Automatic generation of simplified weakest preconditions for integrity constraint verification
A. Ai T-Bouziad, Irene Guessarian, L. Vieille

To cite this version:
A. Ai T-Bouziad, Irene Guessarian, L. Vieille. Automatic generation of simplified weakest preconditions for integrity constraint verification. 2006. hal-00020682

HAL Id: hal-00020682
https://hal.archives-ouvertes.fr/hal-00020682
Submitted on 14 Mar 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Automatic Generation of Simplified Weakest Preconditions for Integrity Constraint Verification

Ahmed Ait-Bouziad *  Irène Guessarian **  Laurent Vieille ***

March 14, 2006

Abstract

Given a constraint c assumed to hold on a database B and an update u to be performed on B, we address the following question: will c still hold after u is performed? When B is a relational database, we define a confluent terminating rewriting system which, starting from c and u, automatically derives a simplified weakest precondition wp(c, u) such that, whenever B satisfies wp(c, u), then the updated database u(B) will satisfy c, and moreover wp(c, u) is simplified in the sense that its computation depends only upon the instances of c that may be modified by the update. We then extend the definition of a simplified wp(c, u) to the case of deductive databases; we prove it using fixpoint induction.

Keywords: Database updates, integrity constraints, weakest preconditions, program verification and simplification.

1 Introduction

We assume a constraint c, given by a universal sentence, on a database B and an update u to be performed on B, and we address the following question: will c still hold after u is performed? When B is a relational database, we define a terminating rewriting system which, starting from c and u, automatically derives a simplified weakest precondition wp(c, u) such that, 1. whenever B satisfies wp(c, u), then the updated database u(B) will satisfy c, and moreover wp(c, u) is the weakest such precondition, and 3. the computation of wp(c, u) depends only upon the instances of c that may be modified by the update. The definition of a weakest precondition wp(c, u) ensuring the safety of update u with respect to constraint c extends easily to deductive databases with recursive rules and constraints, and even updates which can add (delete) recursive rules. When the update is an insertion update, we give an algorithm which defines a simplified weakest precondition. We will abbreviate weakest precondition into wp.

A large amount of research work has been devoted to optimizing the verification of integrity constraints at transaction commit. These optimizing efforts use the fact that the constraints are verified when the transaction starts, so that the evaluation can focus on those constraints which can be violated by the updates and on the data relevant to the updates and to the constraints. Work in this area started for relational databases with techniques to simplify domain-independent first-order formula [N82]. More recently, techniques based on propagating updates through the rules of deductive databases have been developed [BDM88, SK88, LST87]. These methods are well-understood by now; they have been tested in prototype implementations and start appearing in commercial products [V98]. [VBKL99] has shown that when general formulae are properly rewritten into rules defining intermediate predicates, and when update propagation is adequately formalized, the simplification approach can be seen as a special case of update propagation.

However, these methods still involve computation at the end of the transaction. It is sometimes claimed that, in practice, this negative effect on transaction commit time explains why integrity constraints are rarely used. While there are many applications where this is not true, such a negative effect is probably not acceptable in production systems where response time is critical.

This is the reason why another line of research has been proposed. The objective is now to take into account both the integrity constraints and the structure of the transaction program, to try and determine at compile-time whether executing the program can violate this constraint or not. Early work in this line of research include [GM79, CB80, SS89]. For more recent
work, see [BS98, L95, L98, LTW93, M97].

To illustrate the differences between the two approaches, consider the constraint: ‘forall x: p(x) \rightarrow q(x)’ and the transaction program (expressed here in a Prolog-like syntax): ‘prog(x) := -insert p(x), insert q(x).’ Running this program with x = a results in the insertion of both p(a) and q(a). Optimizing methods will avoid checking the constraint on the whole databases and will focus on the data relevant to the updates. A transaction compiling approach will determine at program or constraint compile-time that, executing prog(x) with any parameter will never violate this constraint whatever instance is provided for x in prog(x), and no transaction-time activity will occur.

Known theoretical results put a limit on what can be expected from such an approach [AHV95, BGL96]. Further, not all transaction programs are such that they can be proved compliant with the constraints at compile-time. However, (1) it is natural programming practice to write transaction programs as safe\(^1\) as possible, and, (2) if the compile-time check fails, it is always possible to resort to optimizing techniques. Finally, simple examples like the one above indicate that it is worth attempting to design effective methods to prove the compliance of transaction programs with integrity constraints.

While predicate calculus appears today as the language of choice to express integrity constraints, the choice of an update language is more open. For instance, [GM79] or [BS98] focus on existing general-purpose programming languages, for which proving formal properties is notoriously difficult. In this paper, we follow [L95, L98, LTW93] by choosing a "pure" (no cut!) logic-programming based language, more easily amenable to proofs, in particular against constraints expressed in predicate calculus. This choice remains practical for database systems, as they have a tradition of providing specific procedural languages, different from general-purpose programming languages (e.g., stored procedures).

A second parameter is the degree of minimality of the language. While pure theory would tend to prefer a clean, mininal language, the design of effective methods is often facilitated by the use of additional programming constructs. This is in particular true of theorem proving techniques as clearly outlined by W. Bibel in [BLSB92]. In this paper, we add an if – then – else operator to the language of [LTW93].

In this paper, following the approach of [LTW93, L95, L98], we apply techniques coming from program verification and program transformations, see for instance [D75, TS84, PP94, SS89]. Several approaches have been taken along these lines. One can either generate weakest preconditions, as in [LTW93, N82, BDM88, M97] or one can generate postconditions, as in [BS98]. Once the (pre or post)conditions are generated, usually the task of verifying them is left to the user: hints to simplify the precondition are given in [LTW93], the decidability of checking the weakest precondition in relational databases is studied in [M97], while [N82] suggests a method for checking the precondition on the only relevant part of the database (e.g. the ‘new’ facts produced by an insertion update). [BS98] define post-conditions post(u, c) and they implement a theorem prover based approach to check the safety of updates at compile-time: it consists in proving that post(u, c) \rightarrow c holds, as this is clearly a sufficient condition for ensuring that the update is safe. A different approach to the constraint preservation problem has been studied in [BD98]: it consists in constructing generalized transaction schemes which ensure that classes of dynamic constraints are preserved.

**Contribution of the paper.** It is twofold.

1. In the case of relational databases, we define a terminating rewriting system which, starting from constraint c and update u automatically derives a wp ensuring the safety of the update; assuming that c holds, this wp is simplified into a formula which depends only upon those instances of subformulas of c that might be modified by the update. We prove that our simplified wp is simpler than the wp obtained in [LTW93, L95, L98] in the following sense: our simplified wp is implied by the wp of [L98], but the converse does not hold.

2. For deductive databases allowing for recursion, we describe an algorithm computing an efficient simplified wp in the case of insertion updates.

As soon as recursion is allowed, several problems come up: 1. it is undecidable to check if a transaction preserves a constraint[AHV95], 2. the wp is usually not expressible in first-order logic [BGL96], and moreover, 3. if we want to ensure that both the wp and the constraint are expressed in the same language [BDM88, SK88, LST87, GSUW94, L98], we have to assume a language allowing for both negation and recursion, which severely limits efficient checking of the truth of the wp. Thus, we can only hope for special cases when the wp can be shown to hold and/or can be simplified.

Our update language generalizes the language of [LTW93] and of [L98] by allowing for conditional updates; it is more expressive than SQL and the languages of [BDM88, LST87, N82]; e.g. in the relational case, in [BDM88], only elementary single fact insert/delete updates are considered, whilst we can in-

\(^1\) A transaction program is safe if it preserves the truth of the constraints.
sert/delete subsets defined by first-order formulas as in [BD88, LTW93, L95, L98]. Our simplified wp are simpler than the wp of [LTW93, L95, L98].

The paper is organized as follows: our update language is defined in section 2, the terminating rewriting system deriving the simplified wp for relational databases is described in section 3, heuristics for treating the deductive case are presented in section 4, and finally section 5 consists of a short discussion.

2 Update language

In the present section, we define our update language, which is a mild generalization of the update language of [LTW93], and we define the corresponding weakest preconditions.

2.1 Definitions

Let \( \mathcal{U} \) be a countably infinite set of constants.

1. a database (DB) \( B \) is a tuple \((R_1, R_2, ..., R_n)\) where \( R_i \) is a finite relation of arity \( k_i \) over \( \mathcal{U} \). The corresponding lower-case letters \( r_1, r_2, ..., r_n \) denote the predicate symbols naming relations \( R_1, R_2, ..., R_n \); they are called extensional predicates (or EDBs).

2. an update \( u \) is a mapping from \( B = (R_1, R_2, ..., R_n) \) to \( u(B) = (R'_1, R'_2, ..., R'_n) \) where \( R_i \) and \( R'_i \) have the same arity.

3. a constraint \( c \) is a domain independent sentence (a closed first-order formula which is domain independent, see [ToS88, VGT87]).

4. formula \( f \) is a precondition for update \( u \) and constraint \( c \) if for every database \( B \), if \( B \models f \) then \( u(B) \models c \).

5. formula \( f \) is a weaker than formula \( g \) iff for every database \( B \) we have \( B \models g \Rightarrow B \models f \). Formula \( f \) is a weakest precondition for update \( u \) and constraint \( c \), if it is weaker than every precondition for \((u, c)\).

Remark 1 In points 4 and 5 above, \( f \) is only a precondition for \( c \), i.e. \( B \models f \Rightarrow u(B) \models c \) regardless of whether \( B \models c \).

Let \( R \) be a relation in database \( B \) and let \( \Phi(\mathcal{U}) \) be a first-order formula with free variables \( \mathcal{U} = x_1, x_2, ..., x_n \).

- \( B' = B[R \rightarrow R'] \) denotes the result of substituting relation \( R' \) for relation \( R \) in \( B \).
- \( c[r \rightarrow r \cup \phi] \) (resp. \( c[r \rightarrow r - \phi] \)) denotes the result of substituting \( r(\mathcal{U}) \cup \phi(\mathcal{U}) \) (resp. \( r(\mathcal{U}) \land \neg \phi(\mathcal{U}) \)) for every occurrence of \( r(\mathcal{U}) \) in \( c \).
- \( c[r \rightarrow r \cup \phi] \) (resp. \( c[r \rightarrow r - \phi] \)) denotes the result of substituting \( r(\mathcal{U}) \lor \phi(\mathcal{U}) \) (resp. \( r(\mathcal{U}) \lor \neg \phi(\mathcal{U}) \)) for every positive\(^2\) occurrence of \( r(\mathcal{U}) \) in \( c \).

2.2 The update language

Let \( \Phi \) be a first-order formula with free variables \( \mathcal{U} \) and \( \text{cond} \) a first-order sentence which are domain independent [ToS88, VGT87]. The instructions of our language are defined as follows.

**Definition 1**

1. \( I_1. \; \text{foreach } \mathcal{U} : \Phi(\mathcal{U}) \text{ do } \text{insert}_R(\mathcal{U}) \)
2. \( I_2. \; \text{foreach } \mathcal{U} : \Phi(\mathcal{U}) \text{ do } \text{delete}_R(\mathcal{U}) \)
3. \( I_3. \; \text{if } i_1 \text{ and } i_2 \text{ are instructions then } (i_1; i_2) \text{ is an instruction; } \)
   \( I_4. \; \text{if } \text{cond} \text{ then } i_1 \text{ else } i_2 \text{ is an instruction. } \)

The alternative else \( i_2 \) is optional.

For instance, adding tuple \( \mathcal{T} \) in relation \( R \), will be denoted by \( \text{foreach } \mathcal{T} : \mathcal{T} \rightarrow \mathcal{T} \text{ do } \text{insert}_R(\mathcal{T}) \). In the sequel, we will abbreviate it by: \( \text{insert}_R(\mathcal{T}) \).

Formula \( \Phi \) is called the qualification of the update.

The update language of [LTW93] consists of instructions \( I_1, I_2, I_3 \). Our language is thus more user-friendly, because of the possibility of conditional updates defined in \( I_4 \). Our language is equivalent to the language considered in [L98]: complex instructions such as

\begin{equation}
\text{foreach } \mathcal{U} : \Phi(\mathcal{U}) \text{ do } (i_1; i_2)
\end{equation}

where e.g. for \( j = 1, 2 \), \( i_j = \text{foreach } \mathcal{T}_j : \Phi_j(\mathcal{T}_j) \text{ do } \text{insert}_{R_j}(\mathcal{T}_j) \) will be expressed in our language by \( i_0; i'_1; i'_2 \), where \( \text{TEMP} \) is a suitable, initially empty, temporary relation, and \( \mathcal{T}_j = \mathcal{U} \setminus \mathcal{T}_j \) for \( j = 1, 2 \):

\begin{align*}
&i_0 = \text{foreach } \mathcal{U} : \Phi(\mathcal{U}) \text{ do } \text{insert}_{\text{TEMP}}(\mathcal{U}) \\
&i'_j = \text{foreach } \mathcal{T}_j : \left( \mathcal{T}_j \land \Phi_j(\mathcal{T}_j) \right) \text{ do } \text{insert}_{R_j}(\mathcal{T}_j).
\end{align*}

Our language can be extended to allow for complex instructions such as 1, as well as non sequential or parallel combinations of updates (such as exchanging the values of two relations). On the other hand, \( I_4 \) could also be simulated by several instructions of [LTW93].

We briefly describe the semantics of our update language. \( I_1 \) (resp. \( I_2 \)) is executed by first evaluating \( \Phi(\mathcal{U}) \) thus producing the set of instances of \( \mathcal{U} \) satisfying \( \Phi(\mathcal{U}) \) in \( B \), and then inserting (resp. deleting) from \( R \) these instances of \( \mathcal{U} \). \( I_3 \) is performed by executing first \( I_1 \) and then \( I_2 \). \( I_4 \) is performed by evaluating condition \( \text{cond} \) on \( B \), if \( \text{cond} \) is true \( \text{insert1} \) is executed, otherwise \( i_2 \) is executed. Complex instructions such as \( \text{if } \text{cond} \text{ then } (i_1; i_2) \text{ else } i_3 \) are performed by evaluating first \( \text{cond} \) and then, if e.g. \( \text{cond} \) is true, executing \( i_1 \) and then \( i_2 \), regardless of whether \( \text{cond} \) holds after executing \( i_1 \).

Formally, the update \([i](B)\) associated with instruction \( i \) is defined by:

1. \([I_1](B) = B[R \rightarrow R \cup \{\mathcal{U}B | \Phi(\mathcal{U})\}]\)
2. \([I_2](B) = B[R \rightarrow R \setminus \{\mathcal{U}B | \Phi(\mathcal{U})\}]\)

\( ^2 \)An occurrence of \( r(\mathcal{U}) \) is positive if it is within the scope of an even number of negations. It is negative otherwise.
Theorem 1 The formulas obtained by the weakest precondition transformation of [LTW93] and defined below in 1–4 are weakest preconditions for instructions $I_1$–$I_4$.

1. $wp(\text{foreach } \mathcal{T} : \Phi(\mathcal{T}) \text{ do } \text{insert}_R(\mathcal{T}), c) = c[r \rightarrow r \cup \Phi]$.
2. $wp(\text{foreach } \mathcal{T} : \Phi(\mathcal{T}) \text{ do } \text{delete}_R(\mathcal{T}), c) = c[r \rightarrow r - \Phi]$.
3. $wp(i_1; i_2) = wp(i_1, (wp(i_2, c))$.
4. $wp(\text{if } \text{cond} \text{ then } \text{inst}_1 \text{ else } \text{inst}_2, c) = (\text{cond} \land wp(\text{inst}_1, c)) \lor (\neg \text{cond} \land wp(\text{inst}_2, c))$.

Example 1 Let $c = \forall x (r(x) \rightarrow q(x))$ and let $i = \text{foreach } \mathcal{T} : p(\mathcal{T}) \text{ do } \text{insert}_R(\mathcal{T})$, then $wp(i, c) = \forall \mathcal{T} ((r(\mathcal{T}) \lor p(\mathcal{T})) \rightarrow q(\mathcal{T}))$

Remark 2 It is noted in [LTW93] that, for constraints which are universal formulas in conjunctive normal form, one can take advantage of the fact that $c$ holds in $B$ to simplify $wp$ into a $wp'$ such that for all $B$ satisfying $c$ in $B \models wp'$ we have $\models \text{cond}$ $B$. Consider for instance the update $i = \text{foreach } \mathcal{T} : p(\mathcal{T}) \text{ do } \text{insert}_R(\mathcal{T})$ of Example 1.1, with $c = \forall x (r(x) \rightarrow q(x))$, then we can simplify $wp(i, c)$ as follows

$$wp(i, c) = \forall \mathcal{T} ((r(\mathcal{T}) \lor p(\mathcal{T})) \rightarrow q(\mathcal{T}))$$

whence $wp' = \forall \mathcal{T} (p(\mathcal{T}) \rightarrow q(\mathcal{T}))$. In the next section, we will apply this simplification in a systematic way, and implement it via a confluent and terminating rewriting system for weakest preconditions.

3. Relational databases

In the present section, we define a confluent terminating rewriting system which, starting from $c$ and $u$, automatically derives a simplified weakest precondition $wp(c, u)$. The idea underlying the simplification is the same as in [LTW93, L98]: it consists in transforming the $wp$ into the form $wp = c \land wp_1$, and to take advantage of the truth of $c$ in the initial database to replace $wp$ by $wp_1$. However,

1. it is not clear in [LTW93, L98] how far this simplification is carried on, and
2. this simplification is carried on by the user.

On the other hand, in our case, the simplification
1. is iteratively applied until no further simplification is possible, and
2. is performed automatically.

It is noted in [LTW93] that constraints involving existential quantifiers cannot be simplified: e.g. let $c$ be the constraint $\exists \mathcal{T} (r(\mathcal{T}) \land let u be the update $\text{delete}_R(\mathcal{T})$; then $wp(c, u) = \exists \mathcal{T} (r(\mathcal{T}) \land \neg (r(\mathcal{T}))$ which cannot be simplified. Hence, for the simplification to be possible, we will assume the following restrictions to our language:

1. constraint $c$ and conditions $\text{cond}$ are universal sentences, and
2. the qualifications $\Phi(\mathcal{T})$ are formulas without quantifications.

Recall that a clause is a sentence of the form $c = \forall x (l_1 \lor l_2 \lor \ldots \lor l_m)$ where the $l_i$s are literals. Without loss of generality, we can restate our hypotheses as

1. constraint $c$ is a single clause; indeed if $c = \forall x (D_1 \land D_2 \land \ldots D_n)$ is in conjunctive normal form, then we can replace $c$ by the $n$ constraints $\forall x D_1$ which are clauses and can be studied independently.
2. the qualifications $\Phi(\mathcal{T})$ are conjunctions of literals; we can write $\Phi(\mathcal{T}) = \Phi_1(\mathcal{T}) \lor \Phi_2(\mathcal{T}) \lor \ldots \lor \Phi_n(\mathcal{T})$ in disjunctive normal form, and we replace the instruction qualified by $\Phi(\mathcal{T})$ by a sequence of similar instructions qualified by $\Phi_1(\mathcal{T}), \Phi_2(\mathcal{T}), \ldots, \Phi_n(\mathcal{T})$.
3. the conditions $\text{cond}$ are single clauses: assuming as above that $c = \forall x (D_1 \land D_2 \land \ldots D_n)$ is in conjunctive normal form, we let $\text{cond}_i = \forall x D_i$ and we replace e.g. $\text{if } \text{cond} \text{ then inst}$ by $\text{if } \text{cond}_i \text{ then inst}$ (if $\text{cond}_i \text{ then inst}$).

Note that in many practical cases, constraints are indeed given by clauses, even quite simple clauses involving just 2 or 3 disjuncts, and the restrictions on $\text{cond}$ and $\Phi(\mathcal{T})$ are also satisfied.

We will use the following notation: let $S$ be a set of clauses and $r$ a predicate symbol; $\text{Res}_S(S)$ is the set of simplified binary resolvents of pairs of clauses in $S$ such that

1. each resolvent is obtained by a unification involving $r$. 

2. if the resolvent contains a literal of the form \( \neg(\exists x \Phi(x)) \) (which we write as \( \exists x \not\Phi(x) \)), then the resolvent is simplified by deleting that literal and substituting \( \exists x \not\Phi(x) \) for \( \exists x \Phi(x) \) in the resolvent.

**Example 2.** Let \( S = \{ \neg r(x, y) \vee q(y, z), r(x, y) \vee \neg q(x, y) \} \); then \( Res_c(S) = \{ q(y, z), \neg q(x, y) \} \).

2. Let \( S = \{ \neg r(x, y) \vee \neg r(x, z) \vee q(y, z), r(x, y) \vee (x, y) \neq (a, b) \} \); then the binary resolvents obtained by unifying over predicate \( r \) all pairs of clauses in \( S \) are

\[
\{ \neg r(x, z) \vee q(y, z) \vee ((x, y) \neq (a, b)) \}, \neg r(x, y) \vee q(y', z) \vee ((x, y) \neq (a, b)) \}
\]

they are then simplified into

\[
Res_c(S) = \{ \neg r(a, z) \vee q(b, z), \neg r(a, y') \vee q(y', b) \}.
\]

The idea governing our simplification method is as follows: we define a rewriting system with rules of the form

\[
wp(u, c) \rightarrow \bigwedge wp(u, c_i)
\]

and with the property that all formulas in a derivation are logically equivalent. Rewriting steps consist in computing the instances of the constraint which could be modified by the update. Let us define the following abbreviations: we write \( insert_R \Phi \) for update \( \Phi \) : \( \forall x \exists y \Phi(x,y) \) do \( insert_R(\exists x \Phi(x,y)) \), and we write \( r \vee \neg \phi \) instead of \( \forall x \neg R(x) \vee \neg \Phi(x) \). Similarly, \( delete_R \Phi \), (resp. \( \neg r \vee \neg \phi \)) abbreviates \( I_2 : \forall x \exists y \Phi(x,y) \) do \( delete_R(\exists x \Phi(x,y)) \), (resp. \( \forall x \neg R(x) \vee \neg \Phi(x) \)). In updates \( insert_R \Phi \) or \( delete_R \Phi \), we may assume that \( r \) does not occur in \( \Phi \): indeed any occurrence of \( r \) in \( \Phi \) is preventively renamed before applying our rewriting rules.

Then our rewriting system consists of the nine rules given in Figure 1, where \( I_1 - I_4 \) are defined in Definition 1, \( c \) is a clause and \( c_1, c_2 \) are universal sentences:

**Theorem 2.** The rules given in Figure 1 define a terminating and confluent rewriting system.

2. The weakest precondition generated by our system is in the form \( wp = sup \land c' \), with \( c' \) such that \( c \implies c' \); if \( c \) holds, \( wp \) can be simplified into the weakest precondition \( sup \) which is weaker than \( wp \), the weakest precondition defined in Theorem 1.

**Proof idea:** Rules are repeatedly applied till saturation, i.e., until an explicit form to which no rule applies is obtained.

1. The termination is proved by structural induction: each subformula \( wp(i, c_j) \) derived from \( wp(i, c) \) either is in an explicit form, or has a \( c_j \) which is strictly smaller than \( c \). Confluence follows from the fact that each \( wp(i, c) \) has a unique derivation (up to the order in which the rewritings are applied).

2. We prove by induction that our rewriting system generates a \( wp \) in the form \( wp = sup \land c' \), with \( c' \) such that \( c \implies c' \) holds. Therefore, taking into account that \( c \) holds in \( B \), we can simplify \( wp \) into \( sup \), and \( wp \implies sup \) holds. Because \( wp \) is equivalent to the weakest precondition of Theorem 1, and because the weakest preconditions of \([LTW93, L95, L98]\) and of Theorem 1 are equivalent, \( sup \) is simpler than the weakest preconditions of \([LTW93, L95, L98]\) and Theorem 1. Example 3 1 shows that \( sup \not\Leftrightarrow wp \), i.e., our \( sup \) can be strictly simpler than \( wp \).

**Example 3.** Let \( c = \forall x, y \exists z ((p(x, y) \land q(y, z)) \rightarrow (p(x, z) \lor q(x, z))) \) and let \( i = \text{foreach } x, y \exists z ((x, y) = (a, a)) \) do \( insert_P(x, y) \), i.e. \( \Phi(\exists x \exists y) \) is \( (x, y) = (a, a) \), then the method of \([LTW93]\) gives

\[
wp(i, c) = \forall x, y, z (((p(x, y) \lor x, y) = (a, a)) \lor q(y, z)) \rightarrow ((p(x, z) \lor q(x, z)))
\]

and our method gives the sequence of rewritings:

\[
wp(i, c) \rightarrow \cdots \\
\neg \exists (q(x, z) \lor p(x, z) \lor q(a, z)) \\
\land (p(x, y) \rightarrow p \lor (x, y) = (a, a)) \\
\neg \exists (q(a, z) \lor x, y = (a, a)) \\
\rightarrow \text{true}
\]

The simplification of line 2 is obtained by taking into account the fact that \( c \) holds in \( B \), hence \( [c \vdash p \rightarrow p \lor (x, y) = (a, a)] \) also holds. Because \( \neg q(a, z) \lor q(a, z) = \)
true, our simplified weakest precondition equivalent to true and we obtain line 3.

2. Consider again Example 1.2. We have the sequence of rewritings, where we have underlined the rewritten term whenever a choice was possible:

\[
\begin{align*}
wp((i_1;i_2), c) & \rightarrow_{R_0} wp(i_1, wp(i_2, c)) \\
& \rightarrow_{R_2} wp(i_1, c \land wp(i_2, \neg p \lor q)) \\
& \rightarrow_{R_1} wp(i_1, c \land (\neg p \lor q)) \\
& \rightarrow_{R_8} wp(i_1, c) \land wp(i_2, (\neg p \lor q)) \\
& \rightarrow_{R_3} (c \rightarrow \neg r \rightarrow \neg r \lor s) \land (\neg p \lor q)) \\
& \rightarrow (\neg p \lor q)
\end{align*}
\]

(4)

Note that line 4 gives us the simplified form of the weakest precondition of Example 1.1, (see Remark 2). The last simplification is obtained by taking into account the fact that \( c \) holds in \( B \), hence \( c \rightarrow \neg r \rightarrow \neg r \lor s \) also holds.

3. Let \( c = \forall x, y, z \ (\neg p(x, y) \lor \neg p(x, z) \lor q(y, z)) \) and let \( i = \text{foreach } x, y : (x, y) = (a, b) \text{ do insert}_P(x, y) \), then our method gives:

\[
\begin{align*}
wp(i, c) & \rightarrow_{R_2} wp(i, \neg p(a, z) \lor q(b, z)) \\
& \land wp(i, (\neg p(a, y) \lor q(b, y)) \land c) \\
& \rightarrow_{R_2} wp(i, (q(b, b)) \land (\neg p(a, z) \lor q(b, z)) \\
& \land (\neg p(a, y) \lor q(y, y)) \land c \\
& \rightarrow_{R_1} (q(b, b) \land (\neg p(a, z) \lor q(b, z)) \\
& \land (\neg p(a, y) \lor q(y, y)) \land c
\end{align*}
\]

(5)

(\( \rightarrow_{R_2}^* \) means that several \( \rightarrow_{R_2} \) rewriting steps are performed); assuming \( c \) holds in \( B \), we can simplify 5 into \( swp = \forall y, z \ q(b, b) \land (\neg p(a, z) \lor q(b, z)) \land (\neg p(a, y) \lor q(y, y)) \) which is to be verified in \( B \). Assuming \( c \) holds in \( B \), the methods of [N82, BDM88, BD88] yield the postcondition \( \forall y, z \ (\neg p(a, z) \lor q(b, z)) \land (\neg p(a, y) \lor q(y, y)) \) which must be verified in the updated database \( i(B) \). [BDM88, BD88] go one step further: embedding \( B \) in a deductive framework, they simulate the evaluation of the postcondition via predicates \( \text{delta} \) and \( \text{new} \) which are evaluated in \( B \) before update \( i \) is performed.

4 Deductive Databases

We now extend the definition and verification of a weakest precondition \( wp(u, c) \) to the deductive database setting, where both \( c \) and \( u \) can be defined by DATALOG programs. We chose DATALOG because it is the best understood, most usual and simplest setting for deductive databases. We will first define our framework, the weakest preconditions, and then we will give heuristics for

1. proving that the weakest precondition holds without actually evaluating it
2. computing simplified weakest preconditions.

Recall that on a language consisting of the EDBs \( r_1, \ldots, r_k \) and new predicate symbols \( q_1, \ldots, q_t \) – called intensional predicates (or IDBs) –, a DATALOG program \( P \) is a finite set of function-free Horn clauses, called rules, of the form:

\[
q(y_1, \ldots, y_n) \leftarrow q_1(y_1,1, \ldots, y_{1,n_1}) \ldots, q_t(y_p,1, \ldots, y_{p,n_p})
\]

where the \( y_i \)'s and the \( y_{i,j} \)'s are either variables or constants, \( q \) is an intensional predicate in \( \{q_1, \ldots, q_t\} \), the \( q_i \) 's are either intensional predicates or extensional predicates.

In our framework, both updates and constraints range over deductive queries, possibly involving recursion. Formally, updates and constraints are defined as in Section 2, but in addition:

- the qualifications \( \Phi(x) \) in both \( \text{insert} \) and \( \text{delete} \) statements are conjunctions of literals which may contain atoms defined by recursive DATALOG programs,
- similarly, constraints \( c \) and conditions \( \text{cond} \) in \( \text{if - then - else} \) statements may contain atoms defined by recursive DATALOG programs: constraints and conditions are general clauses, but the atoms occurring in them are defined by Horn clauses.

The definition of the weakest preconditions extends easily: we follow here the approach of [L95].

**Notation 1.** Let \( Q \) and \( R \) be the programs obtained by adding to \( P \) new IDB symbols \( q' \) for each predicate \( q \) depending on \( c \), and new rules; for each rule \( \rho \) of \( P \) defining an IDB \( q \) depending on \( r \), a new rule \( \rho' \) defining \( q' \) is added: \( \rho' \) is obtained from \( \rho \) by substituting \( s' \) for each occurrence of a symbol \( s \) depending on \( r \) (hence \( r' \) is substituted for each occurrence of \( r \)). If \( r \) is an EDB predicate, we add a new IDB symbol \( r' \), but there is no rule defining \( r' \) yet: the rules defining \( r' \) depend on the update and will be given later.

2. Let \( P_\rho^c \) be the program obtained by adding to \( P \) new IDB symbols \( q' \) for each predicate \( q \) depending on \( c \), and new rules; for each rule \( \rho \) of \( P \) defining an IDB \( q \) depending on \( r \), a new rule \( \rho' \) defining \( q' \) is added: \( \rho' \) is obtained from \( \rho \) by substituting \( s' \) for each occurrence of a symbol \( s \) depending on \( r \) (hence \( r' \) is substituted for each occurrence of \( r \)). If \( r \) is an EDB predicate, we add a new IDB symbol \( r' \), but there is no rule defining \( r' \) yet: the rules defining \( r' \) depend on the update and will be given later.

3. \( c[r \rightarrow r'] \) denotes \( c \) where all the predicates depending on \( c \) (including \( r \) when \( r \) is an EDB) are replaced by the corresponding primed predicates.

We will assume the following hypotheses:

- \( H_1 \): the qualifications \( \Phi(x) \) are conjunctions of literals, and all the rules defining the IDBs in constraint \( c \),
qualifications \( \Phi(\mathfrak{T}) \) and conditions \( \text{cond} \) are given in program \( P \).

\( H_2 \): in statements of the form \( \text{foreach} \ \mathfrak{T} : \Phi(\mathfrak{T}) \ \text{do} \ \text{insert}_R(\mathfrak{T}) \), or \( \text{foreach} \ \mathfrak{T} : \Phi(\mathfrak{T}) \ \text{do} \ \text{delete}_R(\mathfrak{T}) \), none of the literals in \( \Phi(\mathfrak{T}) \) depends on \( r \).

**Theorem 3** Assume constraint \( c \) and instruction \( i \) satisfy \( H_1 \) and \( H_2 \), then the formulas defined are weakest preconditions for \( c \) and \( i \).

1. \( \text{wp}(\text{foreach} \ \mathfrak{T} : \Phi(\mathfrak{T}) \ \text{do} \ \text{insert}_R(\mathfrak{T}), c) = c[r \rightarrow r'] \) where the program defining the IDBs is \( P'_r \cup \{ r'(\mathfrak{T}) \leftarrow r(\mathfrak{T}), r'(\mathfrak{T}) \leftarrow \Phi(\mathfrak{T}) \} \).

2. \( \text{wp}(\text{foreach} \ \mathfrak{T} : \Phi(\mathfrak{T}) \ \text{do} \ \text{delete}_R(\mathfrak{T}), c) = c[r \rightarrow r'] \) where the program defining the IDBs is \( P'_r \cup \{ r'(\mathfrak{T}) \leftarrow r(\mathfrak{T}) \land \neg \Phi(\mathfrak{T}), t(\mathfrak{T}) \leftarrow \Phi(\mathfrak{T}) \} \), with \( t \) a new IDB predicate,

3. \( \text{wp}(i_1; i_2, c) = \text{wp}(i_1, (\text{wp}(i_2, c)) \),

4. \( \text{wp}(\mathfrak{T} \ 	ext{cond then} \ \text{inst1 else inst2} = \text{cond} \land \text{wp}(\text{inst1}, c) \lor \neg \text{cond} \land \text{wp}(\text{inst2}, c)) \).

Theorem 3 calls for some remarks.

1. When none of \( r, c \) or \( \Phi \) is recursive, we obtain again the weakest preconditions of Theorem 1.

2. Our definition of insertions is quite liberal, allowing us to add new rules, which is not permitted in [L95]. Similarly, deletions can suppress tuples, sets of tuples or even rules.

3. Deletions and/or qualifications \( \Phi \) containing negations force us out of the DATALOG framework, because the weakest precondition of \( \text{foreach} \ \mathfrak{T} : \Phi(\mathfrak{T}) \ \text{do} \ \text{delete}_R(\mathfrak{T}) \) and constraint \( c \) is defined by the DATALOG\(^-\) program \( P'_r \cup \{ r'(\mathfrak{T}) \leftarrow r(\mathfrak{T}) \land \neg \Phi(\mathfrak{T}), t(\mathfrak{T}) \leftarrow \Phi(\mathfrak{T}) \} \); this was already noted in [GSUW94]. In [L95, LST87] a stratified DATALOG\(^-\) framework is assumed: this ensures that both the constraint and the \( \text{wp} \) are expressible in the same framework.

4. In what follows, we will consider only insertions and positive qualifications in order to be able to express both constraints and their \( \text{wp}s \) in DATALOG. The \( \text{wp}s \) defined in Theorem 3 are correct without this restriction, but they are defined by stratified DATALOG\(^-\) programs, and not by Horn clauses.

**Example 4** Let \( c \) be the constraint \( \forall x, y \ (\neg t_c(x, y) \lor i_3(x, y)) \), where \( i_3 \) does not depend on \( t_c \), and consider the update \( \text{foreach} \ x, y : \text{path}(x, y) \ \text{do} \ \text{insert}_{TC}(x, y) \), where all the predicates are defined by program \( P \):

\[
P = \begin{align*}
\text{tc}(x, y) & \leftarrow \text{arc}(x, y) \\
\text{tc}(x, y) & \leftarrow \text{arc}(x, z), \text{tc}(z, y) \\
\text{path}(x, y) & \leftarrow \text{edge}(x, y) \\
\text{path}(x, y) & \leftarrow \text{edge}(x, z), \text{path}(z, y) \\
i_3(x, y) & \leftarrow \text{body}(x, y)
\end{align*}
\]

Then \( P'_c \) is \( P \) together with a new predicate \( t_c' \) and the rules 6 and 7.

\[
\begin{align*}
t_c'(x, y) & \leftarrow \text{arc}(x, y) \\
t_c'(x, y) & \leftarrow \text{arc}(x, z), t_c'(z, y)
\end{align*}
\]

and \( P'_{\text{insert}_{TC}} \) is \( P'_c \) together with the rules 8 and 9.

\[
\begin{align*}
t_c'(x, y) & \leftarrow \text{tc}(x, y) \\
\text{path}(x, y) & \leftarrow \text{path}(x, y)
\end{align*}
\]

Finally, \( \text{wp}(u, c) = \forall x, y \ (\neg t_c'(x, y) \lor i(x, y)) \), where \( t_c \) is defined by \( P'_{\text{insert}_{TC}} = P \cup \{ 6, 7, 8, 9 \} \).

We now turn our attention towards the goal of proving that the weakest precondition holds without actually evaluating it. One method is to show that \( c \iff \text{wp}(u, c) \).

The problem is that implication 10 is undecidable except in some special cases: e.g., if both \( c \) and \( \text{wp}(u, c) \) are unions of conjunctive queries, and at least one of them is not recursive [C91, CV92]; some special classes of formulas for which 10 is decidable are studied in [M97]. So we can only hope for heuristics to find sufficient conditions ensuring that implication 10 will hold. The idea, coming from Dijkstra’s loop invariants [D76], consists in proving 10 by recursion induction, without actually computing \( \text{wp}(u, c) \). We illustrate this idea on an example.

**Example 5** Let \( c \) be the constraint \( \forall x, y \ (\neg t_c(x, y) \lor i_3(x, y)) \), where \( I \) and \( TC \) are defined by \( P \):

\[
\begin{align*}
tc(x, y) & \leftarrow \text{arc}(x, y) \\
tc(x, y) & \leftarrow \text{edge}(x, y) \\
tc(x, y) & \leftarrow \text{arc}(x, z), \text{tc}(z, y) \\
i(x, y) & \leftarrow i_1(x, z_1), \text{edge}(z_1, z_2), i_1(z_2, y) \\
i_1(x, y) & \leftarrow \text{arc}(x, z), i_1(z, y) \\
i_1(x, y) & \leftarrow \text{edge}(z, x), i_1(z, y) \\
i_3(x, y) & \leftarrow \text{body}(x, y)
\end{align*}
\]

and consider the update \( u = \text{foreach} \ x, y : \text{edge}(x, y) \ \text{do} \ \text{insert}_{AR}(x, y) \). Then \( \text{wp}(u, c) = \forall x, y \ (\neg t_c'(x, y) \lor i_3(x, y)) \), where \( I \) and \( TC' \) are defined by \( P'_{\text{insert}_{AR}} \) consisting of \( P \) together with the rules (because \( I = I' \) here):

\[
\begin{align*}
arc'(x, y) & \leftarrow \text{arc}(x, y) \\
arc'(x, y) & \leftarrow \text{edge}(x, y) \\
tc'(x, y) & \leftarrow \text{arc}(x, y) \\
tc'(x, y) & \leftarrow \text{arc}(x, z), t_c'(z, y)
\end{align*}
\]

We prove that \( c \iff \text{wp}(u, c) \) by induction. To this end, let \( C[R] = (R \subset I) \); we note that \( c \) holds iff \( C[TC] \) holds, and \( \text{wp}(u, c) \) holds iff \( C[TC'] \) holds.
Let \( \bowtie \) denote the composition of binary relations. It thus suffices to prove that, for any \( R \), \( C[R] \implies C[\text{Arc'} \cup \text{Arc'} \bowtie R] \) to conclude, by fixpoint induction, that \( C[TC'] \) holds.

We now show that, if \( c \) holds, then \( C[R] \implies C[\text{Arc'} \cup \text{Arc'} \bowtie R] \) holds. Assume that \( C[R] \) holds. Because \( c \) holds, \( C[TC'] \) holds. Note that \( C[\text{Arc'} \cup \text{Arc'} \bowtie R] \) reduces to the conjunction of \( C[\text{Arc'}], C[\text{Arc'} \bowtie R], C[\text{Edge}], C[\text{Edge} \bowtie R] \); each of the conjuncts is easy to verify: for instance \( C[\text{Arc'}] \) holds because \( \text{Arc} \subset TC \) by rule 11 and because \( C[TC'] \) holds; \( C[\text{Arc'} \bowtie R] \) holds because \( C[R] \) holds, and because of rules 14, 15, 14, and similarly for \( C[\text{Edge}] \) and \( C[\text{Edge} \bowtie R] \).

We now study the computation of simplified weakest preconditions, in the case of insertion updates. The basic idea is quite simple and comes from the semi-naive query evaluation method in DATALOG (see [AHV95]). We design a DATALOG program computing all new facts deduced from the insertion update (and preferably only new facts) and we verify the constraint on the new facts computed by that program. The method of [BD88] is based on a similar idea. We will sketch this method on an example, simple, but useful, where the constraint is based on a similar idea. We will sketch this method on an example, simple, but useful, where the constraint is based on a similar idea.

**Example 6** Consider the update \( \text{insert}_{\text{Arc}}(d, b) \), and let \( c \) be the constraint \( \neg \exists x \ t(x, x) \), where \( TC \) is defined by program \( P \).

\[
P = \begin{cases} 
  t(x, y) & \leftarrow \text{arc}(x, y) \\
  t(x, y) & \leftarrow \text{arc}(x, z), t(z, y)
\end{cases}
\]

We assume that constraint \( c \) is verified by database \( B \).

Let \( \Delta_x \) be the potentially new facts which will be inserted in \( TC \) as a consequence of the update. The IDB predicate \( \delta_x \) corresponding to \( \Delta_x \) is defined by the DATALOG program:

\[
P' = \begin{cases} 
  t(x, y) & \leftarrow \text{arc}(x, y) \\
  \delta_x(d, b) & \leftarrow \text{arc}(x, z), t(z, y)
\end{cases}
\]

The weakest precondition \( \wp(c, \text{insert}_{\text{Arc}}(d, b)) \) then is \( \neg \exists x \ \delta_x(t(x, x)) \), which can be evaluated by SLD-AL resolution [V89]. The weakest precondition of [L95] and of Theorem 3 would be in the present case \( \neg \exists x \ t(x, x) \) where \( TC' \) is defined by the program \( P'_{\text{insert}_{\text{Arc}}(d, b)} \):

\[
P'_{\text{insert}_{\text{Arc}}(d, b)} = \begin{cases} 
  t(x, y) & \leftarrow \text{arc}(x, y) \\
  t(x, y) & \leftarrow \text{arc}(x, z), t(z, y) \\
  t(x, y) & \leftarrow t(x, y) \\
  t(x, y) & \leftarrow \text{arc}(x, z) \wedge \delta_x(z, y)
\end{cases}
\]

The program \( P' \) of Example 6 can be obtained by an algorithm: the idea is to compute the rules defining new facts by resolution with the inserted atoms (similar to the idea of "refutation with update as top clause" of [SK88]). A saturation method [AHV95, BD88, SK88, LST87] is used to generate the new rules, i.e. we add rules until nothing new can be added. To simplify the notations, we give the algorithm in the case when the update is of the form \( \text{insert}_{\text{Arc}}(d) \) for \( j = 1, \ldots, k \), with \( r \) an EDB predicate, and the constraint is of the form \( \neg \exists x \ t(x) \) with \( t \) an IDB, possibly depending on \( r \), defined by a linear DATALOG program.

**Algorithm. Inputs:** update \( u = \text{insert}_{\text{Arc}}(d) \) for \( j = 1, \ldots, k \), constraint \( c = \neg \exists \delta_x(t) \), and a linear DATALOG program \( P \) defining relation \( T \).

**Outputs:** A simplified \( \wp(u, c) \), and a DATALOG program \( P' \) defining \( \wp(u, c) \).

**Step 1:** For each \( \delta_q \) of \( P \) depending on \( r \), let \( \delta_q \) be a new IDB; let \( \Pi \) be the set of rules defined as follows: for each rule \( q \leftarrow \text{body} \) of \( P \) whose head \( q \) depends on \( r \) and \( r \) add to \( \Pi \) a rule \( \delta_q \leftarrow \text{body} \). If \( \Pi \) is empty, then update \( u \) is safe; STOP.

**Step 2:** Let \( P_1 = \{ \text{Res}(\rho, r(t_j)) / \rho \in \Pi, j = 1, \ldots, k \} \) be the set of resolvents of rules of \( \Pi \) with updated atoms. If \( P_1 \) is empty, then update \( u \) is safe; STOP.

Otherwise, generate new rules as follows:

\[
i := 1 \text{ WHILE } P_i \neq \emptyset \text{ DO } P_{i+1} = \{ \text{Res}(\rho, r(t_j)) / \rho \in P_i, j = 1, \ldots, k \} ; \ i := i + 1 \text{ ENDDO}
\]

Let \( P' = \cup P_i \).

**Step 3:** Let \( \Sigma_1 \) be the set of rules of \( \Pi \) which contain an IDB \( q \) depending on \( r \) in their body. If \( \Sigma_1 \) is empty, then \( \wp(u, c) = \neg \exists \delta_t(t) \), and program \( P \cup P' \) defines \( \Delta_t \); STOP.

Otherwise, generate new rules as follows: let \( P'' \) be obtained from \( \Sigma_1 \) by substituting \( \delta_q \) for \( q \) in the bodies of the rules of \( \Sigma_1 \); only new predicates \( \delta_q \) appear in rules of \( P'' \).

\[
i := 1 \text{ WHILE } P'' \neq \emptyset \text{ DO } P'''_{i+1} = \{ \text{Res}(\rho, r(t_j)) / \rho \in P'', j = 1, \ldots, k \} ; \ i := i + 1 \text{ ENDDO}
\]

Let \( P'' = \cup P''' \).
\[ wp(u, c) = \neg \exists x \, \delta_{tc}(x), \] and program \( P \cup P' \cup P'' \) defines \( \Delta_t; \) STOP. \[ \square \]

The WHILE loops in steps 2 and 3 terminate, because at each iteration step the number of atoms involving \( r \) decreases in the rules. This algorithm can generate rules which might be useless in some cases: e.g., in Example 6, the useless rule \( \delta_{tc}(d, y) \leftarrow \delta_{tc}(c, y) \) would be generated. This algorithm can be generalized to non-linear DATALOG programs, and to more general insert instructions.

When the insertion is defined by a (possibly recursive) qualification, we can similarly compute the potentially new facts to be inserted as a consequence of the update. Consider the program \( P \) and the update \( u = \text{foreach } x, y : \text{path}(x, y) \text{ do insert}_{TC}(x, y) \) defined in Example 4, and let \( c \) be constraint \( \neg \exists x \, tc(x, x) \). The potentially new\(^4 \) facts \( \Delta_{tc}(x, y) \) which will be inserted in \( TC \) are defined by the DATALOG program \( P' \):

\[
P' = \left\{ \begin{array}{l}
P \\delta_{tc}(x, y) \leftarrow \text{edge}(x, y) \\
\delta_{tc}(x, y) \leftarrow \text{edge}(x, z), \, tc(z, y) \\
\delta_{tc}(x, y) \leftarrow \text{arc}(x, z), \, \delta_{tc}(z, y) \\
\delta_{tc}(x, y) \leftarrow \text{arc}(x, z) \text{ or } \delta_{tc}(z, y) \\
\end{array} \right. \]

The weakest precondition \( wp(c, \text{insert}_{Arc}, \text{path}) \) is again \( \neg \exists x \, \delta_{tc}(x, x) \).

The method of [L95] and of Theorem 3 would now give the weakest precondition \( \neg \exists x \, tc'(x, x) \) where \( TC' \) is defined by the program \( P'_{\text{insert}_{Arc}, \text{edge}} \), namely:

\[
P' = \left\{ \begin{array}{l}
P \text{tc}(x, y) \leftarrow \text{arc}'(x, y) \\
\text{tc}(x, y) \leftarrow \text{arc}'(x, z), \, \text{tc}(z, y) \\
\text{arc}'(x, y) \leftarrow \text{arc}(x, y) \\
\text{arc}'(x, y) \leftarrow \text{path}(x, y) \end{array} \right. \]

5 Conclusion and discussion

In the relational case, we devised a systematic method for computing a simplified weakest precondition for a general database update transaction \( u \) and a constraint \( c \). This yields an efficient way of ensuring that the update maintains the truth of the constraints. In the deductive case, we studied two methods: the first one consists in proving by fixpoint induction that \( c \implies wp(u, c) \) holds without evaluating \( wp(u, c) \); the second one consists in defining, for insertion updates and constraints \( c \) of the form \( \neg \exists x \, q(x) \), a constraint \( c' \) simpler than \( wp(u, c) \) and such that \( c \implies \left( c' \iff wp(u, c) \right) \); this is a first step towards one of the goals stated in the conclusion of [BGL96].

The idea of our method is to preventively check only relevant parts of the precondition which are generated using saturation methods. We preventively check a weakest precondition before performing the update, and perform the update only when the weakest precondition ensures us that it will be safe (see also [BDM88, LTW93, L95, L98]); complex updates are also considered a whole, rather than separately, thus generating simpler weakest preconditions. Following the method initiated in [N82], we check only the relevant part of the weakest precondition (i.e. those facts potentially affected by the update); to this end, we preventively simplify the weakest precondition by using a resolution method: we separate in the weakest precondition the facts which reduce to \( c \) (assumed to hold) from the ‘new’ facts which have to be checked (see also [BDM88, N82, SK88]).

Our update language is more expressive than the ones considered in [BS98, BDM88, LST87, N82] in that we allow for: 1. more complex updates, inserting or deleting sets defined by a qualification which is a universal formula, 2. conditional updates, 3. complex transactions, and 4. recursively defined updates and constraints. The language of [BS98] has an additional statement \( \text{for one } x \text{ where cond do inst} \) which can be simulated in our language; in addition it is object-oriented, as are the languages of [L95, L98]. Our update language is in some respects more user-friendly than the one considered in [L95, L98] (because we allow for conditional updates, and, in the deductive case, we allow for insertions or deletions of rules); in some respects it is less expressive (because our qualifications are universal formulas instead of arbitrary first order formulas in [L95, L98, M97]); our system has been extended to allow for some existential qualtifiers and in practice, our qualifications suffice to model usual update languages. This slight loss in expressivity enables us to explicitly and effectively give an automatic procedure for generating a simplified weakest precondition, implemented via a terminating term rewriting system. [LTW93, L98] give only sufficient conditions under which simplifications are possible, and state the existence of a simplified weakest precondition, without giving an algorithm to compute it.

The parallel time complexity for computing \( wp(u, c) \) is linear in the total size of the formulas involved (constraint, qualification, etc.). The maximum size of \( wp(u, c) \) is also linear in the size of the formulas involved, except for the case of an update of the form \( \text{insert}_{R \Phi} \) paired with a constraint \( c \) containing \( k > 1 \) occurrences of \( \neg r \), when a blow-up exponential in \( k \)

\( \text{Some new facts could be already present in the old database or could be generated twice; this is unavoidable, unless we are willing to perform a semantical analysis of the program which can be expensive.} \)
may occur in the size of $wp(u,c)$ (and similarly for $\text{delete}_g\Phi$ with $k > 1$ occurrences of $r$ in $c$).

Because our weakest precondition is defined independently of whether $c$ holds in $B$, and is then simplified by taking into account whether $c$ holds in $B$, our approach can be extended to handle changes in the integrity constraints. Further steps would be:

1. to apply semantic query optimization techniques for recursive programs [CGM90, M98] to simplify even more our simplified weakest preconditions;

2. to incorporate in our language complex updates (e.g. modifications, exchanges);

3. to generalize our algorithms to allow for constraints which are not given by clauses.

References


