# Ambiguity for Referential Integrity Is Undecidable

Joachim Reinert

University of Kaiserslautern
Department of Computer Science
P.O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: jreinert@informatik.uni-kl.de

**Abstract.** SQL has grown to be *the* language for relational database systems. One vital element of the relational model is referential integrity. This type of integrity constraints is now included in the new SQL2 standard [11] with capabilities to react on violations of specified integrity constraints. These reactions may lead to indeterminism with respect to the outcome of a user operation which is also known from the usage of rules or triggers. In the database context, however, such ambiguities are undesirable. Hence, for each submitted operation one must check whether or not an ambiguity occurs, and in the former case rollback the operation. Since such checks are time consuming, one might consider performing them only for schemas which bear the risk of an indeterminism. This paper shows that it is *undecidable* whether or not a schema may have an instance leading to ambiguities. Therefore, unnecessary checks cannot be avoided in general.

## 1 Introduction

Compared to other paradigms in database systems, the relational technology is long known and well understood. Invented by Codd [4] in the late sixties as a rock-solid mathematical theory for management of data, some concepts remain somewhat vague. One of these concepts is referential integrity. Basically defined to guarantee the existence of referenced objects, it was refined by Date [5, 7] to a more active concept, i.e., the possibility to define limited reactions in order to compensate violations of the referential integrity by so-called referential actions. These ideas have been included in the new SQL2 standard [11]. In this paper, we analyze the referential integrity with respect to the semantics specified in this standard.

Referential integrity with referential actions may lead to some indeterminism during the evaluation of a user operation (see e.g. [13, 14, 17]) which is undesired (and not allowed in the SQL standard). Hence systems implementing referential integrity have to deal with this phenomenon. The problem is well known in literature because on a more abstract level, referential integrity constraints (with referential actions to maintain the integrity) can be viewed as triggers or rules[1] [6, 7, 19]. Ambiguities of rule sequences were addressed for example in [2, 1, 20]. To detect whether a given set of rules may lead to an indeterministic behaviour, general analysis procedures of the rule sets were developed. The main directions are to analyze the read and write sets of the rules (e.g. [20]) or to use concepts developed in the area of term rewriting (e.g. [2, 1]). Both directions have developed analysis procedures which operate on an arbitrary rule set without the necessity to examine any database instance (some sort of schema analysis). Unfortunately these procedures bear two drawbacks:

1. They are based on a schema analysis. This allows to have the check-overhead at compile-time of an operation (and therefore decrease the run-time) but the problems occur within the execution of a concrete user operation on a concrete instance of the schema. Hence, it is possible to have a "problematic" rule set marked as "problematic" and to get still never into trouble, as the "problematic" instances are never generated.
2. The problem is known to be undecidable, i.e., it is impossible to have a detection algorithm which discriminates exactly those schemas having problematic instances from those having none. Therefore the algorithms either cover only a restricted class of rules or are over-pessimistic in all cases.

The SQL2 standard "solves" the above problems by introducing a run-time check. In real applications of the relational model (business administration), the instances of a schema (the databases) are finite, and therefore the problem of ambiguity becomes decidable if each user operation terminates. However, the required checks result in a severe run-time overhead. Therefore a good compile-time check procedure would be valuable to avoid unnecessary checking at run-time.
These observations lead to the development of specific criteria for referential integrity, covering schemas which may have instances with ambiguities. To check these criteria the required algorithms also have been developed. To cover SQL the approach has to be divided into two phases:

1. Test the schema-based criteria at compile-time.
2. If a problem *may* occur, include a check at run-time. If not, no further precautions are required.

A detailed discussion may be found in [13, 14, 17] and is beyond the scope of this paper. Besides the limitation that the criteria are schema-based (so they cannot cover SQL completely at compile-time), the second problem mentioned above remains: The criteria developed so far (and checked by the algorithms) are sufficient only, i.e., they only identify a subset of the schemas which are safe (exhibiting no ambiguity)[2]. The goal of this research was to derive a precise criterion for referential integrity constraints. As opposed to the more general problem

[1] We will use the term trigger and rule interchangeable throughout the paper always referring to the same concept.

[2] Markowitz presented a criterion in [15] claimed to be safe and sound, but it can be shown that this is not correct.

of rules, this goal seemed to be realistic because such referential integrity constraints are structural constraints and therefore limited in their expressiveness. However, we show in this paper that the underlying problem is undecidable and, as a consequence, a precise criterion cannot be developed.
The paper is organized as follows: In the following section, we introduce referential integrity as proposed by Codd and the form now standardized in the SQL2 standard. After this short introduction we present an example to show why referential integrity may lead to ambiguities. This discussion is followed by the presentation of the needed results known from literature and our own proof of the main result. The paper is closed by a discussion of the consequences of this result for referential integrity checking in relational database systems.

## 2 Referential Integrity

In this section we discuss the referential integrity as it was defined by Codd in his fundamental paper [4] and the definition which is now adopted by the standard committees.
It is assumed that the reader is familiar with notions of the relational data model.

### 2.1 Referential Integrity in the Relational Data Model

Referential integrity constraints are a fundamental concept of the relational data model introduced by Codd [4]. To define this concept (at least informally) one needs the notion of tables as disjoint sets of attributes. Attributes themselves are defined as null-ary functions (constants) mapping into a specific domain. In his original paper, Codd defines the primary key to be an attribute or a group of attributes which uniquely identifies every database object (tuple) within a table (key condition). The primary key of each tuple has to be completely defined, i.e., no null-values are allowed as values of attributes forming the primary key. Furthermore, a primary key has to be minimal, i.e., no real subset of the attributes fulfills the key condition (minimality condition). Together the three conditions form the primary key condition. It is possible to have more than one attribute (group of attributes) in a relation that satisfy the key condition and the minimality condition. Such attributes (groups of attributes) are called candidate keys. In contrast to the primary key, null-values may be allowed as attribute values for such candidate keys.
The domains mentioned before are the ranges of the attributes. Domains are independent from attributes and represent the means to express dependencies between attributes[3]. This independence allows the implicit definition of foreign keys, an additional basic concept of the relational data model: A foreign key is an attribute (or group of attributes) defined on the same domain as the primary key of some relation. The property of set inclusion is connected directly to the

[3] Note that the concept of the domain of an attribute is extended in canonical form to groups of attributes: Their domain is the crossproduct of the domains of each attribute in the group.

foreign key, i.e., for every value of the attribute (respectively group of attributes) forming the foreign key, there has to be a tuple in some relation with this value (respectively values) in its primary key (foreign key condition[4]). Exceptions to this rule are special null-values that appear as attribute values in the foreign key.
Note the usage of the term *some* in this definition: It is possible to have more than one primary key defined on a specific domain (let us denote this set of tables with $S$). In this case, a foreign key defined on that domain may reference all these primary keys in the following sense: For each tuple $t_C$ of the child table $C$ with defined values in the foreign key attributes, a matching tuple has to exist in at least one of the tables of $S$. One cannot specify which table. Furthermore, different tuples of $C$ may reference different tuples in different tables.
To summarize the aspects of referential integrity as defined by Codd: It is a static integrity constraint which prevents the existence of defined foreign keys without the existence of a primary key with the same value and results in a vague dependency between tuples and between tables.

### 2.2 Referential Integrity in SQL2

The discussion in the literature (e.g [7]) has shown that it is useful to extent the concept of referential integrity. One minor extension is the possibility to reference not only primary keys but also candidate keys by a foreign key. In connection to this enhancement it was proposed to specify the integrity constraint explicitly, thereby achieving a precise dependency between the foreign key and the referenced primary key or candidate key. A new dimension was added through system-enforced maintenance of the relational invariants in a more active manner. Clearly, each database state has to obey these invariants and the only possible reaction of a non-active database management system is to roll back all operations violating these constraints. The enhancements for referential integrity maintenance (e.g. [5]) describe how the database management system reacts in case a referential integrity constraint is violated. Such reactions have been included in the new SQL2 standard [11] (we call them referential actions). As the useful reactions seem to be limited, the descriptive nature of the relational model remains valid: It is specified *what* has to be done if *one* integrity constraint is violated and it is *not* specified *how* this maintenance is carried out. In the following we will shortly discuss the possibilities of this standard.
In the SQL2 standard, referential integrity constraints are defined when tables are created or altered. For this purpose, there is a sub-clause of the `CREATE TABLE` and the `ALTER TABLE` statement referring to a table $C$ which includes the foreign key (child table):

[4] The primary key condition and the foreign key condition are also known as the relational invariants.

```
FOREIGN KEY (<referencing columns>)
   REFERENCES <table name> [(<referenced columns>)]
   [MATCH {FULL | PARTIAL}]
   [ON UPDATE {CASCADE | SET NULL |
               SET DEFAULT | NO ACTION}]
   [ON DELETE {CASCADE | SET NULL |
               SET DEFAULT | NO ACTION}]
```

The `<referencing columns>` are the attribute names of the foreign key $F = \{f_1, \ldots f_n\}$ in $C$. The `<referenced columns>` denote the corresponding attributes of the primary key $K$ of the table $P$ with name `<table name>` (parent table). The semantics expressible through `MATCH {FULL | PARTIAL}` is the interpretation of null-values in the foreign key of a tuple $t_C$. We assume that a null-value in a foreign key is allowed (unless stated otherwise) and such foreign keys are not considered in the check whether or not a corresponding primary key exists. To express this semantics the `MATCH` sub-clause has to be omitted completely. An in-depth discussion of the various possible interpretations is beyond the scope of this paper (see e.g. [10]).
The sub-clauses `ON UPDATE ...` and `ON DELETE ...` allow to specify the referential action in case referential integrity is violated by a user operation. Six manipulation operations on $P$ or $C$ are possible: Insert into $P$, Update $P$, Delete from $P$, Insert into $C$, Update $C$ and Delete from $C$. Due to the definition of referential integrity, only four out of the six operations may transform a database state which fulfills referential integrity into one where a referential integrity constraint is violated ("Insert into $P$" and "Delete from $C$" cannot cause problems). In the SQL2 standard, the two operations "Insert into $C$" and "Update $f_i$ of $C$" on the child table are forbidden (backed out) if these would result in DB states where referential integrity is not fulfilled. Therefore, only the two operations ("Delete from $P$" and "Update $k_i$ of $P$") on a parent are handled in a special way:

1. `ON UPDATE`. If attributes of a key referenced in a referential integrity constraint are updated in a tuple $t_P$, then depending on the specification in the schema one of the following actions is carried out:
   - `CASCADE`. The new values in the key are propagated to the referencing children $t_C$.
   - `SET NULL`. The attributes in the referencing tuples $t_C$ corresponding to the updated key attributes are set to the null-value.
   - `SET DEFAULT`. The attributes in the referencing tuples $t_C$ corresponding to the updated key attributes are set to a default value (definable for each attribute in the schema).
   - `NO ACTION`. Nothing is done. Referential integrity remains violated and if no other operations take place to correct the mismatch of the corresponding tuples $t_C$, the complete work of the transaction will eventually be backed out. This happens either at the end of the statement

(if the integrity checking is not deferred) or at transaction commit (if the integrity checking is deferred). The implications of deferred integrity checking raise difficult semantical problems and are subject to further research.

2. `ON DELETE`. If a tuple $t_P$ is deleted then the following actions are carried out depending on the specification in the schema:
   - `CASCADE`. The referencing children are also deleted.
   - `SET NULL`. The foreign key attributes of the referencing children are set to the null-value.
   - `SET DEFAULT`. The foreign key attributes of the referencing children are set to their default value.
   - `NO ACTION`. Nothing is done. Referential integrity remains destroyed and if no other operation takes place to correct this, the complete work of the transaction will be backed out.

There is another important referential action which is not introduced in the SQL2 standard, but in nearly all papers which are dealing with referential integrity: `RESTRICT` (or `RESTRICTED` depending on the author). The semantics of this referential action is to forbid any change (update or delete) of a parent tuple $t_P$ as long as there exist referencing child tuples $t_C$. Although this action is not in the SQL2 standard (but scheduled for SQL3 [12]) we will include it in our discussion for the matter of completeness.

## 3 The Problem of Ambiguity

The standard technique proposed in literature (e.g. [8, 9]) for implementing integrity constraint maintenance in database management systems is an independent trigger (or rule) for each integrity constraint. If this technique is used to maintain *descriptively* specified referential integrity constraints, some indeterminism with respect to the outcome of a user operation may occur. In this section we present an example for this sort of ambiguity.

The problem of ambiguity stems from the descriptive nature of the relational database languages which hide any knowledge about *how* something is carried out on underlying structures and let the user specify only *what* he wants to achieve. Up to now (without referential actions or any other active component), the user has specified the complete scope of his operation, i.e., he has specified all components and all operations on these components as far as they are visible at his interface (e.g., no operations are specified on access paths). Elements not mentioned in the operation either are not of any interest (e.g. order of tuples) or do not influence the outcome of the operation. This scenario changes if an active component is involved: Something happens under cover. As the definitions of referential integrity constraints (with referential actions) are autonomous from each other and from any specific instance of the schema, but have influence on the outcome of an operation, a complete semantics has to include this interference. Fig. 1 shows an example of a schema with possible ambiguities. The

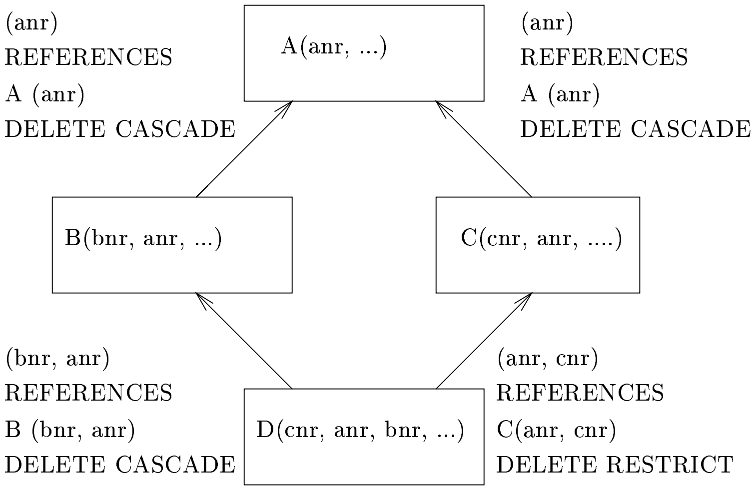


**Fig. 1.** Schema with ambiguities

structure of the schema is rather artificial but nevertheless a possible schema in SQL2. Let us look at the following instance of this schema:

Relation A: (anr, . . . )
(1, . . . )
Relation B: (bnr, anr, . . . )
(2, 1, . . . )
Relation C: (cnr, anr, . . . )
(3, 1, . . . )
Relation D: (cnr, anr, bnr, . . . )
(3, 1, 2, . . . )

Given that each integrity constraint is "implemented" by a separate trigger [9] (or ECA rule [18]) the following happens: If the user deletes the tuple (1, . . . ) in relation A and the path A-C-D is followed first, then the deletion of the tuple (3, 1, . . . ) is prohibited, because the tuple (3, 1, 2, . . . ) in relation D references (3, 1, . . . ). Therefore, the complete operation is backed out. The result is the database state before the deletion of (1, . . . ). If the path A-B-D is followed first, then the tuple (3, 1, . . . ) is deleted and, in turn, the deletion of (3, 1, . . . ) in C is performed, because referencing tuples in D no longer exist. The result is the empty database[5].

Note the difference between *ambiguity* problems and *integrity* problems: The latter can not be tolerated in a database while the former may be tolerable for somebody only interested in a DB state satisfying all integrity constraints.

[5] Similar problems occur with other referential actions, e.g. SET DEFAULT and SET NULL, or even the same pair of referential actions.

Hence approaches dealing with integrity problems (e.g. [16]) have another scope[6] but we think, the acceptance of a system will be rather low in general, if the user is not able to understand what the system is doing. This view is embodied in the new SQL2 standard with an abstract description of a special evaluation procedure for referential integrity constraints. However, the main problem of this evaluation procedure are the costs. These costs result from the necessity to maintain some sort of log for certain operations: In a cascade of referential integrity maintenance, each operation has to check whether or not a problem occurs, by analyzing the log of the operations carried out so far. Apparently this procedure causes a lot of overhead at run-time. To prevent this overhead whenever possible, it is necessary to decide at compile-time of the operation (or the schema) whether or not such an ambiguity may occur. However, to achieve SQL2 compliance the run-time check has to be implemented in order to allow operations (schemas) where ambiguities cannot be ruled out[7].
As mentioned earlier, several authors have already presented different approaches to the problem of compile-time checks. Because the approach of [2, 1] and [20] is rather general, it cannot use the specific knowledge of the special properties of referential integrity constraints. In [14, 17], approaches tailored for this special area of constraints were presented. Both approaches suffer from their incompleteness, i.e., if the check procedure does not find a possible source of ambiguities then there are none (hence, the procedures are sufficient), whereas in the other case, if the procedures identify an ambiguity, there may be none and hence the procedures are not complete.

However, a sufficient *and* complete check procedure would be extremely valuable; therefore, the researchers concentrate on more and more sophisticated approaches. Our main result presented in the next section shows the inability to have such an exact check procedure.

## 4 Ambiguity is Undecidable

A check procedure has to decide the question "are there any ambiguities?" for a given schema. The result of our research presented here is the *undecidability* of this question in general. The subsequent sections prove the following:

*There exists no decidable criterion $P$ over a relational schema $\rho$ with:*

$$(P = \mathit{TRUE}) \iff \textit{no instance of } \rho \textit{ exhibits an ambiguity}$$

To provide a formal proof of this theorem we introduce some preliminaries.

[6] Those approaches try to detect integrity problems and provide some sort of remedy (e.g. some rules which reinforce the integrity) and those remedies may be the *source* of ambiguity problems.

[7] If a system does not provide such a run-time check and ambiguities cannot be tolerated, all schemas (operations) which may lead to ambiguities have to be forbidden.

### 4.1 Some Formal Notation

In order to state our problem it is sufficient to restrict ourselves to the following referential actions:

- If a tuple t is deleted or its primary key is changed then this action is *propagated* to those tuples t' that reference t before the deletion/update has taken place (referential action `CASCADE`).
- If a tuple t is to be deleted or its primary key is changed and there are tuples t' referencing t before the deletion/update has taken place, then the deletion/update of t is *forbidden* (referential action `RESTRICT`)[8].

First, we define the basic elements of a relational schema: Domains, attributes, relations, and keys, followed by the definition of schema instances.
**Relational schema:** A *relational schema* $\rho$ consists of four parts:

1. A set $\mathcal{D} = \{d_1, \ldots, d_k\}$ of domain names.
2. A set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of 0-ary functions, $a_j :\rightarrow \mathcal{D}$, called attribute names.
3. A set $\mathcal{T} = \{t_1, \ldots, t_l\}$ of sets of attributes, called relations.
   $t_h = \{a_{h,1}, \ldots, a_{h,n_h}\} \subseteq \mathcal{A}$ and $t_i \cap t_j = \emptyset$ for $i \neq j$.
4. A set $\mathcal{K} = \{k_{1,1}, \ldots, k_{l,q_l}\}$ of primary keys and candidate keys.
   $k_{m,v} \subseteq \{a_{m,1}, \ldots, a_{m,n_m}\}$.

**Instance:** An *instance* $\mathcal{I}$ of a relational schema $\rho$ depends on sets $\overline{d_1}, \ldots, \overline{d_k}$ of values ($\overline{\mathcal{D}} = \{\overline{d_1}, \ldots, \overline{d_k}\}$) and results in a set of relation instances $r_1, \ldots, r_l$ with the following properties:

- $\overline{d_i}$ is the set of values of the domain $d_i$.
- $r_h \subseteq \overline{d_{h,1}} \times \ldots \times \overline{d_{h,n_h}}$ where $d_{h,i}$ is the domain of the attribute $a_i$ in table $t_h$ and $\overline{d_{h,i}}$ is the set of values of $d_{h,i}$.
- $\forall r_i \forall k_{i,l} = \{a_{i,l,1}, \ldots, a_{i,l,n_{i,l}}\} \in \mathcal{K} : \forall v_1, v_2 \in r_i :$
  $v_1{\downarrow_{k_{i,l}}} = v_2{\downarrow_{k_{i,l}}} \Leftrightarrow v_1 = v_2$[9].

On the basis of these notions we will now define inclusion dependencies, functional dependencies, referential integrity constraints and NULL constraints.

An instance $\mathcal{I}$ satisfies

- a *NULL constraint* $nn = r_i(a_{v_1}, \ldots, a_{v_k})$ if:
  $\forall v \in r_i : v{\downarrow_{\{a_{v_1},\ldots,a_{v_k}\}}} = (w_1, \ldots, w_k) \Rightarrow w_j \neq NULL$ for each $j = 1, \ldots, k$.
- an *inclusion dependency*
  $i = r_m(a_{v_1}, \ldots, a_{v_k}) \subseteq r_{m'}(a_{v_{k+1}}, \ldots, a_{v_{2k}})$ if:
  $\forall v \in r_m : v{\downarrow_{\{a_{v_1},\ldots,a_{v_k}\}}} = (w_1, \ldots, w_k)$ and $w_j \neq NULL \Rightarrow$
  $(\exists v' \in r_{m'} : v'{\downarrow_{\{a_{v_{k+1}},\ldots,a_{2k}\}}} = (w_1, \ldots, w_k))$.

[8] The choice of these referential actions in the proof is done on behalf of simplicity, other actions lead to the same result.
[9] We denote with $v{\downarrow_{\{a_1,\ldots,a_k\}}}$ the projection of $v$ to the attributes $\{a_1, \ldots, a_k\}$.

- a *functional dependency*
  $f = r_m(a_{v_1}, \ldots, a_{v_k}) \to r_m(a_{v_{k+1}}, \ldots, a_{v_{k+l}})$ if:
  $\forall v_1, v_2 \in r_m : v_1{\downarrow}_{\{a_{v_1},\ldots,a_{v_k}\}} = (w_1, \ldots, w_k) = v_2{\downarrow}_{\{a_{v_1},\ldots,a_{v_k}\}}$
  and $w_j \neq NULL \Rightarrow v_1{\downarrow}_{\{a_{v_{k+1}},\ldots,a_{v_{k+l}}\}} = v_2{\downarrow}_{\{a_{v_{k+1}},\ldots,a_{v_{k+l}}\}}$.
- a *referential integrity constraint*
  $ri = r_m(a_{v_1}, \ldots, a_{v_k}) \sqsubseteq r_{m'}(a_{v_{k+1}}, \ldots, a_{v_{2k}})$ if:
  $\forall v \in r_m : v{\downarrow}_{\{a_{v_1},\ldots,a_{v_k}\}} = (w_1, \ldots, w_k)$ and
  $w_j \neq NULL \Rightarrow (\exists v' \in r_{m'} : v'{\downarrow}_{\{a_{v_{k+1}},\ldots,a_{2k}\}} = (w_1, \ldots, w_k)$ and
  $\exists k \in \mathcal{K} : k = \{a_{v_{k+1}}, \ldots, a_{2k}\})$.

Obviously a referential integrity constraint subsumes an inclusion dependency. This observation is central to our proof.

### 4.2 Inclusion Dependencies

We have introduced the concept of inclusion dependencies as dependencies separate from referential integrity constraints. Furthermore, we stated that a referential integrity comprises such an inclusion dependency. In this section, we elaborate on the influence of general inclusion dependencies on possible ambiguities.

The schema of Fig. 2 is slightly enhanced compared to the schema of Fig. 1. Let us now assume that for all instances $\mathcal{I}$ the inclusion dependency $D.anr \subseteq E.anr$[10] holds (the application may enforce this). Then no ambiguity at all will occur because of the following observation:
If a tuple $t$ in $D$ is accessed via the paths $B - D$ and/or $C - D$ then the attribute $t.anr$ is not null. Hence, the inclusion assumption ($D.anr \subseteq E.anr$) ensures the existence of a tuple $\hat{t}$ in $E$ with $\hat{t}.anr = t.anr$. The precondition for an ambiguity to occur is the deletion of the tuple $\tilde{t} \in A$ with $\tilde{t}.anr = t.anr$. But the operation "delete $\tilde{t}$" will be rolled back because of the referencing tuple $\hat{t}$ in $E$ and the `RESTRICT` option. Therefore no ambiguity occurs!
The observation of the influence of inclusion dependencies on ambiguity is central to our proof, since Chandra and Vardi [3] have shown that the implication problem for functional and inclusion dependencies is undecidable. Their proof is based on a reduction of the word problem in monoids to this problem. We will use this result to prove our central theorem.

### 4.3 Result

Let us first state our main theorem and the corollary describing our main result. Afterwards we proceed to the proof of the theorem in some more or less technical steps.

[10] $D.anr$ respectively $E.anr$ denotes the set of all values occurring in the attribute anr of tuples in $D$ (respectively $E$).

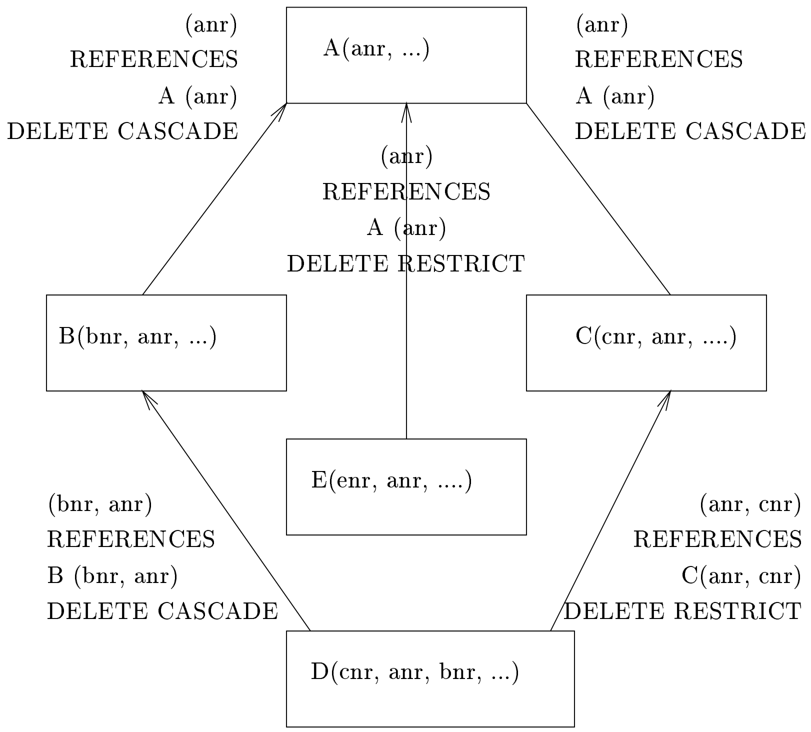


**Fig. 2.** Relevance of inclusion dependencies

**Theorem 1.** *Given a relational schema $\rho$, a set of referential integrity constraints $\mathcal{C}$, a set $\mathcal{N}$ of NULL constraints and an inclusion dependency $i = (a_1, \ldots, a_n) \subseteq (b_1, \ldots, b_n)$ where $a_i$ and $b_i$ are attributes of compatible type, it is undecidable whether $\rho, \mathcal{C}, \mathcal{N} \models i$*[11].

The theorem tailors the result of Chandra and Vardi to a specific sort of inclusion dependencies and functional dependencies. The main result follows directly from theorem 1:

**Corollary 2.** *It is undecidable whether a relational database schema with explicit referential constraints and referential actions is free of ambiguities or not.*

*Proof.* From theorem 1 it follows, that for a given relational schema with a set of attributes, a set of key constraints $\mathcal{K}$, a set of referential integrity constraints $\mathcal{R}$ and an inclusion dependency $i$, it is undecidable whether or not $\mathcal{K}, \mathcal{R} \models i$.

[11] $\rho, \mathcal{C}, \mathcal{N} \models i$ holds if all instances $\mathcal{I}$ of $\rho$ that satisfy all dependencies in $\mathcal{C}$ and all NULL constraints in $\mathcal{N}$ also satisfy $i$.

Furthermore, an inclusion dependency can avoid ambiguities because it initiates rollbacks for the critical situations. Because the existence of such guarding inclusion dependencies is undecidable the question of ambiguity is undecidable. □

To prove theorem 1 we reduce the mentioned problem of general set inclusion dependencies and functional dependencies (which is undecidable according to the result of [3]) to the more specific referential integrity constraints and NULL constraints provided by the new SQL standard. Since the reduction is effective, the more specialized problem is also undecidable. The reduction uses the following idea: Functional dependencies are mapped to key or candidate key constraints and referential integrity constraints are used to model inclusion dependencies. The mapping of functional dependencies to keys (resp. candidate keys) is done in two steps

1. Each such dependency $(a_1, \ldots, a_n) \rightarrow (a_{n+1})$ is mapped to a separate relations $t$ with the key $(a_1, \ldots, a_n)$.
2. Because an attribute may occur within several functional dependencies, it has to be guaranteed that the combinations of values used within the different relation generated by the first step are the same. Therefore given two functional dependencies $f_a = (a_1, \ldots, a_n) \rightarrow (a_{n+1})$ and $f_b = (b_1, \ldots, b_k) \rightarrow (b_{k+1})$ (with corresponding relation $t_a$ and $t_b$) which overlap in some attributes $c_1, \ldots, c_l$ we include the following two *inclusion dependencies* to guarantee the same usage of values:
   $t_a \downarrow c_1, \ldots, c_l \subseteq t_b \downarrow c_1, \ldots, c_l$ and
   $t_b \downarrow c_1, \ldots, c_l \subseteq t_a \downarrow c_1, \ldots, c_l$.

These mappings guarantee to represent all functional dependencies *and* all lossless joins.
The mapping of functional dependencies is carried out in a first step, therefore we can base the following steps on a schema which includes only key constraints and inclusion dependencies. The basis of the mapping of inclusion dependencies are the referential integrity constraints. But those constraints differ from inclusion dependencies in the key constraint on the referenced group of attributes implied by a referential integrity constraint. Hence, the idea of the modeling of inclusion dependencies with referential integrity constraints is to combine the attributes of the inclusion dependency with a special key. The latter ensures the key constraint[12] while the former ensures the inclusion dependency.
To do this formally we introduce a function $\Re$ which maps relational schemas with inclusion dependencies to relational schemas with referential integrity constraints. In order to map the inclusion dependencies to referential integrity constraints we introduce for each relation $r$ a new candidate key attribute $a_{new}$ (ranging over a new domain). Furthermore, an new attribute $(a_{i,new})$ is introduced for each inclusion dependency $i$ with attributes of $r$ on the left hand side which also ranges over the new domain. These attributes are used to derive referential integrity constraints from the inclusion dependencies. Following definitions formalize the introduced ideas.

[12] The referenced group of attributes is not minimal.

Let $\mathcal{C}$ be the set of inclusion dependencies
$i_{m,m',k} = r_m(a_{m,k,1}, \ldots, a_{m,k,n_k}) \subseteq r_{m'}(a_{m',k,1}, \ldots, a_{m',k,n_k})$ and $\mathcal{N}$ the set of NULL constraints.
Then $\Re_{schema}(\rho, \mathcal{C}, \mathcal{N})$ is defined as follows:
$\Re_{schema}(\rho, \mathcal{C}, \mathcal{N}) = (\hat{\rho}, \hat{\mathcal{C}}, \hat{\mathcal{N}})$ ($\rho = (\mathcal{D}, \mathcal{A}, \mathcal{T}, \mathcal{K})$ and $\hat{\rho} = (\hat{\mathcal{D}}, \hat{\mathcal{A}}, \hat{\mathcal{T}}, \hat{\mathcal{K}})$), where

- $\hat{\mathcal{D}} = \mathcal{D} \cup \{d_{new}\}, d_{new}$ is a new domain.
- $\hat{\mathcal{A}} = \mathcal{A} \bigcup_{t_m \in \mathcal{T}} \{a_{m,new} :\rightarrow d_{new}\} \bigcup_{i_{m,m',k} \in \mathcal{C}} \{a_{fk_{m,m',k}} :\rightarrow d_{new}\}$.
- $\hat{\mathcal{T}} = \bigcup_{t_m \in \mathcal{T}} \{\{a_{m,new}\} \cup \{a_{m1}, \ldots, a_{mn_m}\} \cup_{i_{m,m',k} \in \mathcal{C}} \{a_{fk_{m,m',k}}\}\}$.
- $\hat{\mathcal{K}} = \mathcal{K} \cup \{k_{m,l'} \mid k_{m,l'} = \{a_{m,new}\}\}$.
- $\hat{\mathcal{C}} = \bigcup_{i_{m,m',k} \in \mathcal{C}} \{r_m(a_{fk_{m,m',k}}, a_{m,k,1}, \ldots, a_{m,k,n_k}) \sqsubseteq r_{m'}(a_{m',new}, a_{m',k,1}, \ldots, a_{m',k,n_k})\}$.
- $\hat{\mathcal{N}} = \mathcal{N} \bigcup_{i_{m,m',k} \in \mathcal{C}} \{r_m(a_{fk_{m,m',k}})\}$.

An instance $\mathcal{I}$ of a relational schema $\rho$ is mapped by the function $\Re_{instance}(\rho, \mathcal{D}, \mathcal{N}, \mathcal{I})$ to $\hat{\mathcal{I}}(\overline{\hat{d}_1}, \ldots, \overline{\hat{d}_k}, \overline{\hat{d}_{new}})$ which is defined as follows:

1. $\overline{\hat{d}_i} = \overline{d_i}$ for $1 \leq i \leq k$ and $\overline{\hat{d}_{new}}$ is a countable infinite set with a complete order $\preceq$ .
2. All tuples of $r_m$ are mapped to the corresponding relation in $\hat{\rho}$ (denoted as $\hat{r}_m$). The attributes $a_{m,new}$ and possible attributes $a_{fk_{m,m',k}}$ are set to the null-value [13].
3. For each tuple $\hat{v}$ of a particular relation $\hat{r}_m$, set $\hat{v}.a_{m,new}$ to a value which is unique within $\hat{r}_m$.
4. Let $v$ be a tuple of a relation $r_m$ (the original relation). In case there is an inclusion dependency
$i_{m,m',k} = r_m(a_{m,k,1}, \ldots, a_{m,k,n_k}) \subseteq r_{m'}(a_{m',k,1}, \ldots, a_{m',k,n_k})$
and $v\downarrow_{\{a_{m,k,1},\ldots,a_{m,k,n_k}\}}$ does not include a null-value, then there exists a tuple $\tilde{v}$ in $r_{m'}$ with $v\downarrow_{\{a_{m',k,1},\ldots,a_{m',k,n_k}\}} = \tilde{v}\downarrow_{\{a_{m,k,1},\ldots,a_{m,k,n_k}\}}$ [14] and the attribute $a_{fk_{m,m',k}}$ in $\hat{v}$ (the mapped tuple) is set to the value of $\hat{\tilde{v}}.a_{m',new}$. If $v\downarrow_{\{a_{m,k,1},\ldots,a_{m,k,n_k}\}}$ includes a null-value, then $\hat{v}.a_{fk_{m,m',k}}$ is set to an arbitrary value of the domain $\overline{\hat{d}_{new}}$.

With these definitions we can prove the following theorem:

**Theorem 3.** *Given a relational schema $\rho$, a set $\mathcal{C}$ of inclusion dependencies and a set $\mathcal{N}$ of NULL constraints. The following holds: $\mathcal{I}$ is an instance of $\rho$ satisfying $\mathcal{C}$ and $\mathcal{N}$ $\iff$ $\Re_{instance}(\rho, \mathcal{D}, \mathcal{N}, \mathcal{I}) = \hat{\mathcal{I}}$ is an instance of $\Re_{schema}(\rho, \mathcal{C}, \mathcal{N})$.*

*Proof. if* part ("$\Leftarrow$"): Suppose $\hat{\mathcal{I}}$ is an instance of $\Re_{schema}(\rho, \mathcal{C}, \mathcal{N})$ and $\mathcal{I}$ is no instance of $\rho, \mathcal{C}, \mathcal{N}$. While the key constraints cannot be violated by only one of

[13] The key conditions on $a_{m,new}$ and the NULL constraints on the $a_{fk_{m,m',k}}$ are violated at this point.

[14] If there is more than one such tuple, choose one arbitrarily.

the instances, a problem may occur through the inclusion dependencies.
Let $v \in r_m$ be a tuple which violates an inclusion dependency
$i_{m,m',k} = r_m(a_{m,k,1}, \ldots, a_{m,k,n_k}) \subseteq r_{m'}(a_{m',k,1}, \ldots, a_{m',k,n_k})$. By the definition of the inclusion dependency we obtain that there is no null-value in the attributes $\{a_{m,k,1}, \ldots, a_{m,k,n_k}\}$ of $v$. By the definition of $\Re_{instance}$ we have a tuple $\hat{v}$ in $\hat{\mathcal{I}}$ that violates the referential integrity constraint $\Re_{schema}(\ldots, i_{m,m',k}, \ldots)$.
Hence $\hat{\mathcal{I}}$ is no instance of $\Re_{schema}(\rho, \mathcal{C}, \mathcal{N})$.
*only-if* part ("$\Rightarrow$"): Suppose $\mathcal{I}$ is an instance of $\rho, \mathcal{C}, \mathcal{N}$ and $\hat{\mathcal{I}}$ is no instance of $\Re_{schema}(\rho, \mathcal{C}, \mathcal{N})$. The contradiction follows directly from the definition of $\Re_{instance}$. □

**Corollary 4.** *Given a relational schema $\rho$, a set $\mathcal{C}$ of inclusion dependencies, a set $\mathcal{N}$ of NULL constraints and an inclusion dependency $i$. Then the following holds (for $\hat{\rho}, \hat{\mathcal{D}}, \hat{\mathcal{N}}$ defined as above): $\rho, \mathcal{D}, \mathcal{N} \models i \iff \hat{\rho}, \hat{\mathcal{D}}, \hat{\mathcal{N}} \models i$.*

*Proof.* This follows directly from the above theorem. □

Corollary 4 immediately implies theorem 1.

## 5 Conclusions and Outlook

Integrity constraints are fundamental concepts for modeling the real world. This is especially true for database systems. In such systems every committed database state has to obey all specified constraints. Therefore, it is desirable to have an automatic maintenance if a constraint is violated (e.g., through a user operation). For this purpose, the new SQL2 standard, as a specification of a language for the relational data model, includes referential integrity and its extensions.
In this paper, we have shown that the outcome of a user operation may be indeterministic because of these extensions which was also observed in the framework of [13, 17]. Similar effects were analyzed in the area of rule processing [2, 1, 20]. Although the problem of deciding whether or not a database schema allows operations with an indeterministic outcome was known to be unsolvable in general, we hoped to find a solution for simple integrity constraints like referential integrity constraints because they are tightly connected to the structure of a database schema. As proved in this paper, even this restricted case is unsolvable in general and therefore only a subset of the "safe" schemas may be recognized e.g. by the criterions of [13] or [17]. Hence, as long as the known rule processing algorithms are used, situations where ambiguities occur cannot be ruled out. In those cases one of the following alternatives has to be chosen:

1. Live with the danger
   Nothing is done. Therefore, the database system only guarantees that each committed state obeys all defined integrity constraints. Which result state is achieved using the active maintenance facilities remains unspecified and is therefore not known to the user.

2. Expand the syntax (and semantics) of the rules
   The rule syntax (and as a consequence the semantics) may be enhanced with the notion of explicit conflict resolution, e.g., priority assignment. But as a result, the rules introduced as local autonomous elements of integrity maintenance may degenerate to a form of a complex (unreadable) if-then-else programming paradigm. To prevent such a deterioration, only conflicting situations should be burdened with these "enhancements"[15]. Therefore, a good (read: sharp) detection algorithm for ambiguities on a schema basis remains desirable.
3. Live with *some* limitations
   Opposed to the second case where all difficulties are solved through amendments to the constraints, in this case the operation is backed out completely after the detection of an ambiguity. Hence a good detection algorithm is needed. This is the way SQL2 deals with the ambiguity problem: Whether or not an ambiguity occurs has to be detected at run-time. This causes coordination overhead only in case an ambiguity occurs. Therefore, less coordination is required than in the second case. On the other hand, the need of a check at run-time arises. To optimize this overhead a good compile-time detection is needed.

Since a general requirement in the database context are deterministic results of operations the first case is ruled out. Both other cases require a good detection algorithm. Since the general problem is undecidable, it is an interesting problem to determine the largest subset of database schemas that remains decidable.
Furthermore, we want to elaborate on the problem how the semantics of the integrity constraints proposed in SQL2 and SQL3 can be supported in an efficient way. It seems reasonable to use general ECA rules. For this purpose, we develop the required deterministic semantics for such rules. This approach will allow an autonomous definition of rules, do some compile-time checking (of rules and operations) and, if ambiguities are possible, include an appropriate check in the evaluation that provides a coordination at run-time. Implementation concepts are developed in parallel. Together these efforts will allow an answer to the question, whether such deterministic semantics can be achieved in an effective manner acceptable for an end-user.

## Acknowledgements
We would like to acknowledge Prof. T. Härder, B. Reinert, H. Schöning and the unnamed referees for reading the preliminary version of the paper and providing useful hints to improve it.

## References

1. A. Aiken, J. M. Hellerstein, and J. Widom. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM TODS*, 20(1):3–41, March

[15] Note that the SQL2 semantics of referential integrity cannot be achieved using priorities.

1995.
2. A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. *ACM SIGMOD Record*, 20(2):59–68, June 1992.
3. A.K. Chandra and M. Y. Vardi. The Implication Problem for Functional and Inclusion Dependencies is Undecidable. *ACM SIAM*, 14(3):671–677, 1985.
4. E. F. Codd. A Relational Model of Data for Large Shared Databases. *Communication of the ACM*, 13(6):377–387, 1970.
5. C. J. Date. Referential Integrity. In *Proceedings of the VLDB'81*, pages 9–35. IEEE, March 1981.
6. C. J. Date. *A Guide to THE SQL STANDARD.* Addision-Wesley, New York, 1988.
7. C. J. Date. *Relational Databases: Selected Writings 1985-1989.* Addision-Wesley, New York, 1990.
8. K. P. Eswaran. Specification, Implementation and Interactions of a Trigger Subsystem in a Integrated Database System. IBM Research Report RJ-1820(26414), IBM Almaden Research Center, IBM Research Laboratory, San Jose, California 95193, 1976.
9. M. M. Hammer and D. J. McLeod. Semantic Integrity in a Relational Data Base System. In *Proceedings of the VLDB'75*, pages 25–47, 1975.
10. T. Härder and J. Reinert. Access Path Support for Referential Integrity in SQL2. Research Report, Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, D-W 6750 Kaiserslautern, Germany, 1993.
11. ISO/IEC JTC1/SC21. *Information Technology - Database Languages - SQL2.* ANSI, American National Standards Institute, 1430 Broadway, New York, NY 10018, July 1992.
12. ISO/IEC JTC1/SC21/WG3. *ISO/ANSI working draft Database Languages - SQL3.* ANSI, American National Standards Institute, 1430 Broadway, New York, NY 10018, February 1993.
13. V. M. Markowitz. Referential Integrity Revisited: An Object-Oriented Perspective. In *Proceedings of the VLDB'90, Australia*, pages 9–35, 1990.
14. V. M. Markowitz. Safe Referential Integrity Structures. In *Proceedings of the VLDB'91, Barcelona, Spain*, 1991.
15. V. M. Markowitz. Safe Referential Integrity and Null Constraints in Relational Databases. *Information Systems*, 19(4):359–378, 1994.
16. G. Moerkotte. *Inkonsitenzen in deduktiven Datenbanken.* Springer-Verlag, Heidelberg, 1990. in German.
17. J. Reinert. Ensuring Referential Integrity in SQL2 and SQL3. Research Report, University of Kaiserslautern, 1992.
18. J. Widom and S. J. Finkelstein. Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems. IBM Research Report 6880, IBM Almaden Research Center, IBM Research Laboratory, San Jose, California 95193, 1989.
19. J. Widom and S. J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. *ACM SIGMOD Record*, 19(2):259–270, June 1990.
20. Y. Zhou and M. Hsu. A Theory for Rule Triggering Systems. In *Proceedings of the EDBT 92, LNCS 416*, pages 407–421. Springer-Verlag, Berlin, March 1990.