \(U_{\text{max}}\) and \(\Delta(U_{\text{max}})\):

- \(-M_S[n_f] = S_{RA}\),
- \(-U_{\text{max}} = \{\text{del\_R}(\bar{X}) \in U_b \mid M_S[n_f] \models \text{req\_del\_R}(\bar{X})\},\) and
- \(-\Delta(U_{\text{max}}) = \{\text{del\_R}(\bar{X}) \mid M_S[n_f] \models \text{req\_del\_R}(\bar{X})\}\).

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
3.5.1 Implementation in a Procedural Programming Language. The StateLog formalization $P_S$ above is translated into the algorithm given in Fig. 4. Initially, it is assumed that there are only those blockings which result directly from ON DELETE RESTRICT racs. Then, blockings are propagated upwards the ON DELETE CASCADE chains, finally blocking the triggering user requests. For the remaining unblocked user requests, the cascaded requests are recomputed. Thus, some more tuples will remain in the database, which can block other requests. In the next step, all blockings are computed which are caused by ON DELETE NO ACTION racs from tuples which are not reachable via cascaded deletions. These steps are repeated until a fixpoint is reached. Observe that each iteration corresponds to the evaluation of a query with PTIME data complexity. Since the fixpoint is reached after at most $|U_B|$ iterations (Theorem 3.21), the overall algorithm also has polynomial data complexity.

![Algorithm Diagram](https://example.com/algorithm-diagram.png)

**Input:** A consistent database $D$ and a set $U_B$ of user delete requests. $B := \{\text{all blockings which result from ON DELETE RESTRICT racs}\}$.

1. (Re)Compute the set of induced blockings $B^*$, which result from $B$ by propagating blockings upwards the ON DELETE CASCADE chain.

2. (Re)Compute the set $U^*$ of internal requests which result from cascading user delete requests $U_B$ which are not blocked: $U^* := (U_B \backslash B^*)$.

3. Add to $B$ all blockings which are issued by ON DELETE NO ACTION racs from tuples not in $U^*$, i.e., which are not requested for deletion.

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

![Algorithm Diagram](https://example.com/algorithm-diagram.png)

Fig. 4. Algorithm for executing all admissible deletions.

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

$-U_{\max} = \{\text{del}_R(\bar{x}) \in U_B \mid \text{req}_R(\bar{x}) \in U^*\}$, and $\Delta(U_{\max}) = U^*$.

**Proof.** In the $n^{th}$ iteration, $B^* = \{\text{blk}_R(\bar{x}) \mid M_S \models [n] \text{blk}_R(\bar{x})\}$, and $U^* = \{\text{del}_R(\bar{x}) \mid M_S \models [n] \text{req}_R(\bar{x})\}$. $\square$
For given $D$, $U_P$, and $RA$, the algorithm in Figure 4 computes the maximal subset $U_{\text{max}}$ of $U_P$, which can be executed without violating any ric, and the set $\Delta(U_{\text{max}})$ of "internal deletions" which are induced by it. For cases when $U_P$ 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., $R_C \cdot \mathcal{F}(\bar{x})$ and $R_P \cdot \mathcal{K}(\bar{y})$) are selected wrt. the original database state,
— the tuples to be deleted are fixed wrt. the original state, RESTRICT is also evaluated wrt. the original state,
— NO ACTION is evaluated against the (prospective) result.
— an update referential action is executed whenever the parent is updated.
— effectively, $\Delta$ 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 wrt. the intended semantics – except that the specification of the individual rules is correct – is redundant.

Moreover, in case 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 rac'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 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

In case that $\triangleright \text{del}_d R_d(\bar{x}_d) \in U_P$ 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, i.e., where the database schema, containing the ric's and rac's is correct wrt. the semantics of the application domain, and where the program itself that computes $U_P$ 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 ▷ del Ra_d(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((#) to R_C((#) is ON DELETE NO ACTION, it is implicitly assumed that R_C((#) is deleted by some other delete request in U_D, possibly via another cascading chain. Thus, there can be two reasons:

1. the program is wrong: it should derive ▷ del Ra((#) or it should derive a ▷ del R'(x') such that deleting R'(x') cascades to deletion of R_C((#) or
2. another ric or rac is wrong: The program derives such an ▷ del R'(x'), but it does not cascade to R_C((#) .

—the original database state is wrong: R_C((#) should not be there. Then, another transaction before did not execute in the intended way.

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

—where the above situation occurs, and
—why 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 ▷ del Ra_d((#) ∈ U_D, there is a chain of blocked deletions downwards a series of cascading references. The “lowest” blocked deletion concerns the above tuple R_P((#) , and the blocking is caused by a child R_C((#) via a ric that is associated to the above rac:

Definition 3.24. Given a database D and a set U_D, and ¥ := ¥(P_R_A, D, U_D) as above, a quadruple (▷ del Ra_d((#) , R_P((#) , rac , R_C((#) is a problem situation if

—there is an ▷ del Ra_d((#) ∈ U_D, such that (R_P((#) , R_d((#) ) ∈ DC* and ¥(blk del Ra_d((#) ) = true and for all R'(x') on this chain, ¥(blk del R'(x')) = true and

(i) (R_C((#) , R_P((#) ) ∈ DR (then, rac = RESTRICT) or
(ii) (R_C((#) , R_P((#) ) ∈ DN (then, rac = NO ACTION) and ¥(req del R_C((#) ) = false (i.e., R_C((#) is not deleted), and ¥(pot del R_C((#) ) = false.

Note that in case (ii), if ¥(pot del R_C((#) ) = true, this situation is not a problem since R_C((#) would be deleted by U_D, but the update ▷ del R'(x') ∈ U_D 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_D = {▷ del R_1(a), ▷ del R_1(b)} has been rejected. In a real database environment, the rejection can be augmented by a problem description: Here, (▷ del R_1(b), R_1(b), NO ACTION, R_S(b)) is the problem situation since the deletion of R_1(b) must be rejected due to the existence of the tuple R_S(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 \( U'_D \) (delete \( R_5(b) \) too, explicitly). Thus add a program line DELETE \ldots 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 Fig. 2 with \( U'''_D = \{ \uparrow \text{del}_R (a), \uparrow \text{del}_R (b) \} \). It induces the update \( U_D \) above and is again rejected. A possible conclusion from the detailed description 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 \( U'_D \).

Detecting Problems During Computation. The above problem analysis is based on the outcome of computing the well-founded model. However, the problem can easily be located already during the computation of the well-founded model: the first blocking \( \text{blk}_{\text{del}_R} (g) \) 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 \( R_C (x) \) and the reference. Then, the above scenario of “indirect” blockings can be avoided: the detected problem is a problem.

In case that 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 \( \mathcal{F} \), either, to restricted, or via \( DC \circ DN \) to a tuple whose deletion is not founded (in case that its deletion is founded, but also “lost”, the above indirect situation applies, and the game goes on, leading to another final move of \( \mathcal{F} \)).

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 \( U_D \) for different arguments (e.g., checking what happens for \( \uparrow \text{del}_R (c) \) in the above example) to check if there is a general problem, or only for a special value (output: a set of problematic values).
— try extensions of \( U_D \), 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. Especially, in the presence of overlapping foreign keys, interferences between cascaded modifications can occur. 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 $\text{ins}_R(\bar{X})$, $\text{del}_R(\bar{X})$, and $\text{mod}_R(M, \bar{X})$. $M$ 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 $\text{mod}_R([1/d, 3/e], (a, b, c))$, we may write $\text{mod}_R(a/d, b, c/e)$. The restriction of a modification $M$ to a key $\bar{A}$ is denoted by $M[\bar{A}]$; the result of applying a modification $M$ to $\bar{X}$ is denoted by $M(\bar{X})$. E.g., 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, e.g., $\{1[a, 2/b, 3/d] \cup [1/e, 2/b, 4/f] = [1/a, 1/b, 2/b, 3/d, 4/f]$. A modification is consistent if it 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 which a set of updates $\Delta$ may have wrt. $D$, $U_\triangleright$, 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 $\Delta$ 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** wrt. $U_\triangleright$, $\Delta$, $D$, and $RA$ if it can be justified by the user requests and propagations:

- A delete instruction $\text{del}_R(\bar{x})$ is **founded in $n$ steps** wrt. $U_\triangleright$, $\Delta$, $D$, and $RA$ if either $n = 0$ and $\triangleright \text{del}_R(\bar{x}) \in U_\triangleright$, or there is a delete instruction $\text{del}_R(\bar{x}_i) \in \Delta$ which is founded in $< n$ steps, and a rac $\text{R.F}_i \rightarrow \text{R.K}_i$ ON DELETE OF PARENT CASCADE s.t. $\bar{x}[\text{F}_i] = \bar{x}[\text{K}_i]$.

- A modify instruction $\text{mod}_R(M, \bar{x})$ is **founded in $n$ steps** wrt. $U_\triangleright$, $\Delta$, $D$, 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 $\triangleright \text{mod}_R(M_i, \bar{x}) \in U_\triangleright$ or there is a modify instruction $\text{mod}_R(M_i, \bar{x}) \in \Delta$ which is founded in $< n$ steps, and a rac $\text{R.F}_i \rightarrow \text{R.K}_i$ ON UPDATE OF PARENT CASCADE s.t. $\bar{x}[\text{K}_i] \neq M'_i(\bar{x}_i)[\text{K}_i]$ and $\bar{x}[\text{F}_i] = \bar{x}[\text{K}_i]$ and $M_i = F_i/M'_i(\bar{x}_i)[\text{K}_i]$.

- An insert instruction $\text{ins}_R(\bar{x})$ is **founded** wrt. $U_\triangleright$, if $\triangleright \text{ins}_R(\bar{x}) \in U_\triangleright$.

For given $D$, $U_\triangleright$, and $RA$, a set $\Delta$ of updates is called
—*founded wrt. U_D*, D, and RA if every update instruction \( \text{del}_R(\bar{x}) \) or \( \text{mod}_R(M, \bar{x}) \in \Delta \) is founded.

—*complete wrt. D and RA* if it is closed wrt. propagations, i.e., satisfies the following conditions:

1. if \( \text{del}_R(M, \bar{y}) \in \Delta \), \( R_C(\bar{x}) \in D \), \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON DELETE OF PARENT CASCADE** in RA, and \( \bar{x}[\varphi] = \bar{y}[\bar{K}] \) then \( \text{del}_R(\bar{C}) \in \Delta \).

2. if \( \text{mod}_R(M, \bar{y}) \in \Delta \), \( R_C(\bar{x}) \in D \), \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON UPDATE OF PARENT CASCADE** in RA, \( \bar{y}[\bar{K}] \neq M(\bar{y})[\bar{K}] \) and \( \bar{x}[\varphi] = \bar{y}[\bar{K}] \) then there is an \( M' \) s.t. \( \text{mod}_R(M', \bar{x}) \in \Delta \) and \( M' \supseteq \varphi / M(\bar{y})[\bar{K}] \).

—*feasible wrt. D and RA* if all rac’s specified as no action and restrict are satisfied:

1. a) \( \Delta \) contains no delete instruction \( \text{del}_R(\bar{y}) \) s.t. there is a rac \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON DELETE OF PARENT RESTRICT** in RA and a tuple \( R_C(\bar{x}) \in D \) with \( \bar{x}[\varphi] = \bar{y}[\bar{K}] \).

b) \( \Delta \) contains no modify instruction \( \text{mod}_R(M, \bar{y}) \) such that there is a rac \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON UPDATE OF PARENT RESTRICT** in RA, \( \bar{y}[\bar{K}] \neq M(\bar{y})[\bar{K}] \) and a tuple \( R_C(\bar{x}) \in D \) with \( \bar{x}[\varphi] = \bar{y}[\bar{K}] \).

2. for every rac \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON DELETE OF PARENT NO ACTION** or \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON UPDATE OF PARENT NO ACTION** in RA, every delete instruction \( \text{del}_R(M, \bar{y}) \) or modify instruction \( \text{mod}_R(M, \bar{y}) \) s.t. \( \bar{y}[\bar{K}] \neq M(\bar{y})[\bar{K}] \) which is contained in \( \Delta \), and every tuple \( R_C(\bar{x}) \in D \) s.t. \( \bar{x}[\varphi] = \bar{y}[\bar{K}] \), \( \Delta \) contains either a delete instruction \( \text{del}_R(\bar{C}) \) or a modify instruction \( \text{mod}_R(M, \bar{y}) \) s.t. \( M(\bar{y})[\bar{K}] \neq \bar{y}[\bar{K}] \).

3. for every rac \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON CHILD RESTRICT** in RA and every tuple \( R_C(\bar{x}) \in D \) which either results from an insertion \( \text{ins}_R(\bar{C}) \in \Delta \) or a modification \( \text{mod}_R(M, \bar{x'}) \in \Delta \) (i.e., \( \bar{x} = M(\bar{x'}) \)) s.t. \( \bar{x}[\varphi] \neq M(\bar{x'})[\bar{K}] \), \( D \) contains a tuple \( R_P(\bar{y}) \) s.t. \( \bar{y}[\bar{K}] = \bar{x}[\varphi] \) and \( \Delta \) contains neither \( \text{del}_R(\bar{y}) \) nor any \( \text{mod}_R(M, \bar{y}) \) s.t. \( M(\bar{y})[\bar{K}] \neq \bar{x}[\varphi] \).

4. for every rac \( R_C, \varphi \rightarrow R_P, \bar{K} \) **ON CHILD NO ACTION** in RA and every tuple \( R_C(\bar{x}) \in D \) which either results from an insertion \( \text{ins}_R(\bar{C}) \in \Delta \) or a modification \( \text{mod}_R(M, \bar{x'}) \in \Delta \) (i.e., \( \bar{x} = M(\bar{x'}) \)) s.t. \( \bar{x}[\varphi] \neq M(\bar{x'})[\bar{K}] \), one of the following conditions hold:

a) \( D \) contains a tuple \( R_P(\bar{y}) \) s.t. \( \bar{y}[\bar{K}] = \bar{x}[\varphi] \) and \( \Delta \) contains neither \( \text{del}_R(\bar{y}) \) nor any \( \text{mod}_R(M, \bar{y}) \) s.t. \( M(\bar{y})[\bar{K}] \neq \bar{x}[\varphi] \), or

b) there is an \( \text{ins}_R(\bar{y}) \in \Delta \) s.t. \( \bar{y}[\bar{K}] = \bar{x}[\varphi] \), or

c) there is a \( \text{mod}_R(M, \bar{y}) \in \Delta \) s.t. \( R_P(\bar{y}) \in D \) and \( M(\bar{y})[\bar{K}] = \bar{x}[\varphi] \).

—*coherent* if no contradicting updates are issued on the same tuple, i.e., if \( \text{upd} = \text{del}_R(\bar{x}) \in \Delta \), then \( \text{ins}_R(\bar{x}) \notin \Delta \) and there is no \( M \) s.t. \( \text{mod}_R(M, \bar{x}) \in \Delta \); similar for other updates \( \text{upd} \). Note that if \( \Delta \) is coherent, \( D' = D \pm \Delta \) is well-defined.

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

—*admissible* if \( \Delta \) 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_D**. Let \( U_D, RA, \) and \( D \) be given.

—The set of *induced updates* \( \Delta(U) \) of a set of user requests \( U \subseteq U_D \) is the least set \( \Delta \) which contains \( U \) and is complete.

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
(analogously to the case of deletion, $\Delta(U)$ there is a unique least such set by construction).

— A set of user requests $U \subseteq U_D$ is admissible if $\Delta(U)$ is admissible, and maximal admissible if there is no other admissible $U'$, s.t. $U \subseteq U' \subseteq U_D$.

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

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

**Theorem 4.3 Adequacy.**

1. If $U \subseteq U_D$, then $\Delta(U)$ is founded and complete.
2. If a coherent $\Delta$ 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 $\Delta$ is complete, all updates propagated by $RA$ are contained in $\Delta$. Feasibility of $\Delta$ 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 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 in general not hold 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 which 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 $T,(1,2) \rightarrow R,(1,2)$, $T,(3,4) \rightarrow S,(1,2)$, $U,(1,2) \rightarrow T,(2,3)$, and a rac ON UPDATE OF CHILD RESTRICT for $T,(1,4) \rightarrow V,(1,2)$. Assume the database contains $R(a,b)$, $S(c,d)$, $T(a,b,c,d,d')$, $U(b,c,d,d')$, and $V(a,d,d')$, $V(c,d,d')$, $V(a,d,d')$.

Given $mod_R(a,a',b/b')$ and $mod_S(c,c',d/d')$, the rac’s trigger the modifications $mod_T(a,a',b/b',c,c',d/d',d')$. Since these updates to $T$ are coherent, they can be merged into $mod_T(a,a',b/b',c,c',d/d',d')$, which triggers $mod_U(b/b',c/c',d/d')$.  

ACM Transactions on Database Systems, Vol. V. No. N. Month 20YY.
On the other hand, the rac \( T, (1, 4) \to V, (1, 2) \) on update of child 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 which 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 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: \( \text{mod}_{T}(a/a', b/b', c, d', \ldots) \) and \( \text{mod}_{T}(a, b, c'/c, d'/d', \ldots) \) 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 if there is an appropriate parent. If not, the update must be blocked. For deletions, we could list all blocked deletions, i.e., 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 \( U_{P} \). 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 \( \text{chg}_{R} \bar{A}(M, \bar{X}) \) (defined below in (CH1)) and \( \text{prp}_{R_{P}} \bar{K} \to R_{C} \bar{F}(M_{C}, \bar{X}) \) (defined below in (MPC1)) are the "keys" to the solution:

- \( \text{chg}_{R} \bar{A}(M, \bar{X}) \) denotes that the (candidate or foreign) key value \( \bar{X}[\bar{A}] \) in relation \( R \) is modified by \( M \), i.e., changes to \( (M(\bar{X}))[\bar{A}] \).

For candidate keys \( R, \bar{A} \), this modification is propagated to all referencing foreign keys:

- \( \text{prp}_{R_{P}} \bar{K} \to R_{C} \bar{F}(M_{C}, \bar{x}) \) denotes that a modification \( M \) of \( R_{P}. \bar{K} \) is propagated along \( R_{P}. \bar{K} \to R_{C} \bar{F} \) to \( R_{C}(\bar{x}) \) resulting in the modification \( M_{C} = \bar{F}/M_{P}(\bar{y})[\bar{K}] \) of the foreign key value.

From the incoming foreign key changes, changes of overlapping candidate keys are computed (see rule (CH1) 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 Initialization 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:

\[
\begin{align*}
&\text{del}_R(\bar{x}) \iff \neg \text{del}_R(\bar{x}), \neg \text{blk}_{\text{del}}_R(\bar{x}). \\
&\text{ins}_R(\bar{x}) \iff \text{ins}_R(\bar{x}), \neg \text{blk}_{\text{ins}}_R(\bar{x}). \\
&\text{mod}_R(M, \bar{x}) \iff \text{mod}_R(M, \bar{x}), \neg \text{blk}_{\text{mod}}_R(M, \bar{x}).
\end{align*}
\]

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:

—$\text{is}_{\text{ref}}_R.\bar{K}^R(\bar{x})$: the (key) value $R.\bar{K}^R(\bar{x})$ is referenceable in the original state.

—$\text{rem}_{\text{ref}}_R.\bar{K}^R(\bar{x})$: the (key) value $R.\bar{K}^R(\bar{x})$ remains referenceable.

—$\text{new}_{\text{ref}}_R.\bar{K}^R(\bar{x})$: the (key) value $R.\bar{K}^R(\bar{x})$ becomes referenceable.

—$\text{is}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{v})$: in the current database, the key value $R.\bar{K}^R(\bar{v})$ appears as foreign key value of $\bar{F}^R$ in some tuple $R.C(\bar{x})$.

—$\text{rem}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{v})$: there is a reference to the key value $R.\bar{K}^R(\bar{v})$ as foreign key value of $\bar{F}^R$ in some tuple $R.C(\bar{x})$ s.t. $\bar{x}^R[\bar{F}^R]$ does not change.

—$\text{new}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{v})$: a reference to the key value $R.\bar{K}^R(\bar{v})$ as foreign key $\bar{F}^R$ in some tuple $R.C(\bar{x})$ is introduced by some update.

For every candidate key $\bar{K}$ mentioned in some ric $R.C.\bar{F}^R \rightarrow R.p.\bar{K}$:

\[
\begin{align*}
&\text{is}_{\text{ref}}_R.p.\bar{K}^R(\bar{V}) \iff R.p(\bar{x}), \bar{V} = \bar{x}[\bar{K}] . \\
&\text{rem}_{\text{ref}}_R.p.\bar{K}^R(\bar{V}) \iff R.p(\bar{x}), \bar{V} = \bar{x}[\bar{K}], \neg \text{del}_R.p(\bar{x}), \neg \exists M : \text{chg}_R.p.\bar{K}^R(M, \bar{x}) . \\
&\text{new}_{\text{ref}}_R.p.\bar{K}^R(\bar{V}) \iff \text{ins}_R.p(\bar{x}), \bar{V} = \bar{x}[\bar{K}] . \\
&\text{new}_{\text{ref}}_R.p.\bar{K}^R(\bar{V}) \iff \text{chg}_R.p.\bar{K}^R(M, \bar{x}), M(\bar{x})[\bar{K}] = \bar{V} .
\end{align*}
\]

(CK)

For every ric $R.C.\bar{F}^R \rightarrow R.p.\bar{K}$:

\[
\begin{align*}
&\text{is}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{V}) \iff R.C(\bar{x}), \bar{V} = \bar{x}[\bar{F}^R] . \\
&\text{rem}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{V}) \iff R.C(\bar{x}), \bar{V} = \bar{x}[\bar{F}^R], \neg \text{del}_R.C(\bar{x}), \\
&\neg \exists M : \text{chg}_R.C.\bar{F}^R(M, \bar{x}) . \\
&\text{new}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{V}) \iff R.C(\bar{x}), \text{chg}_R.C.\bar{F}^R(M, \bar{x}), M(\bar{x})[\bar{F}^R] = \bar{V} . \\
&\text{new}_{\text{refd}}_R.p.\bar{K}^R_by_R.C.\bar{F}^R(\bar{V}) \iff \text{ins}_R.C(\bar{x}), \bar{V} = \bar{x}[\bar{F}^R] .
\end{align*}
\]
4.2.2 Deletions. We only have to consider rac's of the form \( R_C.\bar{F} \rightarrow R_P.\bar{K} \) on delete of parent (cf. Table I). Logic rules that again describe their local behavior are given 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 NO ACTION: 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 s.t. it references another parent) \((DN)\).

\[
\begin{align*}
\text{del}_{R_C}(\bar{X}) & \leftarrow \text{del}_{R_P}(\bar{Y}), \quad R_C(\bar{X}), \quad \bar{X}[\bar{F}] = \bar{Y}[\bar{K}] . \quad (DC_1) \\
\text{blk}_{\text{del}}_{R_P}(\bar{Y}) & \leftarrow \text{blk}_{\text{del}}_{R_C}(\bar{X}), \quad \bar{X}[\bar{F}] = \bar{Y}[\bar{K}] . \quad (DC_2) \\
\text{blk}_{\text{del}}_{R_P}(\bar{Y}) & \leftarrow \text{i\_refd}_{R_P.\bar{K}}_{by\_R_C.\bar{F}}(\bar{Y}[\bar{K}]). \quad (DR) \\
\text{blk}_{\text{del}}_{R_P}(\bar{Y}) & \leftarrow \text{rem\_refd}_{R_P.\bar{K}}_{by\_R_C.\bar{F}}(\bar{Y}[\bar{K}]). \quad (DN)
\end{align*}
\]

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

\[
\begin{align*}
\text{pot\_del}_{R}(\bar{X}) & \leftarrow \triangleright \text{del}_{R}(\bar{X}), \quad R(\bar{X}). \\
\text{for each} \quad R_C.\bar{F} \rightarrow R_P.\bar{K} \text{ on delete of parent cascade (analogous to } \(DC_1)\): \quad (P) \\
\text{pot\_del}_{R_C}(\bar{X}) & \leftarrow \text{pot\_del}_{R_P}(\bar{Y}), \quad R_C(\bar{X}), \quad \bar{X}[\bar{F}] = \bar{Y}[\bar{K}].
\end{align*}
\]

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 \( \text{chg}_{R.\bar{A}}(M, \bar{X}) \) and \( \text{prp}_{R_P.\bar{K}} \rightarrow R_C.\bar{F}(M_C, \bar{X}) \) 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, i.e., 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 change to a key which is forbidden. For every key (candidate and foreign keys) \( R.\bar{A} \):

\[
\begin{align*}
\text{pot\_prp}_{\triangleright \rightarrow \bar{A}}(M, \bar{X}) & \leftarrow \triangleright \text{mod}_{R}(M', \bar{X}), \quad \bar{X}[\bar{A}] \neq M'[\bar{A}], \quad M = M'[\bar{A}]. \\
\text{blk\_mod}_{R}(M, \bar{X}) & \leftarrow \triangleright \text{mod}_{R}(M', \bar{X}), \quad \text{blk}_{\text{chg}}_{R.\bar{A}}(M', \bar{X}), \quad M' = M'[\bar{A}]. \\
\text{prp}_{\triangleright \rightarrow \bar{A}}(M, \bar{X}) & \leftarrow \text{mod}_{R}(M', \bar{X}), \quad \bar{X}[\bar{A}] \neq M'[\bar{A}], \quad M = M'[\bar{A}]. \quad (EXT_2)
\end{align*}
\]

Interaction of Modifications. Assume a modification \( \text{mod}_{R_P}(M_P, \bar{y}) \) and a rac \( R_C.\bar{F} \rightarrow R_P.\bar{K} \) on update of parent cascade s.t. the key value \( R_P.\bar{K} \) of \( R_P(\bar{y}) \) changes, which is denoted by \( \text{chg}_{R_P.\bar{K}}(M_P, \bar{y}) \) (analogously, there are relations \( \text{pot\_chg} \) and \( \text{blk\_chg} \)). Then, for every referencing child \( R_C(\bar{x}) \), this modification
is propagated to the corresponding foreign key, i.e., \( M_C = \overline{F_1}/M_P(\overline{g})[\overline{K_1}] \). This is stored in the propagation relation \( \text{prop}_{R_P, \overline{K_1}} \rightarrow R_C, \overline{F_1}(M_C, \overline{x}) \).

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, \((C,H)\) defines a finite set of rules for computing all possibilities how 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 which 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 \( R.A \), define the set of sources of influences on \( R.A \), \( S_{R.A} \subseteq \text{Keys}(R) \times (\text{ForeignKeys}(DB) \cup \{ \text{src} \}) \) as follows:

\(- S_{R.A} \) contains all pairs \( (R.P, \overline{K}_1, R.F_i) \) s.t. there is a \( \text{rac}_{R.P, \overline{K}_1} \rightarrow R.F_i \) ON UPDATE OF PARENT CASCADE and \( F_i \) overlaps \( A \) (i.e., key references whose propagation influences the value of \( R.A \)), and

\(- (R.A, \text{src}) \in S_{R.A} \) since external modifications of \( R \) also influence the value of \( R.A \).

For the following rule \((C.H_1)\), let \( S \) range over all subsets of \( S_{R.A} \):

\[
\begin{align*}
\text{pot.chg}_{R.A}(M, \overline{x}) & \leftarrow \left( \bigwedge_{i \in S} \text{pot.chg}_{R.P_i, \overline{K}_i} \rightarrow R.F_i(M, \overline{x}) \right), \\
M & = \bigcup_{i \in S} M_i[A] \text{ consistent, } M(\overline{x})[\overline{A}] \neq \overline{x}[\overline{A}] . \\
\text{chg}_{R.A}(M, \overline{x}) & \leftarrow \left( \bigwedge_{i \in S} \text{prop}_{R.P_i, \overline{K}_i} \rightarrow R.F_i(M_i, \overline{x}) \right), \\
M & = \bigcup_{i \in S} M_i[A] \text{ consistent, } M(\overline{x})[\overline{A}] \neq \overline{x}[\overline{A}] . \tag{C.H_1}
\end{align*}
\]

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 \( \overline{F} \) and foreign or candidate key \( A \) s.t. \( \overline{F} \) and \( A \) overlap:

\[
\text{allow.chg}_{R.F \cap A}(M, \overline{x}) \leftarrow \text{chg}_{R.F}(M, \overline{x}), \neg \text{blk.chg}_{R.F \cap A}(M, \overline{x}), M = M_1[\overline{F} \cap \overline{A}], \\
\text{chg}_{A}(M, \overline{x}), \neg \text{blk.chg}_{R.F \cap A}(M_2, \overline{x}), M = M_2[\overline{F} \cap \overline{A}] .
\]

\[
\text{blk.chg}_{R.F}(M, \overline{x}) \leftarrow \text{pot.chg}_{R.F}(M, \overline{x}), \neg \text{allow.chg}_{R.F \cap A}(M', \overline{x}), M' = M[\overline{F} \cap \overline{A}] . \tag{C.H_2}
\]

\textbf{Remark 4.4.} Consider a foreign or candidate key \( R.A \) such that there is only a single constraint \( R.P, \overline{K} \rightarrow R.F \) ON UPDATE OF PARENT CASCADE such that \( \overline{F} \) and \( A \) overlap. Then, \((C.H_1)\) maps the incoming change on \( \overline{F} \) to a change of \( A \):

\(A\) 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.

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
\text{pot.chg}_{\mathcal{R}} \tilde{A}(M, \tilde{X}) \leftarrow \text{pot.prp}_{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M', \tilde{X}), \ M = M'[\tilde{A}], \ M(\tilde{X})[\tilde{A}] \neq \tilde{X}[\tilde{A}].
\text{chg}_{\mathcal{R}} \tilde{A}(M, \tilde{X}) \leftarrow \text{prp}_{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M', \tilde{X}), \ M = M'[\tilde{A}], \ M(\tilde{X})[\tilde{A}] \neq \tilde{X}[\tilde{A}]. \quad (CH^3)

\textit{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:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{prp}_{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M, \tilde{X})</td>
<td>\leftarrow \text{chg}_{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}), \ R_C(\tilde{X}), \ \tilde{X}[\tilde{F}] = \tilde{Y}[\tilde{K}], \ M_C = \tilde{F}[M_p(\tilde{Y})[\tilde{K}]. \quad (MPC_1)</td>
</tr>
<tr>
<td>\text{pot.prp}_{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M, \tilde{X})</td>
<td>\leftarrow \text{pot.chg}_{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}), \ R_C(\tilde{X}), \ X[\tilde{F}] = Y[\tilde{K}], \ M_C = \tilde{F}[M_p(\tilde{Y})[\tilde{K}]. \quad (MPC_2)</td>
</tr>
<tr>
<td>\text{blk.chg}_{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y})</td>
<td>\leftarrow \text{blk.chg}<em>{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}), \ \text{blk.prop}</em>{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M, \tilde{X}), \ \tilde{X}[\tilde{F}] = \tilde{Y}[\tilde{K}], \ M_C = \tilde{F}[M_p(\tilde{Y})[\tilde{K}]. \quad (MPR)</td>
</tr>
<tr>
<td>\text{blk.prop}_{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M, \tilde{X})</td>
<td>\leftarrow \text{blk.prp}<em>{\mathcal{R}} r_p. \tilde{K} \rightarrow R. \tilde{F}(M, \tilde{X}), \ \text{blk.chg}</em>{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}), \ \text{blk.chg}<em>{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}) \text{ is refd}</em>{\mathcal{R}} r_p. \tilde{K} \text{ by } R. \tilde{F}(\tilde{Y}[\tilde{K}]). \quad (MPC_3)</td>
</tr>
<tr>
<td>\text{blk.chg}_{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y})</td>
<td>\leftarrow \text{blk.chg}<em>{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}), \ \text{blk.chg}</em>{\mathcal{R}} r_p. \tilde{K}(M, \tilde{Y}) \text{ is refd}_{\mathcal{R}} r_p. \tilde{K} \text{ by } R. \tilde{F}(\tilde{Y}[\tilde{K}]). \quad (MPN)</td>
</tr>
</tbody>
</table>

\textit{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 \( R_C.\tilde{F} \) of a child tuple wrt. a ric \( R_C.\tilde{F} \rightarrow R_p.\tilde{K} \) is blocked if the change is influenced from a propagation along another ric \( R_C.\tilde{F} \rightarrow R_p.\tilde{K} \) (i.e., \( R_C.\tilde{F} \rightarrow R_p.\tilde{K} \) on UPDATE OF PARENT CASCADE and \( R_C.\tilde{F} \) and \( R_C.\tilde{F} \) overlap) or from an external modification and the resulting tuple violates \( R_C.\tilde{F} \rightarrow R_p.\tilde{K} \) 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 \( R_C.\tilde{F} \rightarrow R_p.\tilde{K} \) ON UPDATE OF PARENT CASCADE and \( R_C.\tilde{F} \rightarrow R_p.\tilde{K} \) ON UPDATE OF CHILD RESTRICT s.t. \( R_C.\tilde{F} \neq R_C.\tilde{F} \) or \( R_p.\tilde{K} \neq R_p.\tilde{K} \) and \( R_C.\tilde{F} \)

\(^7\)note that it still occurs with every “diamond” (Fig. 1)

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
and $R_C.\vec{F}'$ overlap:

$$\text{blk}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}) \iff \text{pot}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}), \ \text{prp}_{R_p.\vec{F}' \rightarrow R_C.\vec{F}}(M', \bar{X}),$$
\[ M[\vec{F} \cap \vec{F}'] = M'[\vec{F} \cap \vec{F}'], \quad \text{is refutable}_{R_p.\vec{F}}(\vec{F}[M(\bar{X})]). \]

$$\text{blk}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}) \iff \text{pot}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}), \ \text{prp}_{R_p.\vec{F}' \rightarrow R_C.\vec{F}}(M', \bar{X}),$$
\[ M' \subseteq M, \quad \text{is refutable}_{R_p.\vec{F}}(M(\bar{X})[\vec{F}]). \]  

(MCR$_1$)

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 $R_P.\bar{K}$ or deletion of this tuple is blocked: For every $R_C.\vec{F} \rightarrow R_P.\bar{K}$ ON UPDATE OF CHILD RESTRICT:

$$\text{blk}_{\text{chg}}_{R_p.\vec{K}}(M_P, \bar{Y}), \quad \text{R}_{\bar{X}}(\bar{X}), \quad \text{ch}_{R_C.\vec{F}}(M, \bar{X}), \quad M[\bar{X}] = \bar{Y}[\bar{K}]. \quad \text{(MCR$_2$)}$$

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 s.t. $R_C.\vec{F} \neq R_C.\vec{F}'$ or $R_P.\vec{K} \neq R_P.\vec{K}'$ and $R_C.\vec{F}$ and $R_C.\vec{F}'$ overlap:

$$\text{blk}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}) \iff \text{pot}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}), \ \text{prp}_{R_p.\vec{K}' \rightarrow R_C.\vec{F}}(M', \bar{X}),$$
\[ M[\vec{F} \cap \vec{F}'] = M'[\vec{F} \cap \vec{F}'], \quad \text{rem refutable}_{R_p.\vec{K}}(M(\bar{X})[\vec{F}]), \quad \text{new refutable}_{R_p.\vec{K}}(M(\bar{X})[\vec{F}]). \]

$$\text{blk}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}) \iff \text{pot}_{\text{chg}}_{R_C.\vec{F}}(M, \bar{X}), \ \text{prp}_{R_p.\vec{F} \rightarrow R_C.\vec{F}}(M', \bar{X}),$$
\[ M' \subseteq M, \quad \text{rem refunable}_{R_p.\vec{K}}(M(\bar{X})[\vec{F}]), \quad \text{new refutable}_{R_p.\vec{K}}(M(\bar{X})[\vec{F}]). \]  

(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 $S_R = \text{ForeignKeys}(R) \cup \{(1, \ldots, n)\}$.

$$\text{ch}_{R}(1, \ldots, n)(M, \bar{X}) \iff \text{mod}_{R}(M, \bar{X}). \quad \text{(CH$_\oplus$)}$$

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

$$\text{mod}_{R}(M, \bar{X}) \iff \left( \bigwedge_{P \in S} \text{ch}_{R.\vec{F}}(M_P, \bar{X}) \right), \quad M = \bigcup_{P \in S} M_P \text{ consistent},$$
\[ \left( \bigwedge_{P \in S/\bar{R}} \neg \exists M': \text{ch}_{R.\vec{F}}(M', \bar{X}) \right). \quad \text{(MOD)} \]

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 $R_C.\vec{F} \rightarrow R_P.\bar{K}$ ON INSERT OF CHILD RESTRICT:

$$\text{blk}_{\text{ins}_{R_C.\vec{F}}(\bar{X})} \iff \text{ins}_{R_C.\vec{F}}(\bar{X}), \quad \text{is refutable}_{R_p.\vec{K}}(\bar{X}[\vec{F}]). \quad \text{(ICR)}$$

$$\text{blk}_{\text{chg}_{R_p.\vec{K}}(M, \bar{Y})} \iff \text{pot}_{\text{chg}_{R_p.\vec{K}}(M, \bar{Y})}, \quad \bar{Y}[\bar{K}] \neq M_P(\bar{Y})[\bar{K}],$$
\[ \text{ins}_{R_C.\vec{F}}(\bar{X}), \quad \bar{X}[\bar{K}] = \bar{Y}[\bar{K}]. \]

$$\text{blk}_{\text{del}_{R_p.\vec{Y}}(\bar{Y})} \iff \text{ins}_{R_C.\vec{F}}(\bar{X}), \quad \bar{X}[\bar{K}] = \bar{Y}[\bar{K}].$$

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

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
\[
\text{blk\_ins\_}R_C(\bar{X}) \leftarrow \triangleright\text{ins\_}R_C(\bar{X}), \neg \text{rem\_viable\_}R_P.\bar{K}(\bar{X}[\bar{F}]), \quad (ICN) \\
\neg \text{new\_viable\_}R_P.\bar{K}(\bar{X}[\bar{F}]). 
\]

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

\[
\begin{align*}
\text{blk\_ins\_}R(\bar{X}) & \leftarrow \triangleright\text{ins\_}R(\bar{X}), \text{del\_}R(\bar{X}) . \\
\text{blk\_del\_}R(\bar{X}) & \leftarrow \triangleright\text{ins\_}R(\bar{X}) . \\
\text{blk\_mod\_}R(M,\bar{X}) & \leftarrow \triangleright\text{mod\_}R(M,\bar{X}), \text{del\_}R(\bar{X}) . \\
\text{blk\_del\_}R(\bar{X}) & \leftarrow \triangleright\text{mod\_}R(M,\bar{X}) . \quad (C)
\end{align*}
\]

For every rac \( R_P.\bar{K} \rightarrow R_C.\bar{F} \) on update of parent cascade:

\[
\begin{align*}
\text{blk\_prop\_}R_P.\bar{K} \rightarrow R_C.\bar{F}(M,\bar{X}) & \leftarrow \text{pot\_prop\_}R_P.\bar{K} \rightarrow R_C.\bar{F}(M,\bar{X}), \text{del\_}R(\bar{X}) . \\
\text{blk\_del\_}R(\bar{X}) & \leftarrow \text{prp\_}R_P.\bar{K} \rightarrow R.\bar{F}(M,\bar{X}) . 
\end{align*}
\]

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 rac’s \( R_{P1}.\bar{K}_1 \rightarrow R.\bar{F}_1 \) on update of parent cascade and \( R_{P2}.\bar{K}_2 \rightarrow R.\bar{F}_2 \) on update of parent cascade s.t. \( R_{P1} \) and \( R_{P2} \) overlap:

\[
\begin{align*}
\text{blk\_prop\_}R_{P1}.\bar{K}_1 \rightarrow R.\bar{F}_1(M_1,\bar{X}) & \leftarrow \text{pot\_prop\_}R_{P1}.\bar{K}_1 \rightarrow R.\bar{F}_1(M_1,\bar{X}), \\
\text{prp\_}R_{P2}.\bar{K}_2 \rightarrow R.\bar{F}_2(M_2,\bar{X}) & , M_1 \cup M_2 \text{ inconsistent} . \quad (C)
\end{align*}
\]

The uniqueness of a candidate key \( R.\bar{K} \) is guaranteed by the following rules:

\[
\begin{align*}
\text{blk\_ins\_}R(\bar{X}) & \leftarrow \triangleright\text{ins\_}R(\bar{X}), \text{rem\_viable\_}R.\bar{K}(X[\bar{K}]) . \\
\text{blk\_chg\_}R(M,\bar{X}) & \leftarrow \text{pot\_chg\_}R(K(M,\bar{X}), \text{rem\_viable\_}R.\bar{K}(M(\bar{X})[\bar{K}]) . \\
\text{blk\_ins\_}R(\bar{X}) & \leftarrow \triangleright\text{ins\_}R(\bar{X}), \text{ins\_}R(\bar{Y}), \bar{X}[\bar{K}] = \bar{Y}[\bar{K}] . \\
\text{blk\_chg\_}R(M,\bar{X}) & \leftarrow \text{pot\_chg\_}R(K(M,\bar{Y}), \text{ins\_}R(\bar{X}), \bar{X}[\bar{K}] = M(\bar{Y})[\bar{K}] . \\
\text{blk\_chg\_}R(M,\bar{X}) & \leftarrow \text{pot\_chg\_}R(K(M,\bar{X}), \text{chg\_}R.\bar{K}(M',\bar{Y}), M(\bar{X})[\bar{K}] = M'(\bar{Y})[\bar{K}] . 
\end{align*}
\]

On Delete/Update Set Default/Set Null. The additional rac’s \( R_C.\bar{F} \rightarrow R_P.\bar{K} \) on update/delete set default and \( R_C.\bar{F} \rightarrow R_P.\bar{K} \) 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 \{block1 \( \leftarrow \text{exec} \), block2 \( \leftarrow \text{exec} \} \cup \{\text{exec} \leftarrow \neg \text{block} \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 \{\text{exec} \leftarrow \neg \text{block} \), \text{block} \leftarrow \text{exec} \}, where no total stable model exists.

ACM Transactions on Database Systems, Vol. V, No. N, Month YYY.
Definition 4.5. Every 3-valued model $\mathcal{M} := \mathcal{M}(P_{RA}, D, U_D)$ defines sets of updates $\Delta_\mathcal{M}$ and user requests $U_M \subseteq U_D$ which are true, false, or undefined in $\mathcal{M}$. Let $\text{upd}$ be any of $\text{ins}_R(\bar{x}), \text{del}_R(\bar{x}), \text{mod}_R(M, \bar{x})$, then:

$$\Delta^\text{true}_\mathcal{M} := \{\text{upd} \mid M(\text{upd}) = \text{true}\}, \quad \text{and} \quad U^\text{true}_M := \{\text{ upd} \in U_D \mid M(\text{upd}) = \text{true}\}$$

(analogous for false and undefined).

Again, we examine first the well-founded model $\mathcal{W} := \mathcal{W}(P_{RA}, D, U_D)$ which provides a safe, skeptical semantics which 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 a 3-valued model $\mathcal{M} := \mathcal{M}(P_{RA}, D, U_D)$, let $\Delta^\text{true+}_W \subseteq \Delta^\text{true}_W$ denote the set of non-subsumed updates, i.e., the set of all $\text{ upd} \in \Delta^\text{true}_W$ s.t. there is no $M'$ which subsumes $M$ and $\text{mod}_R(M', \bar{x}) \in \Delta^\text{true}_W$.

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

**Lemma 4.7.** $D \pm \Delta^\text{true}_W = D \pm \Delta^\text{true+}_W$.

**Theorem 4.8 Correctness: Well-Founded Semantics.**

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

**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) $\Delta^\text{true}_W \subseteq \Delta(U^\text{true}_W)$ follows from foundedness, $\Delta^\text{true}_W \supseteq \Delta(U^\text{true}_W)$ from completeness.

(3) follows from (1) and (2). $\square$

We see that already $\Delta^\text{true+}_W$ 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 $U \subseteq U_D$ extends $U^\text{true}_W$, and (ii) updates classified as false by $\mathcal{W}$ are definitely not admissible:

**Corollary 4.9.** For every maximal admissible $U \subseteq U_D$:

1. $U^\text{true}_W \subseteq U$,
2. $U^\text{false}_W \cap U = \emptyset$,
3. $\Delta^\text{false}_W \cap \Delta(U) = \emptyset$.

**Proof.** Follows from Theorem 4.8 and model-theoretic properties of the well-founded semantics. $\square$

The different types of undefined updates $\text{ upd} \in U^\text{undef}_W$ can be characterized according to the different types of controversial atoms:

- $\text{ upd} \in U$ for every maximal admissible $U \subseteq U_D$ ("diamond"), or
- there are maximal admissible sets $U, U' \subseteq U_D$ s.t. $\text{ upd} \in U$ and $\text{ upd} \not\in U'$ ("mutex"), or

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
—\textit{upd} \notin U \text{ for any admissible } U \subseteq U_D \text{ ("self-attack").}

As already mentioned, "diamonds" — even if they do not cause a problem — cause a negative cycle which is undefined in the well-founded model. Such a case is described in Example 12 in Section A of the 	extit{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 which executes the user request, and the one which 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 	extit{Mutex} (cf. Example 4), or for self-attacking requests (cf. Example 5).

\textit{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 possibility of total stable models, we have to consider \textit{P-stable} (partial stable) models:

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

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

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

\textbf{Theorem 4.11} \textit{Correctness, Completeness: Stable Semantics.}

—\textit{For every } \textit{P-stable model } S:\n\textit{(i) } \Delta_S^{\text{true}+} \text{ is admissible, (ii) } \Delta_S^{\text{true}+} = \Delta(\text{U}_S^{\text{true}}), \text{ (iii) } U_S^{\text{true}} \text{ is admissible.}

—\textit{For every maximal admissible } U \subseteq U_D, \text{ there is an M-stable model } M_S \text{ s.t. } U = U_M^{\text{true}} \text{ and } \Delta(U) = \Delta_M^{\text{true}+}.

\textit{M-stable models of } P_{RA} \text{ almost capture the notion of "optimal" (maximal admissible) solutions. Note that in case of mutual exclusion, there can be several } \textit{M-stable models which describe maximal admissible solutions. In case of a "diamond" } \{ \text{block } \leftarrow \neg \text{exec}, \text{ exec } \leftarrow \neg \text{block} \} \text{ (cf. Example 9) there are two } \textit{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 \text{ on } \textit{P-stable models according to our intended application:}

\[ S_1 <_a S_2 : \iff U_S^{\text{true}} \subset U_S^{\text{true}}. \]

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
Among M-stable models, \(<_a\) prefers those, which 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 \(S_1\) and \(S_2\), the following is equivalent:

1. \(S_1 <_a S_2\),
2. for every \(\triangledown \text{upd} \in U_D\), \(S_1(\triangledown \text{upd}) \leq S_2(\triangledown \text{upd})\),
3. for every (internal) update \(\text{upd}, S_1(\text{upd}) \leq S_2(\text{upd})\), or \(\text{upd} = \text{mod}_R(M, \bar{x})\) for some \(R, M, \bar{x}\), and there is an \(M'\) which subsumes \(M\) and \(S_1(\text{upd}) \leq S_2(\text{mod}_R(M', \bar{x}))\) (cf. Definition 4.6 and Theorem 4.8).

The maximal stable models wrt. \(<_a\) represent exactly the maximal admissible sets:

**Theorem 4.13** Maximality. Let \(D\) and \(U_D\) be as usual. Then, the following sets coincide:

— the set of all maximal admissible sets \(U \subseteq U_D\),
— the set of all \(U^{\text{true}}_{\text{AS}}\) s.t. \(\text{AS}\) is a \(<_a\)-maximal M-stable model of \(P_{RA}, D\), and \(U_D\).

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 does in general not hold:

**Lemma 4.14.** The well-founded model induces an upper bound for the set of admissible updates:

Let \(W = W(P_{RA}, D, U_D)\) be the well-founded model of \(P_{RA}, D\), and \(U_D\), and

\[ U^{\text{ub}} := U^{\text{true}}_{W} = \{\triangledown \text{upd} \in U_D \mid W(\text{upd}) \in \{\text{true}, \text{undef}\}\}. \]

— If \(U^{\text{ub}}\) is admissible, then there is a unique \(<_a\)-maximal stable model \(ASU\) of RA, \(D\), and \(U_D\), s.t. \(U^{\text{true}}_{ASU} = U^{\text{ub}}\). Then \(\Delta(U^{ub}) = \Delta_{ASU}^{\text{true}}\), and the new database is \(D \pm \Delta_{ASU}^{\text{true}}\).

— for every \(\text{upd} \in U_D\) which is in any admissible subset of \(U_D\), \(\text{upd} \in U^{\text{ub}}\).

Thus, the following necessary condition for admissibility \(U_D\) can be decided in \(\text{PTIME}\) using the well-founded model:

**Corollary 4.15.** If \(U^{ub} \subseteq U_D\), then \(U_D\) is not admissible.

Due to the various predicates, the above \(ASU\) 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 \(W := W(P_{RA}, D, U_D)\). If \(U^{ub}\) is admissible, then for every \(\text{upd} \in \Delta^{\text{true}\cup\text{undef}}_{W}\), \(ASU(\text{upd}) = \text{true}\), or \(\text{upd} = \text{mod}_R(M, \bar{x})\) for some \(R, M, \bar{x}\), and there is an \(M'\) which subsumes \(M\) and \(ASU(\text{mod}_R(M', \bar{x})) = \text{true}\) (cf. Lemma 4.12). (Analogously for \(\text{prop}\) and \(\text{chg}\) 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 \(W := W(P_{RA}, D, U_D)\). 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 consider also a history of the game which was not required for the simpler game-theoretic characterization in Section 3.3 with deletions only (which was in the famous win-move-style).

For given $U_D$, Player I claims a subset $U \subseteq U_D$ 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 $U' \subseteq U$ 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 wrt. 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 optimal). 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, e.g., $[R(a, b, c)|\text{mod}_R([1/x, 2/y], (a, b, c)) (\ldots)]R(x, y, c)$. At the beginning, Player I puts all plates that describe a set $U$ of updates that he claims to be maximal admissible wrt. given $D, RA$, and $U_D$. E.g., for $\text{del}_R(x) \in U$, he puts the plate $[R(\bar{x})|\text{del}_R(\bar{x}) (\emptyset) | c]$ on the tuple $R(\bar{x})$.

The Moves. During the game, the admissibility (i.e., 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 (i.e., 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

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
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” Π 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, Π 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 Π).

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

Model-theoretic aspects. Note that for given $D$, $RA$ and $U_{\text{p}}$ – 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 $U \subseteq U_{\text{p}}$. For a given initial setting, $\text{I}$ either wins or loses, there is no draw. In case that $\text{I}$ wins, the whole $Δ(U)$ is created explicitly on the board – retaining the whole history of the game. For given $D$, $RA$ and $U_{\text{p}}$, there can be several such $U$ such that $\text{I}$ wins the game. Each of them is a maximal admissible subset of $U_{\text{p}}$. 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 $U_{\text{p}}$ 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 $M_{\text{S}}$ has to be considered for the final check, and for computing the actual set $Δ$ of updates to the database. In case that $U_{\text{p}}$ is not admissible, information about maximal admissible subsets and about the problem situations is needed. 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 $U_{\text{p}}$ is admissible, and
—if $U_{\text{p}}$ 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.

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
—The following predicates are **fixed** predicates: pot_del, pot_prog, pot_chg.
—The following ones 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 what plates to play to win the game, including that he does only play 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", $W^+$ can be generated in polynomial time:

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

—for fixed predicates (evaluate to true or to false in $W$), $W^+(atom) := W(atom)$,
—for positive predicates, $W^+(atom) = true$ if $W(atom) \in \{true, undef\}$ and atom
not subsumed by another one.

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

Then, $W^+$ has to checked whether there is a stable model that coincides with $W^+$ on positive and fixed predicates. We call $W^+$ **positive-stable** if this is the case, which corresponds to a game where Player $\Pi$ has no successful attacks.

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

**Theorem 5.2 Positive-stable vs. Stable Models.** If $W^+$ is positive-stable, then there is a (unique) $<_\alpha$-maximal $M$-stable model $\mathcal{A}_S$ that coincides in all atoms over positive and fixed predicates with $W^+$. $\mathcal{A}_S$ is computed by applying the computation of the well-founded model starting with $W^+$ (instead of $\emptyset$).

**Proof.** As described above $W^+$ 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 $W^+$, it is stable (it cannot be attacked by Player $\Pi$).

$W^+$ corresponds to the unique $<_\alpha$-maximal $M$-stable model since in Corollary 4.9 and Lemma 4.14, it has been shown that updates $upd \in U_D$ that are false in $W$ are not contained in any admissible $U \subseteq U_D$. \[\square\]

In the following, we call the above $\mathcal{A}_S$ the **positive-stable model** to $W$. Note again that such a model does not always exist (e.g., if $W^+$ 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 admissibility check for $U_D$, the following result shows that this task can be done in $\text{PTIME}$ since computing $W$ and $\mathcal{A}_S$ is polynomial:

**Corollary 5.3 Admissibility of $U_D$.** If $U^{ub} = U_D$ and $W^+$ is positive-stable (i.e., there is an $\mathcal{A}_S$ which is the positive-stable model to $W$), then $U_D$ is admissible.

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

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
Lemma 5.4 Internal Updates in $\mathcal{AS}$ vs. Game. If $\mathcal{AS}$ is the positive-stable model to $\mathcal{W}$, then $\Delta_{\mathcal{AS}} = \Delta_{\mathcal{W}^+} = \Lambda$ where $\Lambda$ 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 $U_D$ (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 in case that 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 $U_{\mathcal{AS}} = U_D$, then $U_D$ is admissible and $\Delta_{\mathcal{AS}} = \Delta(U_D)$. The whole problem can be solved in polynomial time.

Thus, the steps for executing a set $U_D$ of updates on a database $D$ are as follows:

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

Note that if $U_D$ 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. Similar 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 in case that $U_D$ is rejected.

5.2 Troubleshooting in Case of Rejected Updates

Similar to Section 3.7, in case that $U_D$ 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 $U^{ub} = U_{\mathcal{W}}^{true, undef}$ that provides an upper bound which subset of $U_D$ can possibly be admissible. Thus, first problems are already identified during the computation of $\mathcal{W}$. The first time where a blocking for an upd $\in U_D$ or for a propagation along a cascading reference is derived in an underestimate, a problem situation is identified that can be reported.

Later, if $U^{ub} = U_D$, but it is still not admissible, the check 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 $\Delta_{\mathcal{W}^+}^{true+}$ identifies the problem situation. The conclusions how to cure the problem are the same as described in Section 3.7.
5.3 Refined Analysis

In the above well-founded model $W$, in general most updates will be undefined, thus, the subsequent investigation of $W^+$ is in general necessary. There are several typical situations, where even single, admissible updates yield only undefined updates in $W$, e.g., diamonds or cyclic dependencies. These can be detected by schema analysis, and then they 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; 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 which 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 $W^+$ 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, by regarding the transitive closure of referential actions, potential problems can be detected.

*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 has been cancelled and referential actions have been 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; ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY].
Markowitz 1990; 1991b] due to different execution orderings of referential actions (see also [Date 1990; Date and Darwen 1994]). The same characterization was also used in the early SQL3 working drafts, e.g. [ANSI/ISO 1991; 1992b; 1994].

Motivated by the problem of ambiguities, several research directions have been followed. Schema-based analysis provides conditions which 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; Horowitz 1989].

A “semantic” approach is followed by [Markowitz 1990]; Starting from an object-oriented or EER model of an application, criteria are given 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], i.e., that 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, i.e., 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 already 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, i.e., 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, e.g., [Abiteboul and Vianu 1991; Picouet and Vianu 1995; Ceri and Widom 1990; 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 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 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 (which 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 presence of referential actions of the form ON UPDATE CASCADE (which is still only partly supported in ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
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 how to change the application specification.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: http://www.acm.org/pubs/citations/journals/tods/20YY-V-N/p1-May.

It contains an example for the logic-programming characterization of referential actions in case of update operations hat 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 how to improve the presentation of the different aspects of the paper.

REFERENCES


Cochrane, R., Pirahesh, H., and Matthews, N. 1996. Integrating triggers and declarative constraints in SQL database systems. In *Int. Conf. on Very Large Data Bases (VLDB)*. 567-578.


Date, C. 1981. Referential Integrity. In *Int. Conf. on Very Large Data Bases (VLDB)*. 2-12.


ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.


received: TO BE INSERTED

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
This document is the online-only appendix to:

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

The electronic appendix contains an example for the logic-programming characterization of referential actions in 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.

A. Example for the Logic Programming Characterization of Referential Actions with Updates

In Section 4.2, a logic programming characterization for full referential actions, including modifications (i.e., ON UPDATE CASCADE) has been given. The following example instantiates the logic rules for a given situation.

Example 12 Modifications: Diamond. Consider again Figure 1, all references labeled with ON UPDATE OF PARENT CASCADE and ON UPDATE OF CHILD NO ACTION, which is a completely plausible setting. Consider the external modification request \( \triangleright \text{mod}_{R_1}(1/n, (a, \ldots)) \). Among many others, we have the following rules:

External modification on \( R_1 \) (\( \text{EXT}_1 \)):

\[
\text{mod}_{R_1}(M, \bar{X}) \leftrightarrow \triangleright \text{mod}_{R_1}(M, \bar{X}), \neg \text{blk}_{\text{mod}_{R_1}}(M, \bar{X}).
\]

From \( R_{2.1} \rightarrow R_{1.1} \) ON UPDATE OF PARENT CASCADE (\( \text{Refd} \)), \( (MPC_1) \), \( (MPC_2) \): Propagate the modification of the primary key \( R_{1.1} \) downwards to the foreign key \( R_{2.1} \) of the child tuple (if possible, otherwise block the update), and do the bookkeeping that the new value of \( R_{1.1} \) is referenced:

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires specific permission and/or a fee.

© 20YY ACM 0362-5915/20YY/0300-0001 $5.00
\[
\text{new_refd}_R_1.1 \text{ by } R_2.1(\vec{V}) \leftarrow R_2(\vec{X}), \ \text{chg}_R_2.1(M, \vec{X}), \ M(\vec{X})[1] = \vec{V}.
\]
\[
\text{prp}_R_1.1 \rightarrow R_2.1(M_C, \vec{X}) \leftarrow \text{chg}_R_1.1(M_P, \vec{Y}), \ R_2(\vec{X}), \ \vec{X}[1] = \vec{Y}[1],
\]
\[
M_C = 1/M_P(\vec{Y})[1].
\]
\[
\text{pot_prp}_R_1.1 \rightarrow R_2.1(M_C, \vec{X}) \leftarrow \text{potchg}_R_1.1(M_P, \vec{Y}), \ R_2(\vec{X}), \ \vec{X}[1] = \vec{Y}[1],
\]
\[
M_C = 1/M_P(\vec{Y})[1].
\]
\[
\text{blk_chg}_R_1.1(M_P, \vec{Y}) \leftarrow \text{potchg}_R_1.1(M_P, \vec{Y}), \ \text{blkprop}_R_1.1 \rightarrow R_2.1(M_C, \vec{X}),
\]
\[
\vec{X}[1] = \vec{Y}[1], \ M_C = 1/M_P(\vec{Y})[1].
\]
\[
\text{blkprop}_R_1.1 \rightarrow R_2.1(M, \vec{X}) \leftarrow \text{prp}_R_1.1 \rightarrow R_2.1(M, \vec{X}), \ \text{blkchg}_R_2.1(M, \vec{X}).
\]

\text{Analogously, from } R_3.1 \rightarrow R_1.1 \text{ on update of parent cascade (Refd), } (MPC_1), (MPC_2):

\[
\text{new_refd}_R_1.1 \text{ by } R_3.1(\vec{V}) \leftarrow R_3(\vec{X}), \ \text{chg}_R_3.1(M, \vec{X}), \ M(\vec{X})[1] = \vec{V}.
\]
\[
\text{prp}_R_1.1 \rightarrow R_3.1(M_C, \vec{X}) \leftarrow \text{chg}_R_1.1(M_P, \vec{Y}), \ R_3(\vec{X}), \ \vec{X}[1] = \vec{Y}[1],
\]
\[
M_C = 1/M_P(\vec{Y})[1].
\]
\[
\text{pot_prp}_R_1.1 \rightarrow R_3.1(M_C, \vec{X}) \leftarrow \text{potchg}_R_1.1(M_P, \vec{Y}), \ R_3(\vec{X}), \ \vec{X}[1] = \vec{Y}[1],
\]
\[
M_C = 1/M_P(\vec{Y})[1].
\]
\[
\text{blk_chg}_R_1.1(M_P, \vec{Y}) \leftarrow \text{potchg}_R_1.1(M_P, \vec{Y}), \ \text{blkprop}_R_1.1 \rightarrow R_3.1(M_C, \vec{X}),
\]
\[
\vec{X}[1] = \vec{Y}[1], \ M_C = 1/M_P(\vec{Y})[1].
\]
\[
\text{blkprop}_R_1.1 \rightarrow R_3.1(M, \vec{X}) \leftarrow \text{prp}_R_1.1 \rightarrow R_3.1(M, \vec{X}), \ \text{blkchg}_R_3.1(M, \vec{X}).
\]

\text{Analogously, from } R_4.(1, 2) \rightarrow R_2.(1, 2) \text{ on update of parent cascade (Refd), } (MPC_1), (MPC_2):

\[
\text{new_refd}_R_2.(1, 2) \text{ by } R_4.1.(1, 2)(\vec{V}) \leftarrow R_C(\vec{X}), \ \text{chg}_R_4.1.(1, 2)(M, \vec{X}), \ M(\vec{X})[1, 2] = \vec{V}.
\]
\[
\text{prp}_R_2.(1, 2) \rightarrow R_4.1.(1, 2)(M_C, \vec{X}) \leftarrow \text{chg}_R_2.(1, 2)(M_P, \vec{Y}), \ R_4(\vec{X}),
\]
\[
\vec{X}[1, 2] = \vec{Y}[1, 2], \ M_C = (1, 2)/M_P(\vec{Y})[1, 2].
\]
\[
\text{pot_prp}_R_2.(1, 2) \rightarrow R_4.(1, 2)(M_C, \vec{X}) \leftarrow \text{potchg}_R_2.(1, 2)(M_P, \vec{Y}), \ R_4(\vec{X}),
\]
\[
\vec{X}[1, 2] = \vec{Y}[1, 2], \ M_C = (1, 2)/M_P(\vec{Y})[1, 2].
\]
\[
\text{blk_chg}_R_2.(1, 2)(M_P, \vec{Y}) \leftarrow \text{potchg}_R_2.(1, 2)(M_P, \vec{Y}),
\]
\[
\text{blkprop}_R_2.(1, 2) \rightarrow R_4.(1, 2)(M_C, \vec{X}), \ \vec{X}[1, 2] = \vec{Y}[1, 2], \ M_C = (1, 2)/M_P(\vec{Y})[1, 2].
\]
\[
\text{blkprop}_R_2.(1, 2) \rightarrow R_4.(1, 2)(M, \vec{X}) \leftarrow \text{prp}_R_2.(1, 2) \rightarrow R_4.(1, 2)(M, \vec{X}), \ \text{blkchg}_R_4.1.(1, 2)(M, \vec{X}).
\]

\text{Analogously, from } R_4.(1, 3) \rightarrow R_3.(1, 2) \text{ on update of parent cascade (Refd), } (MPC_1), (MPC_2):

ACM Transactions on Database Systems, Vol. V, No. N. Month 20YY.
new_refd_R3.(1, 2) by R4.(1, 3)(\bar{V}) \leftarrow R_C(\bar{X}), \ chg_R4.(1, 3)(M, \bar{X}), M(\bar{X})[1, 3] = \bar{V}.

prp_R3.(1, 2) \rightarrow R4.(1, 3)(M_C, \bar{X}) \leftarrow chg_R3.(1, 2)(M_P, \bar{Y}), R_4(\bar{X}), \bar{X}[1, 3] = \bar{Y}[1, 2], M_C = (1, 3)/M_P(\bar{Y})[1, 2].

pot_prp_R3.(1, 2) \rightarrow R4.(1, 3)(M_C, \bar{X}) \leftarrow pot_chg_R3.(1, 2)(M_P, \bar{Y}), R_4(\bar{X}), \bar{X}[1, 3] = \bar{Y}[1, 2], M_C = (1, 3)/M_P(\bar{Y})[1, 2].

blk_chg_R3.(1, 2)(M_P, \bar{Y}) \leftarrow pot_chg_R3.(1, 2)(M_P, \bar{Y}), blk_prop_R3.(1, 2) \rightarrow R4.(1, 3)(M_C, \bar{X}), \bar{X}[1, 3] = \bar{Y}[1, 2], M_C = (1, 3)/M_P(\bar{Y})[1, 2].

blk_prop_R3.(1, 2) \rightarrow R4.(1, 3)(M, \bar{X}) \leftarrow pot_prp_R3.(1, 2) \rightarrow R4.(1, 3)(M, \bar{X}), blk_chg_R4.(1, 3)(M, \bar{X}).

For the (primary and foreign) key R1.1 (CH1) and (RCK), translate the external updates to R1 into their effects on R1.1 and do bookkeeping of referenceable values:

pot_chg_R1.1(M, \bar{X}) \leftarrow pot_prp_D \rightarrow R1.1(M', \bar{X}), M(\bar{X})[1] \neq \bar{X}[1].

chg_R1.1(M, \bar{X}) \leftarrow prp_D \rightarrow R1.1(M', \bar{X}), M(\bar{X})[1] \neq \bar{X}[1].

new_refable_R1.1(\bar{V}) \leftarrow chg_R1.1(M, \bar{X}), M(\bar{X})[1] = \bar{V}.

The foreign key R2.1 does not overlap with other foreign keys, thus (CH1) considers only its "own" parent key (again we omit D_R2):

pot_chg_R2.1(M, \bar{X}) \leftarrow pot_prp_R1.1 \rightarrow R2.1(M', \bar{X}), M = M'[1], M(\bar{X})[1] \neq \bar{X}[1].

chg_R2.1(M, \bar{X}) \leftarrow prp_R1.1 \rightarrow R2.1(M', \bar{X}), M = M'[1], M(\bar{X})[1] \neq \bar{X}[1].

Analogous for R3.1.

For the key R2.(1, 2) (CH1) and (RCK), translate the incoming updates along R2.1 \rightarrow R1.1 on UPDATE OF PARENT CASCADE to updates into their effects on R2.(1, 2) and do bookkeeping of referenceable values (for space restrictions, we omit the influence of D_R2 in (CH1)):

pot_chg_R2.(1, 2)(M, \bar{X}) \leftarrow pot_prp_R1.1 \rightarrow R2.1(M', \bar{X}), M = M'[1, 2], M(\bar{X})[1, 2] \neq \bar{X}[1, 2].

chg_R2.(1, 2)(M, \bar{X}) \leftarrow prp_R1.1 \rightarrow R2.1(M', \bar{X}), M = M'[1, 2], M(\bar{X})[1, 2] \neq \bar{X}[1, 2].

new_refable_R2.(1, 2)(\bar{V}) \leftarrow chg_R2.(1, 2)(M, \bar{X}), M(\bar{X})[1, 2] = \bar{V}.

Analogous for the key R3.(1, 2).

The foreign key R4.(1, 2) can be influenced either by its own parent key (which has to be regarded as atomic, thus, it cannot be augmented by any propagation along R4.(1, 3) \rightarrow R3.(1, 2)), or if there is no change at the parent) by a propagation along R4.(1, 3) \rightarrow R3.(1, 2) (again we omit D_R4). The rule schema (CH1) yields the
following rules:

\[
pot_{\text{chg}} R_4.(1, 2)(M, \tilde{X}) \leftarrow pot_{\text{prp}} R_2.(1, 2)^{\text{↔}} R_4.(1, 2)(M', \tilde{X}),
M = M_1[1, 2], M(\tilde{X})[1, 2] \neq \tilde{X}[1, 2].
\]

\[
chg R_4.(1, 2)(M, \tilde{X}) \leftarrow \text{ppp}_R_2.(1, 2)\text{↔} R_4.(1, 2)(M_1, \tilde{X}),
M = M_1[1, 2], M(\tilde{X})[1, 2] \neq \tilde{X}[1, 2].
\]

\[
pot_{\text{chg}} R_4.(1, 2)(M, \tilde{X}) \leftarrow pot_{\text{prp}} R_3.(1, 2)\text{↔} R_4.(1, 2)(M', \tilde{X}),
M = M_1[1, 2], M(\tilde{X})[1, 2] \neq \tilde{X}[1, 2].
\]

\[
chg R_4.(1, 2)(M, \tilde{X}) \leftarrow \text{ppp}_R_5.(1, 2)\text{↔} R_4.(1, 2)(M_1, \tilde{X}),
M = M_1[1, 2], M(\tilde{X})[1, 2] \neq \tilde{X}[1, 2].
\]

Analogous for \( R_4.(1, 3) \).

\( MCN \) contributes rules for \( R_4.(1, 2) \) and \( R_4.(1, 3) \) since there we have overlapping foreign keys which are changed by different parent key propagations:

\[
blk_{\text{chg}} R_4.(1, 2)(M, \tilde{X}) \leftarrow pot_{\text{chg}} R_4.(1, 2)(M, \tilde{X}), \text{ppp}_R_3.(1, 2)\text{↔} R_4.(1, 3)(M', \tilde{X}),
M[1] = M'[1], \text{rem}_{\text{refable}} R_2.(1, 2)(M(\tilde{X})[1, 2],
\neg new_{\text{refable}} R_2.(1, 2)(M(\tilde{X})[1, 2]).
\]

\[
blk_{\text{chg}} R_4.(1, 3)(M, \tilde{X}) \leftarrow pot_{\text{chg}} R_4.(1, 3)(M, \tilde{X}), \text{ppp}_R_2.(1, 2)\text{↔} R_4.(1, 2)(M', \tilde{X}),
M[1] = M'[1], \text{rem}_{\text{refable}} R_3.(1, 3)(M(\tilde{X})[1, 3],
\neg new_{\text{refable}} R_3.(1, 3)(M(\tilde{X})[1, 3]).
\]

Finally, the schema \( CH_2 \) adds the following rules for interferences between the overlapping foreign keys \( R_4.(1, 2) \) and \( R_4.(1, 3) \):

\[
allow_{\text{chg}} R_4.(1, 2)\cap(1, 3, M, \tilde{X}) \leftarrow
chg R_4.(1, 2)(M_1, \tilde{X}), blk_{\text{chg}} R_4.(1, 2)(M_1, \tilde{X}), M = M_1[1],
chg R_4.(1, 3)(M_2, \tilde{X}), blk_{\text{chg}} R_4.(1, 3)(M_2, \tilde{X}), M = M_2[1].
\]

\[
blk_{\text{chg}} R_4.(1, 2)(M, \tilde{X}) \leftarrow pot_{\text{chg}} R_4.(1, 2)(M, \tilde{X}),
\neg allow_{\text{chg}} R_4.(1, 2)\cap(1, 3, M', \tilde{X}), M' = M[1].
\]

\[
blk_{\text{chg}} R_4.(1, 3)(M, \tilde{X}) \leftarrow pot_{\text{chg}} R_4.(1, 3)(M, \tilde{X}),
\neg allow_{\text{chg}} R_4.(1, 2)\cap(1, 3, M', \tilde{X}), M' = M[1].
\]

Evaluating this program wrt. the well-founded semantics (via the AFP characterization) yields the following sequence of truth values (e.g., “ttf…” denoting “true-false-true-false-…”):

\[
\triangleright mod R_1.(1/n, a, \ldots)
\]
\[
pot_{\text{chg}} R_1.(1/n, a, \ldots)
\]
\[
pot_{\text{prp}} R_1.(1/n, a, b, b, \ldots)
\]
\[
pot_{\text{chg}} R_2.(1, 2)(1/n, a, b, b, b, \ldots)
\]
\[
pot_{\text{prp}} R_2.(1, 2)\leftrightarrow R_4.(1, 2)(1/n, a, b, b, b, \ldots)
\]
\[
pot_{\text{chg}} R_4.(1, 2)(1/n, a, b, b, b, b, \ldots)
\]
\[
pot_{\text{prp}} R_4.(1, 2)\leftrightarrow R_4.(1, 2)(1/n, a, b, b, b, b, b, \ldots)
\]
\[
pot_{\text{chg}} R_4.(1, 2)(1/n, a, b, b, b, b, b, b, \ldots)
\]
\[
pot_{\text{prp}} R_4.(1, 2)\leftrightarrow R_4.(1, 2)(1/n, a, b, b, b, b, b, b, b, \ldots)
\]
\[
pot_{\text{chg}} R_4.(1, 2)(1/n, a, b, b, b, b, b, b, b, b, \ldots)
\]
\[
pot_{\text{prp}} R_4.(1, 2)\leftrightarrow R_4.(1, 2)(1/n, a, b, b, b, b, b, b, b, b, b, \ldots)
\]
\[
pot_{\text{chg}} R_4.(1, 2)(1/n, a, b, b, b, b, b, b, b, b, b, b, \ldots)
\]
\[
pot_{\text{prp}} R_4.(1, 2)\leftrightarrow R_4.(1, 2)(1/n, a, b, b, b, b, b, b, b, b, b, b, b, \ldots)
\]
Thus, all actual modifications and changes, as well as all blockings are undefined in the well-founded model. Nevertheless, the modification is admissible.

B. GAME-THEORETIC CHARACTERIZATION OF REFERENTIAL ACTIONS WITH UPDATES

In this section, we present an equivalent game-theoretic characterization of maximal admissible sets of updates. Here, we need to consider also a history of the game which was not required for the simpler game-theoretic characterization in Section 3.3 with deletions only (which was in the famous win-move-style).

For a given \( U \subseteq U \rightarrow \), Player I claims a subset \( U \subseteq U \rightarrow \) 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 \( U' \subseteq U \rightarrow \) 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 wrt. 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 $\Delta(U)$. 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.

B.1 Rules of the Game

B.1.1 Setting and Initialization

*Setting.* The positions of the game are all tuples of the database $D$: $\left[R(\overline{x})\right]$ s.t. $R(\overline{x}) \in D$ and $\left[\left\{ \text{\texttt{ins}}_R(\overline{X}) \text{ s.t. } \text{\texttt{ins}}_R(\overline{X}) \in U_\text{p} \right\} \right]$ many positions $\square$. Thus, the “board” is practically a graphical representation of the database (see Example 15 and Figures 6–8). The game is played by putting plates that represent the update operations performed on the database: Each plate consists of a source tuple ($\in D$), an update (over the active domain of the database and the updates; $\text{\texttt{adom}}(D) \cup \text{\texttt{adom}}(U_\text{p})$), and a result tuple, e.g., $\left[R(\langle a, b, c \rangle)\text{mod}_R(\langle \frac{1}{x}, \frac{2}{y} \rangle, \langle a, b, c \rangle )\right] \xrightarrow{\langle x, y, c \rangle} R$. The update plates also contain positions $\square$ (as many boxes as there are atomic updates described by the plate), indicating how many cascading steps are needed for founding the update. Each position can be filled with a pebble (as a question) or with a number. E.g., for the above plate, there are two positions (for $\frac{1}{x}$ and for $\frac{2}{y}$): $\left[R(\langle a, b, c \rangle)\text{mod}_R(\langle \frac{1}{x}, \frac{2}{y} \rangle, \langle a, b, c \rangle )\right] \xrightarrow{\langle x, y, c \rangle} R$. This means the modification of position 1 to $x$ is founded by two cascading steps, and $\Pi$ currently forces $\Pi$ to tell her how many steps are needed for founding the modification $\frac{2}{y}$. There are four kinds of update plates:

- **No change:** $\left[R(\overline{x})\text{unchanged}\right] R(\overline{x})$,
- **Deletion:** $\left[R(\overline{x})\text{del}_R(\overline{X}) (\square) \varepsilon \right]$,
- **Insertion:** $\left[\varepsilon \text{\texttt{ins}}_R(\overline{X}) (\square) \right] R(\overline{x})$,
- **Modification:** $\left[R(\overline{x})\text{mod}_R([m_1, \ldots, m_n], \overline{x}) (\square^n) \right] R(M(\overline{x}))$; $M = [m_1, \ldots, m_n]$.

**Player I: Start.** Player I claims that a set $U \subseteq U_\text{p}$ of updates is maximal admissible and puts the following plates with numbers on suitable positions:

$$\left[R(\overline{x})\text{del}_R(\overline{x}) (0) \varepsilon \right] : \text{\texttt{del}}_R(\overline{x}) \in U$$

$$\left[R(\overline{x})\text{mod}_R([m_1, \ldots, m_n], \overline{x}) (0^n) \right] R(M(\overline{x})) : \text{\texttt{mod}}_R(M, \overline{x}) \in U \ ; M = [m_1, \ldots, m_n]$$

$$\left[\varepsilon \text{\texttt{ins}}_R(\overline{x}) (0) \right] R(\overline{x}) : \text{\texttt{ins}}_R(\overline{x}) \in U$$

**B.1.2 Questions and Answers.** The game is played such that Player $\Pi$ asks “questions”, attacking the claim of Player $I$: $\Pi$ attacks the admissibility (cf. Definition 4.1) of $U$, i.e., foundedness, completeness, and feasibility. Coherence is inherent to the game, and uniqueness of key values is guaranteed by the winning conditions. $I$ has to answer each attack – if he has no answer, he loses. We will describe each aspect below by *attack-defense-pairs:* $\Pi$ 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. In most answers, I will put a new plate (thereby constructing $\Delta(U)$); then the number positions of the plate are initially empty.

Completeness/Cascading. If an update plate is positioned on a tuple that has a child tuple with a CASCADE reference, II can ask I to cascade the update (e.g., when trying to follow a $DC^* \circ DR$ or $DC^* \circ DN$ chain to a problem situation). II answers by materializing the cascaded update:

Cascade Attack. For a deletion plate $[R_P(\bar{y}) \mid \text{del}_P(M_P, \bar{y}) \langle \omega \rangle] \epsilon$ or a modification plate $[R_P(\bar{y}) \mid \text{mod}_P(M_P, \bar{y}) \langle \omega \rangle \ldots]$, a rac $R_C.F \rightarrow R_{P.K}$ on DELETE/UPDATE OF PARENT CASCADE, and a tuple $R_C(\bar{x})$ s.t. $\bar{x}[\bar{F}] = \bar{y}[\bar{K}]$ and – in case of a modification – $\bar{y}[\bar{K}] \neq M_P(\bar{y})[\bar{K}]$, Player II can put a propagate-wire labelled with the ric from the updated parent tuple to the referencing tuple, asking “what about this reference?”:

$$
\begin{array}{c}
R_C.F \rightarrow R_{P.K} \text{ CASCADE}\ \\
\bar{x}[\bar{F}] = \bar{y}[\bar{K}] \\
\end{array}
\begin{array}{c}
\text{RC}(\bar{x}) \\
\end{array}
\begin{array}{c}
\Rightarrow \ \\
\text{RC}(\bar{x}) \rightarrow R_{C.K} \text{ CASCADE}\ \\
\bar{x}[\bar{F}] = \bar{y}[\bar{K}] \\
\end{array}
\begin{array}{c}
\text{RC}(\bar{x}) \\
\end{array}
\begin{array}{c}
R_P(\bar{y}) \mid \text{del}_P(M_P, \bar{y}) \langle \omega \rangle \mid R_P(M_P(\bar{y})) \\
R_P(\bar{y}) \mid \text{mod}_P(M_P, \bar{y}) \langle \omega \rangle \mid R_P(M_P(\bar{y})) \\
\end{array}
\begin{array}{c}
\Rightarrow \ \\
\text{RC}(\bar{x}) \mid \text{del}\text{-}_{RC}(M_C, \bar{x}) \langle \square^n \rangle | R_C(M_C(\bar{x})) \\
\text{RC}(\bar{x}) \mid \text{mod}\text{-}_{RC}(M_C, \bar{x}) \langle \square^n \rangle | R_C(M_C(\bar{x})) \\
\end{array}
$$

Cascade Answer. A propagate-wire from a plate $[R_P(\bar{y}) \mid \text{del}_P(M_P, \bar{y}) \langle \omega \rangle] \epsilon$ or $[R_P(\bar{y}) \mid \text{mod}_P(M_P, \bar{y}) \langle \omega \rangle \ldots]$ to a tuple $R_C(\bar{x})$ is answered by putting a plate $[R_C(\bar{x}) \mid \text{del}_C(M_C, \bar{x}) \langle \square^n \rangle] \epsilon$ resp. $[R_C(\bar{x}) \mid \text{mod}_C(M_C, \bar{x}) \langle \square^n \rangle \ldots]$ onto $R_C(\bar{x})$, and putting an action wire labelled with $R_C.F \rightarrow R_{P.K}$ from the update component of the parent to the update component of the child.

If the target tuple already holds a plate, there is nothing to do than adding an action wire from the update component of the parent to the update component of the child. Player I loses if the plates are inconsistent (i.e., the child plate does not represent the correct key value – trapped).

Feasibility: Restricting. If an update plate is positioned on a tuple that has a child tuple with a RESTRICT reference, II can show the reference and wins immediately:
Restrict Attack. For a deletion plate \( R_P(\overline{y}) \triangleleft_{\text{del}} R_P(\overline{y}) \) or a modification plate \( R_P(\overline{y}) \triangleleft_{\text{mod}} R_P(M, \overline{y}) \), a rac \( R_C.\overline{F} \rightarrow R_P.\overline{K} \) on DELETE/UPDATE of PARENT RESTRICT, and a tuple \( R_C(\overline{x}) \) s.t. \( \overline{x}[\overline{F}] = \overline{y}[\overline{K}] \), Player II can point to this tuple, and Player I immediately loses.

Feasibility: No Action. If an update plate is positioned on a tuple that has a child tuple with a NO ACTION reference, II can show the reference, asking what happens to that child/reference. I answers by showing (claiming) that the child is deleted or modified (which must be founded from somewhere else and which must be admissible, leading to II’s next move):

No Action Attack. For a deletion plate \( R_P(\overline{y}) \triangleleft_{\text{del}} R_P(\overline{y}) \) or a modification plate \( R_P(\overline{y}) \triangleleft_{\text{mod}} R_P(M, \overline{y}) \), a rac \( R_C.\overline{F} \rightarrow R_P.\overline{K} \) on DELETE/UPDATE of PARENT NO ACTION, and a tuple \( R_C(\overline{x}) \) s.t. \( \overline{x}[\overline{F}] = \overline{y}[\overline{K}] \) and – in case of a modification – \( \overline{y}[\overline{K}] \neq M_P(\overline{y})[\overline{K}] \), Player II can put a no-action-wire labelled with the ric from the source tuple of the update to the referencing tuple, asking “what about this reference?” (the deletion situation on the left side will be continued in Example 13 with the answer and a following Founding Attack step):

No Action Answer. For showing how the referencing tuple is adapted, Player II puts some (consistent) update plate on the result tuple.

Such situations frequently occur in a diamond, where the update cascades along another way (cf. the Example 15 that describes a complete game).

Example 13 Update Game – Selected Steps.
Continuing the above situation (left side; deletion), I puts a deletion plate on the referencing tuple, arguing that this tuple will also be deleted. Note that this “deletion” is at that time just a claim, and its foundedness will very probably be attacked by II (as will be shown below).

In case that the referencing tuple already holds a plate, there is nothing to do. In this case, player I loses if the plates are inconsistent (i.e., the child plate does not represent the correct key value – again trapped).

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
Founding. If \( \Pi \) answers – e.g. as described above – with putting a delete or update plate somewhere, \( \Pi \) can ask him how this modification is founded (or, if it is an update plate where several foreign keys change, how each component of it is founded). \( \Pi \) has to answer by showing a parent that cascades an appropriate modification, and by saying how many steps are needed to prove the correctness.

**Founding Attack.** For any plate with some empty number position, 
\[
\begin{align*}
R(\vec{x})|\text{del}_R(\vec{x}) (\varepsilon) \varepsilon & \quad \text{or} \quad R(\vec{x})|\text{mod}_R([m_1, \ldots, m_n], \vec{x}) (k_1, \ldots, k_i, \ldots, k_n) \varepsilon \\
\end{align*}
\]
Player \( \Pi \) can place a pebble on the number position, asking “why \( m_i \)?”:
\[
\begin{align*}
R(\vec{x})|\text{del}_R(\vec{x}) (\bullet) \varepsilon & \quad \text{or} \quad R(\vec{x})|\text{mod}_R([m_1, \ldots, m_n], \vec{x}) (k_1, \ldots, k_i, \bullet, k_{i+1}, \ldots, k_n) \varepsilon \\
\end{align*}
\]

**Example 13 Update Game – Selected Steps (Cont’d).**
Continuing the above situation, \( \Pi \) will ask how the newly placed deletion plate is founded (by putting a pebble on the empty “\( \square \)” on the plate).

**Founding Answer.** For a new pebble, 
\[
R_C(\vec{x})|\text{del}_R(\vec{x}) (\bullet) \varepsilon
\]
resp.
\[
R_C(\vec{x})|\text{mod}_R([m_1, \ldots, m_n], \vec{x}) (k_1, \ldots, k_{i-1}, k_\bullet, k_{i+1}, \ldots, k_n) \varepsilon
\]
I chooses a suitable plate for founding the questioned update (this can be either a plate that is already present, or I puts a new plate):

— in case of a deletion: 
\[
R_P(\vec{g})|\text{del}_R(\vec{g}) (\varepsilon) \varepsilon
\]
such that \( \vec{x} = g_K \), or

— in case of a modification: 
\[
R_P(\vec{g})|\text{mod}_R(M_P, \vec{g}) (\varepsilon) \varepsilon
\]
such that \( \vec{x} = g_K \),
\[
[m_1, \ldots, m_n] [\vec{F}] = \vec{F}/M_P(\vec{g})[K], \text{ and } m_i \in \vec{F}/M_P(\vec{g})[K].
\]
Then, I adds an action-wire marked with a ric \( R_C(\vec{x}) \rightarrow R_P(\vec{K}) \) from the update component of the founding plate to the update component \( \text{del}_R(\vec{x}) \) resp. \( \text{mod}_R([m_1, \ldots, m_n], \vec{x}) (k_1, \ldots, k_{i-1}, k_i, k_{i+1}, \ldots, k_n) \) of the plate under consideration and replaces the pebble by some \( k_i \in \mathbb{N} \) (claiming that the update is founded in \( k_i \) steps):

\[
\begin{align*}
R_P(\vec{g})|\text{del}_R(\vec{g}) (\varepsilon) & \varepsilon \\
R_P(\vec{g})|\text{mod}_R(M_P, \vec{g}) (\varepsilon) & \varepsilon \\
R_C(\vec{x})|\text{del}_R(\vec{x}) (k) & \varepsilon \\
R_C(\vec{x})|\text{mod}_R([m_1, \ldots, m_n], \vec{x}) (k_1, \ldots, k_{i-1}, k_i, k_{i+1}, \ldots, k_n) & \varepsilon
\end{align*}
\]

**Example 13 Update Game – Selected Steps (Cont’d).** Consider again the above situation. Player \( \Pi \) answers the pebble by showing another parent tuple (now, \( R_P(\vec{g}) \) comes actually into play) of the referencing tuple from which the deletion
cascades, and telling how many steps are required from a founding update:

\[
\begin{align*}
R_P(\overline{y}) | \text{del} & \text{ } R_P(\overline{y}) \langle \bot \rangle | \varepsilon \\
R_C(\overline{F}) \rightarrow R_P.\overline{K} & \text{ NO ACTION} \\
\overline{a}[\overline{F}] = \overline{y}[\overline{K}] \\
R_C(\overline{z}) & \text{ CASCADE} \\
\overline{a}[\overline{F}'] = \overline{y'}[\overline{K}'] \quad & \text{CASCADE}
\end{align*}
\]

The next move by II is probably again a Founding Attack against the new plate.
This is continued until I's answers reach a founding external update (showing only the
right side of the example board): I places the pebble on the update plate at \(R_P(\overline{y}')\),
II replaces it by a “1” and links it to the founding update plate for an external update,
\(\text{del} R_P'(\overline{y}'')\) (which has been put in the initialization).

Feasibility: Referred Parents Needed. Feasibility of an update is not only concerned with the children of the tuple (as handled above by Restrict Attack and No Action Attack), but has also to consider updates of foreign keys, searching for parents.

If I answers with putting an update plate somewhere that changes a foreign key, II can ask him what parent is referenced (note that this is different from asking how the update is founded). I has to answer by showing a parent that provides an appropriate key value (this can either be an unchanged tuple of the original database, or the result of a modification or insertion).

Referencing Attack. For a modification \(R(\overline{x}) | \text{mod} \_ R(M, \overline{x}) (\bot) \ldots\) or insertion \(\varepsilon | \text{ins} \_ R(\overline{x}) (\bot) | R(\overline{x})\) and a ric \(R.\overline{F} \rightarrow R_P.\overline{K}\), Player II can put a reference-wire with a loose end, labelled with the ric, on the result entry, asking “referencing what?”:

\[
\begin{align*}
R_P.\overline{K}(M(\overline{x}))[\overline{F}] & \Rightarrow R.\overline{F} \rightarrow R_P.\overline{K} \\
R_P.\overline{K}(\overline{F}) & \Rightarrow R.\overline{F} \rightarrow R_P.\overline{K} \\
R(\overline{x}) | \text{mod} \_ R(M, \overline{x}) (\bot) | R(M(\overline{x})) & \varepsilon | \text{ins} \_ R(\overline{x}) (\bot) | R(\overline{x})
\end{align*}
\]

Example 14 Update Game: Referencing. A Reference Attack is e.g. played if a tuple is “moved” from one parent to another: Consider relations \(R_C, R_P\) and \(R_P'\) with ric’s \(R_C.(1,2) \rightarrow R_P.(1,2)\) CASCADE and \(R_C.(1,3) \rightarrow R_P'.(1,2)\) NO ACTION where the foreign keys overlap. Let \(D\) contain the tuples \(R_P(a,x), R_P'(a,y), R_P'(b,y), R_C(a,x,y)\). Then, the modification \(\text{mod} \_ R_P'(1/b')[(a,x)]\) cascades to \(\text{mod} \_ R_C([1/b'],(a,x,y))\), resulting in the tuple \(R_C(b,x,y)\) that
now references $R'_p(b, y)$. The game starts with the database and the plate 
$R_p(a, x)|mod_{R_p}[1/b_1; (a, x)](0)|R_p(b, x)$.
After playing Cascade Attack and Cascade Answer, the situation looks as described below. In this situation, Player II – knowing that the reference from $R'_p(a, y)$ breaks – plays Reference Attack for the reference $R_c.(1, 3) \rightarrow R'_p.(1, 2)$:

![Diagram](image)

Player I has then to find a target for the open reference.

Referencing Answer. A dangling reference wire starting in a plate $\ldots \ldots R_c(\bar{x})$ labelled with a ric $R_c.\bar{F} \rightarrow R_p.\bar{K}$ is answered by connecting it to a plate (which also can be positioned in this move on a tuple not holding a plate) such that the result tuple provides the referenced key value $R_p(\bar{y})$, i.e., $\bar{y}[\bar{K}] = \bar{x}[\bar{F}]$:

![Diagram](image)

In case that I puts a new plate, II can again attack its founding and admissibility.

**Example 14 Update Game: Referencing (Cont’d).** In the above situation, I takes the open end, puts an unchanged plate on $R'_p(b, y)$ and connects the open end to its result:

![Diagram](image)

Key Condition. If I answers by placing an update or insert plate whose key value does already exist, II can ask him how to retain the uniqueness of the key. I can answer by deleting or modifying the other tuple.

Key Attack. Player II can put an empty plate $R(\bar{x})[?]$ on a database tuple $R(\bar{x})$ s.t. $\bar{K}$ is a key of $R$ and there exists an insertion or modification plate with a result tuple $R(\bar{y})$ s.t. $\bar{x}[\bar{K}] = \bar{y}[\bar{K}]$ asking “what will happen to this tuple?”.
**Key Answer.** Player I has to replace/fill the empty plate \( \overline{R(x)} \) by a delete or update plate (which then has to change the key value).

**Maximality.** Player II has another way to refute I’s claim that his proposed set is maximal admissible: with the above attacks, only admissibility was checked. In her first move, II can also claim, that \( U \) is not maximal by choosing a subset \( U’ \) s.t. \( U \subseteq U’ \subset U_D \). Then, the roles are changed: I now asks questions in order to prove that \( U’ \) is not admissible (or still not maximal).

**B.1.3 Winning Conditions and Termination.** Some winning conditions have already been mentioned above, e.g., when II traps I by showing a restricting reference, or when I cheats by putting inconsistent plates, or when he has no defending answer. The more intricate situations have to do with the number of steps to justify an update. Since the game uses a “history”, there are no infinite cycles (that lead to drawn positions in the delete-game).

For \( N_0 \cup \{ \bullet \} \) let \( \prec \) be the complete ordering \( \prec N \cup \{(\bullet, n) | n \in N_0 \} \).

For every deletion plate \( P_C = \{ R_C(\overline{x}) \}_{\text{del}} = R_C(\overline{x})(k) \epsilon \) let

\[
k_C := 1 + \min_{k_P} |k_P| \text{ there is a plate } P_P = \{ R_P(\overline{y}) \}_{\text{del}} = R_P(\overline{y})(k_P) | \epsilon \text{ and an action wire labeled with a ric } R_{C,F} \rightarrow R_{P,K} \text{ ON DELETE CASCADE from } P_P \text{ to } C \}.
\]

For every modification plate \( C = \{ R_C(\overline{x}) \}_{\text{mod}} = R_C(\overline{x})(k_1, \ldots, k_n) | \cdots \) let

\[
k_{C_i} := 1 + \min_{l_j} |l_j| \text{ there is a plate } P_P = \{ R_P(\overline{y}) \}_{\text{mod}} = R_P(\overline{y})(l_1, \ldots, l_m) | \cdots \text{ and an action wire labeled with a ric } R_{C,F} \rightarrow R_{P,K} \text{ ON UPDATE OF PARENT CASCADE from } P \text{ to } C, c_i \in \overline{F} \text{ and } p_j \in \overline{K} \}.
\]

A situation is won for Player II if one of the following conditions holds:

— Player I cannot answer.
— Player II shows a restriction (cf. Restrict Attack).
— There are two plates with the same key value.
— Player I “lies”:
  — either, when an action wire connects two plates of different types, or when the numbers of founding steps are inconsistent:
  — there is an action wire connecting two deletion plates \( R_P(\overline{x}) \) \text{del} \( R_P(\overline{x})(k_P) | \epsilon \) and \( R_C(\overline{y}) \) \text{del} \( R_C(\overline{y})(k_C) | \epsilon \) s.t. \( k_C \leq k_P \).
  — an action wire labeled with a ric \( R_{C,F} \rightarrow R_{P,K} \) connects two modification plates \( R_P(\overline{x}) \) \text{mod} \( (M_P, \overline{x}) | (k_P) \cdots \) and \( R_C(\overline{y}) \) \text{mod} \( (M_C, \overline{y}) | (k_C) \cdots \) such that \( M_C \supseteq \overline{F}/M_P(\overline{x}) | \overline{K} \) and \( k_C(\overline{F}(i)) \leq k_P(\overline{K}(i)) (i \in \{1, \ldots, |\overline{F}|\}) \).
  — an action wire labeled with a ric \( R_{C,F} \rightarrow R_{P,K} \) connects two modification plates \( R_P(\overline{x}) \) \text{mod} \( (M_P, \overline{x}) | \omega \cdots \) and \( R_C(\overline{y}) \) \text{mod} \( (M_C, \overline{y}) | \omega \cdots \) s.t. \( M_C \not\supseteq \overline{F}/M_P(\overline{x}) | \overline{K} \).

(Note that this is enforced in a regular game by the rules Cascade Answer and Referencing Answer.)
A no-action wire labeled with a ric $R_C.\vec{F} \rightarrow R_P.\vec{K}$ goes from a parent plate $[R_P(\vec{y})]_{\vec{F}}$ to a child plate $[R_C(\vec{x})]_{\vec{F}} = [\vec{x}']_{\vec{F}}$.

A situation is won for Player I if Player II has no more questions. Obviously, the game is finite since there are only finitely many positions, where a plate can be placed and every plate has only finitely many $\square$ positions.

**Definition B.1.** A starting set $U$ is won for Player I iff he has a winning strategy, i.e., no matter how Player II moves, Player I can win the game.

**B.2 Example**

The following example illustrates the game-theoretic characterisation by playing a complete game for a given situation.

**Example 15 Update Game.** Consider again Example 7 with the database with rac's as given in Figure 2 (using only $R_1, \ldots, R_3$), where the **ON UPDATE** rac is the same as the rac given for **ON DELETE**. Consider the user request $U_{\rightarrow} = \{\triangleright \text{del}_R(a), \triangleright \text{mod}_R(1/c,b)\}$.

The first moves are described in Figure 6: I claims that $U_{\rightarrow}$ is admissible. Thus, the initialization consists of placing the corresponding plates $[R_1(\vec{a})]_{\triangleright \text{del}_R(a) (\square)\varepsilon}$ and $[R_1(\vec{b})]_{\triangleright \text{mod}_R(1/c,b) (\square)\varepsilon}$ at $R_1(\vec{a})$ and $R_1(\vec{b})$, respectively.

Player II challenges first the deletion (targeting to the **NO ACTION** reference from $R_4$ to $R_3$) by **Cascade Attack** and puts an action wire from $[R_1(\vec{a})]_{\triangleright \text{del}_R(a) (\square)\varepsilon}$ to $R_3(\vec{a},y)$. I answers by **Cascade Answer**, putting $[R_3(\vec{a},y)]_{\triangleright \text{del}_R(a,y) (\square)\varepsilon}$ on $R_3(\vec{a},y)$ and adding the action wire from the first plate to the second one.

Now, Player II uses the **NO ACTION** child $R_4(a,x,y)$ for a **No Action Attack** and puts a wire from $[R_3(\vec{a},y)]_{\triangleright \text{del}_R(a,y) (\square)\varepsilon}$ to $R_4(\vec{a},y)$. Player I answers by putting a deletion plate $[R_4(\vec{a},x,y)]_{\triangleright \text{del}_R(a,x,y) (\square)\varepsilon}$ on $R_4(\vec{a},x,y)$ (cf. Figure 6). Continue with Figure 7 with the next moves.

Player II now asks "why can you do this" by **Founding Attack**, placing a pebble on the $\square$ position of the plate that now looks like $[R_4(\vec{a},x,y)]_{\triangleright \text{del}_R(a,x,y) (\bullet)\varepsilon}$. I replaces the pebble by a "2" and puts a $[R_2(\vec{a},x)]_{\triangleright \text{del}_R(a,x) (\bullet)\varepsilon}$ plate on $R_2(\vec{a},x)$ and connects them by a **CASCADE** wire (action from $R_2$ to $R_4$). Again, Player II applies **Founding Attack**, placing a pebble on the $\square$ position of that plate, yielding $[R_2(\vec{a},x)]_{\triangleright \text{del}_R(a,x) (\bullet)\varepsilon}$. Now, I replaces the pebble by a "1" and connects the plate by another action wire from $[R_1(\vec{a})]_{\triangleright \text{del}_R(a) (\square)\varepsilon}$ (that has been placed in the initialization).

As an intermediate result, I has now won the deletion of $R_1(a)$ since II has no more questions. Player I has generated all internal updates that are necessary to execute $\triangleright \text{del}_R(a)$ (cf. Figure 7).

An analogous sequence of moves can e.g. be played for showing that I wins the modification of $R_1(\vec{a})$ for $R_2$, $R_3$, and $R_4$ — but II already knows this so she doesn't play this.
Instead, Player II challenges then the modification of \( R_1(b) \) with a No Action
Attack by putting a wire from \( R_1(b)\mod R_1(1/c, b) (0)\) to \( R_5(b) \). She has not
yet won! Player I can still answer by putting a \( R_5(b)\mod R_5(1/c, b) (\square)\) on
\( R_5(b) \). This move is completely legal. But now, Player II applies Founding Attack,
placing a pebble on the \( \square \) position of that plate. Then, Player I has no answer
(since this update is unfounded) and loses (see Figure 8).

B.3 Equivalence

For a game on given \( D, U_D, \) and RA with a starting set \( U \) which is played until
Player I wins, let

\[
\Lambda := \{ \text{upd} \mid \exists R(\bar{x}) : R(\bar{x})[\text{upd}] \ldots \text{is played} \} \quad \text{and}
\]

\[
D' := \{ R'(\bar{x}') \mid \exists R(\bar{x}) : R(\bar{x})[\\ldots R'(\bar{x}')] \text{is played} \} \cup
\]

\[
\cup \{ R(\bar{x}) \mid R(\bar{x}) \in D \text{ and there is no plate on position } R(\bar{x}) \}.
\]

As mentioned above, the main difference between the game and the logic program-
ing characterization is that only non-subsumed updates are "played", correspond-

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
Fig. 7. Update Game: Situation after I won the "a"-deletion

Fig. 8. Update Game: Excerpt of the situation after I lost the "b"-modification in R5

ing to already guessing a (\(\prec_a\)-maximal) stable model and showing its admissibility.

**Theorem B.2.** If a game is won for Player I, \(A = \Delta(U)\).

**Proof.** The starting situation guarantees that \(U \subset \Lambda\). Completeness is guaran-
ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.
teed by the “cascade” question, and minimality of $\Lambda$ wrt. $U$ is guaranteed by the “founding” question.

**Corollary B.3.** $D' = D \pm \Lambda$ is the database obtained by executing $U$ on $D$.

The final theorem states that the game characterizes exactly the $<_\alpha$-maximal, i.e., maximal admissible subsets:

**Theorem B.4.** A starting set $U \subseteq U_\triangledown$ is won for Player I, iff $U$ is maximal wrt. $U_\triangledown$ and admissible.

**Proof.** For every property, Player $\Pi$ can ask questions:

- Maximality: If $U$ is not maximal, Player $\Pi$ chooses a maximal admissible superset and wins his game.

The above lemma proved that $\Lambda = \Delta(U)$.

- $\Lambda$ satisfies conditions (1) and (2) of feasibility because otherwise Player $\Pi$ would have won by *Restrict Attack* (if she finds a restriction, Player $\Pi$ shows it and wins) or *No Action Attack* (Player $\Pi$ has to show what to do with the child tuple s.t. it no longer references the parent key value).

- $\Lambda$ satisfies conditions (3) and (4) of feasibility because otherwise Player $\Pi$ would have won by “referencing” (Player $\Pi$ has to show which tuple is referenced).

- $\Lambda$ is coherent since Player $\Pi$ can only place one plate on each tuple. $\Lambda$ is key-preserving since otherwise Player $\Pi$ would have won by “key condition”.

-----

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.