Referential Actions as Logical Rules

Bertram Ludäscher
Institut für Informatik
Universität Freiburg
Germany
ludaesch@informatik.uni-freiburg.de

Wolfgang May*
Institut für Informatik
Universität Freiburg
Germany
may@informatik.uni-freiburg.de

Georg Lausen
Institut für Informatik
Universität Freiburg
Germany
lausen@informatik.uni-freiburg.de

Abstract

Referential actions are specialized triggers used to automatically maintain referential integrity. While their local behavior can be grasped easily, it is far from clear what the combined effect of a set of referential actions, i.e., their global semantics should be. For example, different execution orders may lead to ambiguities in determining the final set of updates to be applied. To resolve these problems, we propose an abstract logical framework for rule-based maintenance of referential integrity: First, we identify desirable abstract properties like *admissibility* of updates which lead to a non-constructive global semantics of referential actions. We obtain a constructive definition by formalizing a set of referential actions RA as logical rules, and show that the declarative semantics of the resulting logic program P_{RA} captures the intended abstract semantics: The well-founded model of P_{RA} yields a unique set of updates, which is a safe, sceptical approximation of the set of all maximal admissible updates; the third truth-value *undefined* is assigned to all controversial updates. Finally, we show how to obtain a characterization of all maximal admissible subsets of a given set of updates using certain maximal stable models.

1 Introduction

We study the following problem: Given a relational database D, a set of user-defined update requests $U_{\triangleright}$, and a set of referential actions RA, find those sets of updates Δ which (i) preserve referential integrity in the new database D', (ii) are maximal wrt. $U_{\triangleright}$, and (iii) reflect the intended meaning of RA.

The problem is important both from a practical and theoretical point of view: *Referential integrity constraints* (*ric*'s) are a central concept of the relational database model and frequently used in real world applications. *Referential actions* (*rac*'s) are specialized triggers used to automatically maintain referential integrity [Dat81]. While the *local* behavior of *rac*'s is quite intuitive and easy to understand, it is far from clear what their *global semantics* should be. In particular, different execution orders of *rac*'s may lead to different outcomes, i.e. to *ambiguities* in determining the above Δ and D'.

*Supported by grant no. GRK 184/1-97 of the Deutsche Forschungsgemeinschaft.

Proc. 16th ACM Symposium on Principles of Database Systems (PODS), Tucson, Arizona, 1997, ©ACM Press

Due to their practical importance, *rac*'s have been included in the SQL2 standard and SQL3 proposal [ISO92, ISO94]. However, the standards describe the meaning of *rac*'s in a lengthy and procedural way, making it difficult to understand or predict their global behavior. The problem of ambiguous global semantics is "solved" by fixing a rather ad-hoc run-time execution model [Hor92, CPM96]. In a different approach, [Mar94] presents safeness conditions which aim at avoiding ambiguities at the schema level. However, as shown in [Rei96], it is in general undecidable whether a database schema with *rac*'s is ambiguous. Summarizing, the problem is complex and, from a theoretical point of view, has not been solved in a satisfactory way.

In contrast to previous work, we present an abstract logical framework for rule-based maintenance of referential integrity: After introducing a generic language for *rac*'s, we identify general abstract properties which a set of updates Δ wrt. a given set of *rac*'s RA may possess (Section 3). These abstract properties give rise to a natural but non-constructive global semantics. To obtain a constructive definition, we associate with every set of *rac*'s RA a *logic program* P_{RA} (Section 4), and show that the declarative semantics of P_{RA} captures the abstract semantics (Section 5). This solves the above-mentioned problem in a rigorous and comprehensive way.

Our logical formalization has the following benefits:

- the *local behavior* of an individual *rac* $ra \in RA$ is precisely specified, and can be understood by solely looking at the corresponding rules $P_{ra} \subseteq P_{RA}$,
- the *interaction* between different update requests is precisely defined by certain other rules,
- the *global behavior* is precisely specified and understandable from the declarative semantics:
  - the well-founded model $\mathcal{W}$ of P_{RA} yields a *unique* set of updates Δ, which is a safe, sceptical approximation of the set of all *maximal admissible* Δ's (*safe* means that applying Δ does not lead to violation of *ric*'s, *sceptical* means that all controversial updates have the truth-value *undefined* in $\mathcal{W}$),
  - the maximal admissible Δ's can be obtained as certain stable models of P_{RA}.

2 Preliminaries and Notation

A relation schema consists of a relation name R and a vector of attributes $(A_1, \ldots, A_n)$. W.l.o.g. we identify attribute names A_i of a relation R with the integers between 1 and

n. By $\vec{A} = (i_1, \ldots, i_k)$ we denote a vector of $k \leq n$ distinct attributes (usually $\vec{A}$ will be some key).

Tuples of a relation R are denoted by first-order logic atoms $R(\bar{X})$ where R is an n-ary relation symbol, and $\bar{X} = X_1, \ldots, X_n$ is a vector of variables or constants from the underlying domain. If we want to emphasize that such a vector is ground, i.e., comprises only constants, we write $\bar{x}$ instead of $\bar{X}$. The *projection* of tuples $\bar{X}$ to an attribute vector $\vec{A}$ is denoted by $\bar{X}[\vec{A}]$, if e.g. $\bar{X} = (a, b, c)$, $\vec{A} = (1, 3)$, then $\bar{X}[\vec{A}] = (a, c)$.

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

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

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

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

where $\vec{F}$ is a *foreign key* of the *child relation* R_C, referencing a candidate key $\vec{K}$ of the *parent relation* R_P. Clearly, the arities of $\vec{F}$ and $\vec{K}$ have to coincide. Note that R_C and R_P may be the same relation.

A *ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ is *satisfied* by a given database D, if for every tuple $\bar{x}$ in the child R_C with foreign key values $\bar{x}[\vec{F}]$, there exists a tuple $\bar{y}$ in R_P with matching key value, i.e., $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$. Thus, for a database instance D, a *ric* is satisfied if $D \models \varphi_{ric}$, where φ_{ric} is the first-order sentence

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

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

Update requests (*updates*) to a relation R are represented by auxiliary relations $\mathsf{ins}\_R(\bar{X})$, $\mathsf{del}\_R(\bar{X})$, and $\mathsf{mod}\_R(M, \bar{X})$. Here, M is a list $[a_1/v_1, \ldots, a_n/v_n]$ of pairs prescribing that attribute a_i of $R(\bar{X})$ should be set to the value v_i. As a more abstract notation, $[a_1/v_1, \ldots, a_n/v_n]$ is written as $(a_1, \ldots, a_n)/(v_1, \ldots, v_n)$. For brevity, we sometimes write $\mathsf{mod}\_R(a/d, b, c/e)$ instead of $\mathsf{mod}\_R([1/d, 3/e], a, b, c)$.

Using our list notation, two modifications can be *merged* by simply appending both lists, provided the resulting list assigns at most one value to every attribute.

The *restriction of a modification* M to a key $\vec{K}$ is denoted by $M[\vec{K}]$; the result of *applying a modification* M to a tuple $\bar{X}$ 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 $M(\bar{X}) = (d, b, e)$.

Finally, given a modification $\mathsf{mod}\_R_P(M, \bar{X})$ of a tuple in R_P, $M' = \vec{F}/(M(\bar{X})[\vec{K}])$ denotes a modification which replaces the values of the attributes $\vec{F}$ (of some other relation R_C) with the values of the tuple which results from applying M to $\bar{X}$ and then projecting on $\vec{K}$.

3 Referential Actions

Rule-based approaches to referential integrity maintenance are attractive since they describe how *ric*'s should be enforced using "local repairs": Given a *ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ and an update operation on R_P or R_C, a *rac* defines an operation to be applied to R_C resp. R_P. We call this the *locality principle*.

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

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

Table 1: Operations and Possible Repairs

The updates insert, delete, and modify can be applied to R_P or R_C, leading to six basic cases. It is easy to see from the logical implication in (φ_{ric}) above that insert R_P and delete R_C cannot introduce a violation, while the other four operations can. For these, there are in general three possible actions (cf. Table 1):

- propagate: propagate (cascade) the update from the parent to the child,
- restrict: (i) reject an update on the *parent* if there exists a child referencing the parent in the current database state, or (ii) reject an update on the *child* if the referenced parent does not exist in the current database state,
- wait: similar to restrict, but look at the database state after (hypothetically) applying all updates.

As can be seen from Table 1, not all combinations are meaningful: e.g. it is perfectly reasonable to propagate (cascade) a modification from the parent to the referencing child, but not vice versa.

Syntax. 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:

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

Referential Actions in SQL. In SQL, *rac*'s for a referential integrity constraint $R_C.\vec{F} \rightarrow R_P.\vec{K}$ are specified with the definition of the child table:

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

The correspondence of SQL *rac*'s to the above-mentioned strategies is roughly as follows:

CASCADE ≈ propagate, NO ACTION ≈ wait, and RESTRICT ≈ restrict. The operations insert into R_C and update R_C on the child are evaluated in the current database state, and immediately backed out if they would result in a violation.

Thus, for modifications on child tuples, the SQL behavior is less flexible than our presented formalization. The actions `SET DEFAULT` and `SET NULL` of SQL are also covered by our approach, since these operations can be modeled as special cases of modifications.

3.1 Ambiguities

Since *rac*'s only specify local behavior, there are several types of ambiguities leading to potentially different final states. Given a database D, and a set of user requests $U_\triangleright$, a set of *rac*'s RA is called *ambiguous wrt. D and $U_\triangleright$*, if there are different final states D' (depending on the execution order of referential actions). A database *schema* S with *rac*'s RA is *ambiguous*, if RA is ambiguous wrt. some database D over S, and some $U_\triangleright$. As shown in [Rei96] it is in general undecidable, whether a database schema with referential actions is ambiguous.

The SQL standards [ISO92, ISO94] solve the problem of ambiguity of *rac*'s by fixing a certain run-time execution model and a marking algorithm as described in [Hor92, CPM96]. In case a set of updates causes referential problems, the transaction is simply aborted without giving further information, e.g. which tuples or updates caused the problems. Often, although not all requested updates can be accomplished, it is still possible to execute some of them while postponing the others. Thus, the information which tuple or update really caused problems is valuable for preparing a revised request which realizes the intended changes *and* is accepted by the system. In Section 5, 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 set of all maximal solutions is characterized by certain stable models.

Example 1 (Diamond) *Consider the database with rac's as depicted in Figure 1. Solid arcs represent ric's and point from R_C to R_P, rac's are denoted by dashed (propagate) or double (restrict) arcs. Let $U_\triangleright = \{\mathsf{del}\_R_1(a)\}$ be a user request to delete the tuple $R_1(a)$. Depending on the order of execution of rac's, one of two different final states may be reached:*

1. *If execution follows the path R_1–R_3–R_4, the tuple $R_3(a,c)$ 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 $R_1(a)$. Consequently, the user request $\mathsf{del}\_R_1(a)$ is rejected, and the database state remains unchanged, i.e. $D' = D$.*
2. *If execution follows the path R_1–R_2–R_4, the tuple $R_2(a,b)$ and —as a consequence— $R_4(a,b,c)$ are requested for deletion. Hence, the rac for $R_4.(1,3) \rightarrow R_3.(1,2)$ can assume that $R_4(a,b,c)$ is deleted, thus no referencing tuple exists in R_4. Therefore, all deletions can be executed, resulting in a new database state $D' \neq D$.*

We argue that the second execution order is preferable to the first, since it accomplishes the desired user request without violating referential integrity. Here, the ambiguity arises since the restrict *rac* considers the current database state, which makes the outcome dependent 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. However, the situation is more complex for *rac*'s of type wait which have to look at the *final* database state. As it turns out, in the presence of modifications, there are in general several "equally justified" final states, each of which has to be considered:

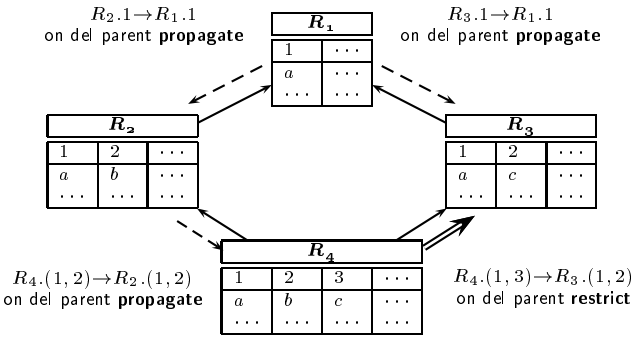

Figure 1: Database with Referential Actions

Example 2 (Mutex) *Consider modifications $\mathsf{mod}\_R(a/b)$ and $\mathsf{mod}\_R(a/c)$. 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.*

The final type of ambiguity may arise due to "self-contradictory" requests:

Example 3 (Self-Attack) *Imagine a database with rac's such that $\mathsf{mod}\_R_1(a/b, a/c)$ triggers both $\mathsf{mod}\_R_2(a/b)$ and $\mathsf{mod}\_R_3(a/c)$. Then, $\mathsf{mod}\_R_2(a/b)$ triggers $\mathsf{mod}\_R_4(a/b)$, and $\mathsf{mod}\_R_3(a/c)$ triggers $\mathsf{mod}\_R_4(a/c)$. Since executing the original request $\mathsf{mod}\_R_1(a/b, a/c)$ causes a conflict at R_4, 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.*

3.2 Abstract Semantics

Let RA be a set of rac's, D a database instance, and $U_\triangleright$ a set of update requests given by the user. For an arbitrary set Δ of updates, we define several abstract properties Δ may have wrt. RA, D and $U_\triangleright$. These allow to define the intended meaning of a set of rac's in an abstract (and non-constructive) way. $D' = D \pm \Delta$ denotes the database obtained by applying Δ to D. We confine ourselves to a semi-formal definition; technical details can be found in [LML96].

Definition 1 (Abstract Properties) *A* single update *is called* founded (in n steps) *wrt. given RA, D, $U_\triangleright$, and Δ, if it can be justified by the user requests and propagations:*

- *A deletion $\mathsf{del}\_R(\bar{x})$ is* founded in n steps, *if there is $\mathsf{del}\_R(\bar{x}) \in U_\triangleright$ or there is a deletion $\mathsf{del}\_R_i(\bar{x}_i) \in \Delta$ which is founded in $< n$ steps, and a rac $R.\vec{F_i} \rightarrow R_i.\vec{K_i}$* on del parent propagate *s.t. $\bar{x}[\vec{F_i}] = \bar{x}_i[\vec{K_i}]$.*
- *A modification $\mathsf{mod}\_R(M, \bar{x})$ is* founded in n steps, *if there are modifications $M_1, \ldots, M_k$ s.t. $M = \bigcup_{i=1..k} M_i$ (not necessarily disjoint) and for every i, $\mathsf{mod}\_R(M_i, \bar{x}) \in U_\triangleright$, or $\mathsf{mod}\_R(M_i, \bar{x})$ results from propagating a modification, i.e. there is a modification $\mathsf{mod}\_R_i(M_i', \bar{x}_i) \in \Delta$ which is founded in $< n$ steps, and a rac $R.\vec{F_i} \rightarrow R_i.\vec{K_i}$*

on mod parent propagate *such that* $\bar{x}_i[\vec{K}_i] \neq M_i'(\bar{x}_i)[\vec{K}_i]$, $\bar{x}[\vec{F}_i] = \bar{x}_i[\vec{K}_i]$, *and* $M_i = \vec{F}_i/(M_i'(\bar{x}_i)[\vec{K}_i])$.

- *An insertion* ins_$R(\bar{x})$ *is* founded *if* ins_$R(\bar{x}) \in U_\triangleright$.

Given RA, D, *and* $U_\triangleright$, a set Δ of updates *is called*

- founded wrt. RA, D, and $U_\triangleright$ *if every update* del_$R(\bar{x})$, mod_$R(M, \bar{x})$, *or* ins_$R(\bar{x}) \in \Delta$ *is founded wrt.* RA, D, $U_\triangleright$, *and* Δ.
- complete wrt. RA and D *if it is closed wrt. propagations, i.e., it satisfies the following conditions:*
  – *if* del_$R_P(\bar{y}) \in \Delta$, $R_C(\bar{x}) \in D$, $R_C.\vec{F} \to R_P.\vec{K}$ on del parent propagate $\in RA$, *and* $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$ *then* del_$R_C(\bar{x}) \in \Delta$.
  – *if* mod_$R_P(M, \bar{y}) \in \Delta$, $R_C(\bar{x}) \in D$, $R_C.\vec{F} \to R_P.\vec{K}$ on mod parent propagate $\in RA$, $\bar{y}[\vec{K}] \neq M(\bar{y})[\vec{K}]$, *and* $\bar{x}[\vec{F}] = \bar{y}[\vec{K}]$ *then there is a* M' *s.t.* mod_$R_C(M', \bar{x}) \in \Delta$ *and* $M' \supseteq \vec{F}/(M(\bar{y})[\vec{K}])$.
- feasible *if every* rac *of the form* $R_C.\vec{F} \to R_P.\vec{K}$ *on* ... $\{$*restrict* | *wait*$\}$ *is "obeyed" by* Δ, *i.e.*
  – *if* $R_C.\vec{F} \to R_P.\vec{K}$ on del parent restrict *is in* RA *and for a tuple* $R_P(\bar{y})$ *there is a referencing child* $R_C(\bar{x}) \in D$, *then* $R_P(\bar{y})$ *is* not *deleted by* Δ;
  – *if* $R_C.\vec{F} \to R_P.\vec{K}$ on del parent wait *is in* RA *and* del_$R_P(\bar{x}) \in \Delta$, *then all children* $R_C(\bar{y})$ *referencing* $R_P(\bar{x})$ *are deleted or "modified-away" by some updates in* Δ;
  – *if* $R_C.\vec{F} \to R_P.\vec{K}$ on ins child restrict *is in* RA *and* ins_$R_C(\bar{x}) \in \Delta$, *then a referencable parent* $R_P(\bar{y})$ *exists in* D *and is neither deleted nor modified-away by* Δ;
  – *if* $R_C.\vec{F} \to R_P.\vec{K}$ on ins child wait *is in* RA *and* ins_$R_C(\bar{x}) \in \Delta$, *then a referencable parent* $R_P(\bar{y})$ *exists in the new database state* D'.
  (similar for mod_$R(M, \bar{x})$*)*
- coherent *if no contradicting updates are issued on the same tuple, i.e. if* $upd =$ del_$R(\bar{x}) \in \Delta$, *then* ins_$R(\bar{x}) \notin \Delta$, *and there is no* M *s.t.* mod_$R(M, \bar{x}) \in \Delta$; *similar for other updates* upd. *Note that if* Δ *is coherent,* $D' = D \pm \Delta$ *is well-defined.*
- key-preserving *if in* $D' = D \pm \Delta$ *all key dependencies are satisfied.*
- admissible *if* Δ *is founded, complete, feasible, coherent, and key-preserving.*

These abstract properties are used to formalize our *intended semantics*:

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

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

This semantics reflects the intended behavior of the database system, i.e., it does neither "invent" nor "forget" updates and guarantees referential integrity:

Theorem 1 (Adequacy)

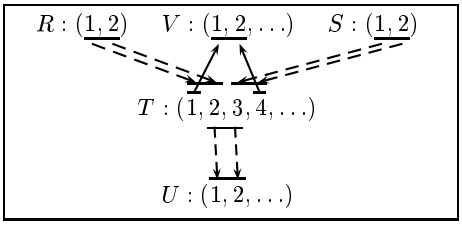

Figure 2: Database Schema with Overlapping Keys

1. *If* $U \subseteq U_\triangleright$, *then for every* RA *and* D, $\Delta(U)$ *is founded and complete.*
2. *If a coherent* Δ *is complete and feasible, then* $D' = D \pm \Delta(U)$ *satisfies all* ric*'s in* RA.

PROOF:

1. $\Delta(U)$ is defined as the *least* complete set. It follows that $\Delta(U)$ is founded.
2. Since Δ is complete, all updates propagated by RA are contained in Δ. Feasibility of Δ guarantees that no $upd \in \Delta$ is restricted, and all *rac*'s which are maintained by on ... wait are satisfied in D'. ∎

The abstract semantics specifies the notions of maximal admissible sets U and induced updates $\Delta(U)$, but provides no direct method how to compute them: Given a set of n user requests, there are 2^n subsets which may be admissible. 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 "naive" way [LMR96], in the presence of modifications, simultaneous updates have to be taken into account. This can lead to an exponential number of rules describing how modifications have to be propagated (see (CH) in Appendix A and Theorem 2 which describes how $\Delta(U)$ can be computed).

Finally, considering the effect of *rac*'s in isolation as suggested by the locality principle (Section 3) is not sufficient if the admissible subsets of $U_\triangleright$ are unknown:

Example 4 *Consider the database schema depicted in Figure 2. Among others there are* rac*'s of type* on mod parent propagate *for the* ric*'s* $T.(1,2) \to R.(1,2)$, $T.(3,4) \to S.(1,2)$, $U.(1,2) \to T.(2,3)$, *and a* rac $T.(1,4) \to V.(1,2)$ on mod child restrict.

a) *Assume* D *contains* $R(a,b)$, $S(c,d)$, $T(a,b,c,d,\ldots)$, $U(b,c,\ldots)$, *and* $V(a,d,\ldots)$, $V(a',d,\ldots)$, $V(a,d',\ldots)$. *For given* mod_$R(a/a', b/b')$ *and* mod_$S(c/c', d/d')$, *the* rac*'s trigger* mod_$T(a/a', b/b', c, d, \ldots)$ *and* mod_$T(a, b, c/c', d/d', \ldots)$. *Since these updates to* T *are coherent, they can be merged, resulting in* mod_$T(a/a', b/b', c/c', d/d', \ldots)$, *which then triggers* mod_$U(b/b', c/c', \ldots)$.

On the other hand, the rac $T.(1,4) \to V.(1,2)$ on mod child restrict *restricts this modification since there is no tuple* $V(a', d', \ldots)$. *So each of the updates is admissible, but they are not admissible together, even though they do not contradict each other directly.*

b) *Assume now, that the situation is the same as in (a), except that* $V = \{(a,d,\ldots),(a',d',\ldots)\}$. *Then, neither* mod_$R(a/a', b/b')$ *nor* mod_$S(c/c', d/d')$ *is admissible in isolation: the triggered updates* mod_$T(a/a', b/b', c, d, \ldots)$ *resp.* mod_$T(a, b, c/c', d/d', \ldots)$ *are both blocked since* V *contains*

neither $(a', d, \ldots)$ *nor* $(a, d', \ldots)$. *On the other hand, simultaneous execution of both updates is allowed: the triggered updates are merged to* $\mathsf{mod\_}T(a/a', b/b', c/c', d/d', \ldots)$ *which is allowed. This shows that the merge of modifications is an important concept for dealing with simultaneous updates.*

Note that in both cases, it is completely irrelevant, which modifications are raised on the dotted parts of the tuples.

Example 4 illustrates some of the problems which may arise due to overlapping foreign keys and candidate keys, and gives a first impression of the inherent complexity of rule-based referential integrity maintenance. We suspect that these problems are the reason that commercial database systems do not (yet) provide means to propagate modifications.

We argue that the propagation of updates should be handled *key-oriented* and cannot be seen tuple-oriented or attribute-oriented, since keys play the central role in the concept of referential integrity. Our claim is supported by the observations made in Example 4a):

An attribute-oriented approach would be too fine: Both $\mathsf{mod\_}T(a/a', b/b', c, d, \ldots)$ and $\mathsf{mod\_}T(a, b, c/c', d/d', \ldots)$ are allowed in isolation, but their combination is forbidden due to the fact that the foreign key $T.(1, 4)$ has to match the parent key $V.(1, 2)$.

On the other hand, a tuple-oriented view is too coarse since then, the two updates $\mathsf{mod\_}T(a/a', b/b', c, d, \ldots)$ and $\mathsf{mod\_}T(a, b, c/c', d/d', \ldots)$ would be merged into a single modification of $T(a, b, c, d, \ldots)$, neglecting the fact that they can also be carried out independently.

Furthermore, a key-oriented approach allows to model the connection between modifications of parent keys and the corresponding foreign keys in a very natural way, which is not the case for an attribute-oriented or a tuple-oriented approach. In our framework, keys are regarded as the atomic units to be considered for modifications. Not surprisingly, parent keys, foreign keys, propagated modifications, references, and overlapping keys play an important role in our logical formalization.

4 Logical Formalization

The meaning of a set RA of *rac*'s is formalized as a logic program P_{RA}, consisting of the sets P_{ra} which specify the local behavior of every *rac* ra, and a set of rules specifying the meaning of interacting update requests.

Here, we only show some rules embodying the main ideas, i.e., the handling of deletions and some aspects of modifications. The remaining rules for handling references, modifications of child tuples, insertions, interferences between updates, coherence, and key-preservation are listed in the Appendix.

Recall that an update request *upd* can be any of $\mathsf{ins\_}R(\bar{X})$, $\mathsf{del\_}R(\bar{X})$, $\mathsf{mod\_}R(M, \bar{X})$. $U_{\triangleright}$ is given as a set of facts of the form $\triangleright upd$. For each update type *upd*, pot_*upd* holds all *potential updates*, i.e. those which are founded by RA and $U_{\triangleright}$. blk_*upd* $\subseteq$ pot_*upd* holds all *blocked updates*, i.e. those which cannot be executed due to some interfering constraints.

User Requests. The handling of user requests incorporates the selection of admissible update sets: every user request raises an update to the database if it is not blocked:

$$\begin{array}{l}\mathsf{pot\_del\_}R(\bar{X}) \leftarrow \ \triangleright\mathsf{del\_}R(\bar{X}).\\ \mathsf{del\_}R(\bar{X}) \leftarrow \ \triangleright\mathsf{del\_}R(\bar{X}), \neg\, \mathsf{blk\_del\_}R(\bar{X}).\end{array} \qquad (EXT_1)$$

Analogous rules are used for $\mathsf{ins\_}R(\bar{X})$ and $\mathsf{mod\_}R(M, \bar{X})$. Additionally, modifications are decomposed into their effects on keys. For every candidate or foreign key $R.\vec{A}$:[1]

$$\begin{array}{l}\mathsf{pot\_mod\_}{\triangleright\!\!\rightsquigarrow}R.\vec{A}(M, \bar{X}) \leftarrow\\ \quad \triangleright\mathsf{mod\_}R(M', \bar{X}),\ \bar{X}[\vec{A}] \neq M'(\bar{X})[\vec{A}],\ M = M'[\vec{A}]\ .\\ \mathsf{mod\_}{\triangleright\!\!\rightsquigarrow}R.\vec{A}(M, \bar{X}) \leftarrow\\ \quad \triangleright\mathsf{mod\_}R(M', \bar{X}),\ \bar{X}[\vec{A}] \neq M'(\bar{X})[\vec{A}],\ M = M'[\vec{A}]\ ,\\ \quad \neg\, \mathsf{blk\_mod\_}R(M', \bar{X})\ .\end{array} \qquad (EXT_2)$$

Deletions. Recall that we only need to consider *rac*'s of the form $R_C.\vec{F} \rightarrow R_P.\vec{K}$ on del parent ... (see Table 1). Logical rules are generated for these *rac*'s as follows (cf. Table 2):

- on del parent propagate: Deletions of parent tuples are propagated downwards to every child tuple by rule (DP_1). Additionally, blockings are propagated upwards: if the deletion of a child tuple is blocked, the deletion of the parent tuple is also blocked (DP_2).
- on del parent restrict: The deletion of a parent tuple is blocked, if there is a referencing child tuple (DR). Here, $\mathsf{is\_ref'd\_}R_P.\vec{K}\mathsf{\_by\_}R_C.\vec{F}(\bar{v})$ holds, if in D, the key value $R_P.\vec{K}(\bar{v})$ appears as foreign key value of $\vec{F}$ in some tuple $R_C(\bar{y})$.
- on del parent wait: 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) (DW). $\mathsf{rem\_ref'd\_}R_P.\vec{K}\mathsf{\_by\_}R_C.\vec{F}(\bar{v})$ specifies that there is a reference to the key value $R_P.\vec{K}(\bar{v})$ by some tuple $R_C(\bar{x})$ s.t. $\bar{x}[\vec{F}]$ does not change between D and D'.

Modifications of Parent Tuples. The handling of modifications follows the same principle as presented for deletions, but since modifications are handled key-oriented, the details are more involved (cf. Table 3).

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. 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. Imagine a modification $\mathsf{mod\_}R_P(M_P, \bar{y})$ and a *rac* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ on mod parent propagate s.t. the key value $R_P.\vec{K}$ of $R_P(\bar{y})$ changes, denoted by $\mathsf{chg\_}R_P.\vec{K}(M_P, \bar{y})$. Then, for every referencing child $R_C(\bar{x})$, this modification is raised for the corresponding foreign key, i.e. $M_C = \vec{F}/(M_P(\bar{y})[\vec{K}])$. This is stored in $\mathsf{mod\_}R_P.\vec{K}{\rightsquigarrow}R_C.\vec{F}(M_C, \bar{x})$.

- on mod parent propagate: Changes of parent keys are propagated downwards to foreign keys (MPP_1). If the resulting modification of the foreign key of some child is blocked, the change of the parent key is also blocked (MPP_2).
- on mod parent restrict: The change of the parent key $R_P.\vec{K}$ is blocked, if there is a referencing child in the original database D (MPR).

[1] As a mnemonic aid, we encode some hints on the meaning of auxiliary relations into relation names. Therefore, relation names may contain unusual characters like "$\rightsquigarrow$", "$\triangleright$", etc.

$$\mathsf{del\_}R_C(\bar{X}) \leftarrow \mathsf{del\_}R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]. \qquad (DP_1)$$
$$\mathsf{pot\_del\_}R_C(\bar{X}) \leftarrow \mathsf{pot\_del\_}R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}].$$
$$\mathsf{blk\_del\_}R_P(\bar{Y}) \leftarrow \mathsf{pot\_del\_}R_P(\bar{Y}),\ \mathsf{blk\_del\_}R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]\ . \qquad (DP_2)$$
$$\mathsf{blk\_del\_}R_P(\bar{Y}) \leftarrow \mathsf{pot\_del\_}R_P(\bar{Y}),\ \mathsf{is\_ref'd\_}R_P.\vec{K}\mathsf{\_by\_}R_C.\vec{F}(\bar{Y}[\vec{K}])\ . \qquad (DR)$$
$$\mathsf{blk\_del\_}R_P(\bar{Y}) \leftarrow \mathsf{pot\_del\_}R_P(\bar{Y}),\ \mathsf{rem\_ref'd\_}R_P.\vec{K}\mathsf{\_by\_}R_C.\vec{F}(\bar{Y}[\vec{K}])\ . \qquad (DW)$$

Table 2: Local Rules for Deletions

- on mod parent wait: The change of the parent key $R_P.\vec{K}$ is blocked, if there is a referencing child which is neither requested for deletion nor modified away (MPW).

5 Declarative Semantics and Formal Results

In this section, we show how the well-founded model and certain stable models of P_{RA} are related to the abstract semantics presented in Section 3.2. Note that P_{RA} contains non-stratified negation due to possible negative cyclic dependencies between updates (see e.g. the rules (ABC) in Table 7). In contrast, computing the set $\Delta(U)$ of updates induced by a given set of updates U can be accomplished using a negation-free set of rules:

Let P_{pot} be the subset of $EXT_1 \cup EXT_2 \cup DP_1 \cup MPP_1 \cup CH$ consisting of all rules where the head is of the form pot_.... P_{pot} models the propagation of changes (but not the propagation of blockings). Note that P_{pot} is positive and has a unique minimal model $\mathcal{M}_{P_{pot}}$. Since $EXT_{1/2}$ guarantee that all user requests are considered, and DP_1, MPP_1, and CH guarantee completeness wrt. deletions and modifications, we have the following result:

Theorem 2 *For every database D, and every set $U_\triangleright$ of external updates:*

$$\Delta(U_\triangleright) = \{upd \mid \mathcal{M}_{P_{pot}}(D \cup U_\triangleright) \models \mathsf{pot\_}upd\}\ .$$

The examples in Section 3.1 illustrate different types of ambiguities which can occur for a set of *rac*'s RA. These ambiguities become apparent by the declarative semantics of P_{RA}:

Given the logical formalization P_{RA} of a set of *rac*'s RA, a database D, and a set of user requests $U_\triangleright$, the well-founded model $\mathcal{W}(P_{RA}, D, U_\triangleright)$ assigns truth-values true and false to all uncontroversial update requests, i.e., which are true or false under any "well-behaved"[2] semantics of P_{RA}. The atoms which are undefined in $\mathcal{W}$ are controversial due to some kind of ambiguity (cf. Section 3.1):

Diamond: Consider Example 1 and the "diamond" in Figure 1. Assume the *rac* $R_4.(1,3) \rightarrow R_3.(1,2)$ on del parent restrict is replaced by $R_4.(1,3) \rightarrow R_3.(1,2)$ on del parent wait. Then the rules of P_{RA} define that the deletion of $R_1(a)$ is blocked (via R_4–R_3–R_1) if $R_4(a,b,c)$ cannot be deleted. $R_4(a,b,c)$ can be deleted (via R_1–R_2–R_4) if the deletion of $R_1(a)$ is not blocked. Hence there is a negative cycle of the form

$$block \leftarrow \neg\, exec. \qquad exec \leftarrow \neg\, block.$$

thus, either setting all requests in the diamond to true or to false will result in a stable model.

[2] Dix [Dix95] formally defines this notion using certain abstract properties of semantics.

Mutex: For two mutually exclusive operations (cf. Example 2), if one of them is rejected, the other can be executed: Here, some requests which are undefined in $\mathcal{W}$ can be set to false, resulting in other undefined requests to be set to true such that eventually, a stable model is obtained. This situation is analogous to the program:

$$block_1 \leftarrow exec_2. \qquad block_2 \leftarrow exec_1.$$
$$exec_1 \leftarrow \neg\, block_1. \qquad exec_2 \leftarrow \neg\, block_2.$$

Self-Attack: For a self-attacking request (cf. Example 3), there is no other support for rejecting it than its "internal contradiction". Therefore, neither assigning true nor false to such a request will yield a stable model. This situation corresponds to

$$exec \leftarrow \neg\, block. \qquad block \leftarrow exec.$$

where no total stable model exists.

Every 3-valued model $\mathcal{M}(P_{RA}, D, U_\triangleright)$ defines sets of updates $\Delta_\mathcal{M}$ and user requests $U_\mathcal{M} \subseteq U_\triangleright$ which are true ($\boldsymbol{t}$), false ($\boldsymbol{f}$) or undefined ($\boldsymbol{u}$) in $\mathcal{M}$. Let upd be any of $\mathsf{ins\_}R(\bar{x})$, $\mathsf{del\_}R(\bar{x})$, $\mathsf{mod\_}R(M,\bar{x})$. Then:

$$\Delta^{\boldsymbol{t}}_\mathcal{M} := \{upd \mid \mathcal{M}(upd) = \boldsymbol{t}\}\ ,\ \text{and} \quad U^{\boldsymbol{t}}_\mathcal{M} := \Delta^{\boldsymbol{t}}_\mathcal{M} \cap U_\triangleright\ .$$

$\Delta^{\boldsymbol{f}}_\mathcal{M}$, $U^{\boldsymbol{f}}_\mathcal{M}$, and $\Delta^{\boldsymbol{u}}_\mathcal{M}$, $U^{\boldsymbol{u}}_\mathcal{M}$ are defined analogously.

The well-founded model $\mathcal{W}(P_{RA}, D, U_\triangleright)$ provides a safe and sceptical semantics which is computable in polynomial time. Here, sceptical means that all controversial updates are assigned the truth-value *undefined*.

By safe, we mean that updates which are true in $\mathcal{W}$ can be executed without violating referential integrity. More precisely, the set $\Delta(U^{\boldsymbol{t}}_\mathcal{W})$ of updates induced by $U^{\boldsymbol{t}}_\mathcal{W}$ is admissible and equal to $\Delta^{\boldsymbol{t}}_\mathcal{W}$; submitting $U^{\boldsymbol{t}}_\mathcal{W}$ results in the new database $D' = D \pm \Delta^{\boldsymbol{t}}_\mathcal{W}$:

Theorem 3 (Correctness: Well-Founded Semantics) *Let $\mathcal{W}$ be the well-founded model of $P_{RA} \cup D \cup U_\triangleright$. Then:*

i) $\Delta^{\boldsymbol{t}}_\mathcal{W}$ is admissible,
ii) $\Delta^{\boldsymbol{t}}_\mathcal{W} = \Delta(U^{\boldsymbol{t}}_\mathcal{W})$,
iii) $U^{\boldsymbol{t}}_\mathcal{W}$ is admissible.

Proof: (Sketch)

i) 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.
ii) $\Delta^{\boldsymbol{t}}_\mathcal{W} \subseteq \Delta(U^{\boldsymbol{t}}_\mathcal{W})$ follows from foundedness, $\Delta^{\boldsymbol{t}}_\mathcal{W} \supseteq \Delta(U^{\boldsymbol{t}}_\mathcal{W})$ from completeness.
iii) follows from (i), (ii), and Definition 2. ∎

The relation between the well-founded model and maximal admissible sets will be investigated in Theorem 6.

The different types of undefined update requests $upd \in U^{\boldsymbol{u}}_\mathcal{W}$ can be characterized according to the different types of controversial atoms:

$$\mathsf{mod\_}R_P.\vec{K}{\rightsquigarrow}R_C.\vec{F}(M_C,\bar{X}) \leftarrow \mathsf{chg\_}R_P.\vec{K}(M_P,\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}]=\bar{Y}[\vec{K}],\ M_C=\vec{F}/(M_P(\bar{Y})[\vec{K}])\ . \quad (MPP_1)$$

$$\mathsf{pot\_mod\_}R_P.\vec{K}{\rightsquigarrow}R_C.\vec{F}(M_C,\bar{X}) \leftarrow \mathsf{pot\_chg\_}R_P.\vec{K}(M_P,\bar{Y}),\ R_C(\bar{X}),\ \bar{X}[\vec{F}]=\bar{Y}[\vec{K}],\ M_C=\vec{F}/(M_P(\bar{Y})[\vec{K}])\ . \quad (MPP_1)$$

$$\mathsf{blk\_chg\_}R_P.\vec{K}(M_P,\bar{Y}) \leftarrow \mathsf{pot\_chg\_}R_P.\vec{K}(M_P,\bar{Y}),\ \mathsf{blk\_mod\_}R_P.\vec{K}{\rightsquigarrow}R_C.\vec{F}(M_C,\bar{X}),\ \bar{X}[\vec{F}]=\bar{Y}[\vec{K}],\ M_C=\vec{F}/(M_P(\bar{Y})[\vec{K}])\ . \quad (MPP_2)$$

$$\mathsf{blk\_chg\_}R_P.\vec{K}(M_P,\bar{Y}) \leftarrow \mathsf{pot\_chg\_}R_P.\vec{K}(M_P,\bar{Y}),\ \mathsf{is\_ref'd\_}R_P.\vec{K}\mathsf{\_by\_}R_C.\vec{F}(\bar{Y}[\vec{K}])\ . \quad (MPR)$$

$$\mathsf{blk\_chg\_}R_P.\vec{K}(M_P,\bar{Y}) \leftarrow \mathsf{pot\_chg\_}R_P.\vec{K}(M_P,\bar{Y}),\ \mathsf{rem\_ref'd\_}R_P.\vec{K}\mathsf{\_by\_}R_C.\vec{F}(\bar{Y}[\vec{K}])\ . \quad (MPW)$$

Table 3: Local Rules for Modifications

- $upd \in U$ for every maximal admissible $U \subseteq U_\triangleright$ ("diamond"), or
- there are maximal admissible sets $U, U' \subseteq U_\triangleright$ s.t. $upd \in U$ and $upd \notin U'$ ("mutex"), or
- $upd \notin U$ for any admissible $U \subseteq U_\triangleright$ ("self-attack").

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

Definition 3 (P-, M-Stable Models) *[ELS96] Let $I = \langle I^t, I^f\rangle$ be a 3-valued interpretation. The reduction P/I of a ground instantiated logic program P is obtained by replacing every negative literal in P by its truth-value wrt. I. Thus, P/I is positive and has a unique minimal (wrt. the truth-order $\boldsymbol{f} <_t \boldsymbol{u} <_t \boldsymbol{t}$) 3-valued model $\mathcal{M}_{P/I}$.*

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

In contrast to the well-founded model which is the "most sceptical" P-stable model, M-stable models are "more brave" and handle mutually exclusive requests as expected; in particular, *all* maximal admissible solutions are represented by the set of P-stable models. This fact, and the generalization of Theorem 3 is expressed by

Theorem 4 (Correctness and Completeness: Stable Semantics)

- *For every P-stable model $\mathcal{PS}$ of $P_{RA} \cup D \cup U_\triangleright$:*
  - *i) $\Delta^t_{\mathcal{PS}}$ is admissible,*
  - *ii) $\Delta^t_{\mathcal{PS}} = \Delta(U^t_{\mathcal{PS}})$,*
  - *iii) $U^t_{\mathcal{PS}}$ is admissible.*
- *For every maximal admissible $U \subseteq U_\triangleright$, there is an M-stable model $\mathcal{MS}$ s.t. $U = U^t_{\mathcal{MS}}$ and $\Delta(U) = \Delta^t_{\mathcal{MS}}$.*

Proof: (Sketch) The first part is proven analogously as in the proof of Theorem 3. The second part follows from the definition of M-stable. ∎

Theorem 4 implies the following logical characterization of admissible subsets of user requests:

Corollary 5 *A set $U_\triangleright$ of user requests is admissible iff there is a P-stable model $\mathcal{PS}$ of $P_{RA} \cup D \cup U_\triangleright$ s.t. $U_\triangleright = U^t_{\mathcal{PS}}$. Then $\Delta(U_\triangleright) = \Delta^t_{\mathcal{PS}}$, and submitting $U_\triangleright$ results in the new database $D' = D \pm \Delta^t_{\mathcal{PS}}$.*

The following theorem states that the well-founded model represents the "least common denominator" of all maximal solutions:

Theorem 6 *Every maximal admissible $U \subseteq U_\triangleright$ extends $U^t_{\mathcal{W}}$, and updates classified as false by $\mathcal{W}(P_{RA}, D, U_\triangleright)$ are not contained in any admissible set:*

i) If U is maximal admissible, then $U^t_{\mathcal{W}} \subseteq U$.

ii) $U^f_{\mathcal{W}} \subseteq U_\triangleright \setminus U$.

Proof: (Sketch)

i) By Theorem 4, there is an M-stable model for every maximal admissible set. Since every M-stable model extends the well-founded model, every $upd \in U^t_{\mathcal{W}}$ is true in every M-stable model.

ii) Given an update $upd \in U^f_{\mathcal{W}}$, for every P-stable model $\mathcal{PS}$ of $P_{RA} \cup D \cup U_\triangleright$, $upd \in U^f_{\mathcal{PS}}$ since every P-stable model extends the well-founded model. Together with Theorem 4 this implies that upd is not contained in any (maximal) admissible set. ∎

M-stable models of P_{RA} almost capture the notion of "optimal" (maximal admissible) solutions. The only exception is that in case of a "diamond" $\{block \leftarrow \neg\, exec,\ exec \leftarrow \neg\, block\}$ there are two M-stable models:

Example 5 *Recall Example 1. For $U_\triangleright = \{\triangleright del\_R_1(a)\}$, both*

$\mathcal{M}_1 = \{blk\_del\_R_1(a), blk\_del\_...(...), pot\_del\_..(...),$
$rem\_ref'd\_R_1.1\_by\_R_2.1(a),$
$rem\_ref'd\_R_1.1\_by\_R_3.1(a),$
$rem\_ref'd\_R_2.(1,2)\_by\_R_4.(1,2)(a,b),$
$rem\_ref'd\_R_3.(1,2)\_by\_R_4.(1,3)(a,c), \ldots\}$, *and*

$\mathcal{M}_2 = \{del\_R_1(a), del\_R_2(a,b), del\_R_3(a,c), del\_R_4(a,b,c),$
$pot\_del\_\ldots(\ldots), \ldots\}$

(where only the true atoms of $\mathcal{M}_1$ and $\mathcal{M}_2$ are sketched) are total and M-stable.

However, executing an update should be preferred to blocking it in order to capture the notion of maximal admissibility. Therefore, we define an ordering $<_a$ on P-stable models which reflects this "application-specific" preference.

$$\mathcal{PS}_1 <_a \mathcal{PS}_2 \ :\Leftrightarrow\ U^t_{\mathcal{PS}_1} \subset U^t_{\mathcal{PS}_2}.$$

Finally, our main result can be stated. The maximal stable models wrt. $<_a$ represent exactly the maximal admissible sets:

Theorem 7 (Maximality) *The set of all maximal admissible sets $U \subseteq U_\triangleright$, and the set of all $U^t_{\mathcal{AS}}$ s.t. $\mathcal{AS}$ is an M-stable model of $P_{RA} \cup D \cup U_\triangleright$ which is maximal wrt. $<_a$ coincide.*

6 Conclusion

By formalizing referential actions as logical rules and exploiting the power of declarative semantics, we have solved the problem given in the introduction in a rigorous and comprehensive way. In [LMR96] we presented preliminary steps towards a logical semantics of referential actions in SQL. However, the complex case of modifications was not considered, and no abstract, SQL-independent semantics was given.

Production rules have recently been reconsidered, since they seem well-suited as a language for *active rules*. Therefore, referential actions – which are specialized active rules – can also be formalized by production rules, e.g. in the style of [AV91, PV95]. However, by axiomatizing referential actions as a logic program P and employing a declarative semantics, the resulting set of updates can be "justified" and explained in a more intuitive way using the rules of P. This is due to the fact that declarative semantics like the well-founded or stable semantics treat negative cyclic dependencies (which occur from inherent interdependencies between requests and blockings) in a more adequate way than production rule semantics (see e.g. [Via97]).

In contrast to the somewhat ad-hoc execution model of referential actions in SQL [ISO94, Hor92, CPM96], which simply aborts a transaction if a violation is detected, our semantics also provides valuable information in that case, i.e., if the given set of user requests is *not* executable: The additional information about maximal admissible sets can be used to explain the user why her updates are not admissible, and allows to revise the desired update in such a way that it is accepted by the system.

Acknowledgments. The first author would like to thank Joachim Reinert for fruitful discussions, especially on the peculiarities of triggers in SQL.

References

[AV91] S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[CPM96] R. Cochrane, H. Pirahesh, and N. Mattos. Integrating Triggers and Declarative Constraints in SQL Database Sytems. In *Proc. Intl. Conference on Very Large Data Bases*, pages 567–578, Mumbai (Bombay), India, 1996.

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

[Dix95] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. In A. Fuhrmann and H. Rott, editors, *Logic, Action and Information*. de Gruyter, 1995.

[ELS96] T. Eiter, N. Leone, and D. Saccà. The Expressive Power of Partial Models for Disjunctive Deductive Databases. In D. Pedreschi and C. Zaniolo, editors, *Proc. Intl. Workshop on Logic in Databases (LID)*, number 1154 in LNCS, pages 197–222, San Miniato, Italy, 1996. Springer.

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

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

[ISO94] ISO/IEC JTC1/SC21/WG3. ISO/ANSI working draft Database Languages – SQL3, August 1994. J. Melton (Ed.), ANSI, 1430 Broadway, New York, NY 10018.

[LML96] B. Ludäscher, W. May, and G. Lausen. Triggers, Games, and Stable Models. Technical report, Institut für Informatik, Universität Freiburg, 1996. http://www.informatik.uni-freiburg.de/~ludaesch/Paper/tgsm.ps.gz.

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

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

[PV95] P. Picouet and V. Vianu. Semantics and Expressiveness Issues in Active Databases. In *Proc. ACM Symposium on Principles of Database Systems*, 1995.

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

[Via97] V. Vianu. Rule-Based Languages. *Annals of Mathematics and Artificial Intelligence*, 19(I–II):215–259, 1997.

A The Remainder of the Logical Formalization

We present the remaining rules needed to formalize a set of *rac*'s RA as the logic program P_{RA}.

Auxiliary Relations. There are several auxiliary relations which have to be maintained. They contain the following information about referenced and referencable candidate key values:

- is_ref'able_$R.\vec{K}(\bar{x})$: the key value $R.\vec{K}(\bar{x})$ is referencable.
- rem_ref'able_$R.\vec{K}(\bar{x})$: the key value $R.\vec{K}(\bar{x})$ remains referencable.
- new_ref'able_$R.\vec{K}(\bar{x})$: the key value $R.\vec{K}(\bar{x})$ becomes referencable by some update.
- is_ref'd_$R_P.\vec{K}$_by_$R_C.\vec{F}(\bar{v})$: in the current database, the key value $R.\vec{K}(\bar{v})$ appears as foreign key value of $\vec{F}$ in some tuple $R_C(\bar{y})$.
- rem_ref'd_$R_P.\vec{K}$_by_$R_C.\vec{F}(\bar{v})$: there is a reference to the key value $R.\vec{K}(\bar{v})$ as foreign key value of $\vec{F}$ in some tuple $R_C(\bar{x})$ s.t. $\bar{x}[\vec{F}]$ does not change.
- new_ref'd_$R_P.\vec{K}$_by_$R_C.\vec{F}(\bar{v})$: a reference to the key value $R.\vec{K}(\bar{v})$ as foreign key $\vec{F}$ in some tuple $R_C(\bar{x})$ is introduced by some update.

The rules for maintaining these additional relations are shown in Table 4.

User Requests. The following rules have to be added to the rules (EXT_1) from Section 4:

$$\begin{array}{ll}
\mathsf{pot_ins}_R(\bar{X}) & \leftarrow \rhd \mathsf{ins}_R(\bar{X})\ . \\
\mathsf{ins}_R(\bar{X}) & \leftarrow \rhd \mathsf{ins}_R(\bar{X}), \neg\, \mathsf{blk_ins}_R(\bar{X})\ . \\
\mathsf{pot_mod}_R(M,\bar{X}) & \leftarrow \rhd \mathsf{mod}_R(M,\bar{X}). \\
\mathsf{mod}_R(M,\bar{X}) & \leftarrow \rhd \mathsf{mod}_R(M,\bar{X}), \neg\, \mathsf{blk_mod}_R(M,\bar{X}).
\end{array} \qquad (EXT_1)$$

For dealing with several user-requested modifications to the *same* tuple, for every foreign or candidate key $\vec{A}$, the following rules have to be added to EXT_2:

$$\begin{array}{ll}
\mathsf{pot_mod}_\rhd \rightsquigarrow R.\vec{A}(M,\bar{X}) & \leftarrow \mathsf{pot_mod}_\rhd \rightsquigarrow R.\vec{A}(M_1,\bar{X}), \\
 & \quad \mathsf{pot_mod}_\rhd \rightsquigarrow R.\vec{A}(M_2,\bar{X}), \\
 & \quad M = M_1 \cup M_2 \text{ is consistent.} \\
\mathsf{mod}_\rhd \rightsquigarrow R.\vec{A}(M,\bar{X}) & \leftarrow \mathsf{mod}_\rhd \rightsquigarrow R.\vec{A}(M_1,\bar{X}), \\
 & \quad \mathsf{mod}_\rhd \rightsquigarrow R.\vec{A}(M_2,\bar{X}), \\
 & \quad M = M_1 \cup M_2 \text{ is consistent.}
\end{array} \qquad (EXT_2)$$

Modifications on Child Tuples. A modification of a foreign key value $R_C.\vec{F}'$ of a child tuple can be problematic due to a *ric* $R_C.\vec{F}' \rightarrow R'_P.\vec{K}'$ only if the it is influenced by a propagation along *another ric* $R_C.\vec{F} \rightarrow R_P.\vec{K}$ (i.e., $R_C.\vec{F} \rightarrow R_P.\vec{K}$ on mod parent propagate and $R_C.\vec{F}$ and $R_C.\vec{F}'$ overlap) or by an external modification.

Thus, for a *ric* $R_C.\vec{F}' \rightarrow R'_P.\vec{K}'$ on mod child restrict, in those cases it is checked whether there is a referencable tuple in the current database. If there is no such tuple, then the modification is blocked, otherwise any modification of the attributes $R_P.\vec{K}$ or deletion of this tuple is blocked ((MCR_1) and (MCR_2) in Table 5).

For $R_C.\vec{F}' \rightarrow R'_P.\vec{K}'$ on mod child wait, the situation is analogous, but now the database after execution of Δ is checked (cf. (MCW_1) and (MCW_2)).

By considering only changes which are propagated along *another ric*, the 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" does not matter (i.e. on modify parent propagate has priority over on modify child restrict).

Insertions. Since insertions on parent tuples are not critical, only insertions of child tuples have to be handled. This is done analogously to (MCR) and (MCW) by (ICR) and (ICW) (see Table 6).

Interaction. The changes of candidate and foreign key values are determined depending on the elementary modify requests. Modifications can be founded either on external requests or by propagating modifications from parent relations. For a given database schema, (CH) (see Table 7) defines a set of rules for computing all possibilities how a key can change.
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 each key can change agreeing with the value on the intersection. Furthermore, a change of a key is forbidden, if its effect on the intersection with another key is not allowed (ABC) (see Table 7).
If a propagated modification would change a foreign key in a forbidden way, the propagation of the modification is forbidden (which by (MPP_2) further blocks the change of the respective parent key) (BMC_2) (see Table 7).
As blockings propagate upwards by rac's of the form $R.\vec{F} \rightarrow R_P.\vec{K}$ on mod parent propagate, they finally cause a blocking on their founding external requests (EXT_2) (also see Table 7).

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

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

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

$$\begin{array}{l}
\mathsf{blk_del}_R(\bar{X}) \leftarrow \mathsf{pot_del}_R(\bar{X}),\ \mathsf{mod}_R_P.K \rightsquigarrow R.\vec{F}(M,\bar{X})\ . \\
\mathsf{blk_mod}_R_P.K \rightsquigarrow R.\vec{F}(M,\bar{X}) \leftarrow \qquad\qquad (C) \\
\qquad \mathsf{pot_mod}_R_P.K \rightsquigarrow R.\vec{F}(M,\bar{X}),\ \mathsf{del}_R(\bar{X})\ .
\end{array}$$

Since propagated modifications are handled key-oriented as foreign-key-modifications, it is sufficient to handle contradicting modifications at this granularity: For every pair of *rac*'s $R_{P_1}.\vec{K}_1 \rightarrow R.\vec{F}_1$ on mod parent propagate and $R_{P_2}.\vec{K}_2 \rightarrow R.\vec{F}_2$ on mod parent propagate s.t. $R.\vec{F}_1$ and $R.\vec{F}_2$ overlap, overlapping but contradictory modifications are forbidden:

$$\begin{array}{l}
\mathsf{blk_mod}_R_{P_1}.\vec{K}_1 \rightsquigarrow R.\vec{F}_1(M_1,\bar{X}) \leftarrow \\
\qquad \mathsf{pot_mod}_R_{P_1}.\vec{K}_1 \rightsquigarrow R.\vec{F}_1(M_1,\bar{X}), \\
\qquad \mathsf{mod}_R_{P_2}.\vec{K}_2 \rightsquigarrow R.\vec{F}_2(M_2,\bar{X}), \\
\qquad M_1 \cup M_2 \text{ inconsistent}\ .
\end{array} \qquad (C)$$

The uniqueness of a candidate key $R.\vec{K}$ is guaranteed by the rules (K) (see Table 8).

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

$$
\begin{array}{lcl}
\mathsf{remains}\_R(\bar{X}) & \leftarrow & R(\bar{X}),\ \neg\,\mathsf{del}\_R(\bar{X}),\ \neg\,\exists M : \mathsf{mod}\_R(M, \bar{X})\ . \\
\mathsf{is\_ref'able}\_R_P.\vec{K}(\bar{V}) & \leftarrow & R_P(\bar{X}), \bar{V} = \bar{X}[\vec{K}]\ . \\
\mathsf{rem\_ref'able}\_R_P.\vec{K}(\bar{V}) & \leftarrow & R_P(\bar{X}), \bar{V} = \bar{X}[\vec{K}]\ ,\ \neg\,\mathsf{del}\_R_P(\bar{X}), \neg\,\exists M : \mathsf{chg}\_R_P.\vec{K}(M, \bar{X})\ . \\
\mathsf{new\_ref'able}\_R_P.\vec{K}(\bar{V}) & \leftarrow & \mathsf{ins}\_R_P(\bar{X}),\ \bar{V} = \bar{X}[\vec{K}]\ . \\
\mathsf{new\_ref'able}\_R_P.\vec{K}(\bar{V}) & \leftarrow & \mathsf{chg}\_R_P.\vec{K}(M, \bar{X}),\ M(\bar{X})[\vec{K}] = \bar{V}\ .
\end{array}
\qquad (RA)
$$

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

$$
\begin{array}{lcl}
\mathsf{is\_ref'd}\_R_P.\vec{K}\_\mathsf{by}\_R_C.\vec{F}(\bar{V}) & \leftarrow & R_C(\bar{X}),\ V = \bar{X}[\vec{F}]\ . \\
\mathsf{rem\_ref'd}\_R_P.\vec{K}\_\mathsf{by}\_R_C.\vec{F}(\bar{V}) & \leftarrow & \mathsf{remains}\_R_C(\bar{X}),\ V = \bar{X}[\vec{F}]\ . \\
\mathsf{rem\_ref'd}\_R_P.\vec{K}\_\mathsf{by}\_R_C.\vec{F}(\bar{V}) & \leftarrow & R_C(\bar{X}),\ V = \bar{X}[\vec{F}],\ \neg\,\mathsf{del}\_R_C(\bar{X}),\ \neg\,\exists M : \mathsf{chg}\_R_C.\vec{F}(M, \bar{X})\ . \\
\mathsf{new\_ref'd}\_R_P.\vec{K}\_\mathsf{by}\_R_C.\vec{F}(\bar{V}) & \leftarrow & \mathsf{ins}\_R_C(\bar{X}),\ \bar{V} = \bar{X}[\vec{F}]\ . \\
\mathsf{new\_ref'd}\_R_P.\vec{K}\_\mathsf{by}\_R_C.\vec{F}(\bar{V}) & \leftarrow & R_C(\bar{X}),\ \mathsf{chg}\_R_C.\vec{F}(M, \bar{X}),\ M(\bar{X})[\vec{F}] = \bar{V}\ .
\end{array}
\qquad (RD)
$$

Table 4: Rules for Maintaining Auxiliary Relations

For every *ric* $R_C.\vec{F}' \to R'_P.\vec{K}'$ and *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on mod child restrict 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:

$$
\begin{array}{lcl}
\mathsf{blk\_chg}\_R_C.\vec{F}(M, \bar{X}) & \leftarrow & \mathsf{pot\_chg}\_R_C.\vec{F}(M, \bar{X}),\ \mathsf{mod}\_R'_P.\vec{K}' \rightsquigarrow R_C.\vec{F}'(M', \bar{X}),\ M[\vec{F} \cap \vec{F}'] = M'[\vec{F} \cap \vec{F}'], \\
 & & \neg\,\mathsf{is\_ref'able}\_R_P.\vec{K}(M(\bar{X})[\vec{F}])\ . \\
\mathsf{blk\_chg}\_R_P.\vec{K}(M_P, \bar{Y}) & \leftarrow & \mathsf{pot\_chg}\_R_P.\vec{K}(M_P, \bar{Y}),\ \mathsf{chg}\_R_C.\vec{F}(M_C, \bar{X}),\ \mathsf{mod}\_R'_P.\vec{K}' \rightsquigarrow R_C.\vec{F}'(M', \bar{X}), \\
 & & M[\vec{F} \cap \vec{F}'] = M'[\vec{F} \cap \vec{F}'],\ M(\bar{X})[\vec{F}] = \bar{Y}[\vec{K}]\ .
\end{array}
\qquad (MCR_1)
$$

For every *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on mod child restrict:

$$
\begin{array}{l}
\mathsf{blk\_chg}\_R_C.\vec{F}(M, \bar{X}) \leftarrow \mathsf{pot\_chg}\_R_C.\vec{F}(M, \bar{X}),\ \mathsf{mod}\_\triangleright\!\rightsquigarrow R_C.\vec{F}(M', \bar{X}),\ M' \subseteq M,\ \neg\,\mathsf{is\_ref'able}\_R_P.\vec{K}(M(\bar{X})[\vec{F}])\ . \\
\mathsf{blk\_chg}\_R_P.\vec{K}(M_P, \bar{Y}) \leftarrow \mathsf{pot\_chg}\_R_P.\vec{K}(M_P, \bar{Y}),\ \mathsf{chg}\_R_C.\vec{F}(M, \bar{X}),\ \mathsf{mod}\_\triangleright\!\rightsquigarrow R_C.\vec{F}(M', \bar{X}),\ M' \subseteq M,\ M(\bar{X})[\vec{F}] = \bar{Y}[\vec{K}]\ . \\
\mathsf{blk\_del}\_R_P(\bar{Y}) \leftarrow \mathsf{pot\_del}\_R_P(\bar{Y}),\ R_C(\bar{X}),\ \mathsf{chg}\_R_C.\vec{F}(M_C, \bar{X}),\ M(\bar{X})[\vec{F}] = \bar{Y}[\vec{K}]\ .
\end{array}
\qquad (MCR_2)
$$

For every *ric* $R_C.\vec{F}' \to R'_P.\vec{K}'$ and *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on mod child wait 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:

$$
\begin{array}{lcl}
\mathsf{blk\_chg}\_R_C.\vec{F}(M, \bar{X}) & \leftarrow & \mathsf{pot\_chg}\_R_C.\vec{F}(M, \bar{X}),\ \mathsf{mod}\_R'_P.\vec{K}' \rightsquigarrow R_C.\vec{F}'(M', \bar{X}),\ M[\vec{F} \cap \vec{F}'] = M'[\vec{F} \cap \vec{F}'], \\
 & & \neg\,\mathsf{rem\_ref'able}\_R'_P.\vec{K}(M(\bar{X})[\vec{F}']),\ \neg\,\mathsf{new\_ref'able}\_R'_P.\vec{K}(M(\bar{X})[\vec{F}'])\ .
\end{array}
\qquad (MCW1)
$$

For every *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on mod child wait:

$$
\begin{array}{lcl}
\mathsf{blk\_chg}\_R_C.\vec{F}(M, \bar{X}) & \leftarrow & \mathsf{pot\_chg}\_R_C.\vec{F}(M, \bar{X}),\ \mathsf{mod}\_\triangleright\!\rightsquigarrow R_C.\vec{F}(M', \bar{X}),\ M' \subseteq M, \\
 & & \neg\,\mathsf{rem\_ref'able}\_R'_P.\vec{K}(M(\bar{X})[\vec{F}']),\ \neg\,\mathsf{new\_ref'able}\_R'_P.\vec{K}(M(\bar{X})[\vec{F}'])\ .
\end{array}
\qquad (MCW2)
$$

Table 5: Rules for Handling Modifications

For every *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on ins child restrict:

$$
\begin{array}{lcl}
\mathsf{blk\_ins}\_R_C(\bar{X}) & \leftarrow & \mathsf{pot\_ins}\_R_C(\bar{C}),\ \neg\,\mathsf{is\_ref'able}\_R_P.\vec{K}(\bar{X}[\vec{F}])\ . \\
\mathsf{blk\_chg}\_R_P.\vec{K}(M, \bar{Y}) & \leftarrow & \mathsf{pot\_chg}\_R_P.\vec{K}(M, \bar{Y}),\ \mathsf{ins}\_R_C(\bar{X})\ , \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]\ . \\
\mathsf{blk\_del}\_R_P(\bar{Y}) & \leftarrow & \mathsf{pot\_del}\_R_P(\bar{Y}),\ \mathsf{ins}\_R_C(\bar{X}),\ \bar{X}[\vec{F}] = \bar{Y}[\vec{K}]\ .
\end{array}
\qquad (ICR)
$$

For every *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on ins child wait:

$$
\mathsf{blk\_ins}\_R_C(\bar{X}) \leftarrow \mathsf{pot\_ins}\_R_C(\bar{X}),\ \neg\,\mathsf{rem\_ref'able}\_R_P.\vec{K}(\bar{X}[\vec{F}]),\ \neg\,\mathsf{new\_ref'able}\_R_P.\vec{K}(\bar{X}[\vec{F}])\,. \qquad (ICW)
$$

Table 6: Rules for Handling Insertions

For a given foreign resp. candidate key $R.\vec{A}$, let

$$M_{\vec{A}} := \{(R'.\vec{K}, R.\vec{F}) \mid R.\vec{F} \to R'.\vec{K} \text{ on mod parent propagate} \in RA \text{ and } \vec{F} \text{ overlaps } \vec{A}\} \cup \{(\triangleright, R.\vec{A})\}$$

be the set of referential dependencies along which modifications can be propagated which influence the value of $\vec{A}$. Note that the cardinality n of $M_{\vec{A}}$ only depends on the given database schema, but not on the size of the database. Moreover, we may assume that the elements of $M_{\vec{A}}$ are numbered by $i = 1, \ldots, n$ such that $(R_i.\vec{K}_i, R.\vec{F}_i)$ denotes the i-th element of $M_{\vec{A}}$. For every set of indices $I \subseteq \{1, \ldots, n\}$ there are rules

$$\begin{array}{lcl}
\mathsf{pot_chg}_R.\vec{A}(M, \bar{X}) & \leftarrow & \left(\bigwedge_{i \in I} \mathsf{pot_mod}_R_i.\vec{K}_i \rightsquigarrow R.\vec{F}_i(M_i, \bar{X}),\ M_i(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}]\right),\ M = \bigcup_{i \in I} M_i\ . \\
\mathsf{chg}_R.\vec{A}(M, \bar{X}) & \leftarrow & \left(\bigwedge_{i \in I} \mathsf{mod}_R_i.\vec{K}_i \rightsquigarrow R.\vec{F}_i(M_i, \bar{X}),\ M_i(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}]\right), \\
 & & \left(\bigwedge_{i \in \{1,\ldots,n\} \setminus I} \neg\, \exists M_i : \mathsf{mod}_R_i.\vec{K}_i \rightsquigarrow R.\vec{F}_i(M_i, \bar{X}) \wedge M_i(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}]\right),\ M = \bigcup_{i \in I} M_i\ .
\end{array} \qquad (CH)$$

To illustrate this definition, the rules obtained when instantiating the schema for $n = 2$ (i.e., only one foreign key $\vec{F}$ overlaps $\vec{A}$) are given:

$$\begin{array}{lcll}
\mathsf{pot_chg}_R.\vec{A}(M, \bar{X}) & \leftarrow & M = \emptyset\ . & \%\ I = \emptyset \\
\mathsf{pot_chg}_R.\vec{A}(M, \bar{X}) & \leftarrow & \mathsf{pot_mod}_R_1.\vec{K}_1 \rightsquigarrow R.\vec{F}(M_1, \bar{X}),\ M_1(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}],\ M = M_1\ . & \%\ I = \{1\} \\
\mathsf{pot_chg}_R.\vec{A}(M, \bar{X}) & \leftarrow & \mathsf{pot_mod}_\triangleright \rightsquigarrow R.\vec{A}(M_2, \bar{X}),\ M_2(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}],\ M = M_2\ . & \%\ I = \{2\} \\
\mathsf{pot_chg}_R.\vec{A}(M, \bar{X}) & \leftarrow & \mathsf{pot_mod}_R_1.\vec{K}_1 \rightsquigarrow R.\vec{F}(M_1, \bar{X}),\ M_1(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}], & \%\ I = \{1, 2\} \\
 & & \mathsf{pot_mod}_\triangleright \rightsquigarrow R.\vec{A}(M_2, \bar{X}),\ M_2(\bar{X})[\vec{A}] \neq \bar{X}[\vec{A}],\ M = M_1 \cup M_2\ . &
\end{array}$$

For every foreign key $\vec{F}$ and foreign or candidate key $\vec{A}$ s.t. $\vec{F}$ and $\vec{A}$ overlap:

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

For every *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on mod parent propagate:

$$\mathsf{blk\_mod}\_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M, \bar{X}) \leftarrow \mathsf{pot\_mod}\_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M, \bar{X}), \mathsf{blk\_chg}\_R_C.\vec{F}(M, \bar{X})\ . \qquad (BMC)$$

For every *rac* $R_C.\vec{F} \to R_P.\vec{K}$ on mod parent propagate:

$$\begin{array}{rcl}
\mathsf{blk_mod}_R_C(M, \bar{X}) \leftarrow & \triangleright \mathsf{mod}_R_C(M, \bar{X}),\ \mathsf{blk_mod}_\triangleright \rightsquigarrow R_C.\vec{F}(M', \bar{X}),\ M' = M[\vec{F}]\ . & \\
\mathsf{mod}_R_C(M, \bar{X}) \leftarrow & \mathsf{mod}_R_P.\vec{K} \rightsquigarrow R_C.\vec{F}(M, \bar{X})\ . & (EXT_2)
\end{array}$$

Table 7: Rules for Dealing with Interfering Modifications

For every candidate key $R.\vec{K}$:

$$\begin{array}{lcl}
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow & \mathsf{pot_ins}_R(\bar{X}),\ \mathsf{rem_ref'able}_R.\vec{K}(\bar{X}[\vec{K}])\ . \\
\mathsf{blk_chg}_R.\vec{K}(M, \bar{X}) & \leftarrow & \mathsf{chg}_R.\vec{K}(M, \bar{X}),\ \mathsf{rem_ref'able}_R.\vec{K}(M(\bar{X})[\vec{K}])\ . \\
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow & \mathsf{pot_ins}_R(\bar{X}),\ \mathsf{ins}_R(\bar{Y}),\ \bar{X}[\vec{K}] = \bar{Y}[\vec{K}]\ . \\
\mathsf{blk_ins}_R(\bar{X}) & \leftarrow & \mathsf{pot_ins}_R(\bar{X}),\ \mathsf{chg}_R.\vec{K}(M, \bar{Y}), \bar{X}[\vec{K}] = M(\bar{Y})[\vec{K}]\ . \\
\mathsf{blk_chg}_R.\vec{K}(M, \bar{Y}) & \leftarrow & \mathsf{pot_chg}_R.\vec{K}(M, \bar{Y}),\ \mathsf{ins}_R(\bar{X}),\ \bar{X}[\vec{K}] = M(\bar{Y})[\vec{K}]\ . \\
\mathsf{blk_chg}_R.\vec{K}(M, \bar{X}) & \leftarrow & \mathsf{pot_chg}_R.\vec{K}(M, \bar{X}),\ \mathsf{chg}_R.\vec{K}(M', \bar{Y}),\ M(\bar{X})[\vec{K}] = M'(\bar{Y})[\vec{K}]\ .
\end{array} \qquad (K)$$

Table 8: Rules for Preserving Key Dependencies