TOWARDS EFFECT PRESERVATION OF UPDATES WITH LOOPS

Steffen Jurk
Brandenburg Technical University at Cottbus
P.O. Box 101344, Cottbus 03044, Germany
sj@informatik.tu-cottbus.de

Mira Balaban
Ben-Gurion University
P.O. Box 653, Beer-Sheva 84105, Israel
mira@cs.bgu.ac.il

Abstract Todays technologies are capable to assist database developers in designing complex actions (e.g. stored procedures, active database rules). However, automatic generation and extention of given actions holds a danger — The original intention of actions might get canceled or even reversed, e.g. by some appended repairing action enforcing an integrity constraint. This problem might cause non-desired situations where an action commits although one of the intended database modifications was not applied.

In this paper, we deal with the characterization and preservation of effects (intentions) of database actions. As an extention of our current approach for effect preservation, we present a method for handling updates including non-nested loops. A transformation process is proposed that modifies a given action S to S', such that S' does preserve the effects. In order to reduce additional run-time overhead of S', most of the computation is shifted to compile-time.

Keywords: Intention/Effects of Database Updates, Contradictory Updates, Code Transformation, Transaction Design

1. Introduction

Management of large data-intensive systems requires automatic preservation of semantical correctness. Semantical properties are specified by integrity constraints which can be verified by querying the information base. In dynamic situations, however, where operations can violate necessary properties, the role of integrity enforcement is to guarantee a consistent information base by applying additional

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35693-8_16

M. Gertz (ed.), *Integrity and Internal Control in Information Systems V*

repairing actions. Todays technologies are capable to assist database developers in designing complex actions (e.g. stored procedures, active database rules) by handling standard sets of integrity constraints and its repairs, as well as used defined constraints. For example, [10, 11, 9, 15] and tools like ERWin, DBMain and Silverrun.

The problem of *effect preservation* arises when repairing actions undo the *intention* of the original update or of previous repairing actions. For example, the update $insert(a, e)$ intends that in the database state to which the update evaluates, the entity e is an element of the set a. Effect preservation is not handled within *Rule Triggering Systems (RTS)*, which are implemented in almost every commercial database product ([15, 8, 16]).

In previous works [4, 5] we introduced an intention (effect) preserving mechanism, that combines static analysis of an update with run-time tests. It is based on characterization of contradictory primitive updates.. The effect preserving mechanism that was introduced is a program transformation. The mechanism combines static transformation with run-time tests. At compile-time, a computation tree that analyzes all computation paths of the program is constructed, and the program is augmented with tree navigation and intention tests. The nodes of the tree are associated with maximally reduced constraints that capture the intention preservation requirements. The mechanism achieves a better performance than run-time methods for intention preservation since the time consuming operation of tree construction and constraint reduction is applied at compile-time. Only statically unresolved intentions are left for run-time evaluation.

In this paper we present a first extention of this method for handling updates including non-nested loops. Such updates are needed, for example, for updates that involve insertions of selected values to relations, as in

```
INSERT INTO TABLE1 (SELECT * FROM TABLE2 WHERE <CONDITION>)
```

The main difficulty is that in the presence of loops a node in the tree can correspond to multiple applications of its update. Therefore, the association of maximally reduced constraints with tree nodes requires static analysis for multiple repetitions of updates. In order to achieve effect preservation for loops we modify the computation tree into a computation graph. The reduced constraints associated with the graph nodes involve application of run-time methods, needed for run-time instantiation of variables. The program transformation stays as before. The burden of statically analyzing the intention preservation requirements needed for dynamical repetitions is put on the computation of the reduced constraints. The novelty lies in the insertion of run-time methods into the static constraints.

In section 2 the language of updates is defined, and section 3 introduces the main notions for sequence updates. Section 4 presents the extended method for non-nested loops. Related works in the field of effect preservation are introduced in section 5. The paper is concluded by section 6.

2. AN IMPERATIVE LANGUAGE OF UPDATES

The updates are built over a finite set of typed *state variables* X (a *state space*). A *state* is a well typed value assignment to the variables in X. For example, in a relational database with relations $R_1, \ldots, R_n$, the state space is $\{R_1, \ldots, R_n\}$, and any assignment of concrete relations to the relation variables results a database state. A language over X is denoted $\mathcal{L}(X)$.

The language symbols include, besides the state variables, regular variables, and self-evaluating symbols (language constants). The primitive update commands are *skip* (a no-op operation), $fail$ (rollback, the impossible update), and *assignment updates* – well typed assignments to variables. Assignment updates to state variables are restricted to include no other state variable. The $fail$ update leads to the *undefined state*, which in transactional databases corresponds to the *rollback* operation, which undoes undefined states by restoring the old state.

The three constructors that are studied in this paper are sequential composition, conditional, and non-nested loop. For a condition P, and updates S, S_1, S_2, $S_1; S_2$ denotes the sequential composition of S_1 and S_2, `if` P `then` S_1 `else` S_2 is a P conditioned update and `while` P `do` S is a P conditioned S loop with S as the body of the loop. `if` P `then` S is an abbreviation for `if` P `then` S `else` *skip*. The formal semantics of the language can be defined as in Dijkstra's guarded commands language ([3, 7]).

3. CHARACTERIZATION AND PRESERVATION OF EFFECTS OF SEQUENCE UPDATES OF STATE VARIABLES

In this section we shortly define effects of sequence updates of state variables, and introduce the notion of a *delta-condition* that captures possible contradictory interaction between assignments. Delta-conditions are used to define a compile-time effect preservation transformation of sequence updates. The characterization of effects and the notion of an effect-preserving sequence update are fully defined in [4, 5]. Below we summarize these notions, using examples.

3.1 Effects of Sequence Updates of State Variables

Effects of an update S, denoted $effects(S)$ are postconditions that should hold following the update. The effects of a sequence are computed on the basis of the primitive effects. Note that this section only handled assignments on state space variables. Assignments to local variables are not discussed, since they do not affect the state and therefore are not considered within the effects of an update.

Effects of Primitive Updates: For the two primitives $skip$ and $fail$, their effects derive from their intended semantics: $effects(skip) = true$, $effects(fail) = false$. For assignments to state variables, the effects are domain specific and developer provided. Clearly, we expect that developer provided effects are non-trivial, e.g. $effects(S) = true$.

Example 3.1 (Possible Effects of Primitive Assignments)

- **Sets** – *insert or delete a value e from a set x:*
  $effects(x := insert(x, e)) = (e \in x)$, *and*
  $effects(x := delete(x, e)) = (e \notin x)$.
- **Lists** – *insert an nth element e to a list x:*
  $effects(x := insert(x, n, e)) = (e = element(x, n))$.
- **Trees** – *insert an element e to a tree x:*
  $effects(x := insert(x, path, e)) = (e = element(x, path))$.

Effects of Sequence Updates of State Variables: In [4, 5] we distinguish between *desired effects*, denoted $effects_D(S)$ and *executed effects*, denoted $effects_E(S)$: The first, expresses the aggregated desired effects of all primitive updates in an update and it might not hold after the update is completed. The second, expresses the historical effects of all primitive updates in an update, as they were when executed, and it always holds after an update is computed. These notions are necessary for defining the *effect-preservation* property of sequence updates. The following example demonstrates the difference:

Example 3.2 (Desired and Executed Effects) *Consider the sequence* $S = (x := insert(x, e_1); x := insert(x, e_2); x := delete(x, e_3))$. *The desired effects are:* $effects_D(S) = e_1 \in x \wedge e_2 \in x \wedge e_3 \notin x$, *while the executed effects are:* $effects_E(S) = (e_1 \in delete(insert(x, e_3), e_2 - e_1)) \wedge (e_2 \in insert(x, e_3)) \wedge (e_3 \notin x)$ *where the element subtraction stands for singleton set subtraction. Clearly,* $effects_D(S)$ *does not hold in case that* $e_1 = e_3$ *or* $e_2 = e_3$, *while* $effects_E(S)$ *always holds following the update.*

Under the restriction that assignments to state variables include no additional state variables, desired effects of sequence updates to state variables are defined as conjunctions of the effects of the primitive updates in the sequences. For an empty sequence the desired effects are simply $true$. The precise definition of executed effects is more complex since it takes into account repetitive assignments to state variables. We skip the definition here, since it is needed only for proving effect preservation properties, and is not used in the effect preservation transformation.

A sequence update is *effect preserving*, if whenever the executed effects hold, also the desired effects hold. That is, the desired post-conditions of all primitive updates are preserved by later updates:

Definition 3.1 *A sequence S is effect preserving, iff $effects_E(S) \Rightarrow effects_D(S)$ is valid.*

3.2 Using Delta-Conditions for Preserving the Effects of Sequence Updates

A naive approach for preserving the effects of assignments in a sequence is to test the effects of all previously executed primitives, following every assignment. A compile-time transformation that follows this idea should replace every assignment A_i $(1 \leq i \leq n)$ in a sequence $A_1; \ldots; A_n$ of assignments by A_i; `if` $\neg effects_D(A_1, \ldots, A_{i-1})$ `then` $fail$. Obviously, any successful execution of S (no execution of $fail$) must result into a state where $effects(A_i)$ and $effects_D(A_1, \ldots, A_{i-1})$ hold which are exactly the desired effects of S. Hence, the transformation results an effect preserving update. Note that a sequence including a $fail$ primitive is always effect preserving, since the state space is not affected.

Example 3.3 *The sequence*

$$S \;=\; x := insert(x, e_1); x := insert(x, e_2); x := delete(x, e_3)$$

is transformed to

$$\begin{aligned} S' \;=\; & x := insert(x, e_1); \\ & x := insert(x, e_2); \texttt{if } \neg e_1 \in x \texttt{ then } fail; \\ & x := delete(x, e_3); \texttt{if } \neg e_2 \in x \wedge \neg e_1 \in x \texttt{ then } fail; \end{aligned}$$

This approach provides no advantage over a run-time test for preservation of past effects. That is, executing the compile-time transformed update or adding the tests at run-time is exactly the same.

A worthwhile compile-time transformation should optimize the task in a way that run-time performance is improved. For that purpose, we introduce the notion of *delta-conditions*, which are minimal conditions that guarantee effect preservation. The above update can be transformed into an effect preserving update more efficiently, as shown in the next example.

Example 3.4

$$\begin{aligned} S' \;=\; & x := insert(x, e_1); \\ & x := insert(x, e_2); \\ & x := delete(x, e_3); \texttt{if not } e_3 \neq e_1 \wedge e_3 \neq e_1 \texttt{ then } fail \end{aligned}$$

or

$$\begin{aligned} S' \;=\; & x := insert(x, e_1); \\ & x := insert(x, e_2); \\ & \texttt{if } e_3 \neq e_1 \wedge e_3 \neq e_1 \texttt{ then } x := delete(x, e_3) \texttt{ else } fail \end{aligned}$$

The first transformation uses a *post delta-condition*, i.e., a condition that should be tested following the assignment, while the second uses a *pre delta-condition.*

Definition 3.2 (Delta Conditions) *Let S be a sequence of assignments on state space variables, and $A = (x := f(\bar{e}))$ for some state variable x, and an assignment expression $f(\bar{e})$.*

1 *A post delta-condition for S and A is any condition P satisfying: $effects_D(S) \wedge P_{\{x/f(\bar{e})\}} \Rightarrow effects_D(S)_{\{x/f(\bar{e})\}}$ (recall that the desired effects of an empty sequence are $true$).*

2 *A pre delta-condition for S and A is any condition P satisfying: $effects_D(S) \wedge P \Rightarrow effects_D(S)_{\{x/f(\bar{e})\}}$*

$x/f(\bar{e})$ denotes variable substitution.

Thus, the delta-conditions guarantees that the desired past effects hold after or before the assignment. Clearly, $false$ and $effects_D(S)$ are delta-conditions. The first implies rejection of the assignment, while the second is not minimal. The best delta-condition is $true$, which indeed is the case if A does not interfere with $effects_D(S)$.

Algorithm 3.1 (Transformation with Delta-Conditions)
Let $A_1; \ldots; A_n$ be a sequence of assignments on state space variables, and $post_i$ and pre_i be post/pre delta-conditions for $A_1; \ldots; A_{i-1}$ and A_i $(1 \leq i \leq n)$. The transformed update is obtained by replacing every assignment A_i $(1 \leq i \leq n)$ in the sequence by either A_i; `if` $\neg post_i$ `then` $fail$, or `if` pre_i `then` A_i `else` $fail$.

Again, it can be shown that the resulting update is effect preserving.

Computation of Delta-Conditions: In general, delta-conditions can be computed using methods developed in the field of *Integrity Constraint Checking* [14, 6], where static analysis results minimal conditions that are left to be tested at runtime. For the domain of sets with insert and delete updates, delta-conditions can be inequality tests between assignment expressions that occur in inverse assignments (insertion - deletion) to the same state variables. The delta-condition $e_3 \neq e_1 \wedge e_3 \neq e_2$ in the above example was obtained in this way. The following algorithm computes delta-conditions based on this view. Assignments can be expressed as $x := u(x, e)$ where u is either *insert* or *delete*. u^{-1} denotes the reverse operation of u.

Algorithm 3.2 (Computation of Delta-Conditions) *Let S be a sequence of assignment updates to state variables, and $x := u(x, e)$ an update of x. The algorithm computes a delta-condition for set insertion/deletions.*

```
computeDC( S, x := u(x,e) )
  delta-condition = true
  for each u'(x',e') in S do
    if u' = u^-1 and x = x' and not not_equal(e,e') then
      delta-condition = delta-condition ∧ e ≠ e'
  return delta-condition
```

The test $not\_equal(e, e')$ is an expression inequality comparison (static). It succeeds only if the expressions can be determined to be different on a syntactic basis. The redundant case is $not\_equal \equiv false$.

Proposition 3.1 (Correctness of the computeDC algorithm)
The algorithm computeDC(S, A), for a sequence of assignment updates to state variables S, and a set state variable update A, is a post/pre delta condition for S and A.

4. PRESERVING THE EFFECTS OF UPDATES WITH CONDITIONALS, LOOPS AND LOCAL VARIABLES

Compile-time effect preservation in updates with conditionals, loops and local variables requires analysis of the different execution paths in the update. In order to define effects on paths, we use an auxiliary data structure termed the *computation graph.* The main idea of constructing an effect preserving operation is given in section 4.1 which defines the computation graph of an operation, and in section 4.2 which introduces the transformation process.

The difficulty is to find an abstraction for characterizing the effects of paths including loops, where the number of assignments is not known at compile-time. We suggest to derive delta-condition is two steps. The first, handled in section 4.3, characterizes effects statically. The second, handled in section 4.4, transforms the static effects into delta-conditions by considering the dynamics of loops.

4.1 The Computation Graph of an Update

An update is associated with a computation graph, whose paths span all the possible execution paths. The nodes of the computation graph are labeled with primitive updates. They are used for carrying interference conditions between assignments on a path.

Definition 4.1 (Computation Graph (CG)) Let $Node(U)$ be a constructor for a node labeled with a primitive update U (assignment, skip, fail) or a constructor, and $addLeft(node, T)$ and $addRight(node, T)$ methods that add a left or right child T to a node. The computation graph $CG(S)$ of a transaction S is inductively defined as follows:

- For a primitive update U: $CG(U) = Node(U)$.
- **Sequence:**
  $CG(\ S_1; S_2\ )$ = for all leaves l of $CG(S_1)$ do: $addLeft(l, CG(S_2))$
- **Branching:**
  $CG(\ \texttt{if}\ P\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ ) =$
  $root = Node(if)$,
  $addLeft(root, CG(S_1))$, $addRight(root, CG(S_2))$
- **Loop:**
  $CG(\ \texttt{while}\ P\ do\ \{S\}\ ) =$
  $root = Node(loop)$,
  $addLeft(root, CG(S))$, $addRight(root, Node(skip))$
  for all leaves l of $CG(S)$ do: $addLeft(l, root)$

Example 4.1 *Consider the following update:*

$S(a, b, c)$ =
if P_1 *then* $x :=$ *insert(x,b);*
else $y :=$ *insert(y,c);*
while P_2 *do*
$l :=$ *avg(x);*
$y :=$ *delete(y,l);*
if P_3 *then* $x :=$ *insert(x,l);*
else $x :=$ *delete(x,l);*
$x :=$ *delete(x,a);*

The expression $avg(x)$ *represents some SQL expression for computing the average value of* x. *The computation graph of* S *is depict at figure 1.*

Note that the constructed graphs have always a root node.

A node n that is labeled by an assignment to a state space variable is associated with a condition, $DC(n)$ which is path-sensitive. That is, an assignment in the update, that possibly corresponds to multiple nodes (see example 4.1), might yield different conditions, based on different path information. The intention is that the execution of n preserves the effects of previous updates, if $DC(n)$ is true. The computation of $DC(n)$ is described later on.

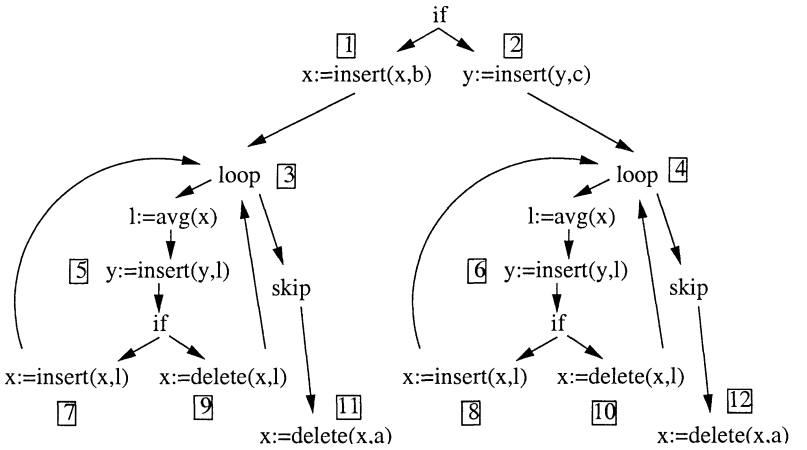

Figure 1. Database update S and its computation graph. Numbers in boxes represent the name of nodes.

4.2 Enforce Effect Preservation by Navigating the Graph and Check Delta-Conditions

We introduce a compile-time transformation of an update S into an effect preserving one. The transformation has four parts:
(1) Construct the computation graph $CG(S)$,
(2) Compute the conditions DC for all nodes in $CG(S)$ that assign state space variables,
(3) Attach $CG(S)$ as a constant to S and
(4) Transform S into an effect preserving update.

The graph is completely computed at compile-time. In step (3) the graph is added as a constant to S, and S is extended by a variable ref that initially points to the root node of the graph, and is used to navigate the graph. Step (4) results an operation that navigates along a path of the graph and checks path-sensitive delta-conditions of assignment nodes. In case a condition does not hold, the update is aborted (rejected) due to the violation of previous effects.

Algorithm 4.1 (Code Transformation) *The algorithm transforms a given operation S by syntactically replacing fragments of code by navigation instructions. The navigation instructions are:* ref`:=left(`ref`)` *and* ref`:=right(`ref`)` *which move the pointer to the left or right child (if exists) according to the current position. Following every assignment to a state space variable the algorithm adds code for evaluating the conditions in* $DC(ref)$.

```
reviseUpdate( S ) =
  replace each [while P do {T}] in S by:
    (while P do {ref:=left(ref);T}
    ref:=right(ref);)
  replace each [if P then S1 else S2] in S by:
    (if P then ref:=left(ref);S1
    else ref:=right(ref);S2)
  for each primitive U in S
    if U = skip or U = fail or U assigns local variable
    then replace U by:
      (U;ref:=left(ref))
    else
      replace U by:
        (U;if not DC(ref) then fail;
        ref:=left(ref))
```

Example 4.2 *The transformation of S as in example 4.1. For readability we set* `checkDC = if not` $DC(ref)$ `then` $fail$.

$S(a, b, c)$ =
if P_1 *then* ref :=*left(ref); x := insert(x,b); checkDC; ref* :=*left(ref);*
else ref :=*right(ref); y := insert(y,c); checkDC; ref* :=*left(ref);*
while P_2 *do*
ref :=*left(ref);*
l := *avg(x); ref* :=*left(ref);*
y := delete(y,l); ref :=*left(ref);*
if P_3 *then ref* :=*left(ref); x := insert(x,l); checkDC; ref* :=*left(ref);*
else ref :=*right(ref); x := delete(x,l); checkDC; ref* :=*left(ref);*
ref :=*right(ref);*
x := delete(x,a); checkDC; ref :=*left(ref);*

Further optimization can remove tests of delta-condition that are equivalent to true. It can be shown that the transformation results an effect preserving update.

4.3 Static Characterization of Past Effects of Graph Nodes

Our aim is to associate with an assignment node in the graph all the effects of nodes that are applied before it in a computation. But, a computation graph with loops includes an infinite number of computation paths. Therefore, a node in the graph might be preceded by an unbounded number of nodes. Yet, node repetition within paths is meaningless since repeated nodes carry the same static information. Therefore, the past effects associated with a node are characterized by the sequence

of all nodes that might precede it, without repetitions. Since the number of nodes in the graph is finite, the past effects associated with a node are also finite. The past effects of a node n are split into the sequence $path_{fix}(n)$ of nodes that must precede n (given by the shortest path to the root), and sequences $path_{var}(n)$ of nodes that might precede it, by following paths within loops. Repetitions, following loop cycles are not considered in the static analysis, but are accounted for by the later transformation of delta-conditions.

The assignment nodes of the computation graph are associated with two lists of nodes sequences. Both are only defined for assignments on state space variables. Assignments to local variables are not considered.

Definition 4.2 ($path_{fix}$) *The property $path_{fix}(n) = (n_1, \ldots, n_k)$ of an assignment node n provides the sequence of all assignment nodes on the shortest path from the root node to n (excluding the node n itself).*

The shortest path guarantees that $path_{fix}(n)$ is finite and does not contain loop nodes that might, or might not, precede n.

For the purpose of the $path_{var}$ definition we introduce the notion of a *loop-path*: Let n be a loop node and S the body of the loop. The loop-paths of n are all sequences of assignment nodes from n to leaf nodes of $CG(S)$.

Definition 4.3 ($path_{var}$) *Let n be an assignment node of the graph. Let $n_1, \ldots, n_k$ be all the loop nodes on the path from the root to n. Then, the property $path_{var}(n)$ provides the set of loop-paths for all loop nodes $n_1, \ldots, n_k$.*

The property $path_{var}(n)$ characterizes sequences of nodes that might precede n, due to loop repetitions.

Example 4.3 *Consider nodes n_5, n_7, n_9 and n_{11} of example 4.1.*

$$\begin{array}{llll} path_{fix}(n_5) = (n_1) & & path_{var}(n_5) = \{(n_5, n_7), (n_5, n_9)\} \\ path_{fix}(n_7) = (n_1, n_5) & & path_{var}(n_7) = \{(n_5, n_7), (n_5, n_9)\} \\ path_{fix}(n_9) = (n_1, n_5) & & path_{var}(n_9) = \{(n_5, n_7), (n_5, n_9)\} \\ path_{fix}(n_{11}) = (n_1) & & path_{var}(n_{11}) = \{(n_5, n_7), (n_5, n_9)\} \end{array}$$

The Problem of Static Characterization of Effects. Consider node n_{11} of the assignment $x := delete(x, a)$ of example 4.1. Its variable path effects are $path_{var}(n_{11}) = \{(n_5, n_7), (n_5, n_9)\}$ which states that the effects of the sequences (n_5, n_7) and (n_5, n_9) might need to be preserved by n_{11}. Clearly, both (n_5, n_7) and (n_5, n_9) can be executed multiple times. Computing the delta-conditions for the variable path effects of n_{11} results *true* for the loop-path n_9 and $a \neq l$ for the loop-path n_7. The condition $a \neq l$ is meaningless since l might be repeatedly assigned.

The problem involves the unknown number of repetitions of the loop. Therefore, the condition $a \neq l$ has to be duplicated and instantiated according to the number of loop repetitions, resulting a run-time test of the form $a \neq l_1 \wedge \ldots \wedge a \neq l_k$, for k repetitions of the loop. The test requires access to all previous values of l.

In order to cope with this problem we introduce in the next section two run-time functions *exec* and *history*, that are used in conditions. For the above example, the resulting condition is: $\forall 1 \leq k \leq exec(n_7) : history_F(n_{11}) \neq history_V(n_7, n_7, k)$. That is: For all executions of the loop-path n_7 check whether the last deleted value of n_{11} is different from the inserted value of the k-th execution of node n_7 on the loop-path n_7.

4.4 Delta-Conditions under the Presence of Loops

The basic idea is that while executing a complex update S, each inserted or deleted value by assignments of S is kept in a run-time history.

Definition 4.4 ($history_F$) *The method $history_F(n)$ returns the inserted or deleted value of the last execution of assignment node n.*

For non-loop assignment nodes n there exists only one value, since they are only executed once. Loop assignment nodes might result into multiple values according to the number of repetitions. However, $history_F()$ returns always the value of the last execution. Since loops are non-nested, the leaf nodes of the computation graph of a loop body uniquely identify paths to the loop node. Therefore leaf-nodes of loop bodies determine loop-paths.

Definition 4.5 ($history_V$, *exec*) *Let n' be a loop-path. Then, $history_V(n, n', k)$ returns the inserted or deleted value of the execution of assignment node n on the k-th completed (full) execution of the loop-path n'. The function $exec(n')$ returns the number of executions of the loop-path n'.*

Note that $history_V$ returns only values for loop-paths that have been completely (all assignments of the path) executed. The additional information of a loop-path n' is necessary, since it identifies the executed path of the loop. This is necessary for the path specific computation of the delta-conditions.

Example 4.4 *Consider the update S of example 4.1 and the execution of $S(1, 2, 3)$ resulting the following sequence of assignment nodes: n_1, n_5, n_7, n_5, n_9, n_{11}. Assume that the function $avg(x)$ results 4 in the first loop repetition and 5 in the*

second. After executing n_{11} the content of the history is as follows:

$$history_F(n_1) = 2 \qquad history_F(n_5) = 5$$
$$history_F(n_7) = 4 \qquad history_F(n_9) = 5$$
$$history_F(n_{11}) = 1$$

$$history_V(n_5, n_7, 1) = 4 \qquad history_V(n_7, n_7, 1) = 4$$
$$history_V(n_5, n_9, 1) = 5 \qquad history_V(n_9, n_9, 1) = 5$$

The history for the second repetition is not defined, since each loop-path has only executed once, that is $exec(n_7) = 1$ and $exec(n_9) = 1$.

We present an algorithm that computes a delta-condition for each node of the computation graph for set insertion and deletion. The basic idea is to derive this condition out of $path_{\text{fix}}$ and $path_{var}$. The algorithm constructs the condition on a syntactical basis, using two string variables str and str'. The function `replace(`str, X, n`)` replaces substring X in str by the value of the variable n.

Algorithm 4.2 (Compute Run-Time Delta-Conditions)
Assignments are expressed as $x := u(x, e)$ where u is either insert or delete. u^{-1} denotes the reverse operation of u.

```
compute-RT-DC( CG )
 for each assignment node n = u(x,e) in DG do:
   str = 'TRUE'
   for each node n' of path_fix do:
     let u'(x',e') be the label of n'
     if u' = u^-1 and x = x' then
       str' = 'history_F(X_1) ≠ history_F(X_2)'
       str' = replace(str',X_1,n)
       str' = replace(str',X_2,n')
       str = str + ' AND ' + str'
   for each sequence S of path_var(n) do:
     for each node n' of S do:
       let u'(x',e') the corresponding assignment statement of n'
       if u' = u^-1 and x = x' then
         str' = '∀1 ≤ k ≤ exec(X_1) : history_F(X_2) ≠ history_V(X_3,X_4,k)'
         str' = replace(str',X_1,last(S))
         str' = replace(str',X_2,n)
         str' = replace(str',X_3,n')
         str' = replace(str',X_4,last(S))
         str = str + ' AND ' + str'
   DC(n) = str
```

Example 4.5 *Consider again example 4.1 and its delta-conditions computed by the above algorithm.*

$$
\begin{array}{ll}
DC(n_1) = & DC(n_2) = DC(n_5) = DC(n_6) = \mathit{TRUE} \\
DC(n_7) = & \forall 1 \leq k \leq exec(n_9) : history_{var}(n_9, n_9, k) \neq history_{fix}(n_7) \\
DC(n_8) = & \forall 1 \leq k \leq exec(n_{10}) : history_{var}(n_{10}, n_{10}, k) \neq history_{fix}(n_8) \\
DC(n_9) = & history_{fix}(n_1) \neq history_{fix}(n_9)\ AND \\
 & \forall 1 \leq k \leq exec(n_7) : history_{var}(n_7, n_7, k) \neq history_{fix}(n_9) \\
DC(n_{10}) = & \forall 1 \leq k \leq exec(n_8) : history_{var}(n_8, n_8, k) \neq history_{fix}(n_{10}) \\
DC(n_{11}) = & history_{fix}(n_1) \neq history_{fix}(n_{11})\ AND \\
 & \forall 1 \leq k \leq exec(n_7) : history_{var}(n_7, n_7, k) \neq history_{fix}(n_{11}) \\
DC(n_{12}) = & \forall 1 \leq k \leq exec(n_8) : history_{var}(n_8, n_8, k) \neq history_{fix}(n_{12})
\end{array}
$$

Note that assignment $x := delete(x, a)$ represented by the nodes n_{11} and n_{12} indeed has different delta-conditions for different execution paths.

5. RELATED WORKS

In the field of active databases the problem of effect violation occurs when a rule fires another rule which performed an update contradicting to an update preformed by the first rule. However, automated generation and static analysis of active database rules [15, 1, 2] do neither handle, nor provide a solution for that problem. In fact, contradicting updates are allowed, since the case of inserting and deleting the same tuple is considered as skip-operation.

The GCS theory [12, 13] follows the compile-time transaction preprocessing approach. Intuitively, the GCS of an operation S with respect to a given set of integrity constraints $\mathcal{I}$ is a transaction which is *consistent* with respect to $\mathcal{I}$, and preserves the effects of S with at least as possible additional actions (*greatest*). In the GCS theory, effect preservation is captured by a *specialization* partial ordering between operations. Intuitively, an operation T specializes an operation S ($T \sqsubseteq S$), if all the effects of S are preserved by T. That is, T must do everything that S does, and maybe more. In particular, T cannot undo S. However, due its generality the proposed specialization ordering is too coarse as show by the following example:

Example 5.1 *Consider*

$$
\begin{array}{lll}
S & = & \texttt{if } P(y) \texttt{ then } x := insert(x, a) \texttt{ else } fail \\
T & = & \texttt{if } P(y) \texttt{ then } x := insert(x, a); y := insert(y, b) \texttt{ else } fail
\end{array}
$$

with $P(y)$ as some condition over y. Intuitively, T should preserve the effects of S, but the specialization order suggests $T \not\sqsubseteq S$ (effects are not preserved).

The method proposed in this paper improves the preservation of effects for set insertions and deletions, since the resulting operation T is effect preserving.

6. CONCLUSION AND FUTURE WORK

In this paper we extended our previous method for effect preservations in updates, for handling non-nested loops. We provide an effect preservation algorithm with a small run-time overhead (linear for non-loop updates). The algorithm relies on an exhaustive static analysis of a program, and on constraint reduction techniques, that integrate run-time histories within constraint evaluation. The static analysis includes path sensitive construction of effects, and reduction of interference conditions (delta-conditions). The resulting transformation is as expressive as the original program, but prevents effect violation.

The run-time overhead for updates without loops is linear, under the assumption that the size of the delta-condition for the non-loop part of a path is small and is independent of the size of the path. Our experience so far supports this assumption ([4, 5]). This assumption relies on the following observations: (1) The number of state modification of a path for most applications is rather small (e.g less than 100). (2) The number of contradicting updates in a database update seems to be very low. Yet, an experiment with a real application is still necessary.

In the presence of loops, the run time application of delta-conditions requires additional tests that depend on the number of loop repetitions. Based on our previous analysis, we expect the extra load is not more than quadratic in path length. However, this estimation still needs to be checked.

The proposed method suggests to transform the code of an update to achieve effect preservation. For the above notion of history we did not precisely define its integration into the transformation process. The main idea is to further extend the computation graph, such that computed values of assignment expressions are directly attached to the nodes of the graph. By using sophisticated indexing on the graph we hope to obtain a constant time access to the required information at run-time. For example, a three-dimensional array, although rather space-consuming, can be used. Then, the methods $history$ and $exec$ can be defined as sub-routines of the transformed update.

The transformation process as presented here, is independent from the database system where updates are executed. We believe that in practice, any implementation of this method should be a plug-in for a database system, with an efficient handling of graphs and histories. One approach for achieving such integration is to compute the graph while pre-compiling database updates (e.g. stored procedures) and store the graph within the database system. Then the run-time machine is responsible for efficient testing of delta-conditions.

In the future, we plan to extend the effect preservation method for nested loops, and remove the restriction on assignments to state variables (no other state variable in such assignments). these extensions might require further changes in the construction of past effects associated with graph nodes, and handling of variables in the code transformation.

Another extension is in the direction of more powerful dynamic effects, i.e., effects that allow simultaneous reference to initial and final values of state variables. Such effects are necessary for expressing evolution intentions such as "the value of a state variable can only grow".

Finally, we intend to apply our approach in multiple domains. For web based systems it is necessary to understand what are the characteristic effects of programs. For *Rule Triggering Systems* there is a need to study different modes of embedding an effect preservation algorithms within a system. For concurrent, reactive and mobile systems there is a need to study how effects of primitive actions should be defined. Our goal is to construct a generic open tool that can be extended in terms of language and effects, and can be applied to new domains.

Acknowledgments

This research was supported by the DFG, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316) and by the Paul Ivanir Center for Robotics and Production Management at Ben-Gurion University of the Negev.

REFERENCES

[1] E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. In *ACM Transactions on Database Systems*, volume 25(3), pages 269–332, September 2000.

[2] S. Ceri, P.Fraternali, S. Paraboschia, and L. Tanca. Automatic gerneration of production rules for integrity maintenance. In *ACM Transactions on Database Systems*, volume 19(3), pages 367–422, 1994.

[3] E.W. Dijkstra and C.S. Scholten. Predicate calculus and program semantics. *Springer-Verlag, Texts and Monographs in Computer Science*, 1989.

[4] M. Balaban, S. Jurk. Effect Preservation As A Means For Achieving Update Consistency. In *5th International Conf. on Flexible Query Answering Systems*, Lecture Notes in Computer Science, 2002 (to appear).

[5] M. Balaban, S. Jurk. Intentions of Operations — Characterization and Preservation. In *2nd International Workshop on Evolution and Change in Data Management*, Lecture Notes in Computer Science, 2002 (to appear).

[6] F. Bry. Intensional updates: Abduction via deduction. In *Proc. 7th Conf. on Logi Programming*, 1990.

[7] Greg Nelson. A generalization of dijkstras calculus. *ACM Transactions on Programming Languages and Systems*, 11:517–561, 1989.

[8] P. Fraternali and S. Paraboschi and L. Tanca. Automatic Rule Generation for Constraints Enforcement in Active Databases. Springer WICS, 1993.

[9] Joan Antoni Pastor. Extending the synthesis of update transaction programs to handle existential rules in deductive databases. In *Deductive Approach to Information Systems and Databases*, pages 189–218, 1994.

[10] Joan Antoni Pastor-Collado and Antoni Olive. Supporting transaction design in conceptual modelling of information systems. In *Conference on Advanced Information Systems Engineering*, pages 40–53, 1995.

[11] D. Plexousakis and J. Mylopoulos. Accommodating integrity constraints during database design. *Lecture Notes in Computer Science*, 1057, 1996.

[12] K.D. Schewe. Consistency enforcement in entity-relationship and object-oriented models. *Data and Knowledge Eng.*, 28(1):121–140, 1998.

[13] K.D. Schewe and B. Thalheim. Towards a theory of consistency enforcement. *Acta Informatics*, 36:97–141, 1999.

[14] S.Y. Lee, T.W. Ling. Further Improvement on Integrity Constraint Checking for Stratisfiable Deductive Databases. In *Proc. 22th Conf. on VLDB*, 1996.

[15] J. Widom and S. Ceri. Deriving production rules for constraint maintenance. In *Proc. 16th Conf. on VLDB*, pages 566–577, 1990.

[16] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, 1996.