

Effect Preservation in Transaction Processing in Rule Triggering Systems

Mira Balaban*
Ben-Gurion University
Beer-Sheva, Israel
mira@cs.bgu.ac.il

Steffen Jurk†
Brandenburg University of Technology
Cottbus, Germany
sj@informatik.tu-cottbus.de

Abstract

Rules provide an expressive means for implementing database behavior: They cope with changes and their ramifications. Rules are commonly used for integrity enforcement, i.e., for repairing database actions in a way that integrity constraints are kept. Yet, Rule Triggering Systems fall short in enforcing *effect preservation*, i.e., guaranteeing that repairing events do not undo each other, and in particular, do not undo the original triggering event.

A method for enforcement of effect preservation on updates in general rule triggering systems is suggested. The method derives transactions from rules, and then splits the work between compile time and run time. At compile time, a data structure is constructed, that analyzes the execution sequences of a transaction and computes minimal conditions for effect preservation. The transaction code is augmented with instructions that navigate along the data structure and test the computed minimal conditions.

This method produces *minimal effect preserving transactions*, and under certain conditions, provides meaningful improvement over the quadratic overhead of pure run time procedures. For transactions without loops, the run time overhead is linear in the size of the transaction, and for general transactions, the run time overhead depends linearly on the length of the execution sequence and the number of loop repetitions. The method is currently being implemented within a traditional database system.

Keywords: Rule Triggering Systems, Effect Preservation, Minimal Conditions, Consistency, Static Analysis, Transaction Processing.

1 Introduction

Rules provide an expressive means for implementing database behavior: They cope with changes and their ramifications. Rule mechanisms are used in almost every commercial database system, using features such as `CREATE TRIGGER` or `CREATE RULE`. Rules are commonly used for *Integrity enforcement*, i.e., for *repairing* database actions in a way that integrity constraints are kept ([38, 16, 39]). Yet, *Rule Triggering Systems (RTSs)*, also termed *Active Databases*, fall short in enforcing *effect preservation*, i.e., guaranteeing that repairing events do not undo each other, and in particular, do not undo the original triggering event.

A natural expectation in database maintenance is that a fact that was successfully added is retrievable, as long as it was not intentionally removed. Reliability in question answering, and

*This work was supported in part by the Paul Ivanir Center for Robotics and Production Management at Ben-Gurion University of the Negev. Contact: POB 653, Beer Sheva 84105, ISRAEL, Phone: +972-8-6472222, FAX: +972-8-6477527

†This research was supported by the DFG, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316). Contact: POB 101433, 03013 Cottbus, Germany, Phone: +49-355-692711, FAX: +49-355-692766

faithfulness to the intended semantics of operations, require no contradictory operations. Such behavior is achievable if rule application that can cause contradictory updates is avoided. Active databases do not meet this expectation since it is possible that a rule application undoes the actions of previous rule applications in a single repair transaction. Moreover, in a distributed active database a user might not be aware of rules that trigger contradicting actions, since the rules might reside in independently developed sites.

Example 1 Consider a database with a table T_1 with two attributes A, B , a table T_2 with an attribute C , and two integrity constraints: an inclusion constraint ($A \subseteq C$) and an exclusion constraint ($B \cap C = \emptyset$)¹. The inclusion constraint is enforced by the rules:

$$\begin{aligned} R_1: & \text{ ON } \text{insert}(T_1, (x, -)) : \text{ IF } (x) \notin T_2 \text{ THEN } \text{insert}(T_2, (x)) \\ R_2: & \text{ ON } \text{delete}(T_2, (x)) : \text{ IF } (x) \in T_1.A \text{ THEN } \text{delete}(T_1, (x, -)) \end{aligned}$$

The exclusion constraint is enforced with the rules:

$$\begin{aligned} R_3: & \text{ ON } \text{insert}(T_1, (-, x)) : \text{ IF } (x) \in T_2 \text{ THEN } \text{delete}(T_2, (x)) \\ R_4: & \text{ ON } \text{insert}(T_2, (x)) : \text{ IF } (x) \in T_1.B \text{ THEN } \text{delete}(T_1, (-, x)) \end{aligned}$$

An insertion update $\text{insert}(T_1, (a, b))$ triggers rules R_1 and R_3 , in order to preserve the integrity constraints. The value a is inserted into T_2 (if not exists) to preserve the inclusion constraint, which in turn causes a deletion of tuples $(-, a)$ from T_1 in order to preserve the exclusion constraint. Analogously the value b is removed from T_2 (if exists) to preserve the exclusion constraint which causes deletion of tuples $(b, -)$ from T_1 . If the event-repair policy is that a rule is fired immediately AFTER its event (triggering update) is executed, and the repairing rules R_1 and R_3 are triggered in that ordering, the order of rule application for this insertion event is R_1, R_4, R_3, R_2 . A database design tool would result the following repair update:

```
INSERT INTO T1 VALUES (a,b);
IF NOT EXISTS (SELECT * FROM T2 WHERE C=a) THEN
  INSERT INTO T2 VALUES (a);
  DELETE FROM T1 WHERE B=a;
END IF;
IF EXISTS (SELECT * FROM T2 WHERE C=b) THEN
  DELETE FROM T2 VALUES (b);
  DELETE FROM T1 WHERE A=b;
END IF;
```

If $a = b$, then the repairing update might undo its own actions, since tuples inserted to the tables might be deleted:

Rule	Triggering Primitive Update	T_1	T_2
—	—	(a, a)	\emptyset
R_1	$\text{insert}(T_1, (a, a))$	(a, a)	(a)
R_4	$\text{insert}(T_2, (a))$	\emptyset	(a)
R_3	$\text{insert}(T_1, (a, a))$	\emptyset	\emptyset
R_2	$\text{delete}(T_2, (a))$	\emptyset	\emptyset

If the rule application ordering is R_3, R_1, R_4 , the result is that the inserted tuple is deleted from T_1 , while a new tuple is still inserted to T_2 as a repair for the deleted insertion to T_1 :

Rule	Triggering Primitive Update	T_1	T_2
—	—	(a, a)	\emptyset
R_3	$\text{insert}(T_1, (a, a))$	(a, a)	\emptyset
R_1	$\text{insert}(T_1, (a, a))$	(a, a)	(a)
R_4	$\text{insert}(T_2, (a))$	\emptyset	(a)

¹these constraints are simplified versions of possibly more natural constraints like $f(A) \subseteq C$ and $g(B) \cap C = \emptyset$, where f and g are some functions.

This example demonstrates the problem of *effect preservation*. A seemingly successful insertion update ends up in a state where the insertion is not performed. Moreover, although the insertion actually failed, its repairing updates have taken place. The problem is caused by allowing contradictory updates, i.e., updates that undo the expected effects of each other, within the context of a successful repairing transaction. Rather, the insertion $insert(T_1, (a, a))$ must fail (be rejected) since there is no way to achieve consistency together with effect preservation.

The problem of *effect preservation* goes back to the early days of *Planning* in Artificial Intelligence, when planners tried to cope with the problem of *interacting goals*, where one action undoes something accomplished by another (e.g., Strips [15], Noah [32]). In the field of databases, [33, 36] provide a general framework for consistency enforcement under an effect preservation constraint. Static composition of refactoring ([20, 19]) also needs to cope with possibly contradicting effects of successive refactorings.

In the field of active databases the problem of effect violation occurs when a rule fires other rules that perform updates that contradict an update of a previous rule. Nevertheless, automated generation and static analysis of active database rules [38, 6, 12] do neither handle, nor provide a solution for that problem. Updates triggered within the scope of a single repair can contradict each other.

In this paper we suggest a combined, compile time – run time, method for enforcing effect preservation on updates. The method is applied in two steps:

1. **Static derivation of transactions from rules:** For each primitive (atomic) update, e.g., insertion or deletion of a tuple, obtain a transaction, based on a given rule application policy.
2. **Effect Preservation Transformation:** Enforce effect preservation using “minimal” modifications.

Our method assumes that primitive (atomic) updates are associated with intended effects (post-conditions). At compile time, we construct a data structure that analyzes all execution sequences of an update and computes minimal conditions necessary for effect preservation. The update code is augmented with instructions that navigate along the data structure and test the computed minimal conditions. This method produces *minimal effect preserving transactions*, and under certain conditions, provides meaningful improvement over the quadratic overhead of pure run time procedures. For transactions without loops, the run time overhead is linear in the size of the transaction, and for general transactions, the run time overhead depends linearly on the length of the execution sequence and the number of loop repetitions (while for pure run time methods it is necessarily quadratic in the length of the execution sequence).

Our method is domain independent, but in this paper we demonstrate it on the relational domain with element insertion and deletion operations, alone. The method is currently being implemented within a traditional database system.

Section 2 introduces a small imperative update language *While* that we use in the rest of this paper, and describes the algorithm for static derivation of transactions from rules. Section 3 introduces the major notions of effect preservation. In Section 4 algorithms for effect preservation of *While* updates are described and proved to enforce only *minimal effect preservation*, i.e., do not cause unnecessary failures. The algorithms are introduced gradually, first for *While* updates without loops, and then for general *While* updates. Section 5 describes related work, and section 6 concludes the paper. Proofs are postponed to the Appendix.

2 Static Rule Repair

Rule repair in active databases is performed at run time. The database tool is usually equipped with an *event-repair policy* (e.g., an AFTER policy), while the order of firing the applicable rules, henceforth *rule-ordering policy*, is determined at run time. The event-repair policy determines when an event that is applied by a rule is repaired. For example, using a DELAYED event-repair

policy in Example 1, and SEQUENTIAL rule-ordering policy, the order of rule application for the $insert(T_1, (a, b))$ insertion event is R_1, R_3, R_4, R_2 . The rule-ordering policy is responsible for sequencing the set of rules that should repair for an event. For example, using a DELAYED event-repair policy and a REVERSED SEQUENTIAL rule-ordering policy in Example 1, the order of rule application for the $insert(T_1, (a, b))$ insertion event is R_3, R_1, R_4 .

If the rule-ordering policy is determined statically, the overall task of repairing for an event can be determined statically, by associating every database event with a transaction. The advantage is that an update transaction that is constructed at compile time can be also optimized at compile time, thereby saving some run time load.

Subsection 2.1 describes the small theoretical imperative update language *While* [26], that is used in the rest of the paper. Subsection 2.2 introduces a compile time algorithm for derivation of updates from rules. The rest of the paper deals with enforcing effect preservation on statically derived updates.

2.1 The Repair Language *While* – An Imperative Language of Updates (Transactions)

The *While* language [26] is a small theoretical imperative language that includes the three major control structures in sequential imperative languages: sequencing, conditionals and loops. We adapt it for database update usage by adding a *fail* atomic statement that stands for *rollback*, and by having state variables that stand for relations. In the experimental evaluation the language is replaced by a real database maintenance language. Likewise, all the examples in the paper deal with a relational database and the atomic updates *insert* and *delete* of a tuple to a relation. However, our method applies to any atomic updates for which effects can be provided (see subsection 3.1).

Syntax: Updates are built over a finite set of typed *state variables*. For example, in a relational database with relations R_1, \dots, R_n , the state variables are $\{R_1, \dots, R_n\}$, and any assignment of concrete relations to the relation variables results a *database state*. The symbols of the *While* language include, besides the state variables, *parameter variables*, *local variables*, and *constant symbols* like numerals, arithmetic operations and comparisons, boolean constants and boolean connectives, and set operations (*language constants*). We use the letters r, s, t for state variables, u, v, w, \dots for parameter or local variables, and e, f, g, \dots for expressions.

The primitive update statements of *While* are *skip*, *fail*, and *well-typed assignments*. The *skip* update is a no-op update statement: Its execution does not affect the state of the update. The *fail* update denotes a failure of the update: Updates following *fail* are not executed. In transactional databases without choice, *fail* corresponds to the *rollback* operation, which undoes the failed update by restoring the old state. Assignment updates have the form $x := e$, where x is a variable and e is an expression.

Compound updates (transactions) in the *While* language are formed by three constructors: *sequence*, *condition*, and *while*. If P is a condition, and S, S_1, S_2 are updates, then “ $S_1; S_2$ ” is 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. The statement “if P then S ” is an abbreviation for “if P then S else *skip*”. The *empty update* ϵ is the neutral (unit) element of update sequencing. That is, for every update S , $\epsilon; S \equiv S; \epsilon \equiv S$, where \equiv stands for “is the same as” or “can be replaced by” (ϵ is needed for the definition of the terminal configurations below). A compound update S with parameter variables \vec{x} is sometimes denoted $S(\vec{x})$.

Semantics: A *state* is a well-typed *value assignment* to all variables. A *database state* is a restriction of a state to the state variables. A non-ground expression (that includes variables) can be evaluated only with respect to a state. The value of an expression e in a state s is denoted e^s .

[<i>ass</i>]	$\langle x := e, s \rangle \Rightarrow s[x \mapsto e^s]$
[<i>skip</i>]	$\langle \text{skip}, s \rangle \Rightarrow s$
[<i>fail</i>]	$\langle \text{fail}, s \rangle$
[<i>comp</i>]	$\text{if } \langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle \text{ then } \langle S_1; S_2, s \rangle \Rightarrow \langle S'_1; S_2, s' \rangle$ $\text{if } S'_1 \text{ is } \epsilon \text{ then the conclusion is } \langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle$
[<i>if^T</i>]	$\langle \text{if } P \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_1, s \rangle \text{ if } s \models P$
[<i>if^F</i>]	$\langle \text{if } P \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_2, s \rangle \text{ if } s \not\models P$
[<i>while</i>]	$\langle \text{while } P \text{ do } S, s \rangle \Rightarrow$ $\langle \text{if } P \text{ then } (S; \text{while } P \text{ do } S) \text{ else skip}, s \rangle$

Figure 1: Axioms and rules defining the semantics of the transition relation \Rightarrow

A state s that satisfies a condition P is denoted as $s \models P$. Variable substitution on a state s is denoted $s[x \mapsto e^s]$, which is a state that differs from s only in the value of x which is e^s .

The semantics of *While* updates is defined operationally, using the *structural operational semantics* described in [26]. This semantical approach emphasizes individual steps of the execution, that are captured by a *transition relation* denoted \Rightarrow between update configurations. An *update configuration* (*configuration* for short) is a pair $\langle S, s \rangle$ of an update S and a state s . A configuration $\langle \epsilon, s \rangle$ is called a *terminal configuration* and abbreviated as s . All non-terminal configurations are *intermediate*. A configuration $\langle \text{fail}, s \rangle$ is called a *failing configuration*. A configuration $\langle A; S, s \rangle$ where A is an assignment update and S is any update (possibly the empty update), is called an *assignment configuration*.

The *transition relation* between configurations $\langle S, s \rangle \Rightarrow \gamma$, expresses the first step in the execution of S from state s . If the execution of S from s is not completed by the transition, γ is an intermediate configuration $\langle S', s' \rangle$, where S' is the remaining computation to be executed from state s' . If the execution of S from s is completed by the transition, γ is a terminal configuration $\langle \epsilon, s' \rangle \equiv s'$. A non-terminal configuration γ is a *dead-end* if there is no γ' such that $\gamma \Rightarrow \gamma'$. Failing configurations are dead-end configurations. The transition relation \Rightarrow is defined by the axioms and rules in Figure 1.

An *execution sequence* (denoted as Ψ , Φ or $\text{seq}(S, s)$) of a statement S starting in state s , is either a finite or an infinite sequence of configurations. In a finite execution sequence, $\text{seq}(S, s) = \gamma_0, \dots, \gamma_k$, where $\gamma_0 = \langle S, s \rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$ ($0 \leq i < k$) and γ_k is either a terminal or a dead-end configuration. In an infinite execution sequence, $\text{seq}(S, s) = \gamma_0, \dots$, where $\gamma_0 = \langle S, s \rangle$, and $\gamma_i \Rightarrow \gamma_{i+1}$, $0 \leq i$. A finite sequence is *successful* if it ends in a terminal configuration, and *failing* if it ends in a dead-end configuration. The states of the first and last (if exists) configurations of an execution sequence Ψ are denoted $\text{start}(\Psi)$ and $\text{end}(\Psi)$, respectively, and the i -th configuration ($0 \leq i$) is denoted Ψ_i .

Extended While: *While* is extended with calls to external procedures that do not affect the state of a *While* computation. Such external procedures can operate on elements of an external environment, e.g., print or draw. Formally, the extended *While* includes an additional primitive update *proc*, where *proc* is a procedure that can be applied in the environment where *While* is run. The semantics of *proc* is like that of *skip*:

$$\langle \text{proc}, s \rangle \Rightarrow s$$

The transformations introduced in Section 4 map a *While* update without external calls into a *While* update with external calls to procedures that read and evaluate data from a data structure built as part of the transformation.

2.2 Static Derivation of Transactions from Rules

Event-Condition-Action (ECA) rules consist of *event*, *condition*, and *action* (also termed *body*), with the intuitive semantics that an execution of the event update implies an execution of the action update, provided that the condition holds. For simplicity, we embed the *condition* part into the *action* part, since an ECA rule $\langle E, C \rangle \rightarrow A$ can be captured by the rule $E \rightarrow \text{if } C \text{ then } A$.

Compile time rule application requires careful management of parameter and local variables. Prior to rule application, all such variables must be consistently renamed by fresh variables. Then, the expression in the triggering event must unify with the expression in the rule event, and the resulting substitution applied to the rule body. For example, if the update under construction includes the primitive update $\text{insert}(r, x + 3)$, and the event in an applicable rule is $\text{insert}(r, x)$, then the variable x must be renamed in the rule by a new name, say y , yielding the event $\text{insert}(r, y)$. The two events $\text{insert}(r, x + 3)$ and $\text{insert}(r, y)$ should be unified, yielding the matching substitution $y \mapsto x + 3$, which should be applied to the rule body, replacing all occurrences of y by $x + 3$.

Compile time rule processing builds for each primitive event a complex update, that exhausts all necessary rule applications. It requires *static event-repair policy*, *rule-ordering policy*, and *static termination policy*. A termination policy is necessary since rule triggering can be cyclic. Much research have been devoted to termination analysis ([37, 5]). In this work we take the approach of full propagation, that is controlled by a static termination condition.

Algorithm 1 below, performs static derivation of transactions from rules. It is designed for the AFTER event-repair semantics, which characterizes commercial databases. The AFTER semantics is implemented by sequencing immediately after every occurrence of an assignment to a state variable event, the bodies of all rules that are applicable to that event. Rule application handles variable renaming, rule event matching and substitution application.

Algorithm 1 [DTA – Derive TransAction]

input: a primitive update U , a set of rules \mathcal{R} , a rule-ordering policy RO , and a termination condition TC .

output: A *While* program S .

method:

1. $S := U$
2. While there is an unmarked assignment E in S do:
 - (a) Mark E .
// The marking is used to avoid multiple replacements of the same primitive update.
 - (b) For every rule in \mathcal{R} , consistently rename all parameter and local variables, by fresh variables.
 - (c) Let $E_1 \rightarrow B_1, \dots, E_k \rightarrow B_k$ be all rules whose event can be unified with E , and are such ordered by RO . Let ϕ_i be the matching substitution for E and E_i , i.e., $E\phi_i = E_i\phi_i$, $1 \leq i \leq k$.
 - (d) If $k \neq 0$ and $\neg TC$: Replace E in S by $E; B_1\phi_1; \dots; B_k\phi_k$
 - (e) If $k \neq 0$ and TC : Replace E in S by $E; \text{fail}$
3. Return S

The following example presents the rules from Example 1 as event-action rules in the *While* language, and the update constructed by the DTA algorithm, for the triggering event $\text{insert}(T_1, (x, y))$, using a sequential rule-ordering policy.

Example 2 (Static transaction derivation with the DTA algorithm.)

$$\begin{aligned} R_1 & : \text{insert}(T_1, (x, y)) \rightarrow \text{if } (x) \notin T_2 \text{ then } \text{insert}(T_2, (x)); \\ R_2 & : \text{delete}(T_2, (x)) \rightarrow \text{while } \sigma_{A=x}(T_1) \neq \emptyset \text{ do} \\ & \quad \text{delete}(T_1, \text{first}(\sigma_{A=x}(T_1))); \\ R_3 & : \text{insert}(T_1, (x, y)) \rightarrow \text{if } (y) \in T_2 \text{ then } \text{delete}(T_2, (y)); \\ R_4 & : \text{insert}(T_2, (x)) \rightarrow \text{while } \sigma_{B=x}(T_1) \neq \emptyset \text{ do} \\ & \quad \text{delete}(T_1, \text{first}(\sigma_{B=x}(T_1))); \end{aligned}$$

The While program derived for the update $\text{insert}(T_1, (x, y))$:

$$\begin{aligned} & \text{insert}(T_1, (x, y)); \\ & \text{if } (x) \notin T_2 \text{ then} \\ & \quad \text{insert}(T_2, (x)); \\ & \quad \text{while } \sigma_{B=x}(T_1) \neq \emptyset \text{ do} \\ & \quad \quad \text{delete}(T_1, \text{first}(\sigma_{B=x}(T_1))); \\ & \text{if } (y) \in T_2 \text{ then} \\ & \quad \text{delete}(T_2, (y)); \\ & \quad \text{while } \sigma_{A=y}(T_1) \neq \emptyset \text{ do} \\ & \quad \quad \text{delete}(T_1, \text{first}(\sigma_{A=y}(T_1))); \end{aligned}$$

Assignment Preprocessing: Prior to the transaction derivation a preprocessing of rule actions is needed. Our effect characterization methods require that every assignment to a state variable $r := e(r, x_1, \dots, x_n)$ is rewritten such that the variables x_1, \dots, x_n are not assigned after the assignment is executed. The rationale behind this requirement is that each such assignment has effects which are a set of constraints, expressed by the variables in the assignment. If the variables x_1, \dots, x_n are reassigned after the assignment is executed, then the effects of the assignment must be reassigned as well. In order to avoid this hard procedure, we rewrite the assignment, using new variables, as follows:

The assignment $r := e(r, x_1, \dots, x_n)$, is replaced by the sequence update

$$x'_1 := x_1; \dots, x'_n := x_n; r := e(r, x'_1, \dots, x'_n);$$

where x'_1, \dots, x'_n are new variables. The new sequence command has the same semantics as the original assignment, when restricted to the original update variables.

3 Effect Preservation – Goals and Associated Terminology

In this section we introduce the main concepts of *effect preservation* and analyze their basic properties. We deal with three main issues: (1) Define what are effects and where do they come from; (2) Define when an update is effect preserving; (3) Define criteria for effect preserving transformations. The section ends with the definition of a *minimal effect preserving transformation*.

3.1 Characterization of Effects

The effects of a primitive update are constraints (postconditions) that characterize the intended impact of the update:

Definition 1 (Effects of Primitive Updates) *The effects of a primitive update U is a collection of First Order Logic formulae $\text{effects}(U)$ that always hold following the update. That is, for every state s , if $\langle U, s \rangle \Rightarrow s'$, then $s' \models \text{effects}(U)$ ².*

²For a set of formulae Φ and a state s , $s \models \Phi$ holds iff for every formula ϕ in Φ , $s \models \phi$.

Effects are used as a criterion for acceptance or rejection of execution sequences. Therefore, if they are too restrictive, they lead to needless rejections, while if they are too permissive, they lead to unwanted acceptance that possibly violates some data integrity constraints. For example, taking $\{false\}$ as the effects of an update causes constant rejection, while taking $\{true\}$ causes constant acceptance.

For the primitive *skip* the effects are $\{\}$ since $\langle skip, s \rangle \Rightarrow s$. For the primitive *fail* the effects can be anything since, $\langle fail, s \rangle$ is a dead-end configuration. For assignments to regular variables the effects are also $\{\}$ since such updates have no impact on database states. For assignments to state variables, the effects are domain specific and developer provided. In the relational domain with the operations of element insertion and deletion we assume the following effects:

- $effects(r := insert(r, e))$ is $\{e \in r\}$ and
- $effects(r := delete(r, e))$ is $\{e \notin r\}$.

Clearly, different effects can be considered. For example, the effects of element insertion and deletion can be the dynamic constraints: If $\{e \notin r\}$ ($\{e \in r\}$) before the assignment, then $\{e \in r\}$ ($\{e \notin r\}$) after the assignment, respectively. Yet, these effects are still questionable, and in the current work we do not handle dynamic effects.

3.2 Definition of Effect Preservation

Consider the update:

$$\begin{aligned} &insert(r, e_1); \\ &\text{if } P \text{ then } insert(r, e_2) \text{ else } skip; \\ &delete(r, e_3) \end{aligned}$$

Starting from any state, there are two possible execution sequences, based on whether P holds after the first insertion. Effect preservation requires that both sequences preserve the $\{e_1 \in r\}$ condition, but only one should preserve the $\{e_2 \in r\}$ condition. Therefore, effect preservation requires distinction between execution sequences. That is, the definition of effect preservation should be execution sequence sensitive. This observation leads us to the following definition:

Definition 2 (The effect preservation property of updates) *An update S is Effect Preserving if all of its execution sequences are effect preserving.*

Now we need to define effect preservation for execution sequences. For that purpose, we need first to assign to a configuration within an execution sequence, the effects of previous assignments along the sequence, so to maintain the history of the sequence.

Definition 3 (Effects of Configurations within Execution Sequences) *Let $\Psi = \gamma_0, \gamma_1, \dots$ be an execution sequence. The effects of the i -th configuration in Ψ , denoted $effects_i(\Psi)$, are: $effects_0(\Psi) = \{\}$, and for $i \geq 0$:*

$$effects_{i+1}(\Psi) = \begin{cases} effects_i(\Psi) \cup effects(A) & \text{if } \gamma_i \text{ is an assignment} \\ & \text{configuration } \langle A; S, s_i \rangle \\ & \text{to a state variable} \\ effects_i(\Psi) & \text{otherwise} \end{cases}$$

For a successful (finite length k) execution sequence, the effects are those of its last configuration: $effects(\Psi) = effects_k(\Psi)$. For a failing (finite) execution sequence the effects are $\{false\}$. Note that the effects set of an assignment configuration does not include the effects of its assignment update.

Property 1 *For a successful execution sequence Ψ , the size of $effects(\Psi)$ is proportional to the length of Ψ .*

Example 3 (Effects-of-configurations) *The effects for the execution sequence*

$$\Psi = \langle r := \text{insert}(r, x); r := \text{delete}(r, y), s \rangle \Rightarrow \langle r := \text{delete}(r, y), s' \rangle \Rightarrow \langle \epsilon, s'' \rangle$$

are: $\text{effects}_0(\Psi) = \{\}$, $\text{effects}_1(\Psi) = \{x \in r\}$, $\text{effects}_2(\Psi) = \{x \in r, y \notin r\} = \text{effects}(\Psi)$.

Note: Due to the rewriting preprocessing of assignments to state variables (see Subsection 2.2), the effects of configurations indeed maintain the intended history of effects. Consider an assignment $A = r := e(r, y_1, \dots, y_n)$ to a state variable r . $\text{effects}(A)$ imposes a constraint on r in terms of the values of y_1, \dots, y_n in the state that follows the update execution. If the variables y_1, \dots, y_n are later on assigned, then $\text{effects}(A)$ should be modified to reflect this change. Since the assignment preprocessing guarantees that these variables are not assigned along the sequence, $\text{effects}(A)$ can be taken as the final effect of the assignment.

The *effect preservation* property is concerned with maintaining the effects of primitive updates along a sequence:

Definition 4 (The Effect Preservation Property of Execution Sequences) *An execution sequence $\Psi = \gamma_0, \gamma_1, \dots$, where $\gamma_i = \langle S_i, s_i \rangle$ $i \geq 0$ is Effect Preserving (EP) if it is either failing or $s_i \models \text{effects}_i(\Psi)$ for all $i \geq 0$.*

Example 4 (Effect preservation)

1. Consider the execution sequence from Example 3. The sequence is effect preserving for $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 4]$ but is not effect preserving for $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 3]$. Therefore, the update $r := \text{insert}(r, x); r := \text{delete}(r, y)$ is not effect preserving.

2. Consider the execution sequence

$$\langle r := \text{insert}(r, x); r := \text{insert}(r, y), s \rangle \Rightarrow \langle r := \text{insert}(r, y), s' \rangle \Rightarrow \langle \epsilon, s'' \rangle$$

and states s, s', s'' where $s' = s[r \mapsto \text{insert}(r, x)^s]$ and $s'' = s'[r \mapsto \text{insert}(r, y)^{s'}]$. The sequence is effect preserving for every state s .

Therefore, the update $r := \text{insert}(r, x); r := \text{insert}(r, y)$ is effect preserving.

Note that a failing execution sequence is effect preserving, and an infinite execution sequence can be effect preserving as well.

Proposition 1 *An execution sequence $\Psi = \langle S_0, s_0 \rangle, \langle S_1, s_1 \rangle, \dots$ in which $s_{i+1} \models \text{effects}_i(\Psi)$ for all $i \geq 0$, is effect preserving.*

3.3 Criteria for Effect Preserving Transformations.

The first question that we need to answer when coming to define effect preserving transformation concerns the kind of transformations we wish to produce. One option is *semantics based code transformation*, e.g., the replacement of $r := \text{insert}(r, e); r := \text{delete}(r, e)$ by *skip*³. However, in this case, we need to set criteria for the desired transformation. For example, Schewe and Thalheim in [33, 36], require that all post conditions of the input update are preserved by the transformed code. But in that case, even a simple update such as $r := \text{insert}(r, e); r := \text{delete}(r, e)$ cannot be replaced by *skip* since, for example, in the state $s = [r \mapsto \{3, 4, 5\}, e \mapsto 3]$, the input update has the postcondition $\{e \notin r\}$ which is not satisfied by *skip*.

Therefore, we adopt a more modest approach where non preservation of effects causes rejection. We term it “repair by rejection”. We still need to characterize desirable repairs since otherwise, the simplest transformation can repair every non effect preserving update by replacing it with *fail*. We set two criteria for good effect preserving transformations:

³This is the approach taken in the Generalized Consistent Specialization theory of [33, 36]. This approach is discussed in Section 5.

1. Minimize rejections inserted for enforcing effect preservation.
2. Minimize run time overhead, i.e., minimize tests overhead.

The first criterion is handled by introducing a *minimal restriction relation* between updates. The restriction relation reflects the desired criterion for minimizing the enforced rejections. An effect preserving transformation is required to produce an update which is a minimal restriction of the input update. The second criterion is handled by replacing update effects by *delta-conditions*, which are minimal conditions that guard against effect violation. A desired effect preserving transformation should produce a minimal restriction using delta-conditions.

3.3.1 Update Restriction – Minimizing Enforced Rejections

We already noted that effect preservation should be execution sequence sensitive. That is, it should take into consideration the different effects of different execution sequences. Therefore, the restriction relation on updates is defined on the basis of a restriction relation on execution sequences.

Restriction Relations on Execution Sequences

We introduce three increasingly tighter relations, termed *restriction*, *EP-restriction* and *minimal-EP-restriction*. The *restriction* relation on execution sequences relates sequences that are either both infinite or terminate in the same state, or one is failing. The *EP-restriction* relation further requires that the restricting sequence is EP. The *minimal-EP-restriction* relation is an EP-restriction with minimal rejections (no needless rejections).

Definition 5 (Restriction relations between execution sequences)

1. An execution sequence Ψ' is a restriction of an execution sequence Ψ , denoted $\Psi' \leq \Psi$, if
 - (a) $start(\Psi') = start(\Psi)$,
 - (b) If Ψ' is infinite then Ψ is also infinite,
 - (c) If Ψ' is successful with $end(\Psi') = s'$ then also Ψ is successful with $end(\Psi) = s'$.

That is, Ψ' might be failing while Ψ is successful and terminates properly, or is infinite.

2. An execution sequence Ψ' is an EP-restriction of an execution sequence Ψ , denoted $\Psi' \leq_{EP} \Psi$, if $\Psi' \leq \Psi$, and Ψ' is EP.
3. An execution sequence Ψ' is a minimal-EP-restriction of an execution sequence Ψ , denoted $\Psi' \leq_{min_EP} \Psi$, if $\Psi' \leq_{EP} \Psi$, and if Ψ is EP then $\Psi \leq_{EP} \Psi'$ (i.e., $\Psi' \equiv \Psi$).

The relations $\leq, \leq_{EP}, \leq_{min_EP}$ have *proper* versions, denoted $<, <_{EP}, <_{min_EP}$, respectively. $\Psi' < \Psi$ means that Ψ' is failing while Ψ is successful or infinite. Also, Ψ and Ψ' are *equivalent*, denoted $\Psi' \equiv \Psi$, if $\Psi' \leq \Psi$, and $\Psi \leq \Psi'$. $\Psi' \equiv \Psi$ means that Ψ and Ψ' are both either failing or successful or infinite, and if they are successful they end in the same state. Ψ and Ψ' are *EP-equivalent*, denoted $\Psi' \equiv_{EP} \Psi$, if $\Psi' \leq_{EP} \Psi$, and $\Psi \leq_{EP} \Psi'$. That is, $\Psi' \equiv_{EP} \Psi$ if $\Psi \equiv \Psi'$ are both effect preserving. Note that a failing execution sequence is a restriction of any other sequence.

Proposition 2 *The relations \equiv and \equiv_{EP} are equivalence relations on execution sequences with a common start state, and the \leq, \leq_{EP} and \leq_{min_EP} relations are partial orders with respect to \equiv .*

Proof: Immediate. \square

Example 5 (Minimal-EP-restriction of execution sequences)

1. Consider the execution sequence Ψ from Example 3. Assume: $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 4]$. The following execution sequence is a minimal-EP-restriction of Ψ :

$$\begin{aligned} &\langle r := \text{insert}(r, x); r := \text{delete}(r, y); \text{if } x \notin r \text{ then fail else skip, } s \rangle \Rightarrow \\ &\langle r := \text{delete}(r, y); \text{if } x \notin r \text{ then fail else skip, } s' \rangle \Rightarrow \\ &\langle \text{if } x \notin r \text{ then fail else skip, } s'' \rangle \Rightarrow \\ &\langle \text{skip, } s'' \rangle \Rightarrow \\ &\langle \epsilon, s'' \rangle \end{aligned}$$

2. Consider the execution sequence

$$\Psi = \langle r := \text{insert}(r, e); r := \text{delete}(r, e), s \rangle \Rightarrow \langle r := \text{delete}(r, e), s' \rangle \Rightarrow \langle \epsilon, s \rangle.$$

Assume: $s = [r \mapsto \{3, 4, 5\}, e \mapsto 4]$. Two minimal-EP-restrictions of ψ are:

$$\begin{aligned} &\langle \text{skip}, s \rangle \Rightarrow \langle \epsilon, s \rangle \leq_{\text{min_EP}} \Psi \text{ and} \\ &\langle \text{fail}, s \rangle \leq_{\text{min_EP}} \Psi. \end{aligned}$$

However, while $\langle \text{fail}, s \rangle <_{EP} \langle \text{skip}, s \rangle \Rightarrow \langle \epsilon, s \rangle$,
we have $\langle \text{fail}, s \rangle \not\prec_{\text{min_EP}} \langle \text{skip}, s \rangle \Rightarrow \langle \epsilon, s \rangle$.

Restriction Relations on Updates

Definition 6 (Restriction relations between updates)

1. An update U' is a restriction of an update U , denoted $U' \leq U$, if for all states s , $\text{seq}(U', s) \leq \text{seq}(U, s)$.
2. An update U' is an EP-restriction of an update U , denoted $U' \leq_{EP} U$, if for all states s , $\text{seq}(U', s) \leq_{EP} \text{seq}(U, s)$.
3. An update U' is a minimal-EP-restriction of an update U , denoted $U' \leq_{\text{min_EP}} U$, if for all states s , $\text{seq}(U', s) \leq_{\text{min_EP}} \text{seq}(U, s)$.

In analogy to the restriction relations of execution sequences, the update restriction relations also induce proper versions and equivalence relations. $U' < U$ means that U' has failures which are successful in U . Updates U and U' are *semantically equivalent* (or just *equivalent*, for short), denoted $U' \equiv U$, if for all states s , $\text{seq}(U', s) \equiv \text{seq}(U, s)$. Update U and U' are *termination equivalent*, denoted $U' \equiv_t U$, if for all states s , the successful sequences are equivalent. That is, there might be states for which one update fails while the other loops.

Proposition 3 *The semantical equivalence is an equivalence relation on the set of updates, and the update relation restriction \leq , \leq_{EP} and $\leq_{\text{min_EP}}$ are partial orders on the set of updates, partitioned by the semantical equivalence relation.*

Example 6 (Minimal-EP-restriction of updates)

Consider the update $S = r := \text{insert}(r, e); r := \text{delete}(r, e)$. Since for the state $s = [r \mapsto \{3, 4, 5\}, e \mapsto 3]$ $\text{seq}(\text{skip}, s) \not\prec_{\text{min_EP}} \text{seq}(S, s)$, we have $\text{skip} \not\prec_{\text{min_EP}} S$.

Properties of the update restriction relations:

1. For every update U , $\text{fail} \leq U$ and $\text{fail} \leq_{EP} U$. That is, the set of updates has a single least element with respect to the relations \leq and \leq_{EP} .

2. $U' \leq_{EP} U$, if $U' \leq U$, and U' is EP.
3. The restriction relations on updates are increasingly tighter. That is: If $U' \leq_{EP} U$ then also $U' \leq U$. If $U' \leq_{\min_EP} U$ then also $U' \leq_{EP} U$.
4. If $U' \leq_{\min_EP} U$ and U is EP, then $U \leq_{EP} U'$ (i.e., $U' \equiv U$). However, the converse is not true since it is possible that all execution sequences of U' are minimal-EP restrictions of the corresponding execution sequences of U , while still U is not effect preserving (some sequences of U are not EP).

The restriction relations between updates are used for characterizing the desirable update transformations as *minimal effect preserving*:

Definition 7 An update transformation Θ is minimal effect preserving if for every update U , $\Theta(U) \leq_{\min_EP} U$.

3.3.2 Delta Conditions – Minimizing Run Time Overhead

Effect preservation can be obtained by inserting a test for past effects following every assignment in the sequence. However, the tests for past effects are costly, since the size of the effects of a configuration $effects_i(\Psi)$ in a sequence Ψ is proportional to i . Therefore, the tests overhead in the worst case is $\mathcal{O}(length(\Psi)^2)$, assuming that testing the effects of primitive updates takes constant time.

The tests overhead can be improved by replacing the tests of past effects by *delta-conditions*, which are minimal formulae that serve as guards against effect violation. Our experience shows that delta-conditions tend to be small and independent of the length of the execution sequence. Therefore, based on this experience, their test overhead for the whole sequence is linear in the length of the execution sequence.

Delta-conditions extract the possible interaction between aggregated past effects and a forthcoming assignment. Consider, for example, the execution sequence Ψ from Example 3, where $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 4]$. The first item in Example 5 presents a minimal-EP-restriction for Ψ . Static analysis of the effects of Ψ reveals that there is a single delta-condition $x \neq y$ to be tested prior to the application of the second assignment in Ψ . Therefore, a more efficient minimal-EP-restriction of Ψ is:

$$\begin{aligned} &\langle r := insert(r, x); \text{ if } x \neq y \text{ then } r := delete(r, y) \text{ else fail}, s \rangle \Rightarrow \\ &\langle \text{ if } x \neq y \text{ then } r := delete(r, y) \text{ else fail}, s' \rangle \Rightarrow \\ &\langle r := delete(r, y), s' \rangle \Rightarrow \\ &\langle \epsilon, s'' \rangle \end{aligned}$$

First we define delta-conditions in general, and then specialize to the simple relational domain with element insertion and deletion alone.

Definition 8 (Delta-Conditions of Assignments) A delta-condition of an assignment update A with respect to a set of formulae P is a minimal collection of First Order Logic formulae $\delta(A, P)$ that guarantees that A does not violate P . That is, for every state s , such that $s \models P$ and $s \models \delta(A, P)$, if $\langle A, s \rangle \Rightarrow s'$, then $s' \models P$.

A delta-condition can be viewed, alternatively, as a first order formula, by taking the conjunction of the formulae in the set. The empty set corresponds to the formula *true*. Henceforth, the exact view of delta-conditions can be understood from the context: A delta-condition in a test is treated as a formula while a delta-condition in a set operation is treated as a set.

Computation of Delta-Conditions: $\delta(A, P)$ is not necessarily unique. The worst (useless) delta-condition for every assignment A and a formula P is $\{false\}$. If A is $x := e$ then $\{P_{x/e}\}$

is another example of a useless delta-condition. The best delta-condition is the empty set, which indicates that the update A does not interfere with P .

Delta-conditions can be computed in a domain specific manner, by considering the assignments and their effects. For the relational domain with element insertions and deletions, and effects formulae of the form $\{e \in r\}$ and $\{e \notin r\}$, $\delta(A, P)$ is computed by the following rules:

1. If $A = r := delete(r, e_1)$ and P includes $e_2 \in r$, then $\delta(A, P)$ includes the formula $e_1 \neq e_2$.
2. If $A = r := insert(r, e_1)$ and P includes $e_2 \notin r$, then $\delta(A, P)$ includes the formula $e_1 \neq e_2$.

Note that if the assigned state variable in A does not occur in P , then $\delta(A, P)$ is $\{\}$. The delta-conditions computed for the examples below are based on these rules.

Definition 9 [Delta-conditions of Configurations within Execution Sequences]

Let $\Psi = \gamma_0, \gamma_1, \dots$ be an execution sequence. The delta-conditions of an assignment configuration is the delta-conditions between its effects and the assignment updates. For non assignment configuration the delta-condition is empty. For $i \geq 0$:

$$\delta_i(\Psi) = \begin{cases} \delta(A, effects_i(\Psi)) & \text{if } \gamma_i \text{ is an assignment} \\ & \text{configuration } \langle A; S, s_i \rangle \\ & \text{to a state variable} \\ \{\} & \text{otherwise} \end{cases}$$

Example 7 (Delta-conditions-of-configurations) Consider the execution sequence Ψ from Example 3. While the effects of its configurations are:

$effects_0(\Psi) = \{\}$, $effects_1(\Psi) = \{x \in r\}$, $effects_2(\Psi) = \{x \in r, y \notin r\}$,

the delta-conditions of its configurations are meaningfully smaller:

$\delta_0(\Psi) = \{\}$, $\delta_1(\Psi) = \{x \neq y\}$, $\delta_2(\Psi) = \{\}$.

Size of delta-Conditions of a Configuration in the relational Domain with Element insertion and Deletion: Consider an assignment configuration $\gamma_i = \langle A; S, s_i \rangle$ in an execution sequence $\Psi = \gamma_0, \gamma_1, \dots$. The size of $\delta_i(\Psi)$ depends on the number of opposite assignments in $\gamma_0, \gamma_1, \dots, \gamma_{i-1}$. If $A = r := op(r, e_i)$ where $op = insert$ ($delete$), then the size of $\delta_i(\Psi)$ depends on the number of deletions (insertions) to r , respectively. In the worst case, if all configurations before γ_i are opposite operations on r : $r := -inverse_op(r, e_j)$, then $\delta_i(\Psi) = \{e_j \neq e_i \mid 0 \leq j < i\}$, which is proportional to the length of the execution sequence. Nevertheless, as already commented above, our experience is that multiple contradictory assignments to the same relation in a single execution sequence are rare. Therefore, based on this experience, we assume that the size of delta-conditions is small (bounded), and their tests take constant time.

4 Algorithms for Enforcing Effect Preservation on Updates

In this section we introduce a minimal effect preserving transformation for *While* updates. That is, an input update S is transformed into an update S' such that $S \leq_{min_EP} S'$. The transformation combines compile time analysis with run time evaluation of delta-conditions. This approach was selected since we already know that:

- Effect preservation requires run time information, as it is execution sequence sensitive. Otherwise, the minimize rejections criterion is not met.
- Naive run time effect preservation has a worst case overhead of $\mathcal{O}(size(S)^2)$ for an update S (assuming that testing the effects of primitive updates takes constant time), since at every modification all past effects must be checked. The delta-conditions optimization does not apply since their computation depends on the full effects of configurations in the actual execution sequence.

Our method includes three steps:

1. Construct a computation tree (graph) that spans all execution sequences of the update.
2. Annotate the computation tree (graph) with delta-conditions.
3. Augment the update with instructions that navigate along the computation tree (graph) and test the computed delta-conditions. The output is an *extended-While* update: A *While* update augmented with external calls.

The transformation is introduced gradually: First, for a restricted subset of *While* without loops, and then it is extended to any *While* update.

4.1 Minimal Effect Preservation for *While* updates without Loops

The effect preserving transformation EP_1 applies two algorithms: *build_CT* for computation tree construction, and *reviseUpdate₁* for code transformation. *build_CT* constructs a tree, associates it with an iterator (a pointer), and returns a static encoding of the resulting iterator. *reviseUpdate₁* interleaves within the input code tree navigation (using the iterator) and tests for delta-conditions that are read from the tree. The EP_1 algorithm returns code in the extended *While* – extended with calls to these external procedures.

Algorithm 2 [EP_1 – Minimal Effect-Preservation for *While* without loops]

input: A *While* update S without loops.

output: A pair: \langle computation tree for S , minimal-EP restriction of S \rangle .

method:

$EP_1(S) = T := \text{build_CT}(S); S' := \text{reviseUpdate}_1(S);$
return $\langle T, S' \rangle$

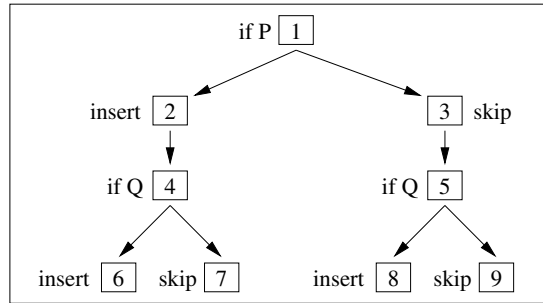
4.1.1 Computation Tree Construction

The *computation tree* is a syntax tree whose nodes are associated with delta-conditions. Paths in the tree correspond to execution sequences of the update, starting from a given state. Nodes correspond to configurations, and are associated with their delta-conditions (note that delta-conditions of configurations are independent from their states).

Example 8 (A computation tree of an update) *The update*

$S(\text{“A”}, \text{“B”}) = \text{if } P \text{ then } Car := \text{insert}(Car, \text{“A”});$
 $\quad \text{if } Q \text{ then } Car := \text{insert}(Car, \text{“B”})$

has the following computation tree:



For this update, the tree captures the four possible execution sequences. Each sequence corresponds to a path of the tree. The effects of the paths are (from left to right): $\{\text{“A”} \in Car, \text{“B”} \in Car\}$, $\{\text{“A”} \in Car\}$, $\{\text{“B”} \in Car\}$, $\{\}$.

The tree construction consists of the actual tree construction, annotation of the tree nodes with effects and delta-conditions, and finally, removal of the effects annotation. The effects annotation is necessary only for the delta-conditions computation. This is important since effects grow linearly with the length of the execution sequence (path in the tree), while delta-conditions tend to be small and independent of path length.

The algorithm uses a node constructor $n = Node(S)$ that returns a node n labeled by S (denoted $label(n)$), and methods $addLeft(n, m)$ ($addRight(n, m)$) that add a left (right) node m to n . The left (right) branch of n are retrieved by $getLeft(n)$ ($getRight(n)$), respectively, and the conjunction of its delta-conditions is retrieved by $getDelta(n)$. The algorithm returns an *iterator* to the tree. The iterator function $node()$ returns the currently pointed node, and the functions $right()$, $left()$ advance the iterator to the right or left child node of the current node, respectively (if the current node is a leaf node they do nothing). The iterator is necessary for interleaving tree navigation in the transformed update.

Algorithm 3 [*build_CT* – Build a computation tree for *While* without loops]

input: A *While* statement S without loops.

output: An iterator for a computation tree for S , initialized to the root node.

method: $build_CT(S) = iterator(annotateT(CT(S)))$,
 where CT and $annotateT$ are the following procedures:

I. CT(S) – the actual tree construction. CT is defined inductively on the structure of *While* statements, following the semantics of the transition relation \Rightarrow . Therefore, the sequence operator $;$ is taken as left associative. That is, in $S_1; S_2$, S_1 can be either primitive or an **if** statement, since it cannot include the sequence operator.

1. For a primitive update U , $CT(U) = Node(U)$.
2. $CT(S_1; S_2) = root$, where $root = CT(S_1)$.
 For all leaves l of $root$, such that $label(l) \neq fail$ do: $addLeft(l, CT(S_2))$.
3. $CT(\text{if } P \text{ then } S_1 \text{ else } S_2) = root$, where $root = Node(\text{if } P \text{ then } S_1 \text{ else } S_2)$,
 $addLeft(root, CT(S_1))$ and $addRight(root, CT(S_2))$.

II. annotateT(T) – the annotation procedure. It annotates a computation tree T built by CT with effects and with delta-conditions with respect to these effects. The effects annotation is only intermediate, and is removed once all the delta conditions are computed.

$annotateT(T) = remove_effects(annotateT_helper(T, \{\}))$, where,

$$\begin{aligned}
 & annotateT_helper(T, eff) = \\
 & \quad effects(root(T)) = eff, \\
 & \quad \delta(root(T)) = \begin{cases} \delta(label(root(T)), eff) & \text{if } label(root(T)) \text{ is a state variable assignment} \\ \{\} & \text{otherwise} \end{cases} \\
 & \quad eff' = \begin{cases} eff \cup effects(label(root(T))) & \text{if } label(root(T)) \text{ is a state variable assignment} \\ eff & \text{otherwise} \end{cases} \\
 & \quad annotateT_helper(getLeft(T), eff') \\
 & \quad annotateT_helper(getRight(T), eff')
 \end{aligned}$$

4.1.2 Code Transformation

The transformation interleaves within the update code commands for navigating the computation tree and for testing delta-conditions. Tree navigation is done using the iterator operations $node()$, $left()$, $right()$. The statically computed delta-conditions are read from the tree and

checked at run time. The transformation is inductively defined on the structure of *While* statements. A syntactic piece of code is denoted $[..]$, and code concatenation is expressed by $[..] \cdot [..]$. Evaluable constructs within the syntactic brackets are understood from context.

Algorithm 4 [*reviseUpdate₁* – Code transformation for *While* without loops]

input: A *While* update S without loops.

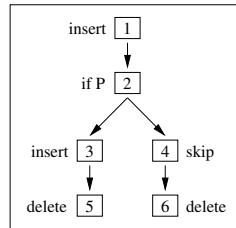
output: An extended *While* update which is a minimal-EP restriction of S , when combined with the computation tree of S .

method:

1. S is a primitive update:
 - (a) S is an assignment to a state variable:
Replace S by $[if\ getDelta(node())\ then\ S;left()\ else\ fail];$
 - (b) Otherwise, S is a regular assignment or skip or fail:
Replace S by: $[S;left()]$
2. $S = [S_1; S_2]$: Replace S by $reviseUpdate_1(S_1) \cdot reviseUpdate_1(S_2)$
3. $S = [if\ P\ then\ S_1\ else\ S_2]$: Replace S by
 $[if\ P\ then\ left();] \cdot reviseUpdate_1(S_1) \cdot$
 $[else\ right();] \cdot reviseUpdate_1(S_2)$

Example 9 (The EP_1 Transformation of a non effect preserving update into an effect preserving one.) Consider the following artificial, but comprehensive, update $S(x, y, z)$, where P is and arbitrary condition:

$S(x, y, z) =$
 1 $r := insert(r, x);$
 2 **if** P **then** $r := insert(r, y);$
 3 $r := delete(r, z)$



Clearly, $r := delete(r, z)$ might violate the effects of both insertions, $insert(r, x)$ and $insert(r, y)$, depending on P and the values of x, y, z . The effects and delta-conditions of nodes are as follows:

node	effects	delta-condition
1	$\{\}$	$\{\}$
2	$\{x \in r\}$	$\{\}$
3	$\{x \in r\}$	$\{\}$
4	$\{x \in r\}$	$\{\}$
5	$\{x \in r, y \in r\}$	$\{x \neq z, y \neq z\}$
6	$\{x \in r\}$	$\{x \neq z\}$

Recall that the effects of a node do not include the effects of the node itself. The delta-conditions of nodes 1, 2, 3 and 4 are empty since there is no previous update whose effects could be violated. The delta-conditions of nodes 5 and 6 are inequalities of the inserted and deleted elements. Applying Algorithm 4 to S returns the following update where the lines are labeled according the original code:

$S'(x, y, z) =$
 1 **if** $getDelta(node())$ **then** $r := insert(r, x);left()$ **else** $fail;$
 2a **if** P **then**

```

2b         left();
2c         if getDelta(node()) then r := insert(r,y);left() else fail;
2d     else
2e         right(); skip; left();
3         if getDelta(node()) then r := delete(r,z);left() else fail

```

Where the external calls $getDelta(node())$, $left()$ and $right()$ refer to the computation tree $build_CT(S)$.

Algorithm 4 can be further statically optimized by removing redundant tests of empty delta-conditions. For this purpose, the $reviseUpdate_1$ procedure should accept also the computation tree as a parameter, and use a tree service $nodes_T(occur(i,U),S)$ that maps the i -th occurrence of a primitive update U in S to the set of tree nodes that correspond to configurations of this occurrence of U . In Example 9, $nodes_T(occur(1,r := insert(r,x)),S) = \{1\}$, and $nodes_T(occur(1,r := insert(r,z)),S) = \{5,6\}$. Such an optimization simplifies the output update and saves redundant run time tests of delta-conditions. The optimization is obtained by revising the primitive update entry in Algorithm 4, as follows:

If S is the i -th occurrence of a primitive update:

1. S is an assignment to a state variable, and for some $n \in nodes_T(i,S)$, $\delta(n) \neq \emptyset$:
Replace S by [if $getDelta(node())$ then S ; $left()$ else $fail$];
2. Otherwise:
Replace S by: [S ; $left()$]

4.2 Minimal Effect Preservation for *While* updates with Loops

The presence of loops introduces difficulties in detecting past effects since the actual assignments being applied are not made explicit by the syntactic structure, but are determined at run time by loop repetitions. Therefore, we need to strengthen the compile time created structure with information that can capture the dynamics of loops, and extend the update so that it can track the actually visited loops. The nodes of the computation tree, which is extended into a *computation graph*, are annotated with additional delta-conditions with respect to all possibly visited nodes, and the code transformation prepares structures for storing and testing the values of variables at run time.

As before, the effect preserving transformation EP_2 applies two algorithms: $build_CG$ for computation graph construction, and $reviseUpdate_2$ for code transformation. The EP_2 algorithm returns code in *While*, extended with external calls for navigating the computation graph, testing the delta-conditions, and recording values of variables in loops.

Algorithm 5 [EP_2 – Minimal Effect-Preservation for *While*]

input: A *While* statement S .

output: A pair: \langle computation graph for S , minimal-EP restriction of S \rangle .

method:

```

 $EP_2(S) = G := build\_CG(S); S' := reviseUpdate_2(S,G);$ 
    return  $\langle G, S' \rangle$ 

```

4.2.1 Computation Graph Construction

As in Algorithm 3, the graph construction consists of the actual graph construction, annotation of the graph nodes with delta-conditions, and finally, removal of redundant annotation that is necessary only for the delta-condition computation.

Algorithm 6 [*build_CG* – Build a computation graph for *While*]**input:** A *While* statement *S*.**output:** An iterator for a computation graph for *S*, initialized to the root node.**method:** $build_CG(S) = iterator(annotateG(CG(S)))$,
where *CG* and *annotateG* are the following procedures:**I. CG(S)** – the actual graph construction. *CG* extends the inductive tree construction of procedure *CT* with a fourth entry for a while update:

4. $CG(\text{while } P \text{ do } S) = root$, where $root = Node(\text{while } P \text{ do } S)$,
 $addLeft(root, CG(\text{if } P \text{ then } S \text{ else skip}))$.
 For all leaves l of $left(left(root))$, such that $label(l) \neq fail$ do: $addLeft(l, root)$

The while entry follows the while semantics:

$$\langle \text{while } P \text{ do } S, s \rangle \Rightarrow \langle \text{if } P \text{ then } (S; \text{while } P \text{ do } S) \text{ else skip}, s \rangle$$

Loop repetitions in a while statement are captured by graph cycles. Note that the graphs are finite and have leaf nodes – the leaf in a while statement graph is the right child node of the added if-then-else statement. Therefore, the construction algorithm is well-defined.

II. annotateG(G) – the annotation procedure. It annotates a computation graph *G* built by *CG* with delta-conditions with respect to its effects. During the annotation phase each node *n* is associated, in addition to its effects and delta-conditions, with:

1. $possible(n)$ – All assignment (to state variable) nodes on cycles through *n*.
2. $\delta Possible(n)$ – Delta-conditions with respect to the effects of the assignments in these nodes. For each node *m* in $possible(n)$, a delta-condition between $label(n)$ and $effects(label(m))$ is computed. $\delta Possible(n)$ is a set of all pairs $\langle m, \delta_m \rangle$ where $m \in possible(n)$ and $\delta_m = \delta(label(n), effects(label(m)))$.

Like the effects annotation, the $possible(n)$ annotation, which grows linearly with the length of the execution sequence (path in the graph), is only intermediate, and is removed when the annotation is completed. The only annotations left in the final graph are the $\delta(n)$ and $\delta Possible(n)$ annotations, which tend to be small and independent of path length.

The set $possible(n)$, for a node *n*, can be computed using any cycle detection algorithm. However, it is not sufficient to consider only simple cycles since cycles through nested loops are not simple (the while node in the cycle must be visited several times). Below we provide separate annotation procedures for the $possible$ and $\delta Possible$ sets. The effects and the delta-conditions annotations are essentially the ones from the tree construction version (Algorithm 3).

$$annotateG(G) = \\ \text{remove}(annotate_deltaPossible(annotate_possible(\\ \text{annotate_effects_delta}(G))))$$

where,

1. $annotate_effects_delta(G)$: Same as $annotateT$ from the tree construction algorithm. The only difference is the escape from looping. For that purpose, the recursive application of $annotate_helper$ on $getLeft(G)$ should be preceded by a test that the child node was not previously visited. For example, test that $effects(getLeft(G))$ is not defined already.
2. Set for all nodes *n* of the computation graph: $possible(n) = \emptyset$.
 $annotate_possible(G) =$

- (a) Mark $\text{root}(G)$ as visited.
- (b) For $n = \text{getLeft}(\text{root}(G))$ or $\text{getRight}(\text{root}(G))$:
- If n is marked as visited:
If $\text{possible}(n) = \text{possible}(\text{root}(G))$: **Stop.**
else $\text{possible}(n) = \text{possible}(n) \cup \{\text{root}(G)\}$
 - $\text{possible}(n) = \text{possible}(n) \cup \text{possible}(\text{root}(G))$
- (c) $\text{annotate_possible}(\text{getLeft}(\text{root}(G)))$
 $\text{annotate_possible}(\text{getRight}(\text{root}(G)))$

3. $\text{annotate_}\delta\text{Possible} =$

For every node n in G , which is an assignment to a state variable node:

For every node $m \in \text{possible}(n)$, which is an assignment to a state variable node:

Add to $\delta\text{Possible}(n)$: $\langle m, \delta(\text{label}(n), \text{effects}(\text{label}(m))) \rangle$.

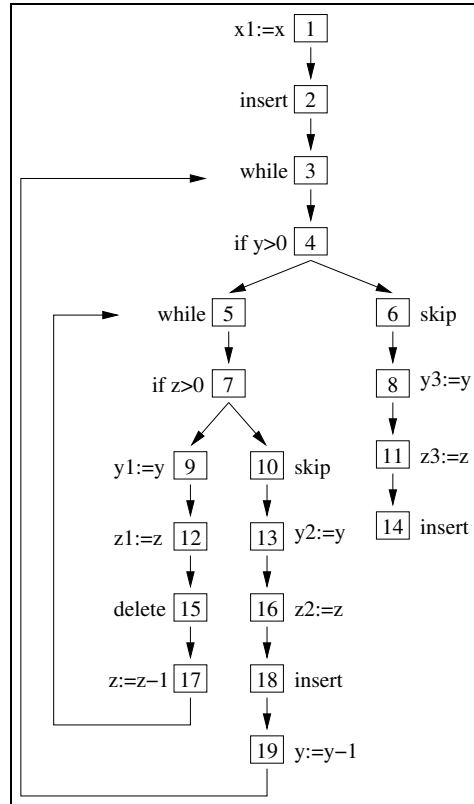
4. The remove procedure removes the effects and the possible annotations from the graph.

Example 10 (Annotated computation graph with loops.) A While statement and its computation graph.

```

S(x, y, z) =
1   x1 := x;
2   r := insert(r, x1);
3   while y > 0 do
4     while z > 0 do
5       y1 := y;
6       z1 := z;
7       r := delete(r, y1 + z1);
8       z := z - 1;
9     y2 := y;
10    z2 := z;
11    r := insert(r, y2 + z2);
12    y := y - 1;
13  y3 := y;
14  z3 := z;
15  r := insert(r, y3 + z3);

```



The graph annotations for the assignment to a state variable nodes.

node	effects	delta-condition	possible	$\delta\text{Possible}$
2	$\{\}$	$\{\}$	$\{\}$	$\{\}$
15	$\{x_1 \in r\}$	$\{x_1 \neq y_1 + z_1\}$	$\{18\}$	$\{(18, y_1 + z_1 \neq y_2 + z_2)\}$
18	$\{x_1 \in r\}$	$\{\}$	$\{15\}$	$\{(15, y_1 + z_1 \neq y_2 + z_2)\}$
14	$\{x_1 \in r\}$	$\{\}$	$\{15\}$	$\{(15, y_1 + z_1 \neq y_3 + z_3)\}$

□

4.2.2 Code Transformation

The full transformation for *While* updates extends the previous one for *While* updates without loops. As before, it interleaves within the update code commands for navigation of the computation graph (using the iterator functions *node()*, *right()*, *left()*) and run time testing of effect preservation (using *getDelta(node())* and a procedure *test δ Possible(node())*). In addition, the transformation adds manipulation of run time value recording, using a procedure *updateValues(node(), $\langle x_1, \dots, x_m \rangle$)*, that records in the given node the current values of the variables x_1, \dots, x_m .

The *updateValues* procedure handles the run time recording of variable values within repeated loop rounds. For that purpose, a node n in the graph that represents an assignment $r := e(r, x_1, \dots, x_m)$ to a state variable r (e.g., $e = \text{insert}$ or delete), is associated at run time with a collection *values*(n) of tuples of values of x_1, \dots, x_m used in the expression e . Whenever the assignment is executed, the procedure *updateValues*(n, x_1, \dots, x_m) is applied, and adds the current values of the variables x_1, \dots, x_m to the collection.

The *test δ Possible* procedure applies a run time test for the *δ Possible* conditions that annotate the node n (the delta conditions are a compile time product). Recall that

$$\delta\text{Possible}(n) = \{ \langle m, \delta_m \rangle \mid \delta_m = \delta(\text{label}(n), \text{effects}(\text{label}(m))), \\ m \text{ is an assignment to a state variable node,} \\ \text{residing on a cycle through } n \}.$$

For each pair $\langle m, \delta_m \rangle$ in $\delta\text{Possible}(n)$, *test δ Possible*(n) applies the associated delta-condition δ_m , when instantiated by *values*(m). This way the delta-conditions between the assignment of n and the effects of all previous executions of the assignment in m are tested. Assume that *values*(m) records the values of variables x_1, \dots, x_m .

test δ Possible(n) =

For every $\langle m, \delta_m \rangle \in \delta\text{Possible}(n)$ such that $\delta_m \neq \{\}$:

For every entry v_1, \dots, v_m in *values*(m):

Substitute in δ_m : $x_1/v_1, \dots, x_m/v_m$, and apply δ_m .

test δ Possible(n) = *true* if all tests are *true* and *false* otherwise.

The *reviseUpdate₂* algorithm extends the previous *reviseUpdate₁* by modifying the assignment to state variable transformation and adding a fourth transformation for a *while* statement. We list only these extensions:

Algorithm 7 [*reviseUpdate₂* – Code transformation for *While*]

input: A *While* update S .

output: An extended *While* update which is a minimal-EP restriction of S , when combined with the computation graph of S .

method:

1.a. $S = [r := e(r, x_1, \dots, x_m)]$, an assignment to a state variable r ,

where expression e is over variables x_1, \dots, x_m :

Replace S by

[if *getDelta*(*node*()) and *test δ Possible*(*node*())

then *updateValues*(*node*(), $\langle x_1, \dots, x_m \rangle$); S ; *left*()

else *fail*];

4. $S = [\text{while } P \text{ do } S']$: Replace S by:

[*while* P do { *left*();*left*(); } · *reviseUpdate₂*(S') · []; *right*()]

Brackets { and } serve to enclose a block of statements. The double *left*();*left*() for loops is required to jump over the added **if-then-else** node. \square

Example 11 (The EP_2 Transformation of a non effect preserving update into an effect preserving one.) For the update S in Example 10, algorithm 5 returns the following update where lines are labelled according to the original line of code:

```

1       $x_1 := x;$ 
2a     if  $getDelta(node())$  and  $test_{\delta}Possible(node())$  then
2b          $updateValues(node(), \langle x_1 \rangle);$ 
2c          $r := insert(r, x_1); left()$ 
2d     else fail ;
3a     while  $y > 0$  do
3b          $left(); left()$ 
4a         while  $z > 0$  do
4b              $left();$ 
5              $y_1 := y; left();$ 
6              $z_1 := z; left();$ 
7a         if  $getDelta(node())$  and  $test_{\delta}Possible(node())$  then
7b              $updateValues(node(), \langle y_1, z_1 \rangle);$ 
7c              $r := delete(r, y_1 + z_1); left()$ 
7d         else fail;
8          $z := z - 1; left();$ 
4c      $right();$ 
9      $y_2 := y; left();$ 
10     $z_2 := z; left();$ 
11a   if  $getDelta(node())$  and  $test_{\delta}Possible(node())$  then
11b        $updateValues(node(), \langle y_2, z_2 \rangle);$ 
11c        $r := insert(r, y_2 + z_2); left()$ 
11d   else fail;
12     $y := y - 1; left();$ 
3c    $right();$ 
13    $y_3 := y; left();$ 
14    $z_3 := z; left();$ 
15a   if  $getDelta(node())$  and  $test_{\delta}Possible(node())$  then
15b        $updateValues(node(), \langle y_3, z_3 \rangle);$ 
15c        $r := insert(r, y_3 + z_3); left()$ 
15d   else fail □

```

4.3 Correctness and Complexity of the Update Transformations

4.3.1 Enforcing Effect Preservation on Execution Sequences

In this subsection we concentrate on effect preservation in the semantic domain of *While* programs, i.e., the set of all execution sequences over a given set of variables. This “pre-processing” study is essential since effect preservation properties of *While* updates are defined in terms of effect preservation of their execution sequences, and correctness of *While* transformations is proved by referring to their impact on their execution sequences. We introduce an execution sequence transformation that enforces *minimal effect preservation*. The transformation is further optimized by using *delta-conditions*. These transformations are used later on to prove the effect preservation properties of the update transformations.

The Conditional Assignment Transformation

The following transform maps an execution sequence Ψ to an execution sequence denoted $CA(\Psi)$, which is a minimal effect preserving restriction of Ψ .

Definition 10 [Conditional assignment transformation]

Let $\Psi = \langle S_0, s_0 \rangle, \langle S_0, s_0 \rangle, \dots$, be an execution sequence. $CA(\Psi)$ is the execution sequence resulting from the replacement of every assignment to a state variable configuration $\langle A; S, s_i \rangle$ in Ψ by the configuration sequence $\gamma_1, \gamma_2, \gamma_3$, where:

$$\gamma_1 = \langle A; \text{if } \neg \text{effects}_i(\Psi) \text{ then fail else skip}; S, s_i \rangle,$$

$$\gamma_2 = \langle \text{if } \neg \text{effects}_i(\Psi) \text{ then fail else skip}; S, s_{i+1} \rangle,$$

$$\gamma_3 = \begin{cases} \langle \text{fail}, s_{i+1} \rangle & \text{if } s_{i+1} \not\models \text{effects}_i(\Psi) \\ \langle \text{skip}; V, s_{i+1} \rangle & \text{otherwise} \end{cases}$$

If the resulting sequence includes a failing configuration cut the sequence after the first failing configuration.

Example 12 (The CA Transformation) Consider the execution sequence from Example 3

$$\langle r := \text{insert}(r, x); r := \text{delete}(r, y), s \rangle \Rightarrow \langle r := \text{delete}(r, y), s' \rangle \Rightarrow \langle \epsilon, s'' \rangle$$

and states s, s', s'' where $s' = s[r \mapsto \text{insert}(r, x)^s]$ and $s'' = s'[r \mapsto \text{delete}(r, y)^{s'}]$. The effects of the configurations in the sequence are $\text{effects}_0(\Psi) = \{\}$, $\text{effects}_1(\Psi) = \{x \in r\}$ and $\text{effects}_2(\Psi) = \{x \in r, y \notin r\}$.

1. Assume: $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 4]$. Then, $CA(\Psi)$ is⁴

$$\begin{aligned} & \langle r := \text{insert}(r, x); \text{if true then fail else skip}; r := \text{delete}(r, y), s \rangle \Rightarrow \\ & \langle \text{if true then fail else skip}; r := \text{delete}(r, y), s' \rangle \Rightarrow \\ & \langle \text{skip}; r := \text{delete}(r, y), s' \rangle \Rightarrow \\ & \langle r := \text{delete}(r, y); \text{if } x \notin r \text{ then fail else skip}, s' \rangle \Rightarrow \\ & \langle \text{if } x \notin r \text{ then fail else skip}, s'' \rangle \Rightarrow \\ & \langle \text{skip}, s'' \rangle \Rightarrow \\ & \langle \epsilon, s'' \rangle \end{aligned}$$

2. Assume: $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 3]$. Then, $CA(\Psi)$ is

$$\begin{aligned} & \langle r := \text{insert}(r, x); \text{if true then fail else skip}; r := \text{delete}(r, y), s \rangle \Rightarrow \\ & \langle \text{if true then fail else skip}; r := \text{delete}(r, y), s' \rangle \Rightarrow \\ & \langle \text{skip}; r := \text{delete}(r, y), s' \rangle \Rightarrow \\ & \langle r := \text{delete}(r, y); \text{if } x \notin r \text{ then fail else skip}, s' \rangle \Rightarrow \\ & \langle \text{if } x \notin r \text{ then fail else skip}, s'' \rangle \Rightarrow \\ & \langle \text{fail}, s'' \rangle \end{aligned}$$

That is, for the state $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 4]$, in which Ψ is effect preserving (see Example 4), $CA(\Psi)$ is also effect preserving, successful and ends in the same state, while for the state $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 3]$, in which Ψ is not effect preserving, $CA(\Psi)$ is failing. In any case, for both states, $CA(\Psi) \leq_{EP} \Psi$.

The following claim shows that the CA transformation does not needlessly restrict execution sequences that are already effect preserving:

Claim 1 [Correctness and Minimality of the CA transformation]

For every execution sequence Ψ , $CA(\Psi) \leq_{min_EP} \Psi$.

Therefore, we conclude that the CA transformation does not needlessly restrict execution sequences that are already effect preserving. Such transformations are termed *minimal effect preserving transformations*.

Conclusion 1 The CA transformation is a minimal effect preserving transformation. That is, $CA(\Psi) \leq_{EP} \Psi$, and if Ψ is already effect preserving, then $CA(\Psi) \equiv \Psi$.

⁴An empty effects set is the formula *true*.

The Delta-Conditions Based Transformation

Delta-conditions were introduced as minimal conditions for guaranteeing that the effects of an intermediate configuration in an execution sequence are preserved by its successor configuration. Therefore, the CA transformation can be revised into a delta-conditions based transformation CA_δ that replaces tests of full effects by tests of delta-conditions. Of course, the revised transformation CA_δ must preserve the properties of the former CA transformation.

Definition 11 [Delta-conditions based transformation]

Let $\Psi = \langle S_0, s_0 \rangle, \dots, \langle S_k, s_k \rangle$ be an execution sequence. $CA_\delta(\Psi)$ is the execution sequence resulting from the replacement of every assignment to a state variable configuration $\langle A; S, s_i \rangle$ in Ψ , by the configuration sequence γ_1, γ_2 , where:

$$\gamma_1 = \langle \text{if } \delta_i(\Psi) \text{ then } A \text{ else fail}; S, s_i \rangle,$$

$$\gamma_2 = \begin{cases} \langle A; S, s_i \rangle & \text{if } s_i \models \delta_i(\Psi) \\ \langle \text{fail}, s_i \rangle & \text{otherwise} \end{cases}$$

If the resulting sequence includes a failing configuration cut the sequence after the first failing configuration.

Example 13 (The delta-conditions based Transformation) Consider the execution sequence from Example 3, and states s, s', s'' , where $s' = s[r \mapsto \text{insert}(r, x)^s]$ and $s'' = s'[r \mapsto \text{delete}(r, y)^{s'}]$. The delta-conditions of the configurations in the sequence are $\delta_0(\Psi) = \{\}$, $\delta_1(\Psi) = \{x \neq y\}$ and $\delta_2(\Psi) = \{\}$.

1. Assume: $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 4]$. Then, $CA_\delta(\Psi)$ is

$$\begin{aligned} & \langle \text{if true then } r := \text{insert}(r, x) \text{ else fail}; r := \text{delete}(r, y), s \rangle \Rightarrow \\ & \langle r := \text{insert}(r, x); r := \text{delete}(r, y), s \rangle \Rightarrow \\ & \langle \text{if } x \neq y \text{ then } r := \text{delete}(r, y) \text{ else fail}, s' \rangle \Rightarrow \\ & \langle r := \text{delete}(r, y), s' \rangle \Rightarrow \\ & \langle \epsilon, s'' \rangle \end{aligned}$$

2. Assume: $s = [r \mapsto \{3, 4, 5\}, x \mapsto 3, y \mapsto 3]$. Then, $CA_\delta(\Psi)$ is

$$\begin{aligned} & \langle \text{if true then } r := \text{insert}(r, x) \text{ else fail}; r := \text{delete}(r, y), s \rangle \Rightarrow \\ & \langle r := \text{insert}(r, x); r := \text{delete}(r, y), s \rangle \Rightarrow \\ & \langle \text{if } x \neq y \text{ then } r := \text{delete}(r, y) \text{ else fail}, s' \rangle \Rightarrow \\ & \langle \text{fail}, s' \rangle \Rightarrow \end{aligned}$$

Compared with the execution sequence that results from the CA -transformation in Example 12, the execution sequence $CA_\delta(\Psi)$ is shorter, and there is a single equality test which precedes the assignment. As before, for both states, $CA_{\text{delta}}(\Psi) \leq_{EP} \Psi$.

Claim 2 [Correctness and Minimality of the CA_δ transformation] For every execution sequence Ψ , $CA_\delta(\Psi) \leq_{\text{min-EP}} \Psi$.

Conclusion 2 The CA_δ transformation is a minimal effect preserving transformation. That is, $CA_\delta(\Psi) \leq_{EP} \Psi$, and if Ψ is already effect preserving, then $CA_\delta(\Psi) \equiv \Psi$.

4.3.2 Minimal Effect Preserving Update Transformation

In this subsection we prove that the update transformations introduced above are minimal effect preserving. Recall that a transformation Θ is minimal effect preserving, if it produces minimal-EP-restrictions of its input updates (Definition 7). Observing the definition of this relation, it means that for all states s , $\text{seq}(\Theta(U), s) \leq_{\text{min-EP}} \text{seq}(U, s)$ (Definition 6). Our proof uses the results

about the CA_δ transformation of execution sequences. For each of the two update transformations Θ introduced in Subsections 4.1 and 4.2, we show that

for every update U , for all states s , $seq(\Theta(U), s) \equiv_{EP} CA_\delta(seq(U, s))$.

Since by Proposition 2:

for every update U , for all states s , $CA_\delta(seq(U, s)) \leq_{min_EP} seq(U, s)$,

it follows that

for every update U , for all states s , $seq(\Theta(U), s) \leq_{min_EP} seq(U, s)$.

Therefore, by Definition 6,

for every update U , $\Theta(U) \leq_{min_EP} U$,

and by Definition 7, Θ is minimal effect preserving.

Correctness of the transformation of *While* updates without loops

The main problem is to show

for every update U , for all states s , $seq(reviseUpdate_1(U), s) \equiv_{EP} CA_\delta(seq(U, s))$.

Once this is proved, the minimal effect preservation property of EP_1 is obtained as outlined above.

Lemma 1 *Let U be a *While* update without loops, and s a state. The execution sequence $seq(U, s)$ corresponds to a full path in the computation tree of U , $CT(U)$, such that:*

1. $seq(U, s)_0$ corresponds to $root(CT(U))$, and if $seq(U, s)_i$ corresponds to node n_i in the tree, then $seq(U, s)_{i+1}$ corresponds to a child node n_{i+1} of n_i . If $seq(U, s)$ is a successful sequence, the last terminal configuration does not correspond to any node, and its previous configuration corresponds to a leaf node. If the sequence is failing, its last configuration corresponds to a leaf node.
This correspondence defines a 1 : 1 mapping between the configurations in $seq(U, s)$ (excluding the terminal configuration, if exists) and the nodes of the path.
2. If $seq(U, s)_i$ corresponds to node n_i , then $effects_i(seq(U, s)) = effects(n_i)$ and $\delta_i(seq(U, s)) = \delta(n_i)$.

Proposition 4 *Let U be an extended *While* update, such that all external calls are to terminating procedures. Let $removeEC(U)$ be the *While* update that is obtained from U by removing all external calls (if there is a syntactic problem, an external call is replaced by skip). Then, for every state s : $seq(removeEC(U), s) \equiv_{EP} seq(U, s)$.*

Proof: The external calls do not affect termination and any variable assignment. Therefore, for every state s , $seq(removeEC(U), s)$ agrees with $seq(U, s)$ with respect to termination and failures, and if $seq(U, s)$ is effect preserving so is $seq(removeEC(U), s)$. \square

Lemma 2 *Let U be a *While* update without loops, and s a state. Then,*

$$removeEC(seq(reviseUpdate_1(U), s)) = CA_\delta(seq(U, s)).$$

Theorem 1 [Correctness and Minimality of Algorithm 4] *For every update S in While without loops, $reviseUpdate_1(S)$ is a minimal EP restriction of S .*

Proof: By Lemma 2, $(seq(removeEC(reviseUpdate_1(U)), s)) = CA_\delta(seq(U, s))$. By Proposition 4, for every update U that does not include non-terminating external calls, for every state s : $seq(removeEC(U), s) \equiv_{EP} seq(U, s)$. Therefore, $seq(reviseUpdate_1(U), s) \equiv_{EP} CA_\delta(seq(U, s))$. The rest of the proof is as outlined above. \square

The following claim holds under the experimental observation that the size of delta-conditions is small, and is independent from the length of the execution sequence. The reason is that multiple contradictory assignments to the same relation in a single execution sequence do not happen frequently.

Claim 3 *The run time overhead of $reviseUpdate_1(S)$ is $\mathcal{O}(size(S))$.*

Proof: By the last theorem, $reviseUpdate_1(S) \leq_{min_EP} S$. Therefore, for every state s , $seq(reviseUpdate_1(S), s)$ is finite. For every state s , the overhead of $seq(reviseUpdate_1(S), s)$ over $seq(S, s)$ is

$$length(seq(S, s)) \times (Time(getDelta(node())) + Time(navigation\ procedures))$$

Since there are no loops, the length of $seq(S, s)$ is bounded by the $size(S)$.

$$Time(getDelta(node())) = size(delta-condition) \times Time(condition\ test).$$

Under the above assumption, the size of the delta-conditions is bounded. Each condition in a delta-condition is a variable inequality, and therefore, its test takes constant time, since it does not depend on the size of a relation. Therefore, $Time(getDelta(node())) = \mathcal{O}(\infty)$. Navigation procedures are $calO(1)$, since they only advance the iterator. Therefore, the overall overhead is $\mathcal{O}(size(S))$. \square

Correctness of the transformation of While updates

The correctness of the $reviseUpdate_2$ transformation is proved, essentially, similarly to the proof for $reviseUpdate_1$. However, there are two tricky points:

1. *While* allows for infinite execution sequences.
2. A node in the computation graph of a *While* update is annotated with delta-conditions with respect to all possible assignments that might precede its statement when executed. Therefore, its set of delta-condition is a superset of the actual delta-conditions of its corresponding configuration in an execution sequence.

Therefore, the two Lemmas on which the correctness Theorem is based are slightly different.

Lemma 3 *Let U be a While update, and s a state. Every finite prefix of $seq(U, s)$ corresponds to a path from the root in the computation graph of U , $CG(U)$, such that:*

1. $seq(U, s)_0$ corresponds to $root(CG(U))$, and if $seq(U, s)_i$ corresponds to node n_i in the tree, then $seq(U, s)_{i+1}$ corresponds to a child node n_{i+1} of n_i . If $seq(U, s)$ is a finite successful sequence, the last terminal configuration does not correspond to any node, and its previous configuration corresponds to a leaf node. If the sequence is finite failing, its last configuration corresponds to a leaf node.

This correspondence defines a partial mapping from configurations in $seq(U, s)$ (excluding the terminal configuration, if exists) to the nodes on the path.

2. If $seq(U, s)_i$ corresponds to node n_i , then $effects_{s_i}(seq(U, s)) = effects(n_i)$ and $\delta_i(seq(U, s)) \subseteq \delta(n_i) \cup \delta Possible(n_i)$ ⁵.

Lemma 4 *Let U be a While update, and s a state. Then,*

$$removeEC(seq(reviseUpdate_2(U), s)) \equiv_{EP} CA_\delta(seq(U, s)).$$

Theorem 2 [Correctness and Minimality of Algorithm 7] *For every update S in While, $reviseUpdate_2(S)$ is a minimal EP restriction of S .*

Proof: By Lemma 4, $(seq(removeEC(reviseUpdate_1(U)), s)) \equiv_{EP} CA_\delta(seq(U, s))$. By Proposition 4, for every update U that does not include non-terminating external calls, for every state s : $seq(removeEC(U), s) \equiv_{EP} seq(U, s)$. Therefore, $seq(reviseUpdate_1(U), s) \equiv_{EP} CA_\delta(seq(U, s))$. The rest of the proof is as outlined above. \square

As before, the complexity claim holds under the experimental observation that the size of delta-conditions is small, and is independent from the length of the execution sequence.

Claim 4 *If for a state s $seq(S, s)$ is terminating, then the run time overhead of $seq(reviseUpdate_2(S), s)$ is proportional to the multiplication of the length of the execution sequence $seq(S, s)$ by the number of loop repetitions.*

Proof: By the last theorem, $reviseUpdate_2(S) \leq_{min-EP} S$. Therefore, if $seq(S, s)$ is finite, then also $seq(reviseUpdate_2(S), s)$ is finite. For every state s , the overhead of $seq(reviseUpdate_2(S), s)$ over $seq(S, s)$ is

$$length(seq(S, s)) \times (\quad Time(getDelta(node())) + Time(test_\delta Possible(node())) + \\ Time(updateValues(node())) + Time(navigation\ procedures))$$

As in the no-loops case, $Time(getDelta(node())) = \mathcal{O}(1)$, and navigation procedures are $\mathcal{O}(1)$, since they only advance the iterator. The procedure $updateValues(node(), x_1, \dots, x_m)$ is also $\mathcal{O}(1)$, since it adds a new value to a collection. The only additional complexity overhead results from the procedure $test_\delta Possible(node())$ that tests multiple tuples that record variable values resulting from loop repetitions. A call to $test_\delta Possible(node())$ takes time proportional to the number of loop repetitions. Therefore, the run time overhead is $\mathcal{O}(length(seq(S, s)) \times \#(\text{loop repetitions}))$. The number of loop repetitions depends on the update arguments, and usually is much smaller than the length of the execution sequence. Therefore, the run time overhead of $seq(reviseUpdate_2(S), s)$ is much smaller than $\mathcal{O}((length(seq(S, s)))^2)$, which is the overhead of a purely run time effect preservation procedure. \square

5 Related Works

Effect preservation is traditionally related to the paradigm of *integrity constraint management*. This direction is relevant in dynamic situations, where operations can violate necessary properties. The role of integrity constraint maintenance is to guard the consistency of the information base. There are, basically, two major approaches to maintain consistency: *Integrity checking*, where operations are tested, either at run time or at compile time, for being integrity preserving [10, 9, 8, 11, 21, 7], and *integrity enforcement*, where operations are *repaired*, so to guarantee consistency [38, 28, 12, 29, 23, 14, 24, 3, 4]. The problem of effect violation arises in the latter approach, where transactions are automatically constructed. It can also arise in situations where

⁵The notation is used imprecisely here, since $\delta Possible(n_i)$ is a set of pairs, and only the second element in each pair is a delta-condition.

transactions are deductively synthesized ([27]). The *Greatest Consistent Specialization* theory of [36] is a compile time enforcement theory with effect preservation. It generates a fully repaired transaction, which is consistent with respect to a given set of constraints, and preserves the effects of the original transaction. A relaxed version, that allows for stronger repairs, is suggested in [31, 22]. A classification of Research efforts in consistency enforcement appears in [25].

The most common approach in integrity enforcement is that of the run time *Rule Triggering Systems (RTS)* (*active databases*), which are implemented in almost every commercial database product [39, 38, 16, 39]. RTSs have to cope with problems of *termination*, *uniqueness* (*confluence*) and *effect preservation* [12, 13, 37, 40]. Since a rule might be repeatedly applied, an application of a set of rules does not necessarily terminate. Practically, the problem is solved by timing-out or counting the level of nested rule activations. Sophisticated methods [12, 13] deal with static cycle analysis of rules. Furthermore, different orders of rule execution do not guarantee a unique database state. This is the *confluence* problem of RTSs. Static analysis of confluence in RTSs is studied in [37, 40].

The problem of effect violation in active databases occurs when a rule fires other rules that perform an update that is contradictory to an update performed by a previous rule. Automated generation and static analysis of active database rules [38, 6, 12] do neither handle, nor provide a solution to that problem. In fact, contradicting updates are allowed, since the case of inserting and deleting the same tuple is considered as a *skip* operation. The problem lies in the locality of rule actions. Since an action is intended to repair a single event, it does not account for the history of the repair, and therefore might not preserve the original intention of the transaction being repaired. The limitations of RTSs in handling effect preservation are studied in [34, 35].

In earlier versions of this work [17, 1, 2, 18], effect preservation was syntactically defined, based on the data structure that is constructed for a transaction. [17] presents an early investigation of the usage of dependency graphs for ordering constraint enforcement. [1, 2] present our first effect preservation efforts for *While* statements without loops: First, sequence transactions were annotated with two kinds of effects (*desired* and *executed*), and a sequence was said to be effect preserving if its desired effects logically followed from its executed effects. Then, a loop-less transaction was said to be effect preserving if all sequence transactions on its computation tree are effect preserving. However, in [18] we tried to extend this approach for handling effect preservation in general *While* transactions, but encountered major problems, since the meaning of the desired and executed effects in the presence of loops is not straightforward. Consequently, we changed our overall approach to effect preservation, looking for a unified framework that can account for all *While* transactions. The result is a new approach, where effect preservation is semantically defined (rather than syntactically). The move from syntax based effect preservation to semantics based one is dramatic, since the latter provides a uniform framework for *While*, on which general criteria for effect preserving transformations can be developed. The effect preservation terminology developed in Section 3, and the algorithms introduced in section 4 could not have been developed on the basis of syntax based effect preservation alone.

6 Conclusion

In this work we introduced a combined, compile time – run time method for enforcing effect preservation in rule triggering systems. Our method enforces effect preservation on updates written in an imperative language with loops. It is based on the assumption that *effects of primitive database updates* are provided by the developer. The transformation is proved to be *minimal effect preserving*, and under certain conditions provides meaningful improvement over the quadratic overhead of pure run time procedures.

Our goal is to produce a database tool in which effect preservation is a correctness condition for transactions. For that purpose, we are currently implementing our effect preservation algorithm by embedding it within a real database platform. We intend to use an open source database, such

as postgres, and apply effect preservation to stored procedures (procedural database application code maintained within the database). Moreover, we plan to extend the effect preservation process so that it will be applicable to general transactions (and not only to isolated primitive updates). In addition, we plan to experimentally test the assumption about the independence of the size of delta-conditions from the length of the execution sequence.

Further research is needed in order to extend the set of primitive updates such that it includes, for example, attribute modification. Study of dynamic constraints requires further research as well. Another future application domain is semi-structured databases. The theory applies to any domain, provided that the developer associates effects with primitive assignments, and provides an algorithm for deriving delta-conditions.

Acknowledgments: We would like to thank B. Thalheim, K.D. Schewe, F. Bry, E. Mayol, A. Cali and M. Kifer for providing constructive comments on our work.

References

- [1] M. Balaban and S. Jurk. Intentions of Operations – Characterization and Preservation. In *Proc. International ER'02 workshop on Evolution and Change in Data Management (ECDM'02)*, pages 100–111, 2002.
- [2] M. Balaban and S. Jurk. Update-Consistent Query Results by Means of Effect Preservation. In *Proc. Fifth International Conf. on Flexible Query Answering Systems (FQAS'02)*, pages 28–43, 2002.
- [3] M. Balaban and P. Shoval. Enhancing the ER model with structure methods. *Journal of Database Management*, 10(4), 1999.
- [4] M. Balaban and P. Shoval. MEER – an EER model enhanced with structure methods. *Information Systems Journal*, pages 245 – 275, 2001.
- [5] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. *Proceedings of the 20. International Conference on Very Large Data Bases*, 1990.
- [6] 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.
- [7] V. Benzaken and X. Schaefer. Static Integrity Constraint Management in Object-Oriented Databases Programming Languages via Predicate Transformers. In *European Conference on Object-Oriented Programming, ECOOP'97*, Lecture Notes in Computer Science, 1997.
- [8] V. Benzaken and D. Themis. A database programming language handling integrity constraints. *VLDB Journal*, pages 493 – 518, 1995.
- [9] F. Bry. Intensional updates: Abduction via deduction. In *Proc. 7th Conf. on Logi Programming*, 1990.
- [10] F. Bry and R. Manthey. Checking consistency of database constraints: A logical basis. In *Proc. of the VLDB int. Conf.*, pages 13–20, 1986.
- [11] M. Celma and H. Decker. Integrity checking in deductive databases. the ultimate method? *Proceedings of 5th Australasian Database Conference*, pages 136–146, 1995.
- [12] S. Ceri, P. Fraternali, S. Paraboschia, and L. Tanca. Automatic generation of production rules for integrity maintenance. In *ACM Transactions on Database Systems*, volume 19(3), pages 367–422, 1994.

- [13] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. *Proceedings of the 16. International Conference on Very Large Data Bases*, pages 566–577, 1990.
- [14] O. Etzion and B. Dahav. Patterns of self-stabilization in database consistency maintenance. *Data and Knowledge Engineering*, 28(3):299–319, 1998.
- [15] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [16] P. Fraternali, S. Paraboschi, and L. Tanca. Automatic rule generation for constraints enforcement in active databases. In U. Lipeck and B. Thalheim, editors, *Modeling Database Dynamics*, pages 153–173. Springer WICS, 1993.
- [17] S. Jurk and M. Balaban. Improving Integrity Constraint Enforcement by Extended Rules and Dependency Graphs. In *Proc. 22th Conf. on DEXA*, 2001.
- [18] S. Jurk and M. Balaban. Towards Effect Preservation of Updates with Loops. In *Proc. Fifth IFIP TC-11 WG 11.5 Working Conf. on Integrity and Internal Control in Information Systems (IICIS'02)*, pages 59–75, 2002.
- [19] G. Kniesel. *ConTraCT – A Refactoring Editor Based on Composable Conditional Program Transformations*. Technical Report, Computer Science Dept., University of Bonn, 2005.
- [20] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming, Special Issue on "Program Transformation"*, Lammel, R. (ed.), 52:9–51, 2004.
- [21] S.Y. Lee and T.W. Ling. Further improvement on integrity constraint checking for stratifiable deductive databases. In *Proc. 22th Conf. on VLDB*, pages 495–505, 1996.
- [22] S. Link. Consistency enforcement in databases. In L. Bertossi, G.O.H. Katona, K.-D. Schewe, and B. Thalheim, editors, *Semantics in Databases, Second International Workshop, Dagstuhl Castle, Germany*, 2003.
- [23] E. Mayol and E. Teniente. Structuring the process of integrity maintenance. In *Proc. 8th Conf. on Database and Expert Systems Applications*, pages 262–275, 1997.
- [24] E. Mayol and E. Teniente. Addressing efficiency issues during the process of integrity maintenance. In *Proc. 10th Conf. on Database and Expert Systems Applications*, pages 270–281, 1999.
- [25] E. Mayol and Ernest Teniente. A survey of current methods for integrity constraint maintenance and view updating. In Chen, Embley, Kouloumdjian, Liddle, Roddick, editor, *Intl. Conf. on Entity-Relationship Approach*, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73, 1999.
- [26] H.R. Nielson and F. Nielson. *Semantics with Applications – A Formal Introduction*. John Wiley & Sons, 1992.
- [27] X. Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, pages 626 – 677, 1993.
- [28] X. Qian, R. Jullig, and M. Daum. Consistency Management in a Project Management Assistant. In *ACM-SIGSOFT'90*, 15(6), 1990.
- [29] K.A. Ross and D. Srivastava. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *ACS-SIGMODF'96*, 1996.

- [30] K.D. Schewe S. Link. Computability and Decidability Issues in the Theory of Consistency Enforcement. In *Electronic Notes in Theoretical Computer Science*, volume 42, 2001.
- [31] K.D. Schewe S. Link. Towards an Arithmetic Theory of Consistency Enforcement Based on Preservation of δ Constraints. In *Electronic Notes in Theoretical Computer Science*, volume 61, pages 1–20, 2002.
- [32] E. Sacerdoti. The nonlinear nature of plans. In *ijcai-75*, pages 206–214, 1975.
- [33] K.D. Schewe. Consistency enforcement in entity-relationship and object-oriented models. *Data and Knowledge Eng.*, 28(1):121–140, 1998.
- [34] K.D. Schewe and B. Thalheim. Consistency enforcement in active databases. In S. Chakravarty and J. Widom, editors, *Research Issues in Data Engineering – Active Databases*, pages 71–76. IEEE Computer Society Press, 1994.
- [35] K.D. Schewe and B. Thalheim. Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybernetica*, 13:277–304, 1998.
- [36] K.D. Schewe and B. Thalheim. Towards a theory of consistency enforcement. *Acta Informatics*, 36:97–141, 1999.
- [37] van der Voort and A. Siebes. Termination and confluence of rule execution. In *In Proceedings of the Second International Conference on Information and Knowledge Management*, November 1993.
- [38] J. Widom and S. Ceri. Deriving production rules for constraint maintenance. In *Proc. 16th Conf. on VLDB*, pages 566–577, 1990.
- [39] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, 1996.
- [40] C. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology-EDBT' 90*, volume 416 of *Lecture Notes in Computer Science*, pages 407–421, 1999.

7 Appendix – Proofs

Proof of Proposition 1:

Proposition: An execution sequence $\Psi = \langle S_0, s_0 \rangle, \langle S_1, s_1 \rangle, \dots$ in which $s_{i+1} \models \text{effects}_i(\Psi)$ for all $i \geq 0$, is effect preserving.

Proof: This property derives from the necessary property of effects of assignments A : For every state s , if $\langle A, s \rangle \Rightarrow s'$ then $s' \models \text{effects}(A)$. By the definition of the effects of configurations in an execution sequence Ψ , if the statement that is executed in the i -th transition is not an assignment, then $\text{effects}_{i+1}(\Psi) = \text{effects}_i(\Psi)$, and if it is an assignment A , then $\text{effects}_{i+1}(\Psi) = \text{effects}_i(\Psi) \cup \text{effects}(A)$. Therefore, in the first case we have $s_{i+1} \models \text{effects}_i(\Psi) = \text{effects}_{i+1}(\Psi)$ and in the second case we have $s_{i+1} \models \text{effects}_i(\Psi)$ and $s_{i+1} \models \text{effects}(A)$ which implies $s_{i+1} \models \text{effects}_i(\Psi) \cup \text{effects}(A) = \text{effects}_{i+1}(\Psi)$. Since for $i = 0$, $s_0 \models \{\text{true}\} = \text{effects}_0(\Psi)$, we have the effect preservation property for all configurations in the sequence. \square

Proof of Claim 1:

Claim: For every execution sequence Ψ , $CA(\Psi) \leq_{\text{min_EP}} \Psi$.

Proof: We have to show that $CA(\Psi)$ is EP and a minimal restriction of Ψ .

1. $CA(\Psi) \leq \Psi$: If $CA(\Psi)$ is failing then it is a restriction of Ψ . Otherwise, the sequence of states in $CA(\Psi)$ is the same as in Ψ (apart from possible intermediate repetitions). Therefore, if $CA(\Psi)$ is infinite, then also Ψ is infinite. If $CA(\Psi)$ is finite and successful then $\text{end}(CA(\Psi)) = \text{end}(\Psi)$.
2. $CA(\Psi)$ is EP: We show by induction on the sequence of states in $CA(\Psi)$ s_0, s_1, \dots that if the sequence is not failing, then for every state s_i , $s_i \models \text{effects}_i(CA(\Psi))$. First, we note that not only the states in $CA(\Psi)$ are states that occur in Ψ and in the same ordering, but also the effects associated with configurations in $CA(\Psi)$ are the same since $CA(\Psi)$ has no additional assignments. Therefore, for every $CA(\Psi)$ configuration there is a corresponding earlier Ψ configuration with the same effects.

Basis: $s_0 \models \{\text{true}\} = \text{effects}_0(CA(\Psi))$.

Inductive step: Assume that the claim holds for all states s_i , for $0 \leq i \leq k$, for some $k \geq 0$. Consider the transition $\langle W, s_k \rangle \Rightarrow \langle W', s_{k+1} \rangle$ in the sequence.

- (a) If $\langle W, s_k \rangle$ is not an assignment configuration, then $s_k = s_{k+1}$ and $\text{effects}_k(CA(\Psi)) = \text{effects}_{k+1}(CA(\Psi))$, and by the inductive hypothesis: $s_{k+1} \models \text{effects}_{k+1}(CA(\Psi))$.
- (b) If $\langle W, s_k \rangle$ is an assignment configuration, then

$$W = A; \text{ if } \neg \text{effects}_i(\Psi) \text{ then fail else skip}; V,$$

where the i configuration in Ψ corresponds to the k configuration in $CA(\Psi)$. That is, $\text{effects}_i(\Psi) = \text{effects}_k(CA(\Psi))$. The following configurations in the $CA(\Psi)$ sequence are $\langle \text{if } \neg \text{effects}_i(\Psi) \text{ then fail else skip}; V, s_{k+1} \rangle \Rightarrow$ either $\langle \text{fail}, s_{k+1} \rangle$ if $s_{k+1} \not\models \text{effects}_i(\Psi)$, or $\langle \text{skip}; V, s_{k+1} \rangle$ if $s_{k+1} \models \text{effects}_i(\Psi)$.

In the first case the sequence is failing, and hence EP. In the second case, $s_{k+1} \models \text{effects}_i(\Psi) = \text{effects}_k(CA(\Psi))$ and by Proposition 1, $s_{k+1} \models \text{effects}_{k+1}(CA(\Psi))$.

Therefore, in either case, $CA(\Psi)$ is EP.

3. $CA(\Psi) \leq_{\text{min_EP}} \Psi$: It can be shown, by induction on the sequence configurations, that if Ψ is an EP execution sequence then $\Psi \leq_{EP} CA(\Psi)$. \square

Proof of Claim 2:

Claim: For every execution sequence Ψ , $CA_\delta(\Psi) \leq_{min_EP} \Psi$.

Proof: The proof is similar to that of Proposition 1. It is based on the correspondence between configurations of $CA_\delta(\Psi)$ to those of Ψ : Ψ configurations that are not changed correspond to themselves, and assignment configurations in Ψ correspond to the pair of configurations in $CA_\delta(\Psi)$ that replaces them.

1. $CA_\delta(\Psi) \leq \Psi$ since it is either failing or follows the same configurations (with some additional intermediate ones).
2. In order to show that $CA_\delta(\Psi)$ is EP we show (by induction on the sequence of states in $CA_\delta(\Psi)$ s_0, s_1, \dots) that if the sequence is not failing, then for every state s_k , $s_k \models effects_{s_k}(CA_\delta(\Psi)) = effects_{s_i}(\Psi)$ (where the k -th configuration in $CA_\delta(\Psi)$ corresponds to the i -th configuration in Ψ).

Basis: $s_0 \models \{true\} = effects_{s_0}(CA_\delta(\Psi)) = effects_{s_0}(\Psi)$.

Inductive step: Assume that the claim holds for the first k configurations of $CA_\delta(\Psi)$, for some $k \geq 0$. Consider the transition $\langle W, s_k \rangle \Rightarrow \langle W', s_{k+1} \rangle$ in $CA_\delta(\Psi)$.

- (a) If $\langle W, s_k \rangle$ is an original Ψ configuration, it is not an assignment configuration. Therefore $s_k = s_{k+1}$ and $effects_{s_k}(CA(\Psi)) = effects_{s_{k+1}}(CA(\Psi))$, and by the inductive hypothesis $s_{k+1} \models effects_{s_{k+1}}(CA(\Psi))$.

- (b) If $\langle W, s_k \rangle$ is the first of a new pair of configurations that replaces the i -th assignment configuration in Ψ , then it is of the form

$\langle if \ \delta_i(\Psi) \ then \ A \ else \ fail; V, s_k \rangle$,

where the following $CA_\delta(\Psi)$ configurations are

either $\langle A; V, s_{k+1} \rangle \Rightarrow \langle V, s_{k+2} \rangle$ (where $s_{k+1} = s_k$)

or $\langle fail, s_k \rangle$ (where $s_{k+1} = s_k$).

In the latter case the overall $CA_\delta(\Psi)$ sequence is failing, and hence EP. In the first case, $s_{k+1} = s_k \models \delta_i(\Psi) = \delta(effects_i(\Psi), A)$. Since by the inductive hypothesis $s_{k+1} = s_k \models effects_i(\Psi)$, we have by the definition of delta-conditions: $s_{k+2} \models effects_i(\Psi)$. By the inductive hypothesis we also have $effects_i(\Psi) = effects_k(CA_\delta(\Psi)) = effects_{k+1}(CA_\delta(\Psi))$, since the k -th configuration is an *if* configuration. Altogether, from

$s_{k+2} \models effects_{k+1}(CA_\delta(\Psi))$, and by Proposition 1, we get

$s_{k+2} \models effects_{k+2}(CA_\delta(\Psi))$.

The second equality in the hypothesis is obtained directly from the definition of effects in execution sequence:

$$\begin{aligned} effects_{k+2}(CA_\delta(\Psi)) &= effects_{k+1}(CA_\delta(\Psi)) \cup effects(A) = \\ &effects_i(\Psi) \cup effects(A) = effects_{i+1}(\Psi). \end{aligned}$$

3. If Ψ is an EP execution sequence then $\Psi \leq_{EP} CA_\delta(\Psi)$. Can be shown by induction on the sequence configurations. □

Proof of Lemma 1:

Lemma: Let U be a *While* update without loops, and s a state. The execution sequence $seq(U, s)$ corresponds to a full path in the computation tree of U , $CT(U)$, such that:

1. $seq(U, s)_0$ corresponds to $root(CT(U))$, and if $seq(U, s)_i$ corresponds to node n_i in the tree, then $seq(U, s)_{i+1}$ corresponds to a child node n_{i+1} of n_i . If $seq(U, s)$ is a successful sequence, the last terminal configuration does not correspond to any node, and its previous configuration corresponds to a leaf node. If the sequence is failing, its last configuration corresponds to a

leaf node.

This correspondence defines a 1 : 1 mapping between the configurations in $seq(U, s)$ (excluding the terminal configuration, if exists) and the nodes of the path.

2. If $seq(U, s)_i$ corresponds to node n_i , then $effects_i(seq(U, s)) = effects(n_i)$ and $\delta_i(seq(U, s)) = \delta(n_i)$.

Proof: The proof is by induction on the structure of U (nesting level of its operators).

1. If U is primitive, then $seq(U, s)$ is either $\langle fail, s \rangle$ or $\langle U, s \rangle \Rightarrow s'$, and $CT(U)$ is a single node. So, the correspondence is established.
2. If U is an **if** statement **if** P **then** S_1 **else** S_2 , then $seq(U, s) = \gamma_0, \dots, \gamma_n$, where $\gamma_0 = \langle \text{if } P \text{ then } S_1 \text{ else } S_2, s \rangle$ and $\gamma_1, \dots, \gamma_n$ is either $seq(S_1, s)$ or $seq(S_2, s)$. The root node of $CT(U)$ has two subtrees for $CT(S_1)$ and $CT(S_2)$. The inductive hypothesis holds for S_1 and S_2 . Therefore, the tree path that corresponds to $seq(U, s)$ consists of $root(CT(U))$ and the path that corresponds to either S_1 or S_2 , according to $seq(U, s)$.
3. If U is a sequence statement $S_1; S_2$, then $seq(U, s)$ is the concatenation of two sequences for S_1 and S_2 , respectively. The inductive hypothesis holds for S_1 and S_2 . $CT(U)$ is $CT(S_1)$ where all non *fail* leaves have left subtrees for $CT(S_2)$. Therefore, the tree path that corresponds to $seq(U, s)$ consists of the path that corresponds to $seq(S_1, s)$ in $CT(S_1)$, concatenated to the path that corresponds to the S_2 sequence in $CT(S_2)$.

The second part of the Lemma holds since the definitions of effects and of delta-conditions for execution sequences are exactly the tree annotations.

□

Proof of Lemma 2:

Lemma: Let U be a *While* update without loops, and s a state. Then,

$$removeEC(seq(reviseUpdate_1(U), s)) = CA_\delta(seq(U, s)).$$

Proof: The proof is obtained from the following three immediate claims:

1. There is an order preserving correspondence (mapping) between the configurations of $seq(U, s)$ and the those of $CA_\delta(seq(U, s))$, such that:
 - (a) If $seq(U, s)_i$ is an assignment to a state variable configuration, then it corresponds to two successive configurations in $CA_\delta(seq(U, s))$ – an **if** configuration and either the assignment or a *fail* configuration –, following the definition of the CA_δ transformation.
 - (b) All other configurations correspond to themselves.
2. There is an order preserving correspondence (mapping) between the configurations of $seq(U, s)$ and the those of $removeEC(seq(reviseUpdate_1(U), s))$, such that:
 - (a) If $seq(U, s)_i$ is an assignment to a state variable configuration, then it corresponds to two successive configurations in $removeEC(seq(reviseUpdate_1(U), s))$ – an **if** configuration and either the assignment or a *fail* configuration –, following the definition of the $reviseUpdate_1$ transformation.
 - (b) All other configurations correspond to themselves.
3. When the sequence $removeEC(seq(reviseUpdate_1(U), s))$ reaches an **if** configuration that corresponds to an assignment to a state variable configuration $seq(U, s)_i$, the $node()$ iterator procedure points to the tree node n_i that corresponds to $seq(U, s)_i$.

Based on the first two claims, the two execution sequences $removeEC(seq(reviseUpdate_1(U), s))$ and $CA_\delta(seq(U, s))$ differ only in the conditions in the added `if` statements. In $CA_\delta(seq(U, s))$ the condition is $\delta_i(seq(U, s))$ – for the corresponding $seq(U, s)_i$ configuration, while in $removeEC(seq(reviseUpdate_1(U), s))$ the condition is $\delta(n_i)$ – for the tree node n_i pointed by the iterator. However, based on the third claim, the conditions are the same since $effects_i(seq(U, s)) = effects(n_i)$, by Lemma 1. Therefore, $removeEC(seq(reviseUpdate_1(U), s)) = CA_\delta(seq(U, s))$. \square

Proof of Lemma 3:

Lemma: Let U be a *While* update, and s a state. Every finite prefix of $seq(U, s)$ corresponds to a path from the root in the computation graph of U , $CG(U)$, such that:

1. $seq(U, s)_0$ corresponds to $root(CG(U))$, and if $seq(U, s)_i$ corresponds to node n_i in the tree, then $seq(U, s)_{i+1}$ corresponds to a child node n_{i+1} of n_i . If $seq(U, s)$ is a finite successful sequence, the last terminal configuration does not correspond to any node, and its previous configuration corresponds to a leaf node. If the sequence is finite failing, its last configuration corresponds to a leaf node.
This correspondence defines a mapping from configurations in the finite prefix of $seq(U, s)$ (excluding the terminal configuration, if exists) to the nodes on the path.
2. If $seq(U, s)_i$ corresponds to node n_i , then $effects_i(seq(U, s)) = effects(n_i)$ and $\delta_i(seq(U, s)) \subseteq \delta(n_i) \cup \delta Possible(n_i)$ ⁶.

Proof: We extend the proof of Lemma 1 by adding the additional entry for a *while* statement:

4. If U is a *while* statement `while P do S` , then a finite prefix of $seq(U, s)$ is a concatenation of repeating sequences for S :
 $\langle \text{while } P \text{ do } S, s \rangle \Rightarrow \langle \text{if } P \text{ then } (S; \text{while } P \text{ do } S) \text{ else skip}, s \rangle \cdot seq(S, s)$,
 where \cdot stands for sequence concatenation. The root node of $CG(U)$ (a *while* labeled node) has a left child for the `if` statement, which has a left subtree for $CG(S)$. The inductive hypothesis holds for S . Therefore, the graph path that corresponds to a single round in the loop consists of $root(CG(U))$, its left child, and the path that corresponds to S in $CG(S)$. For the next round: The non failure leaves of $CG(S)$ have $root(seq(U, s))$ as their left child. Therefore, a path that corresponds to a finite prefix of $seq(U, s)$ consists of cyclic repetition on the graph path for a single loop round.

In the second part of the Lemma, the equality of effects holds since their definition for execution sequences is exactly the graph annotations. The delta-conditions and delta-Possible annotations of a graph node is a superset of the delta-conditions of the corresponding configuration since delta-Possible includes delta-conditions with respect to all possible assignments that might precede its statement when executed. Therefore, its set of delta-conditions is a superset of the actual delta-conditions of its corresponding configuration in an execution sequence. \square

Proof of Lemma 4:

Lemma: Let U be a *While* update, and s a state. Then,

$$removeEC(seq(reviseUpdate_2(U), s)) \equiv_{EP} CA_\delta(seq(U, s)).$$

⁶The notation is used imprecisely here, since $\delta Possible(n_i)$ is a set of pairs, and only the second element in each pair is a delta-condition.

Proof: The proof is the same as that of Lemma 2. The only difference is that the two sequences are not equal due to the difference in the delta-conditions in the added `if` statements. The conditions in the $removeEC(seq(reviseUpdate_2(U), s))$ sequence are taken from the graph-nodes – include the delta-conditions and delta-Possible annotations of a graph node, which form a superset of the delta-conditions of the corresponding configuration in $CA_\delta(seq(U, s))$. However, the tests of the conditions in both sequences give the same results since the extra delta-conditions in a configuration of $removeEC(seq(reviseUpdate_2(U), s))$ are evaluated on an empty set of variable values (they are applied on non-visited graph nodes, whose *values* collection is empty). Therefore, the two sequences, although not syntactically equal, have the same behavior with respect to failure, termination, and effect-preservation.

□