Simplification of database integrity constraints revisited: a transformational approach

Henning Christiansen and Davide Martinenghi

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: {henning,dm}@ruc.dk

Abstract. Complete checks of database integrity constraints may be prohibitively time consuming, and several methods have been suggested for producing simplified checks for each update. The present approach introduces a set of transformation operators that apply to database integrity constraints with each operator representing a concise, semantics-preserving operation. These operators are applied in a procedure producing simplified constraints for parametric transaction patterns, which then can be instantiated and checked for consistency at run-time but before any transaction is executed. The operators provide a flexibility for other database enhancements and the work may also be seen as more systematic and general when compared with other approaches. The framework is formulated with first-order clause logic but with the perspective of being applied with present-day database technology.

1 Introduction

Simplification of integrity constraints is a principle that has been recognized for more than two decades, dating back to at least [20], and elaborated by several other authors. Despite a general recognition, it has not gained ground in standard databases. Our work is an attempt to reconcile and generalize such ideas in a systematic way that may promote practical applications with current database management technology.

An integrity constraint is a logical formula, typically depending on the nature of the application domain, that must hold for any database state for it to represent a meaningful set of data. Integrity constraints often concern the entire database and require linear or worse time complexity for a complete check, which is prohibitive in any non-trivial case. Simplification in this context means to derive specialized versions of the integrity constraints that can be checked more efficiently at each update, employing the hypothesis that the database is consistent before the update itself. Ideally, the simplified constraint should be a test that can be generated at database design time and that can be executed before a potentially offensive update is performed, so that rollback operations become unnecessary.

This paper introduces a framework providing a set of semantics-preserving program transformations by means of which an effective simplification procedure as well as other optimizations and database enhancements can be defined. Integrity constraints, here in the form of denial clauses, are considered as programs in the sense that they can be executed as Prolog queries that must fail or, more in the line with present-day database technologies, as SQL queries that must return the empty answer. Unlike some previous approaches, which only consider single updates, our simplification procedure (and the individual transformations) applies to transaction patterns. Once a specific transaction is proposed, and *before* it is executed, the simplified formulas can be evaluated as pieces of code integrated in database application programs so that only consistency-preserving transactions are eventually given to the database.

We also illustrate other applications of our framework combined with other techniques such as data mining, abductive reasoning and data integration.

The paper is organized as follows. In section 2 we review existing literature in the field. The notation and theoretical setting are introduced in section 3, while in section 4 the program transformations and the simplification procedure are formally defined. Other applications are discussed in section 5 and concluding remarks are provided in section 6.

2 Motivation and related works

Simplification of integrity constraints, is highly relevant for optimizations in database integrity checking. Typically it gives a speed-up of a linear factor (in the size of the database state) for singleton updates but for certain transactions an even higher speed-up can be gained. We find crucial the ability to check consistency of a possibly updated database *before* execution of the transaction under consideration so that inconsistent states are completely avoided. Several approaches to simplification first require the transaction to be performed, and *then* the resulting state to be checked for consistency [20, 18, 23, 8, 11].

The proposal of [14] presents many analogies with our method, although the referenced paper does not present a fully developed method. A series of tests is generated from an integrity constraint C and an update U; if one of these tests succeeds, then U is legal with respect to C. This method is based on *resolution* and *transition axioms*, which provide an effect similar to our After operator (introduced in section 4 below). Updates are limited to be single actions, but can contain so-called dummy constants, very similar to our notion of parameters. The main disadvantage of this approach is that once the set of tests is generated, strategies have to be employed to decide which specific tests to execute and in which order. Furthermore, failure of all tests does not necessarily imply inconsistency. Our simplification algorithm is more straightforward in that it generates a single test whose result is a necessary and sufficient condition for determining consistency of the database if the update were performed.

Grant and Minker [11] introduce a principle called partial subsumption applied among other things to produce simplified integrity constraints. This method applies to singleton additions or deletions and produces conditions to be applied *after* the update; it also handles parametric updates expressed using logical variables. For compound updates (transactions) the principle is explained in terms of examples, but no general procedure is described. Partial subsumption applies also to semantic query optimization; see [12, 10] for an overview.

Qian [22] observes the relationship between Hoare's logic [13, 9] for imperative languages and integrity checking, identifying a simplified integrity constraint as a weakest precondition for having a consistent updated state. This notion is enforced by assuming the consistency of the database before the update. We give a more detailed discussion of this issue in section 3. Qian's method works for a variety of SQL-like ways of updating relations but with the impractical limitation that it does not allow more than one update action in a transaction to operate on the same relation; furthermore, no mechanism corresponding to parameters is present, thus requiring to execute the procedure for each update. We have no such restrictions.

Another problematic issue inherent in most work in the field is the lack of a characterization of what it means for a formula to be simplified; this is traditionally defined in terms of a semantic criterion. Our view is that a transformed integrity constraint, in order to qualify as "simplified", must represent a minimum in some ordering that reflects the effort of actually evaluating it. We propose a simple ordering based on the number of literals but we have no proof that our own algorithm hits a minimum in all cases.

Standard ways of translating integrity constraints into SQL exist. In a recent paper [7], Decker shows how to implement integrity constraint checking by translating first-order logic specifications into SQL triggers. In this way the advantages of declarativity are combined with the efficiency of execution. It is interesting to note that the result of our transformations can be combined with similar translation techniques and thus integrated in a database system. An extension of trigger syntax is proposed in [7] that allows the specification of the positions of the arguments of a relation that are relevant for an update. Our approach using parameters clearly subsumes this one.

The use of constraint techniques for abduction in logic programming may display an incremental evaluation of integrity constraints without an explicit simplification algorithm: each time an abducible atomic update a arises, the current representation of the integrity constraints wakes up, checks a's dependencies and, in case of success, delays a specialized version of the integrity constraints waiting for the next update. This principle is applied in the DemoII system [2, 5] and in the approach of [1] using Constraint Handling Rules for abduction; [3] is an attempt to relate such methods to database applications. However, a common drawback of these techniques is that the delayed constraints typically unroll to a size proportional to the database and that, occasionally, an unsatisfiable set of constraints is delayed where a failure should be reported.

With few exceptions (e.g., [17, 16, 15, 4]), little attention has been devoted to the problem of checking the integrity of a database containing recursive views, although recursion is now part of the current SQL standard. Consider, for example, a directed graph in which a path between two nodes is recursively expressed as the transitive closure of the edge relation and suppose that the graph is acyclic. If a new edge $a \mapsto b$ is added, the optimal way to check whether the new graph still is acyclic is to verify that there is currently no path connecting b to a. None of the methods we are aware of is able to provide such a simplified check. When recursive rules are present, the methods described in [17, 4] produce a set of constraints which is typically the same as the original one, i.e., no actual simplification takes place during the application of the procedure. In [16], low-cost pre-tests are generated which are sufficient conditions that guarantee the integrity of the database; however, these pre-tests typically fail in the presence of recursion, so nothing can be concluded about consistency. In [15], partial evaluation is applied to a general integrity checker to generate logic programs that correspond to simplified constraints. Generally, it is difficult to evaluate the method described in [15], as it depends on a number of heuristics in the partial evaluator as well as in the general checker; furthermore, no results are reported for recursive databases. The method we present here applies to non-recursive databases only.

Technical difficulties related to undecidability may also hinder the realization of a perfect simplification method (see subsection 4.4).

We are not aware of any other approach that reduces simplification into a combination of well-defined program transformations, which furthermore serve as an "algebra" in which different optimizations and interesting transformations can be described (see section 5). An experimental prototype [19] implementing a simplified version of these transformations is available on the World Wide Web.

Finally, we emphasize that we only consider integrity constraints expressed relative to all states; constraints that compare successive states, by some authors called dynamic or transactional integrity constraints, are not considered.

3 Preliminaries

Assume a function-free first-order language equipped with negation and predicates for equality ($\doteq$) and inequality ($\neq$), called *predefined predicates*. *Terms* are either *variables* $(x, y, \ldots)$ or *constants* $(a, b, \ldots)$. However, special constants called *parameters* are written in boldface $(\mathbf{a}, \mathbf{b}, \ldots)$; constants that are not parameters are called *ground* constants. *Predicates* $(p, q, \ldots)$ are used to build *atoms*, i.e. expressions of the form $p(t_1, \ldots, t_n)$, where the t_i's are terms and $n \geq 0$. A predicate that is not predefined is called a *database predicate*. *Formulas* are formed as usual from the atoms and the logical connectives. A formula is ground if its terms are only ground constants. A *literal* is either an atom A or a negated atom $\neg A$; whenever $\neg\neg A$ appears, it should be read as A and $\neg s \doteq t$ as $s \neq t$ and $\neg s \neq t$ as $s \doteq t$. *Clauses* are written in the form *Head* $\leftarrow$ *Body* where the head is an atom and the body a (perhaps empty) conjunction of literals; the head may be left out, understood as *false*, in which case the clause is a *denial*; the body may be left out, understood as *true*, in which case the clause is a *fact*; any other clause is a *rule*. (In)equalities are not allowed in the head and are assumed with their usual meaning of syntactic (in)equality, but the order in which the arguments of $\doteq$ and $\neq$ are written does not matter. Logical equivalence between parameter-free formulas is denoted by $\equiv$. The notation $\vec{t}$ indicates a sequence of terms $t_1, \ldots, t_n$ and $p(\vec{t})$ an atom whose arguments are $t_1, \ldots, t_n$. The expression $\vec{t} \doteq \vec{s}$ is a shorthand for $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$ where t_i's and s_i's are the terms of the two sequences; in a similar way, $\vec{t} \neq \vec{s}$ refers to the disjunction of individual inequalities.

Definition 1 (Parametric instance and equivalence). *For any expression E with parameters $\vec{\mathbf{a}}$ and sequence of constants $\vec{c}$ of the same length as $\vec{\mathbf{a}}$, the notation $E_{\vec{\mathbf{a}}/\vec{c}}$ refers to the expression that arises from E when each element of $\vec{\mathbf{a}}$ is replaced consistently by the matching element of $\vec{c}$; $E_{\vec{\mathbf{a}}/\vec{c}}$ is called a* (parametric) instance *of E.*

Two formulas F and G are equivalent up to instantiation of parameters*, written $F \cong G$, whenever $F' \equiv G'$ for all parametric instances (F', G') of (F, G).*

Clearly $\cong$ is symmetric, reflexive and transitive. Note that name uniqueness is assumed for all ground constants but not for parameters, as different parameters may be instantiated by the same ground constant.

Example 1. Let $\mathbf{a}$, $\mathbf{b}$ be two parameters and c, d two ground constants. Then both $p(c, d, c)$ and $p(c, c, c)$ are instances of $p(\mathbf{a}, \mathbf{b}, \mathbf{a})$, whereas $p(c, c, d)$ is not. We have $c \doteq d \cong \mathit{false}$ but neither $\mathbf{a} \doteq \mathbf{b} \cong \mathit{false}$ nor $\mathbf{a} \doteq \mathbf{b} \cong \mathit{true}$. Note that two formulas being parametrically equivalent does not necessarily indicate that they contain the same parameters, e.g. $\mathbf{a} \doteq \mathbf{a} \cong \mathit{true}$. $\square$

We further assume that all clauses are *range restricted*, as defined below.

Definition 2 (Range restriction). *A variable in a clause is* range bound *if it appears in a positive database literal in the body. A clause is* range restricted *if all variables in it are range bound.*

Notice that parameters in this definition are treated in the same way as ground constants. As already stated, we do not allow recursion, but our method is relevant for all database environments in which range restricted queries produce a finite set of ground tuples. The notion of *subsumption* is applied repeatedly in this paper.

Definition 3 (Subsumption). *A clause C_1* subsumes *another C_2 iff there is a substitution σ such that each literal in the body of $C_1\sigma$ occurs in the body of C_2 and similarly for the heads.*

Example 2. The clause $\leftarrow p(x, y) \wedge a \neq x$ subsumes $\leftarrow p(x, b) \wedge x \neq a \wedge q(b)$. $\square$

The definition of subsumption is syntactic but has the semantic property that the subsuming clause implies the subsumed one.

Complying with [10], a *database* is characterized by three components:

- a set of facts, called the *extensional database*;
- a set of rules (the *intensional database*);
- a set of integrity constraints (here denials), known as the *constraint theory.*

Parameters cannot occur in a database but the transformation operators may produce integrity constraints that contain parameters. We call these *parameterized* integrity constraints. In a recursion-free language, we can limit our attention, without any loss of generality, to integrity constraints that refer to extensional predicates only, as intensional predicates can be (repeatedly) replaced with their definitions, that will eventually only be extensional. We will therefore keep this assumption throughout the rest of the paper. By *database state* we refer to the union of the extensional and the intensional parts only.

As semantics of a database state D, with default negation for negative literals, we take its *standard model*, denoted $\mathcal{M}_D$, as D is here recursion-free and thus *stratified*. The truth value of a closed formula F, relative to D, is defined as its valuation in $\mathcal{M}_D$ and denoted $D(F)$. (See e.g. [21] for exact definitions.) The meaning of a formula that includes parameters can be thought of as an operator taking instantiations of the parameters and producing a truth value. In the following, the overloaded notation $D_1(F_1) \cong D_2(F_2)$ will indicate that $D_1(F_1') = D_2(F_2')$ holds for all parametric instances (F_1', F_2') of (F_1, F_2), where D_1 and D_2 are database states and F_1 and F_2 are formulas. Consistency of the integrity constraints can be defined in different ways; we follow [10] arguing that the following is the most natural choice.

Definition 4 (Consistency). *A database state D is* consistent *with a constraint theory Γ iff $D(\Gamma) = true$.*

Definition 5 (Update and update pattern). *An* update $U = U^+ \cup U^-$ *is a non-empty set of* additions U^+ *and* deletions U^-*, both consisting of ground facts, with the deletions indicated by a $\neg$ sign. The* reverse *of an update U, denoted $\neg U$, contains the same elements as U but with the roles of additions and deletions interchanged. The additions and deletions of an update are required to be disjoint, i.e. $U^+ \cap \neg U^- = \emptyset$. The notation $D \cup U$, where D is a database state, is a shorthand for $(D \cup U^+) \setminus \neg U^-$. An* update pattern *is an expression whose parametric instances are updates.*

This definition of update fits with all cases where the additions and deletions are known independently of the database state. This is not always the case. For example, given the statement "delete all records of computer science books from the library", which is easily expressible in SQL, the set of tuples that will be deleted depends on the actual database state. Our method can be generalized to such more general updates including also SQL's `UPDATE`, but for reasons of space this is omitted in the present version of the paper.

As already emphasized, it is important to be able to test that a prospective database update does not violate the integrity constraints — without actually executing the update, i.e., a test is needed that can be checked in the present state but indicating properties of the prospective new state. A semantic correctness criterion for such a test is given by the notion of weakest precondition.

Definition 6 (Weakest precondition, strongest postcondition). *Let Γ and Γ' be constraint theories and U an update pattern. Γ' is a* weakest precondition *of Γ with respect to U and Γ is a* strongest postcondition *of Γ' with respect to U whenever $D(\Gamma') \cong (D \cup U)(\Gamma)$ for any database state D.*

As also noticed by Qian [22], this definition is similar to the standard axiom for defining assignment statements in a programming language [9], whose side effects are analogous to a database update. Hoare's [13] original version of the axiom used only implication from pre- to postcondition; the notion of *weakest* precondition that we need is due to Dijkstra [9].

The concept of strongest postcondition is not used for simplification but is useful for other purposes. It characterizes how questions concerning the previous state may be answered by considering transformed questions in the updated state. The essence of simplification is the optimization of a weakest precondition based on the invariant that the constraint theory holds in the present state. The semantic characterization of this property is defined as follows.

Definition 7 (Conditional weakest precondition). *Let Γ be a constraint theory and U an update pattern. A constraint theory Γ' is a* conditional weakest precondition *of Γ with respect to U whenever $D(\Gamma') \cong (D \cup U)(\Gamma)$ for any database state D consistent with Γ.*

A weakest precondition is also a conditional weakest precondition but not necessarily the other way round. All other known definitions of simplification are based solely on this or similar semantic notions, but this is not sufficient: a characterization should be given of the sense in which the resulting formula is actually "simpler" than the original one. In example 3 we show that a criterion based on semantic weakness does not capture the intuition behind simplicity.

Example 3. In case a theory Γ_1 holds in more states than another Γ_2, we say that Γ_1 is weaker than Γ_2 and that Γ_2 is stronger than Γ_1. Consider the constraint theory $\Gamma = \{\leftarrow p(a) \wedge q(a),\ \leftarrow r(a)\}$ and the update $U = \{p(a)\}$. The strongest, intuitively simplest, and weakest conditional weakest preconditions of Γ with respect to U are shown in the following table.

Strongest	Simplest	Weakest
$\{\leftarrow q(a),\ \leftarrow r(a)\}$	$\{\leftarrow q(a)\}$	$\{\leftarrow q(a) \wedge \neg p(a) \wedge \neg r(a)\}$

□

We shall use a syntactic selection criterion instead, as discussed in detail in section 4.2.

4 Transformations on integrity constraints

In the following, we define four syntactic transformation operators, each performing a well-defined function that satisfies straightforward semantic conditions. With these, we can compose a simplification procedure and, as shown in section 5, other useful transformations of integrity constraints.

4.1 Translation of integrity constraints back and forth between different states

To consider whether a given property holds after a prospective update means to reason about the truth of a formula in a future state, but arguing in the present state.

The following After operator translates a constraint theory into a weakest precondition with respect to a given update pattern in a straightforward way; it does not assume consistency of the present state and is, thus, applicable for other things than plain simplification. By means of it, we define also a dual operator Before that translates a constraint theory Γ into another theory that can be used after the update to test whether Γ held before the update; it is not used in the simplification procedure, but it proves useful for other purposes, as shown in section 5.

Definition 8. *Consider an update pattern U:*

$$\begin{aligned} U = \{\; & p_1(\vec{a}_{1,1}), p_1(\vec{a}_{1,2}), \ldots, p_1(\vec{a}_{1,n_1}),\\ & p_2(\vec{a}_{2,1}), p_2(\vec{a}_{2,2}), \ldots, p_2(\vec{a}_{2,n_2}),\\ & \ldots\\ & p_k(\vec{a}_{k,1}), p_k(\vec{a}_{k,2}), \ldots, p_k(\vec{a}_{k,n_k}),\\ & \neg p_1(\vec{b}_{1,1}), \neg p_1(\vec{b}_{1,2}), \ldots, \neg p_1(\vec{b}_{1,m_1}),\\ & \neg p_2(\vec{b}_{2,1}), \neg p_2(\vec{b}_{2,2}), \ldots, \neg p_2(\vec{b}_{2,m_2}),\\ & \ldots\\ & \neg p_k(\vec{b}_{k,1}), \neg p_k(\vec{b}_{k,2}), \ldots, \neg p_k(\vec{b}_{k,m_k})\}, \end{aligned}$$

where the p_i's are distinct predicates and the $\vec{a}_{i,j}$'s and $\vec{b}_{i,j}$'s are sequences of constants. For a constraint theory Γ, the notation $\mathsf{After}^U(\Gamma)$ *refers to the set of denials Γ' obtained as follows.*

1. *Let Γ' consist of a copy of Γ in which all occurrences of an atom of the form $p_i(\vec{t})$ have been simultaneously replaced by*

$$(p_i(\vec{t}) \wedge \vec{t} \neq \vec{b}_{i,1} \wedge \cdots \wedge \vec{t} \neq \vec{b}_{i,m_i}) \vee \vec{t} \doteq \vec{a}_{i,1} \vee \cdots \vee \vec{t} \doteq \vec{a}_{i,n_i}.$$

2. *Do the following in Γ' as long as possible (A, B_1, B_2 and C are formulas):*
   – *Replace any formula in Γ' of the form $\leftarrow A \wedge (B_1 \vee B_2) \wedge C$ by the two formulas $\leftarrow A \wedge B_1 \wedge C$ and $\leftarrow A \wedge B_2 \wedge C$.*
   – *Replace any formula in Γ' of the form $\leftarrow A \wedge \neg(B_1 \vee B_2) \wedge C$ by the formula $\leftarrow A \wedge \neg B_1 \wedge \neg B_2 \wedge C$.*
   – *Replace any formula in Γ' of the form $\leftarrow A \wedge \neg(B_1 \wedge B_2) \wedge C$ by the two formulas $\leftarrow A \wedge \neg B_1 \wedge C$ and $\leftarrow A \wedge \neg B_2 \wedge C$.*

The notation $\mathsf{Before}^U(\Gamma)$ *refers to the set of denials* $\mathsf{After}^{\neg U}(\Gamma)$.

The intermediary formulas with disjunctions resulting from step 1 may not be clauses, but step 2 takes care of restoring the clausal form. The following properties follow immediately from the definition of After and are stated without proof.

Proposition 1 (Composition of After over denials). *For any update U and constraint theories Γ_1 and Γ_2 the following property holds:*

$$\mathsf{After}^U(\Gamma_1 \cup \Gamma_2) \quad = \quad \mathsf{After}^U(\Gamma_1) \cup \mathsf{After}^U(\Gamma_2).$$

Proposition 2 (Composition of After over updates). *For any constraint theory Γ and updates U, U_1, U_2, with $U = U_1 \cup U_2$ and U_1 and U_2 disjoint, we have*

$$\mathsf{After}^U(\Gamma) \quad \cong \quad \mathsf{After}^{U_2}(\mathsf{After}^{U_1}(\Gamma)) \quad \cong \quad \mathsf{After}^{U_1}(\mathsf{After}^{U_2}(\Gamma)).$$

The semantic correctness of After is expressed by the following property.

Theorem 1 (After produces weakest precondition). *For any update U and constraint theory Γ, $\mathsf{After}^U(\Gamma)$ is a weakest precondition of Γ with respect to U.*

Proof. We need to show that $D(\mathsf{After}^U(\Gamma)) \cong (D \cup U)(\Gamma)$ for any database state D. Step 2 of definition 8 obviously preserves semantics so we can ignore it. Composition over updates means that we need only consider singleton updates; we start considering a positive update of the form $U = \{p(\vec{a})\}$.

For any database predicate q assume another predicate q' of the same arity, and when writing Φ' for some formula or set of formulas Φ (assumed not to include such primed predicates), we refer to a formula similar to Φ with all occurrences of database predicate q replaced by q'. Let Π be the following program that defines a way of going back and forth between the two classes of predicates.

$$\begin{array}{l} \{p'(\vec{x}) \leftrightarrow \vec{x} \doteq \vec{a} \vee p(\vec{x})\} \ \cup \\ \{q'(\vec{x}) \leftrightarrow q(\vec{x}) \mid \text{for any database predicate } q \text{ different from } p\} \end{array}$$

The theory $D \cup \Pi$ is a combined representation of the states before and after the update, so that formulas without primes are evaluated as in the state before and primed ones as in the state after. Formally we have the following equivalences, where ϕ is any formula without primed predicates.

$$\begin{array}{l} (D \cup \Pi)(\phi) \ \cong D(\phi) \\ (D \cup \Pi)(\phi') \cong (D' \cup U')(\phi') \cong (D \cup U)(\phi) \end{array}$$

To see that the last line holds, notice that the first two members are equivalent because they have the same definitions for all primed predicates: in the former p' holds when p holds or when its argument is $\vec{a}$ and in the latter p' is just a renaming of p plus the fact $p'(\vec{a})$; all other primed predicates are evidently the same. The last two members are equivalent as they differ only by a consistent renaming of symbols. The two expressions (on the left-hand sides below) that we need to prove equivalent can be represented in the combined theory as follows:

$$\begin{array}{ll} D(\mathsf{After}^U(\Gamma)) & \cong (D \cup \Pi)(\mathsf{After}^U(\Gamma)) \\ (D \cup U)(\Gamma) & \cong (D \cup \Pi)(\Gamma') \end{array}$$

However, the two right-hand sides are equivalent as $\mathsf{After}^U(\Gamma)$ can be constructed from Γ' by replacement of expressions that are equivalent in $D \cup \Pi$, i.e., replacing $p'(\vec{t})$ by $(\vec{t} \doteq \vec{a} \vee p(\vec{t}))$ and $q'(\vec{t})$ by $q(\vec{t})$ for any predicate q different from p.

For a negative update of the form $U = \{\neg p(\vec{a})\}$ the proof is symmetric, with the definition of p' in Π being $\{p'(\vec{x}) \leftrightarrow \vec{x} \neq \vec{a} \wedge p(\vec{x})\}$. □

We get immediately composition properties of the Before operator and a dual version of theorem 1 stating that Before^U produces strongest postconditions with respect to update U.

Example 4. Consider a database containing information about marriages, where the binary predicate m indicates that a husband (first argument) is married to a wife (second argument). We expect for this database updates of the form: $U = \{m(\mathbf{a}, \mathbf{b})\}$. The following integrity constraint is given:

$$\phi = \leftarrow m(x, y) \wedge m(x, z) \wedge y \neq z$$

meaning that no husband can be married to two different wives. The first step of definition 8 in the calculation of $\mathsf{After}^U(\{\phi\})$ generates the following:

$$\{\leftarrow (m(x, y) \vee (x \doteq \mathbf{a} \wedge y \doteq \mathbf{b})) \wedge (m(x, z) \vee (x \doteq \mathbf{a} \wedge z \doteq \mathbf{b})) \wedge y \neq z\}.$$

The second step translates it to clausal form:

$$\begin{aligned}\mathsf{After}^U(\{\phi\}) = \{ &\leftarrow m(x, y) \wedge m(x, z) \wedge y \neq z,\\ &\leftarrow m(x, y) \wedge x \doteq \mathbf{a} \wedge z \doteq \mathbf{b} \wedge y \neq z,\\ &\leftarrow x \doteq \mathbf{a} \wedge y \doteq \mathbf{b} \wedge m(x, z) \wedge y \neq z,\\ &\leftarrow x \doteq \mathbf{a} \wedge y \doteq \mathbf{b} \wedge x \doteq \mathbf{a} \wedge z \doteq \mathbf{b} \wedge y \neq z \ \}.\end{aligned}$$
□

Example 5. We shall now consider an example of referential integrity, where a relation f (father) is only meaningful if its first argument (the father) is recorded in a relation p (person) with a specific constant value concerning the gender (m for "male"):

$$\phi = \leftarrow f(x, y) \wedge \neg p(x, m).$$

For transactions of the form $U = \{f(\mathbf{a}, \mathbf{b}), p(\mathbf{a}, m)\}$ we have:

$$\begin{aligned}\mathsf{After}^U(\{\phi\}) = \{ &\leftarrow x \doteq \mathbf{a} \wedge y \doteq \mathbf{b} \wedge \neg(x \doteq \mathbf{a}) \wedge \neg p(x, m),\\ &\leftarrow x \doteq \mathbf{a} \wedge y \doteq \mathbf{b} \wedge \neg(m \doteq m) \wedge \neg p(x, m),\\ &\leftarrow f(x, y) \wedge \neg(x \doteq \mathbf{a}) \wedge \neg p(x, m),\\ &\leftarrow f(x, y) \wedge \neg(m \doteq m) \wedge \neg p(x, m)\}.\end{aligned}$$
□

Notice that when a predicate that appears in a positive literal is updated positively by a single addition, $\mathsf{After}^U(\{\phi\})$ contains a copy of ϕ (see example 4), whereas for a predicate that appears in a negative literal, $\mathsf{After}^U(\{\phi\})$ contains a formula that is a specialization of ϕ, even when the update is not singleton (see example 5). The effect is symmetric for deletions.

4.2 Normalization of formulas

The result of the After transformation is obviously not in any "reduced" or "normalized" form, and we introduce an operator Norm to take care of this.

An ideal Norm procedure should produce a constraint theory as output that is minimal in some ordering that reflects an estimate of the time complexity. This can only be an estimate as the actual execution times depend on the database state (that is not available at the time of the simplification process) and is also highly dependent on the applied database technology that may perform optimizations that are not feasible to include in a general definition. We suggest an ordering based on the simple principle of counting literals, although it may be the case that longer constraints are evaluated more efficiently in particular database states (see also section 5.6). We define $\Gamma \prec \Gamma'$ iff Γ has fewer literals than Γ'.

This may appear a bit coarse as it gives the same measure for, say, $\leftarrow 1 \doteq 2$, $\leftarrow p(a)$, and $\leftarrow p(x)$. However, it should be kept in mind that the Norm procedure (as well as the entire simplification process) should return a minimal constraint theory *among those that satisfy the corresponding semantic condition.*

We give a proposal below for a procedure that implements the Norm operation but we do not have a proof that it produces a minimal constraint theory.[1] We are not alone with this problem, as no other work on simplification that we are aware of provides results of this form; in fact, we are not aware of any other work that considers this criterion at all.

In the following definition we also refer to the notion of *expansion* [10]: the expansion of a clause consists in replacing every constant in a database predicate (or variable already appearing elsewhere in database predicates) by a new variable and adding the equality between the new variable and the replaced item. For example, $\leftarrow p(x, a, x)$ can be expanded to $\leftarrow p(x, y, z) \land y \doteq a \land z \doteq x$.

Definition 9 (Normalization). *For a constraint theory Γ, let $\mathsf{Norm}(\Gamma)$ be the result of iterating the following steps on Γ as long as possible, where x is a variable, t is a term, and A, B are (possibly empty) conjunctions of literals.*

> ***Variable elimination:*** *If a clause $\phi \in \Gamma$ contains an equation $x \doteq t$, remove it and replace all occurrences of x in ϕ by t.*
>
> ***Redundant constraints:*** *Remove any denial that is subsumed by another denial in the current set.*
>
> ***Redundancy within constraints:*** *In any denial that can be written $\leftarrow A \land B$ so that A logically implies B, remove B (can be done by a straightforward procedure searching for specific patterns such as trivially satisfied (in)equalities).*

[1] It is clearly possible to enumerate in finite time all constraint theories preceding the original one in the selected ordering, as the set of symbols is finite. However, the problem of determining whether a constraint theory satisfies a given semantic condition (in this case, being a conditional weakest precondition) is likely to be undecidable. See subsection 4.4 for further discussion.

Contradiction removal: *Remove any denial $\leftarrow B$ where B is unsatisfiable (can be done by a straightforward procedure searching for specific patterns).*

Folding by resolution (FbR): *If there are two denials that, after expansion, have the form $\leftarrow A \wedge L$ and $(\leftarrow A \wedge \neg L \wedge B)\sigma$,*[2] *where L is a literal and σ a substitution, the second denial is replaced by $(\leftarrow A \wedge B)\sigma$.*

Notice that if the empty clause is produced during the process, then $\mathsf{Norm}(\Gamma) = \mathit{false}$, as it subsumes every other denial.

The FbR step takes care of a sort of dependency "across" different integrity constraints that is a bit more subtle than the identification of redundant constraints. These cases are not handled in the other approaches to simplification that we have studied, and we can expect that other specific optimizations of the same sort may be recognized in the effort to prove a minimality property. However, there are cases where an unfortunate order of the steps in the nondeterministic Norm procedure makes it fail to remove certain redundant literals, but a refined version not presented here appears to avoid this problem.

Proposition 3 (Semantic correctness of Norm). *The* Norm *procedure terminates on any input, and for any constraint theory Γ, $\Gamma \cong \mathsf{Norm}(\Gamma)$.*

Proof. Termination follows from the fact that each step in the procedure reduces the constraint theory with respect to the $\prec$ ordering which is obviously well-founded, apart from FbR that may generate an expansion which is anyhow removed by variable elimination. Lemma 1 below shows that FbR preserves the logical meaning. For all the other steps in Norm this property is evident. □

Lemma 1 (Validity of FbR). *Let Γ be a constraint theory and Γ' be another constraint theory obtained by applying FbR on Γ once. Then $\Gamma \cong \Gamma'$.*

Proof. We need only consider the logical equivalence between the two constraints mentioned in definition 9 before and after the replacement in the FbR step (the expansion step clearly preserves equivalence); assume the notation in that definition so that ϕ below represents the before-constraints and ψ the after-constraints, σ a substitution.

$$\phi = \{\leftarrow A \wedge L, (\leftarrow A \wedge \neg L \wedge B)\sigma\}.$$

The first denial is equivalent to $A \rightarrow \neg L$, from which we conclude that $A \wedge \neg L$ and A are equivalent. Therefore $\phi \cong \{\leftarrow A \wedge L, (\leftarrow A \wedge B)\sigma\} = \psi$. □

As a step towards a correct simplification procedure, we notice that the next proposition follows immediately from the previous results.

Proposition 4. *For any constraint theory Γ and update U, $\mathsf{Norm}(\mathsf{After}^U(\Gamma))$ is a weakest precondition of Γ with respect to U.*

[2] According to notational convention, the actual negation may appear in either of the two clauses mentioned in FbR.

4.3 Subsumption checks

An essential step in the achievement of simpler integrity constraints is to employ the fact that they hold in the current database state, and remove those parts of the condition about the possible updated state that are implied by this. For this purpose, we define a transformation RSub that is used to remove those derived integrity constraints produced by other transformations, that are anyhow subsumed by the original ones.

Definition 10 (Remove subsumed). *Given two constraint theories Γ and Γ', $\mathsf{RSub}^{\Gamma}(\Gamma')$ refers to a copy of Γ' in which*

- *first, any denial subsumed by a denial in Γ is removed and*
- *then, any remaining denial expandable to the form $(\leftarrow A \wedge \neg L \wedge B)\sigma$, for which a denial expandable to the form $\leftarrow A \wedge L$ is in Γ, is replaced by $(\leftarrow A \wedge B)\sigma$.*

We have immediately the following.

Proposition 5. *Let Γ' be a weakest precondition of Γ with respect to an update pattern U. Then $\mathsf{RSub}^{\Gamma}(\Gamma')$ is a conditional weakest precondition of Γ with respect to U.*

4.4 Putting together a simplification procedure

The transformation operators described in the previous sections comprise tools that can be used to define a procedure for simplification of integrity constraints, where the updates always take place from a consistent state.

Definition 11. *For a constraint theory Γ and an update U, we define*

$$\mathsf{Simp}^{U}(\Gamma) = \mathsf{RSub}^{\Gamma}(\mathsf{Norm}(\mathsf{After}^{U}(\Gamma))).$$

From the previous results we get immediately the following.

Proposition 6. *Let Γ be a constraint theory and U an update. Then $\mathsf{Simp}^{U}(\Gamma)$ is a conditional weakest precondition of Γ with respect to U.*

Example 6. Consider again the update and the constraint theory from example 4, where we showed the transformation $\mathsf{After}^{U}(\{\phi\})$. In order to obtain $\mathsf{Simp}^{U}(\{\phi\})$, we first calculate $\mathsf{Norm}(\mathsf{After}^{U}(\{\phi\}))$ as follows. The "variable elimination" step of definition 9 applied to $\mathsf{After}^{U}(\{\phi\})$ generates the following set.

$$\begin{array}{l} \{ \leftarrow m(x,y) \wedge m(x,z) \wedge y \neq z, \\ \quad \leftarrow m(\mathbf{a},y) \wedge y \neq \mathbf{b}, \\ \quad \leftarrow m(\mathbf{a},z) \wedge \mathbf{b} \neq z, \\ \quad \leftarrow \mathbf{a} \doteq \mathbf{a} \wedge \mathbf{b} \neq \mathbf{b} \qquad\qquad \}. \end{array}$$

Then, "contradiction removal" eliminates the fourth constraint and, finally, the "redundant constraints" step removes the third constraint, as it is subsumed by the second one[3]. The output of the normalization procedure is therefore the following.

$$\mathsf{Norm}(\mathsf{After}^U(\{\phi\})) = \{ \begin{array}{ll} \leftarrow m(x,y) \wedge m(x,z) \wedge y \neq z, & \\ \leftarrow m(\mathbf{a},y) \wedge y \neq \mathbf{b} & \}. \end{array}$$

The first constraint is obviously subsumed by ϕ and thus removed by RSub in the simplification procedure.

$$\mathsf{Simp}^U(\{\phi\}) = \{\leftarrow m(\mathbf{a},y) \wedge y \neq \mathbf{b}\}. \qquad \square$$

Example 7. Reconsider now the update and the constraint theory from example 5. We have here:

$$\mathsf{Simp}^U(\{\phi\}) = \emptyset. \qquad \square$$

A detailed trace for the evaluation of these simplified formulas will show that RSub removes ϕ from the resulting set of formulas when a predicate with only positive occurrences is updated, and a nontrivial specialization of ϕ when a predicate with a negative occurrence is updated.

The following example shows different combinations of singleton and compound updates for predicates occurring in positive and negative literals.

Example 8. The following transformations hold for the integrity constraint $\phi = \leftarrow p(x) \wedge q(x) \wedge \neg r(x)$ and parameters $\mathbf{a}$ and $\mathbf{b}$.

$$\begin{array}{rl}
\mathsf{Simp}^{\{p(\mathbf{a})\}}(\{\phi\}) = & \{\leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a})\} \\
\mathsf{Simp}^{\{q(\mathbf{b})\}}(\{\phi\}) = & \{\leftarrow p(\mathbf{b}) \wedge \neg r(\mathbf{b})\} \\
\mathsf{Simp}^{\{r(\mathbf{a})\}}(\{\phi\}) = & \emptyset \\
\mathsf{Simp}^{\{p(\mathbf{a}),r(\mathbf{b})\}}(\{\phi\}) = & \{\leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}) \wedge \mathbf{a} \neq \mathbf{b}\} \\
\mathsf{Simp}^{\{p(\mathbf{a}),r(\mathbf{a})\}}(\{\phi\}) = & \emptyset \\
\mathsf{Simp}^{\{p(\mathbf{a}),q(\mathbf{b})\}}(\{\phi\}) = & \{ \leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}), \\
 & \leftarrow p(\mathbf{b}) \wedge \neg r(\mathbf{b}), \\
 & \leftarrow \mathbf{a} = \mathbf{b} \wedge \neg r(\mathbf{a}) \}
\end{array} \qquad \square$$

As shown in examples 6, 7 and 8, the simplified integrity constraints are minimal in the $\prec$ ordering and instantiated as much as possible.

In case the original constraint theory contains redundant constraints (i.e., entailed by other constraints in the set), it is possible to construct examples where the result produced by our procedure would also contain redundancies. Cyclic patterns and recursive definitions can, for instance, be encoded in the integrity constraints, thus rendering the detection of redundancy computationally harder. We have not investigated this phenomenon in depth, but it seems to be undecidable in general whether a given constraint is redundant, as it would amount to the query containment problem in DATALOG, which is known to be undecidable. This means that we can only hope to prove the optimality of our

[3] Alternatively, the second constraint could be removed instead of the third one, as they subsume one another.

procedure under certain restrictions to the original constraint theory, e.g., that no recursion is encoded.

The following proposition demonstrates the idea that integrity constrains can be "pre-compiled" at design time and then used against specific updates.

Proposition 7. *Let $U(\vec{\mathbf{a}})$ be a generic update request with sequence $\vec{\mathbf{a}}$ of parameters, $\vec{c}$ any matching sequence of ground constants, and Γ a constraint theory. Then*

$$\mathsf{Simp}^{U(\vec{c})}(\Gamma) \quad \equiv \quad \big(\mathsf{Simp}^{U(\vec{\mathbf{a}})}(\Gamma)\big)_{\vec{\mathbf{a}}/\vec{c}}.$$

The left-hand side can be thought of as a simplification made at update time, whereas the right-hand side uses a pre-compiled version produced at design time in which the parameters are replaced by the actual ground constants when the update arrives. Note that the former may be simpler (according to $\prec$) than the latter. Apart from [14, 11], most works on simplification have not made such a distinction so that the procedures apply to specific updates only and, thus, need to be employed over and over when the database is running.

5 Other applications and examples

We briefly mention here several applications of our transformation operators that go beyond the scope of simplification.

5.1 Generating consistent updates by abductive reasoning

It may be the case that a proposed update, which in itself will create inconsistency in a given database state, can be extended to an update that preserves consistency. In a database application program, this situation may be handled by entering a dialogue in order to get more information.

Consider as an example the referential integrity constraint of example 5, $\leftarrow f(x,y) \wedge \neg p(x,m)$, and how an application program should handle an update request described by the update pattern $f(\mathbf{a},\mathbf{b})$; the simplified version of the integrity constraint becomes $\leftarrow \neg p(\mathbf{a},m)$.

At runtime, with a specific database state and instance of the parameters, the evaluation of this simplified integrity constraint may signal an inconsistency. In that case, the failing constraint gives a proposal for how the consistency may be repaired, namely by an update described by the pattern $p(\mathbf{a},m)$. This principle is closely related to abduction in logic programming, where literals that otherwise would fail are assumed in order to get a query to succeed.

This extended update should not be suggested to the user in case it conflicts with other integrity constraints. Thus simplified integrity constraints for $\{f(\mathbf{a},\mathbf{b}), p(\mathbf{a},m)\}$ need to be checked.

The interesting point is that the different simplifications can be constructed beforehand and added once and for all and embedded as code in the user interface program. In general, this may be structured in a decision tree with each edge labelled by an extension to the update pattern plus a simplified integrity constraints. The tree splits into different branches when there are more than one remaining literal in a failing constraint, i.e., several different ways to achieve consistency may be possible. Either the database designer has made a choice or the user is asked which way to go.

In the example, the insertion of $p(\mathbf{a}, m)$ may conflict with another integrity constraint saying that a person can only have one gender; in this case the dialogue with the user enters a new level.

5.2 Preventing duplicates

We re-examine here the scenario considered in examples 4 and 6 and include a check to avoid duplicates, i.e., an attempt to add a tuple which is already in the database is rejected. By a simple manipulation of the transformation operators, we can define a new simplification operator that includes this.

Definition 12. *For a constraint theory Γ and an update U, we define*

$$\mathsf{Simp}^{U}_{nd}(\Gamma) = \mathsf{RSub}^{\Gamma}(\mathsf{Norm}(\leftarrow U \cup \mathsf{After}^{U}(\Gamma)))$$

where $\leftarrow U$ is a shorthand for $\{\leftarrow u_1, \ldots, \leftarrow u_n\}$, the elements of U being the variable-free literals $u_1, \ldots, u_n$.

The "nd" subscript stands for "no duplicates", which is characterized here by the fact that any redundancies between the update and the weakest precondition expressed by After should be eliminated by Norm. It is easy to check that with U and ϕ from example 4 we have

$$\mathsf{Simp}^{U}_{nd}(\{\phi\}) = \{\leftarrow m(\mathbf{a}, y)\} \quad \prec \quad \{\leftarrow m(\mathbf{a}, y) \wedge y \neq \mathbf{b}\} = \mathsf{Simp}^{U}(\{\phi\}).$$

In the calculation of $\mathsf{Simp}^{U}_{nd}(\{\phi\})$, FbR applies to the denials $\leftarrow m(\mathbf{a}, \mathbf{b})$ and $\leftarrow m(\mathbf{a}, y) \wedge y \neq \mathbf{b}$ (expanded, respectively, to $\leftarrow m(x, y) \wedge x \doteq \mathbf{a} \wedge y \doteq \mathbf{b}$ and $\leftarrow m(x, y) \wedge x \doteq \mathbf{a} \wedge y \neq \mathbf{b}$), generating $\leftarrow m(x, y) \wedge x \doteq \mathbf{a}$. This example showed how a new assumption could be embedded in the simplification procedure to generate a constraint theory which is clearly minimal in terms of the ordering $\prec$.

5.3 Maintaining integrity constraints in data mining applications

Data mining techniques exist that are used to unveil integrity constraints inherent in a database. These integrity constraints may then be used for various purposes, such as semantic query optimization and integrity checking. Suppose that an update comes up that violates the induced integrity. The data miner might either give up the constraint that has been violated, in that it does not model the underlying database anymore, or extend the integrity constraint to use the offending update as a counter-example. For example, let the following be a mined integrity constraint:

$$\phi = \leftarrow p(x) \wedge q(x)$$

and assume that update $U = \{p(a)\}$ violates it. Instead of rejecting ϕ, a perhaps clever approach is to regard U as an exceptional behavior of ϕ and produce a modified version, for example:

$$\phi^U = \leftarrow p(x) \land x \neq a \land q(x)$$

The transformation from ϕ to ϕ^U is exactly what the Before transformation is doing, as in general $D(\phi) = (D \cup U)(\mathsf{Before}^U(\phi))$ for any database state D.

Some heuristics or application domain information can also be applied here to determine an upper limit to the number of exceptions allowed, above which the system eventually decides to reject ϕ.

Another system might go into a dialogue with the user by questioning whether the mined ϕ is a property of the domain and should be trusted; if yes, the system may continue as described in section 5.1.

5.4 Checking the integrity after the update

Although we generally have argued against it, there may be specific applications where database updates should be executed immediately and consistency checked directly on the updated database. Our Before operator can be used in such a case in order to convert our precondition style simplifications into a form that tests the updated state:

Proposition 8. *Let a database D be consistent with a constraint theory Γ. Then the database state $D \cup U$ is consistent with Γ iff $D \cup U$ is consistent with the following:*

$$\mathsf{Norm}(\mathsf{Before}^U(\mathsf{Simp}^U(\Gamma))).$$

Example 9. Consider integrity constraint $\phi = \leftarrow p(x) \land q(x) \land \neg r(x)$ and update $U = \{p(\mathbf{a}), q(\mathbf{b})\}$. We have:

$$\begin{aligned}\mathsf{Simp}^U(\{\phi\}) = \{ &\leftarrow \mathbf{a} \doteq \mathbf{b} \land \neg r(\mathbf{a}),\\ &\leftarrow p(\mathbf{b}) \land \neg r(\mathbf{b}),\\ &\leftarrow q(\mathbf{a}) \land \neg r(\mathbf{a}) \quad \}.\end{aligned}$$

Assuming that U was applied to a consistent database state, the updated state is consistent iff the following holds in it:

$$\begin{aligned}\{ &\leftarrow \mathbf{a} \doteq \mathbf{b} \land \neg r(\mathbf{a}),\\ &\leftarrow p(\mathbf{b}) \land \neg r(\mathbf{b}),\\ &\leftarrow q(\mathbf{a}) \land \neg r(\mathbf{a}) \quad \}.\end{aligned}$$

□

As it can be seen in this example, the test produced in this way for the updated state is identical to the original, simplified one for the state prior to the update. Whether this indicates a general property is not known at the time of writing.

5.5 Applications for data integration

Data integration is the problem of combining two or more existing source databases into a single global one by means of a so-called mediator schema. The global database may be inconsistent even if each of the sources satisfies its particular constraints. In [6] we have adapted the simplification method described in the present paper to a number of different data integration scenarios, where the consistency of the sources is employed, perhaps together with given *a priori* knowledge on their combination.

Example 10 ([6]). Consider two databases containing information about marriages, each of which is known to satisfy the following integrity constraint (no husband has more than a wife):

$$\phi = \leftarrow m(x, y) \wedge m(x, z) \wedge y \neq z.$$

A simplified constraint for checking consistency of the combined database (formed by the union of the tuples) produced by the method is the following, where the subscripts refer to the different local databases.

$$\phi_{1,2} = \leftarrow m_1(x, y) \wedge m_2(x, z) \wedge y \neq z$$

The simplification procedure can also be applied to validate, at the global level, an update reported from one of the local databases, based on the knowledge that the global database was consistent before the update and that the update was checked by the source. If, for example, the update $\{m_1(\mathit{frederik}, \mathit{mary})\}$ is reported, the simplified check for global consistency is the following:

$$\phi'_{1,2} = \leftarrow m_2(\mathit{frederik}, z) \wedge \mathit{mary} \neq z.$$

If, in addition, it is known that the sets of husbands in the two sources are always disjoint, as expressed by the constraint $\leftarrow m_1(x, y) \wedge m_2(x, z)$, both $\phi'_{1,2}$ and $\phi_{1,2}$ could be further simplified by our method to *true*. □

5.6 Semantic query optimization

We indicate here a potentiality for using the simplification procedure for semantic query optimization by a sketchy example.

Consider again the integrity constraint $\leftarrow f(x, y) \wedge \neg p(x, m)$ and assume that a given database is consistent with it. The query $\leftarrow f(x, y)$ is given to the system. Treat the variables in the query as parameters, thus writing it as $f(\mathbf{a}, \mathbf{b})$, and simplify the integrity constraint with respect to it. This gives for sure that $\leftarrow \neg p(\mathbf{a}, m)$ holds for any $\mathbf{a}$ with $f(\mathbf{a}, \mathbf{b})$ in the database.

This means that we can safely extend the query to the following: $\leftarrow p(x, m) \wedge f(x, y)$. It may be the case that the new literal refers to a very small relation so that the remaining query runs faster (although this is not likely to be the case in the present example, however).

6 Conclusion

We applied program transformation techniques to the generation of simplified integrity constraints. A procedure was constructed that makes use of these transformations and produces the simplification searched for according to a criterion of "minimality" that is relevant for a large class of cases. This minimality and the versatility of the transformation operators are the original contribution of this paper. This, together with the ability of producing a necessary and sufficient condition for checking the integrity before a database transaction, constitutes the main advantage of our method with respect to earlier approaches. Examples are discussed that show how this procedure can be applied and it is also pointed out that the program transformations prove useful in several other contexts.

Although the details were not spelled out, the simplified integrity constraints are assumed to be executable as SQL queries. In this context, an empty answer indicates that the database is consistent, otherwise the tuples returned provide hints for extending the update in order to restore consistency, c.f. section 5.1.

Future directions include the extension of the transaction language to cover the expressive power of today's querying languages, which can be handled by extending the After operator with suitable replacement patterns. Extension of the simplification method to databases with recursive views is under consideration.

As we have indicated, there seem to be undecidability results that make it impossible to achieve a general and optimal simplification procedure, so the modest goal we can hope for is to provide a procedure that is proven to produce optimal results under given restrictions to the constraint theory, and acceptable results in all other cases.

Acknowledgements. The authors are grateful to the reviewers for many suggestions and improvements, including a simpler proof of lemma 1. This research is supported in part by the IT-University of Copenhagen.

References

1. Abdennadher, S., Christiansen, H. (2000). An Experimental CLP Platform for Integrity Constraints and Abduction. Proceedings of FQAS2000, Flexible Query Answering Systems, pp. 141–152, Eds. Larsen, H.L., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H., *Advances in Soft Computing series,* Physica-Verlag (Springer).
2. Christiansen, H. (1998). Automated reasoning with a constraint-based metainterpreter, *Journal of Logic Programming*, Vol 37(1–3) Oct–Dec, pp. 213–253.
3. Christiansen, H. (1999). Integrity constraints and constraint logic programming (Invited talk). *INAP'99; Proceedings of the 12th International Conference on Application of Prolog*, Science University of Tokyo, Japan. pp. 5–12.
4. Chakravarthy, U., Grant, J., Minker, J. (1990). Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems (TODS)* 15(2), pp. 162–207, ACM Press.
5. Christiansen, H., Martinenghi, D. (2000). Symbolic constraints for meta-logic programming, *Journal of Applied Artificial Intelligence,* vol. 14, pp. 345–367.

6. Christiansen, H., Martinenghi, D. (2004). Simplification of integrity constraints for data integration. Proc. Third International Symposium on Foundations of Information and Knowledge Systems (FoIKS), February 17–20, 2004, Vienna, Austria. *Lecture Notes in Computer Science*, Eds. D. Seipel, J.-M. Turull-Torres, vol. 2942, pp. 31–48.
7. Decker, H. (2002). Translating Advanced Integrity Checking Technology to SQL. *Database Integrity: Challenges and Solutions*, Eds. J. Doorn, L.C.Rivero, Idea Group, pp. 203–249.
8. Decker, H., Celma, M. (1994). A slick procedure for integrity checking in deductive databases. *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, June 13-18, 1994*, MIT Press, pp. 456–469.
9. Dijkstra, E.W. (1976). *A Discipline of Programming.* Prentice-Hall.
10. Godfrey, P., Grant, J., Gryz, J., Minker, J. (1998). Integrity Constraints: Semantics and Applications. *Logics for Databases and Information System*, Eds. Chomicki, J. Saake, G., Kluwer, pp. 265–306.
11. Grant, J., Minker, J. (1990). Integrity Constraints in Knowledge Based Systems. In *Knowledge Engineering Vol II, Applications.* H. Adeli, Ed., McGraw-Hill, pp. 1–25.
12. Grant, J., Minker, J. (1992). The Impact of Logic Programming on Databases. *Communications of the ACM* 35(3), pp. 66–81.
13. Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Communications of the ACM* 12, no. 10. pp. 576–580.
14. Henschen, L., McCune, W., Naqvi, S. (1984). Compiling Constraint-Checking Programs from First-Order Formulas. *Advances in Database Theory*, volume 2. Eds. Gallaire, H., Nicolas, J.-M., Minker, J., Plenum Press, New York, pp. 145–169.
15. Leuschel, M., De Schreye, D. (1998). Creating Specialised Integrity Checks Through Partial Evaluation of Meta-Interpreters. *Journal of Logic Programming* 36(2), pp. 149–193.
16. Lee, S. Y., Ling, T. W. (1996). Further Improvements on Integrity Constraint Checking for Stratifiable Deductive Databases. Proc. of 22th International Conference on Very Large Data Bases (VLDB'96), September 3-6, 1996, Mumbai (Bombay), eds. Vijayaraman, T. M. et al., India. Morgan Kaufmann, pp. 495–505.
17. Lloyd, J., Sonenberg, L., Topor, R. (1987). Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming* 4(4), pp. 331–343.
18. Lloyd, J., Topor, R. (1985). A Basis for Deductive Database Systems. *Journal of Logic Programming* 2(2), pp. 93–109.
19. Martinenghi, D. (2003). A Simplification Procedure for Integrity Constraints. *World Wide Web*, http://www.dat.ruc.dk/~dm/spic/index.html.
20. Nicolas, J.-M. (1982). Logic for Improving Integrity Checking in Relational Data Bases., *Acta Informatica* 18, pp. 227–253.
21. Nilsson, U., Małuzyński, J. (1995). *Logic, Programming and Prolog (2nd ed.)*, John Wiley & Sons Ltd.
22. Qian, X. (1988). An Effective Method for Integrity Constraint Simplification. *Proc. of the Fourth International Conference on Data Engineering*, IEEE Computer Society, pp. 338–345.
23. Sadri, F., Kowalski, R. (1988). A Theorem-Proving Approach to Database Integrity. *Foundations of Deductive Databases and Logic Programming*, Ed. Minker, J., Kaufmann, Los Altos, CA, pp. 313–362.