Automatic generation of simplified weakest preconditions for integrity constraint verification

A. Ai T -Bouziad, Irene Guessarian, L. Vieille

Automatic Generation of Simplified Weakest Preconditions for Integrity Constraint Verification

Ahmed Aït-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, 2. $wp(c, u)$ 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

* Ahmed Aït-Bouziad: LIAFA, Université Paris 6, 4 Place Jussieu, 75252 Paris Cedex 5, France, bouziad@liafa.jussieu.fr

** Irène Guessarian: LIAFA, Université Paris 6, 4 Place Jussieu, 75252 Paris Cedex 5, France, *address all correspondence to* ig@liafa.jussieu.fr

*** Laurent Vieille: Next Century Media, Inc., 12 Av. des Prés, 78180 Montigny le Bretonneux, France lvieille@computer.org

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 contructs. 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 only the 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 $post(u, c) \Longrightarrow c$ 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 [BDA98]: 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 $\langle R_1, R_2, ..., R_n\rangle$ 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 $B = \langle R_1, R_2, ..., R_n\rangle$ to $u(B) = \langle R'_1, R'_2, ..., R'_n\rangle$ 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 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 $B \models g \Longrightarrow B \models f$. 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. $B \models f \Longrightarrow u(B) \models c$ *regardless* of whether $B \models c$.

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

- $B' = B[R \rightarrow R']$ 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 $r(\overline{s}) \vee \phi(\overline{s})$ (resp. $r(\overline{s}) \wedge \neg\phi(\overline{s})$) for every occurrence of $r(\overline{s})$ in c.
- $c[+r \rightarrow r \cup \phi]$ (resp. $c[-r \rightarrow \neg r \cup \phi]$) denotes the result of substituting $r(\overline{s}) \vee \phi(\overline{s})$ (resp. $\neg r(\overline{s}) \vee \phi(\overline{s})$) for every positive[2] (resp. negative) occurrence of $r(\overline{s})$ in c.

[2] An occurrence of $r(\overline{s})$ is positive if it is within the scope of an even number of negations. It is negative otherwise.

2.2 The update language

Let Φ be a first-order formula with free variables $\overline{x}$ and *cond* a first-order sentence which are domain independent [ToS88, VGT87]. The instructions of our language are defined as follows.

Definition 1 I_1. `foreach` $\overline{x} : \Phi(\overline{x})$ `do` $insert_R(\overline{x})$

I_2. `foreach` $\overline{x} : \Phi(\overline{x})$ `do` $delete_R(\overline{x})$

I_3. *If i_1 and i_2 are instructions then $(i_1; i_2)$ is an instruction;*

I_4. `if` *cond* `then` *inst*1 `else` *inst*2 *is an instruction.The alternative else inst*2 *is optional.*

For instance, adding tuple $\overline{a}$ in relation R, will be denoted by `foreach` $\overline{x} : \overline{x} = \overline{a}$ `do` $insert_R(\overline{x})$. In the sequel, we will abbreviate it by: $insert_R(\overline{a})$. Formula Φ is called the qualification of the update.

The update language of [LTW93] consists of instructions I_1, I_2 and 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

$$\texttt{foreach } \overline{x} : \Phi(\overline{x}) \texttt{ do } (i_1; i_2) \tag{1}$$

where e.g. for $j = 1, 2$, $i_j =$ `foreach` $\overline{x_j} : \Phi_j(\overline{x_j})$ `do` $insert_{R_j}(\overline{x_j})$ will be expressed in our language by $i_0; i'_1; i'_2$, where *TEMP* is a suitable, initially empty, temporary relation, and $\overline{y_j} = \overline{x} \setminus \overline{x_j}$ for $j = 1, 2$:

$$\begin{aligned} i_0 &= \texttt{foreach } \overline{x} : \Phi(\overline{x}) \texttt{ do } insert_{TEMP}(\overline{x}) \\ i'_j &= \texttt{foreach } \overline{x_j} : \left(\forall \overline{y_j}\ temp(\overline{x}) \wedge \Phi_j(\overline{x_j})\right) \texttt{ do} \\ &\quad insert_{R_j}(\overline{x_j}). \end{aligned}$$

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 $\Phi(\overline{x})$ thus producing the set of instances of $\overline{x}$ satisfying $\Phi(\overline{x})$ in B, and then inserting (resp. deleting) from R these instances of $\overline{x}$. I_3 is performed by executing first I_1 and then I_2. I_4 is performed by evaluating condition *cond* on B, if *cond* is true *inst*1 is executed, otherwise *inst*2 is executed. Complex instructions such as `if` *cond* `then` $(i_1; i_2)$ `else` i_3 are performed by evaluating first *cond* and then, if e.g. *cond* is true, executing i_1 and then i_2, regardless of whether *cond* holds after executing i_1.

Formally, the update $\lfloor i \rfloor(B)$ associated with instruction i is defined by:

1. $\lfloor I_1 \rfloor(B) = B[R \rightarrow R \cup \{\overline{x} | B \models \Phi(\overline{x})\}]$
2. $\lfloor I_2 \rfloor(B) = B[R \rightarrow R - \{\overline{x} | B \models \Phi(\overline{x})\}]$

3. $\lfloor I_3 \rfloor(B) = \lfloor i_2 \rfloor(\lfloor i_1 \rfloor(B))$

4. $\lfloor I_4 \rfloor(B) = \begin{cases} \lfloor inst1 \rfloor(B) & if \quad B \models cond \\ \lfloor inst2 \rfloor(B) & if \quad B \models \neg cond \end{cases}$

To simplify, $\lfloor i \rfloor(B)$ will be denoted by $i(B)$.

We can easily generalize the transformation of [LTW93] to obtain weakest preconditions for our language.

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. $wp(\texttt{foreach } \overline{x} : \Phi(\overline{x}) \texttt{ do } insert_R(\overline{x}), c) = c[r \rightarrow r \cup \Phi]$,

2. $wp(\texttt{foreach } \overline{x} : \Phi(\overline{x}) \texttt{ do } delete_R(\overline{x}), c) = c[r \rightarrow r - \Phi]$,

3. $wp((i_1; i_2), c) = wp(i_1, (wp(i_2, c))$,

4. $wp(\texttt{if } cond \texttt{ then } inst1 \texttt{ else } inst2, c) = \big(cond \wedge wp(inst1, c)\big) \vee \big(\neg cond \wedge wp(inst2, c)\big)$.

Example 1 1. Let $c = \forall \overline{x}\ (r(\overline{x}) \rightarrow q(\overline{x}))$ and let $i = \texttt{foreach } \overline{x} : p(\overline{x}) \texttt{ do } insert_R(\overline{x})$, then

$$wp(i, c) \quad = \quad \forall \overline{x}((r(\overline{x}) \vee p(\overline{x})) \rightarrow q(\overline{x}))$$

2. Let $c = \forall \overline{x}(r(\overline{x}) \rightarrow q(\overline{x}))$ and let $i = (i_1; i_2)$ where

$$\begin{array}{rcl} i_1 & = & \texttt{foreach } \overline{x} : s(\overline{x}) \texttt{ do } delete_R(\overline{x}) \\ i_2 & = & \texttt{foreach } \overline{y} : p(\overline{y}) \texttt{ do } insert_R(\overline{y}) \end{array}$$

then

$$\begin{array}{rcl} wp(i, c) & = & wp((i_1; i_2), c) = wp(i_1, wp(i_2, c)) \\ & = & wp(i_1, \forall \overline{x}((r(\overline{x}) \vee p(\overline{x})) \rightarrow q(\overline{x}))) \\ & = & \forall \overline{x}\big(\big((r(\overline{x}) \wedge \neg s(\overline{x})) \vee p(\overline{x})\big) \rightarrow q(\overline{x})\big) \end{array}$$

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: $B \models wp' \Leftrightarrow B \models wp$. Consider for instance the update $i = \texttt{foreach } \overline{x} : p(\overline{x}) \texttt{ do } insert_R(\overline{x})$ of Example 1 1, with $c = \forall \overline{x}\ (r(\overline{x}) \rightarrow q(\overline{x}))$ then we can simplify $wp(i, c)$ as follows

$$\begin{array}{rcl} wp(i, c) & = & \forall \overline{x}((r(\overline{x}) \vee p(\overline{x})) \rightarrow q(\overline{x})) \\ & \equiv & \forall \overline{x}(r(\overline{x}) \rightarrow q(\overline{x})) \wedge \forall \overline{x}(p(\overline{x}) \rightarrow q(\overline{x})) \\ & = & c \wedge \forall \overline{x}(p(\overline{x}) \rightarrow q(\overline{x})) \end{array}$$

whence $wp' = \forall \overline{x}(p(\overline{x}) \rightarrow q(\overline{x}))$. 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 $wp = c \wedge wp_1$, 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 $\exists \overline{x} r(\overline{x})$ and let u be the update $delete_R(\overline{a})$; then $wp(c, u) = \exists \overline{x}\big(r(\overline{x}) \wedge \neg(\overline{x} = \overline{a})\big)$ 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 $cond$ are universal sentences, and
2. the qualifications $\Phi(\overline{x})$ are formulas without quantifications.

Recall that a clause is a sentence of the form $c = \forall \overline{x}\ (l_1 \vee l_2 \vee ... \vee l_m)$ where the l_is are literals. Without loss of generality, we can restate our hypotheses as

1. constraint c is a single clause; indeed if $c = \forall \overline{x}\ (D_1 \wedge D_2 \wedge ... \wedge D_n)$ is in conjunctive normal form, then we can replace c by the n constraints $\forall \overline{x} D_i$ which are clauses and can be studied independently.
2. the qualifications $\Phi(\overline{x})$ are conjunctions of literals; we can write $\Phi(\overline{x}) = \Phi_1(\overline{x}) \vee \Phi_2(\overline{x}) \vee ... \vee \Phi_n(\overline{x})$ in disjunctive normal form, and we replace the instruction qualified by $\Phi(\overline{x})$ by a sequence of similar instructions qualified by $\Phi_1(\overline{x})$, $\Phi_2(\overline{x})$,..., $\Phi_n(\overline{x})$.
3. the conditions $cond$ are single clauses: assuming as above that $cond = \forall \overline{x}(D_1 \wedge D_2 \wedge ... \wedge D_n)$ is in conjunctive normal form, we let $cond_i = \forall \overline{x} D_i$ and we replace e.g. `if` $cond$ `then` $inst$ by `if` $cond_1$ `then (if` $cond_2$ `then (...(if` $cond_n$ `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 $cond$ and $\Phi(\overline{x})$ are also satisfied.

We will use the following notation: let S be a set of clauses and r a predicate symbol; $Res_r(S)$ 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 $\neg(\overline{x} = \overline{a})$ (which we write as $\overline{x} \neq \overline{a}$), then the resolvent is simplified by deleting that literal and substituting $\overline{a}$ for $\overline{x}$ in the resolvent.

Example 2 1. Let $S = \{\neg r(x,y) \vee q(y,z)\ ,\ r(x,y) \vee \neg q(x,y)\}$; then $Res_r(S) = \{q(y,z) \vee \neg q(x,y)\}$.

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

$$\{\quad \neg r(x,z) \vee q(y,z) \vee \big((x,y) \neq (a,b)\big)\ , \quad \neg r(x,y') \vee q(y',y) \vee \big((x,y) \neq (a,b)\big)\quad \}$$

they are then simplified into

$$Res_r(S) = \{\neg r(a,z) \vee q(b,z)\ ,\ \neg r(a,y') \vee q(y',b)\}.\square$$

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

$$\begin{array}{lcl} wp(u,c) & \longrightarrow & \bigwedge wp(u,c_i) \\ wp(u,c) & \longrightarrow & c' \end{array}$$

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 I_1 : `foreach` $\overline{x}$: $\Phi(\overline{x})$ `do` $insert_R(\overline{x})$, and we write $r \vee \neg\phi$ instead of $\forall\overline{x}\big(R(\overline{x}) \vee \neg\Phi(\overline{x})\big)$. Similarly, $delete_R\Phi$, (resp. $\neg r \vee \neg\phi$) abbreviates I_2 : `foreach` $\overline{x}$: $\Phi(\overline{x})$ `do` $delete_R(\overline{x})$, (resp. $\forall\overline{x}\big(\neg R(\overline{x}) \vee \neg\Phi(\overline{x})\big)$). In updates $insert_R\Phi$ or $delete_R\Phi$, we may assume that r does not occur in Φ: indeed any occurrence of r in Φ 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 *1. 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 $wp = swp \wedge c'$, with c' such that $c \Longrightarrow c'$; if c holds, wp can be simplified into the weakest precondition swp 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 $wp(u,c)$ has a *unique* derivation (up to the order in which the rewritings are applied).

$$\begin{array}{lrcl}
\mathrm{R}_1: & wp(insert_R\Phi,c) & \longrightarrow & c[+r \to r \cup \phi] \\
 & & & \text{if } Res_r(c, r \vee \neg\phi) = \emptyset \\
\mathrm{R}_2: & wp(insert_R\Phi,c) & \longrightarrow & c[+r \to r \cup \phi] \wedge \\
 & & & \bigwedge_{c_j \in Res_r(c, r\vee\neg\phi)} wp(insert_R\Phi, c_j) \quad \text{otherwise} \\
\mathrm{R}_3: & wp(delete_R\Phi,c) & \longrightarrow & c[-r \to \neg r \cup \phi] \\
 & & & \text{if } Res_r(c, \neg r \vee \neg\phi) = \emptyset \\
\mathrm{R}_4: & wp(delete_R\Phi,c) & \longrightarrow & c[-r \to \neg r \cup \phi] \wedge \\
 & & & \bigwedge_{c_j \in Res_r(c, \neg r\vee\neg\phi)} wp(delete_R\Phi, c_j) \quad \text{otherwise} \\
\mathrm{R}_5: & wp(I_3,c) & \longrightarrow & wp(i_1, (wp(i_2,c)) \\
\mathrm{R}_6: & wp(I_4,c) & \longrightarrow & \big(cond \wedge wp(inst1,c)\big) \\
 & & & \vee\big(\neg cond \wedge wp(inst2,c)\big) \\
\mathrm{R}_7: & wp(u,\neg c) & \longrightarrow & \neg wp(u,c) \\
\mathrm{R}_8: & wp(u, c_1 \wedge c_2) & \longrightarrow & wp(u,c_1) \wedge wp(u,c_2) \\
\mathrm{R}_9: & wp(u, c_1 \vee c_2) & \longrightarrow & wp(u,c_1) \vee wp(u,c_2)
\end{array}$$

Figure 1: Simplification rules

2. We prove by induction that our rewriting system generates a wp in the form $wp = swp \wedge c'$, with c' such that $c \Longrightarrow c'$ holds. Therefore, taking into account that c holds in B, we can simplify wp into swp, and $wp \Longrightarrow swp$ 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, swp is simpler than the weakest preconditions of [LTW93, L95, L98] and Theorem 1. Example 3 1 shows that $swp \not\Longleftrightarrow wp$, i.e. our swp can be strictly simpler than wp.$\square$

Example 3 1. Let $c = \forall x,y,z\ \big((p(x,y) \wedge q(y,z)) \longrightarrow (p(x,z) \vee q(x,z))\big)$ and let $i =$ `foreach` $x,y : (x,y) = (a,a)$ `do` $insert_P(x,y)$, i.e. $\Phi(\overline{x})$ is $(x,y) = (a,a)$, then the method of [LTW93] gives

$$\begin{array}{rcl} wp(i,c) & = & \forall x,y,z\ \big(((p(x,y) \vee (x,y) = (a,a)) \wedge q(y,z)) \\ & & \longrightarrow ((p(x,z) \vee (x,y) = (a,a)) \vee q(x,z))\big) \end{array}$$

and our method gives the sequence of rewritings:

$$\begin{array}{lcll} wp(i,c) & \longrightarrow_{R_2} & (\neg q(a,z) \vee p(a,z) \vee q(a,z)) & \\ & & \wedge c[+p \to p \cup (x,y) = (a,a)] & \\ & \longrightarrow & (\neg q(a,z) \vee p(a,z) \vee q(a,z)) & (2) \\ & \longrightarrow & true & (3) \end{array}$$

The simplification of line 2 is obtained by taking into account the fact that c holds in B, hence $c[+p \to p \cup (x,y) = (a,a)]$ also holds. Because $\neg q(a,z) \vee q(a,z) =$

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:

$$\begin{array}{lll} wp((i_1;i_2),c) & \longrightarrow_{R_5} & wp(i_1, wp(i_2,c)) \\ & \longrightarrow_{R_2} & wp\big(i_1, c \wedge wp(i_2, \neg p \vee q)\big) \\ & \longrightarrow_{R_1} & wp\big(i_1, c \wedge (\neg p \vee q)\big) \qquad (4) \\ & \longrightarrow_{R_8} & wp(i_1,c) \wedge \underline{wp\big(i_1, (\neg p \vee q)\big)} \\ & \longrightarrow_{R_3} & wp(i_1,c) \wedge (\neg p \vee q) \\ & \longrightarrow_{R_3} & \big(c[-r \rightarrow \neg r \cup s] \wedge (\neg p \vee q)\big) \\ & \longrightarrow & (\neg p \vee q) \end{array}$$

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 $c[-r \rightarrow \neg r \cup s]$ also holds.

3. Let $c = \forall x, y, z\ \big(\neg p(x,y) \vee \neg p(x,z) \vee q(y,z)\big)$ and let $i =$ `foreach` $x, y : (x,y) = (a,b)$ `do` $insert_P(x,y)$, then our method gives:

$$\begin{array}{lll} wp(i,c) & \longrightarrow_{R_2} & wp(i, \neg p(a,z) \vee q(b,z)) \\ & & \quad \wedge wp(i, \neg p(a,y) \vee q(y,b)) \wedge c \\ & \longrightarrow^*_{R_2} & wp(i, q(b,b)) \wedge \big(\neg p(a,z) \vee q(b,z)\big) \\ & & \quad \wedge \big(\neg p(a,y) \vee q(y,b)\big) \wedge c \\ & \longrightarrow_{R_1} & q(b,b) \wedge \big(\neg p(a,z) \vee q(b,z)\big) \\ & & \quad \wedge \big(\neg p(a,y) \vee q(y,b)\big) \wedge c \qquad (5) \end{array}$$

(where $\longrightarrow^*_{R_2}$ means that several $\longrightarrow_{R_2}$ rewriting steps are performed); assuming c holds in B, we can simplify 5 into $swp = \forall y, z\ q(b,b) \wedge \big(\neg p(a,z) \vee q(b,z)\big) \wedge \big(\neg p(a,y) \vee q(y,b)\big)$ which is to be verified in B. Assuming c holds in B, the methods of [N82, BDM88, BD88] yield the postcondition $\forall y, z\ \big(\neg p(a,z) \vee q(b,z)\big) \wedge \big(\neg p(a,y) \vee q(y,b)\big)$ 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 `delta` and `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 $q_1, \ldots, q_l$ –called intensional predicates (or IDBs)–, a DATALOG program P is a finite set of function-free Horn clauses, called rules, of the form:

$$q(y_1, ..., y_n) \longleftarrow q_1(y_{1,1}, ..., y_{1,n_1})\ ,\ \ldots\ ,\ q_p(y_{p,1}, ..., y_{p,n_p})$$

where the y_is and the $y_{i,j}$s are either variables or constants, q is an intensional predicate in $\{q_1, \ldots, q_l\}$, the q_is 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 *insert* and *delete* statements are conjunctions of literals which may contain atoms defined by recursive DATALOG programs,
- similarly, constraints c and conditions *cond* in `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 1. Let Q and R be relations appearing in a DATALOG program P, and respectively corresponding to the predicate symbols q and r: q is said to *depend directly* on r if $q = r$ or if r appears in the body of a rule defining q; q is said to *depend* on r if q depends directly on r, or q depends directly on q' and q' depends on r.

2. Let P'_r be the program obtained by adding to P new IDB symbols q' for each predicate q depending on r, and new rules; for each rule ρ of P defining an IDB q depending on r, a new rule ρ' defining q' is added: ρ' is obtained from ρ 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 r (including r when r is an EDB) are replaced by the corresponding primed predicates.□

We will assume the following hypotheses:

H_1: the qualifications $\Phi(\overline{x})$ are conjunctions of literals, and all the rules defining the IDBs in constraint c,

qualifications $\Phi(\overline{x})$ and conditions *cond* are given in program P.

H_2: in statements of the form `foreach` $\overline{x} : \Phi(\overline{x})$ `do` $insert_R(\overline{x})$, or `foreach` $\overline{x} : \Phi(\overline{x})$ `do` $delete_R(\overline{x})$, none of the literals in $\Phi(\overline{x})$ depends on r.

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

1. $wp($`foreach` $\overline{x} : \Phi(\overline{x})$ `do` $insert_R(\overline{x}), c) = c[r \to r']$ *where the program defining the IDBs is* $P'_{insert_R\Phi} = P'_r \cup \{r'(\overline{x}) \longleftarrow r(\overline{x}) \ , \ r'(\overline{x}) \longleftarrow \Phi(\overline{x})\}$,

2. $wp($`foreach` $\overline{x} : \Phi(\overline{x})$ `do` $delete_R(\overline{x}), c) = c[r \to r']$ *where the program defining the IDBs is* $P'_{delete_R\Phi} = P'_r \cup \{r'(\overline{x}) \longleftarrow r(\overline{x}) \wedge \neg t(\overline{x}) \ , \ t(\overline{x}) \longleftarrow \Phi(\overline{x})\}$, *with t a new IDB predicate,*

3. $wp((i_1; i_2), c) = wp(i_1, (wp(i_2, c))$,

4. $wp($`if` *cond* `then` *inst*1 `else` *inst*2$, c) = \big(cond \wedge wp(inst1, c)\big) \vee \big(\neg cond \wedge wp(inst2, c)\big)$.

Theorem 3 calls for some remarks.

1. When none of r, c or Φ 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 Φ containing negations force us out of the DATALOG framework, because the weakest precondition of `foreach` $\overline{x} : \Phi(\overline{x})$ `do` $delete_R(\overline{x})$ and constraint c is defined by the DATALOG$^\neg$ program $P'_r \cup \{r'(\overline{x}) \longleftarrow r(\overline{x}) \wedge \neg t(\overline{x}) \ , \ t(\overline{x}) \longleftarrow \Phi(\overline{x})\}$; this was already noted in [GSUW94]. In [L95, LST87] a stratified DATALOG$^\neg$ framework is assumed: this ensures that both the constraint and the 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 wps in DATALOG. The *wp*s defined in Theorem 3 are correct without this restriction, but they are defined by stratified DATALOG$^\neg$ programs, and not by Horn clauses.

Example 4 Let c be the constraint $\forall x, y \ \neg tc(x, y) \vee i(x, y)$, where i does not depend on tc, and consider the update `foreach` $x, y : path(x, y)$ `do` $insert_{TC}(x, y)$, where all the predicates are defined by program P:

$$P \left\{ \begin{array}{lcl} tc(x,y) & \longleftarrow & arc(x,y) \\ tc(x,y) & \longleftarrow & arc(x,z) \ , \ tc(z,y) \\ path(x,y) & \longleftarrow & edge(x,y) \\ path(x,y) & \longleftarrow & edge(x,z) \ , \ path(z,y) \\ i(x,y) & \longleftarrow & body(x,y) \end{array} \right.$$

Then P'_{tc} is P together with a new predicate tc' and the rules 6 and 7.

$$\begin{array}{lcll} tc'(x,y) & \longleftarrow & arc(x,y) & (6) \\ tc'(x,y) & \longleftarrow & arc(x,z) \ , \ tc'(z,y) & (7) \end{array}$$

and $P'_{insert_{TC}path}$ is P'_{tc} together with the rules 8 and 9.

$$\begin{array}{lcll} tc'(x,y) & \longleftarrow & tc(x,y) & (8) \\ tc'(x,y) & \longleftarrow & path(x,y) & (9) \end{array}$$

Finally, $wp(u, c) = \forall x, y \ \neg tc'(x, y) \vee i(x, y)$, where TC' is defined by $P'_{insert_{TC}path} = P \cup \{6, 7, 8, 9\}$.

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

$$c \Longrightarrow wp(u, c) \qquad (10)$$

The problem is that implication 10 is undecidable except in some special cases: e.g., if both c and $wp(u, c)$ 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 $wp(u, c)$. We illustrate this idea on an example.

Example 5 Let c be the constraint $\forall x, y \ \neg tc(x, y) \vee i(x, y)$, where I and TC are defined by P:

$$\begin{array}{lcll} tc(x,y) & \longleftarrow & arc(x,y) & (11) \\ tc(x,y) & \longleftarrow & edge(x,y) & (12) \\ tc(x,y) & \longleftarrow & arc(x,z) \ , \ tc(z,y) & (13) \\ i(x,y) & \longleftarrow & i_1(x,z_1), edge(z_1,z_2), i_1(z_2,y) & (14) \\ i_1(x,y) & \longleftarrow & arc(x,z) \ , \ i_1(z,y) & (15) \\ i_1(x,y) & \longleftarrow & edge(x,z) \ , \ i_1(z,y) & (16) \\ i_1(x,x) & \longleftarrow & & (17) \end{array}$$

and consider the update $u =$ `foreach` $x, y : edge(x, y)$ `do` $insert_{Arc}(x, y)$. Then $wp(u, c) = \forall x, y \ \neg tc'(x, y) \vee i(x, y)$, where I and TC' are defined by $P'_{insert_{Arc}edge}$, consisting of P together with the rules (because $I' = I$ here):

$$\begin{array}{lcl} arc'(x,y) & \longleftarrow & arc(x,y) \\ arc'(x,y) & \longleftarrow & edge(x,y) \\ tc'(x,y) & \longleftarrow & arc'(x,y) \\ tc'(x,y) & \longleftarrow & arc'(x,z) \ , \ tc'(z,y) \end{array}$$

We prove that $c \Longrightarrow wp(u, c)$ by induction. To this end, let $C[R] = (R \subset I)$; we note that c holds iff $C[TC]$ holds, and $wp(u, c)$ holds iff $C[TC']$ holds.

Let $\bowtie$ denote the composition[3] of binary relations. It thus suffices to prove that, for any R, $C[R] \Longrightarrow C[Arc' \cup Arc' \bowtie R]$ to conclude, by fixpoint induction, that $C[TC']$ holds.

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

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 $c = \neg\exists x\ tc(x,x)$ where TC is a transitive closure. Such a constraint is used, for instance, to check that a set of DATALOG clauses defines a non recursive program by verifying that the precedence graph of the IDBs occurring in the program has no cycle. We want to check that adding a new clause does not create recursions, i.e. cycles. Adding a new clause corresponds to an insertion update.

Example 6 Consider the update $insert_{Arc}(d,b)$, and let c be the constraint $\neg\exists x\ tc(x,x)$, where TC is defined by program P:

$$P \left\{ \begin{array}{lll} tc(x,y) & \longleftarrow & arc(x,y) \\ tc(x,y) & \longleftarrow & arc(x,z)\ ,\ tc(z,y) \end{array} \right.$$

We assume that constraint c is verified by database B. Let $\Delta_{tc}(x,y)$ be the potentially new facts which will be inserted in TC as a consequence of the update. The IDB predicate δ_{tc} corresponding to Δ_{tc} is defined by the DATALOG program:

$$P' \left\{ \begin{array}{lll} tc(x,y) & \longleftarrow & arc(x,y) \\ tc(x,y) & \longleftarrow & arc(x,z)\ ,\ tc(z,y) \\ \delta_{tc}(d,b) & \longleftarrow & \\ \delta_{tc}(d,y) & \longleftarrow & tc(b,y) \\ \delta_{tc}(x,y) & \longleftarrow & arc(x,z)\ ,\ \delta_{tc}(z,y) \end{array} \right.$$

The weakest precondition $wp(c, insert_{Arc}(d,b))$ then is $\neg\exists x\ \delta_{tc}(x,x)$, which can be evaluated by SLD-AL resolution [V89]. The weakest precondition of [L95] and of Theorem 3 would be in the present case $\neg\exists x\ tc'(x,x)$ where TC' is defined by the program $P'_{insert_{Arc}(d,b)}$:

$$P'_{insert_{Arc}(d,b)} \left\{ \begin{array}{lll} tc(x,y) & \longleftarrow & arc(x,y) \\ tc(x,y) & \longleftarrow & arc(x,z)\ ,\ tc(z,y) \\ tc'(x,y) & \longleftarrow & arc'(x,y) \\ tc'(x,y) & \longleftarrow & arc'(x,z)\ ,\ tc'(z,y) \\ arc'(x,y) & \longleftarrow & arc(x,y) \\ arc'(d,b) & \longleftarrow & \end{array} \right.$$

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, BDM88, 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 $insert_R(\overline{d_j})$ for $j = 1, \ldots, k$, with r an EDB predicate, and the constraint is of the form $\neg\exists\overline{x}\ t(\overline{x})$ with t an IDB, possibly depending on r, defined by a linear DATALOG program.

Algorithm. *Inputs:* update $u = insert_R(\overline{d_j})$ for $j = 1, \ldots, k$, constraint $c = \neg\exists\overline{x}\ t(\overline{x})$, 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 IDB q of P depending on r, let δ_q be a new IDB; let Π be the set of rules defined as follows: for each rule $q \longleftarrow body$ of P whose head q depends on r add in Π a rule $\delta_q \longleftarrow body$. If Π is empty, then update u is safe; STOP.

Step 2: Let $P_1 = \{Res(\rho, r(\overline{d_j}))\ /\ \rho \in \Pi, j = 1, \ldots, k\}$ be the set of resolvents of rules of Π with updated atoms. If P_1 is empty, then update u is safe; STOP.
Otherwise, generate new rules as follows:
$i := 1$ WHILE $P_i \neq \emptyset$ DO $P_{i+1} = \{Res(\rho, r(\overline{d_j}))\ /\ \rho \in P_i, j = 1, \ldots, k\}$; $i := i + 1$ ENDDO
Let $P' = \cup P_i$.

Step 3: Let Σ_1 be the set of rules of Π which contain an IDB q depending on r in their body. If Σ_1 is empty, then $wp(u,c) = \neg\exists\overline{x}\ \delta_t(\overline{x})$, and program $P \cup P'$ defines Δ_t; STOP.
Otherwise, generate new rules as follows: let P''_1 be obtained from Σ_1 by substituting δ_q for q in the bodies of the rules of Σ_1; only new predicates δ_q appear in rules of P''_1.
$i := 1$ WHILE $P''_i \neq \emptyset$ DO $P''_{i+1} = \{Res(\rho, r(\overline{d_j}))\ /\ \rho \in P''_i, j = 1, \ldots, k\}$; $i := i + 1$ ENDDO
Let $P'' = \cup P''_i$.

[3] $\bowtie$ performs an equijoin on the second attribute of the first relation and the first attribute of the second relation, followed by a projection on the first and third attributes.

$wp(u,c) = \neg\exists\overline{x}\ \delta_t(\overline{x})$, and program $P \cup P' \cup P''$ defines Δ_t; STOP.□

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 $\delta_{tc}(d,y) \longleftarrow \delta_{tc}(b,y)$ 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 $u =$ `foreach` $x,y : path(x,y)$ `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':

$$P' \left\{ \begin{array}{lcl} P & & \\ \delta_{tc}(x,y) & \leftarrow & edge(x,y) \\ \delta_{tc}(x,y) & \leftarrow & edge(x,z)\ ,\ tc(z,y) \\ \delta_{tc}(x,y) & \leftarrow & edge(x,z)\ ,\ \delta_{tc}(z,y) \\ \delta_{tc}(x,y) & \leftarrow & arc(x,z)\ ,\ \delta_{tc}(z,y) \end{array} \right.$$

The weakest precondition $wp(c, insert_{Arc}path)$ 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 $P'_{insert_{Arc}edge}$, namely:

$$P' \left\{ \begin{array}{lcl} P & & \\ tc'(x,y) & \longleftarrow & arc'(x,y) \\ tc'(x,y) & \longleftarrow & arc'(x,z)\ ,\ tc'(z,y) \\ arc'(x,y) & \longleftarrow & arc(x,y) \\ arc'(x,y) & \longleftarrow & path(x,y) \end{array} \right.$$

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 $c \Longrightarrow wp(u,c)$ holds without evaluating $wp(u,c)$; the second one consists in defining, for insertion updates and constraints c of the form $\neg\exists\overline{x}\ q(\overline{x})$, a constraint c' simpler than $wp(u,c)$ and such that $c \Longrightarrow \Big(c' \Longleftrightarrow wp(u,c)\Big)$; 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 `forone` x `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 quatifiers 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 $insert_R\Phi$ paired with a constraint c containing $k > 1$ occurrences of $\neg r$, when a blow-up exponential in k

[4] 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 $delete_R\Phi$ 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

[AHV95] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[BGL96] M. Benedikt, T. Griffin, L. Libkin, Verifiable properties of database transactions, PODS'96, 1996, pp. 117–127.

[BLSB92] W. Bibel, R. Letz, J. Schumann, S. Bayerl, SETHEO: A High-Performance Theorem Prover, Journal of Symbolic Computation 15(5-6), 1992, pp.183-212.

[BDA98] N. Bidoit, S. De Amo, A first step towards implementing dynamic algebraic dependences, Theoretical Computer Science 190, 1998, pp. 115-149.

[BD88] F. Bry, H. Decker, Préserver l'Intégrité d'une Base de Données Déductive : une Méthode et son Implémentation. In Proc. 4èmes Journées Bases de Données Avancées (BDA), May 1988.

[BS98] V. Benzaken, X. Schaeffer, Static management of integrity in Object-Oriented databases: design and implementation, Proc. EDBT'98, LNCS 1377, Springer-Verlag, Berlin, 1998, pp. 311–325.

[BDM88] F. Bry, H. Decker, R. Manthey, A uniform approach to constraint satisfaction and constraint satisfiability in deductive object-oriented databases, Proc. 1st EDBT, 1988, pp. 488–505.

[CB80] M.A. Casanova, P.A. Bernstein, A formal System for Reasoning about Programs accessing a Relational Database, ACM Transactions on Database Systems 2 (3), 1980, pp. 386–414.

[CGM90] U. Chakravarthy, J. Grant, J. Minker, Logic-based approach to semantic query optimization, ACM Transactions on Database Systems 15 (2), 1990, pp. 162–207.

[CV92] S. Chaudhuri, M. Y. Vardi, On the equivalence of datalog programs, In Proc. Eleventh ACM Symposium on Principles of Database Systems, 1992, pp. 55–66.

[C91] B. Courcelle, Recursive queries and context-free graph grammars, Theor. Comput. Sc. 78, 1991, pp. 217–244.

[D75] E. Dijkstra, Guarded commands, nondeterminacy and formal derivations of programs, Comm. of the ACM, 18(8), 1975, pp. 453–457.

[D76] E. Dijkstra, A discipline of programming, Prentice-Hall, London, 1976.

[GM79] G. Gardarin, M. Melkanoff, Proving the Consistency of Database Transactions, Proc. VLDB 1979, pp. 291–298.

[GSUW94] A. Gupta, Y. Sagiv, J. D. Ullman, J. Widom, Constraint checking with partial information, Proc. PODS'94, 1994, pp. 45–55.

[L98] M. Lawley, Program transformations for proving database transaction safety, PhD.Th., Griffith University, 1998.

[L95] M. Lawley, Transaction safety in deductive object-oriented databases, Proc. 4th International Conference on Deductive and Object-Oriented Databases, LNCS 1013, Springer-Verlag, Berlin 1995, pp. 395–410.

[LTW93] M. Lawley, R. Topor, M. Wallace, Using weakest preconditions to simplify integrity constraint checking, Proc. 4th Australian database conf., Brisbane, 1993, pp. 161–170.

[LST87] J. Lloyd, E. Sonenberg, R. Topor, Integrity constraint checking in stratified databases, J. of Logic Programming, 4(4), 1987, pp. 331–343.

[M97] N. Magnier, Validation des transactions dans les bases de données: classes décidables et vérification automatique, PhD Thesis, Bordeaux University, 1997.

[M98] J. Minker, Logic and databases: a 20 year retrospective, LNCS 1154, Springer-Verlag, Berlin 1996, pp. 3–57.

[N82] J.-M. Nicolas, Logic for improving integrity checking in relational databases, Acta Informatica, 18, 1982, pp. 227–253.

[PP94] A. Pettorossi, M. Proietti, Transformation of Logic Programs: Foundations and Techniques, Journal of Logic Programming, Vol. 20, 1994, pp. 261-320.

[SS89] T. Sheard, D. Stemple, Automatic verification of database transaction safety, ACM Trans. on Database Systems, 14(3), 1989, pp. 322–368.

[SK88] F. Sadri, R. Kowalski, A theorem-proving approach to database integrity, Foundations of deductive databases and logic programming, Morgan-Kauffmann, 1988, pp. 313-362.

[TS84] H. Tamaki, T. Sato, Unfold/Fold transformation of logic programs, Proc. 2nd logic programming conference, Uppsala, Sweden, 1984.

[ToS88] R. Topor, E. Sonenberg, On domain independent databases, Foundations of deductive databases and logic programming, Morgan-Kauffmann, 1988, pp. 217-240.

[VGT87] A. Van Gelder, R. Topor, Safety and correct translation of relational calculus formulas, Proc.

PODS'87, 1987, pp. 313–327.

[V89] L. Vieille, Recursive Query Processing: the Power of Logic, Theoretical Computer Science, 69(1) 1989, pp. 1-53.

[V98] L. Vieille, From Data Independence to Knowledge Independence: an on-going Story, Proc. VLDB'98.

[VBKL99] L. Vieille, P. Bayer, V. Kuechenhoff, A. Lefebvre , Checking Integrity and Materializing Views by Update Propagation in the EKS system, in Materialized Views, A. Gupta and I. Mumick (Eds) MIT Press, to appear.