Machine Assisted Constraint Maintenance

Maarten Fokkinga

Version of October 3, 1996

Abstract Constraints in databases may be maintained while transactions are executed, by the automatic invocation of suitable *rules* containing actions that compensate any constraint violation within the transactions. There is ample opportunity for machine assistence in the construction of the rules. The correctness and termination of a collection of rules follows from *local* correctness and termination properties for the individual rules, which are left to the responsibility of the user.

This is a summary of the paper by Ceri and Widom [1], rephrased on a very abstract level and provided with alternative and simpler proofs.

Problem statement Regarding a database, we abstract from all aspects except that a database may change by the execution of certain transactions, built from certain operations. Thus we postulate the existence of a given set of operations:

$$O_1, O_2, \ldots$$

The only aspect of a *transaction* that we'll use is: the set of operations occurring in it. Further, we postulate several *constraints* that are intended to be true of the database throughout its lifetime:

$$C_1, C_2, \ldots$$

In general, an operation O_i might violate the constraints, that is, after the execution of the operation some of the constraints may not hold, even though they did hold before the execution.

The problem is to provide (automatic, or semi-automatic) means for overcoming the constraint violation due to the execution of a transaction.

A proto-type solution For each constraint C_i a *rule* $R_i = \neg C_i \rightarrow A_i$ is constructed; the *compensating action* A_i is provided by the user (and A_i is built from the operations and the so-called *rollback* operation), the "remainder" is provided automatically. (Here the remainder is trivial; in the sequel it will be less trivial.) The intention of A_i is to reestablish constraint C_i; *the user is responsible for this* (and there is ample opportunity for machine assistance here). Note that A_i may invalidate any other constraint C_j for $j \neq i$.

Now, the constraints $C_1, C_2, \ldots$ are maintained, in spite of execution of transactions, by succeeding each transaction T by repetition R:

$$R \;=\; \textbf{do}\ \neg C_1 \rightarrow A_1 \;[\!]\; \neg C_2 \rightarrow A_2 \;[\!]\; \ldots \;\;\textbf{od}$$

That is, repeatedly an arbitrary A_i with a true guard $\neg C_i$ is executed; if all guards are false then the repetition terminates. Clearly, there is danger of nontermination: each A_i may establish another rule's guard $\neg C_j$ (for $j \neq i$). But, *if* R terminates, *then* all the constraints $C_1, C_2, \ldots$ are true after R. We will return later to the question of termination.

So, the constraints are maintained by the composite $T;R$. (The execution of *rollback* within $T;R$ restores the initial state of $T;R$ and terminates the entire $T;R$.)

Improvement The solution proposed above is not satisfactory: The evaluation of the guards $\neg C_i$ in R may be quite costly — if possible at all. Let us therefore consider an additional guard P_i in each rule:

$$R_i \quad = \quad P_i \wedge \neg C_i \rightarrow A_i$$

Now, if P_i is false, there is no need to evaluate C_i. So the efficiency is improved, provided that the cost of evaluation P_i is less than that of C_i. Ceri and Widom write R_i as:

$\boxed{\texttt{when } P_i \texttt{ if } \neg C_i \texttt{ then } A_i}$

They do not provided a notation for R; for them it is just the collection of all rules.

Regarding the correctness, the new R establishes all the constraints upon termination, if the predicates satisfy this property:

(1) $\neg C_i \Rightarrow P_i$

(Indeed, the sufficiency of (1) is proved as follows:

$$\begin{array}{ll} & \text{postcondition of } R \Rightarrow C_i \\ \equiv & \\ & \neg C_i \Rightarrow \neg\,\text{postcondition of } R \\ \equiv & \\ & \neg C_i \Rightarrow (P_1 \wedge \neg C_1) \vee \ldots \\ (2) \;\Leftarrow & \\ & \neg C_i \Rightarrow P_i \wedge \neg C_i \\ \equiv & \\ & \neg C_i \Rightarrow P_i \end{array}$$

In general, that is treating C_i and P_i as *variables*, the weakening step (2) is even an equivalence, so that the sufficient condition '$\neg C_i \Rightarrow P_i$' is even a necessity.)

It remains to look for suitable predicates P_i.

Event predicates Ceri and Widom propose the following procedure for the automatic construction of the predicates P_i.

Let the additional guard P_i for the restoration of constraint C_i take the following form:

$$P_i \quad = \quad O!_{i1} \vee \ldots \vee O!_{in}$$

Here, roughly and informally said (a precise formulation follows below), each $O!_j$ is intended to indicate that operation O_j may have invalidated the constraint (so it needs restoring), or rather: if $O!_j$ is *false*, then *certainly* the constraint has not been invalidated by operation O_j (since the last time that the constraint was known to be true).

To make this precise, we proceed in two steps. First we extend the language with a new predicate symbol $O!_j$, for each operation O_j. These predicates are called *event predicates.* The semantics of event predicate $O!_j$ depends on the context in which it occurs: *within rule R_i of the repetition R that is attached to a transaction T* event predicate $O!_j$ denotes the following truth value:

(3) operation O_j has been executed since the last time that rule R_i has been executed within R, or, if R_i has not been executed at all, since the start of the transaction T.

It should not be problematic to adapt the execution system in such a way that event predicates $O!_j$ do indeed get this semantics.

Secondly, the event predicates $O!_{i1}, \ldots, O!_{in}$ for P_i are selected as follows:

(4) take *all* the event predicates *except* those whose operations are known not to invalidate constraint C_i.

So, collection $O!_{i1}, \ldots, O!_{in}$ can be found automatically; it is correct to take all event predicates $O!_1, O!_2, \ldots$ into P_i.

Clearly, the more is known about the operations and constraints, the less event predicates are taken into P_i, and the more efficient the evaluation of R's guards is. In a simple approach, all such knowledge is based on syntax alone; for example, if none of the variables in a constraint occurs in operation O, then O certainly doesn't invalidate the constraint. In a more intelligent approach, some semantic properties may be taken into account as well.

Remark. The P_i thus defined fulfills requirement (1): $\neg C_i \Rightarrow O!_{i1} \vee \ldots \vee O!_{in}$. Indeed, within repetition R attached to transaction T, we argue as follows:

First assume that all constraints are satisfied at the beginning of T (we are developing a method for constraint *maintenance*, aren't we!), and that each compensation action A_i (hence rule R_i) establishes C_i (the responsibility of the user!).

Now, falsity of C_i within the guards of R can only be caused by the execution of some operation that invalidates C_i. Of course, this must have happened after the last time that C_i was known to be true. So the operation, O_j say, has been executed since the last time that rule R_i was executed within R, or, if R_i has not been executed at all, since the start of the transaction T. By definition (3), that execution of the operation makes $O!_j$ true in rule i.

Finally, observe that j must be one of $i1, \ldots, in$, since by definition (4) of P_i, all other operations do not invalidate C_i.

Termination We are still faced with the problem of avoiding nontermination. In the end the user has to solve this, but the presence of the event predicates provide ample opportunity for machine assistance.

In particular, execution of rule R_i cannot make event predicate $O!$ in rule R_j true, if operation O does not occur in R_i. We say R_i *triggers* R_j if at least one of R_i's operations occurs, as an event predicate, in R_j's guard. (This does *not* mean that an execution of R_i is neccesarily followed by an execution of R_j, or that R_j is executed at all.) The *trigger cycles* can be detected automatically, and be shown to the user. Below we prove that if no such cycle can go on indefinitely, then the entire repetition R terminates. Termination of the individual cycles is the user's responsibility.

> *Proof.* Ceri and Widom [1, Section 7.1] give a combinatorial proof. Quite differently, we shall show the existence of a so-called variant function. A variant function is a function from the set of states into the natural numbers, whose value is decreased after each step in the repetition. Since the value is a natural number, there cannot be an infinite number of decrements, hence the repetition terminates.
> We construct the variant function out of two ingredients.
>
> First, let c be an arbitrary cycle of rules, each one triggering the following in the cycle. The assumption that cycle c cannot go on indefinitely, means that there exists a variant function f_c such that execution of each rule in cycle c decreases f_c, while each other rule does not increase it.
>
> Second, let x be a brand new variable, and put the assigment $x := n$ in front of R and, for each i, the assignment $x := n_i$ at the end of rule i; here n is the number of rules, and n_i is the number of rules accessible from rule i via the trigger relation. Clearly, an execution of a rule in a trigger cycle doesn't change the value of x; each other rule decreases the value of x. Moreover, this program transformation doesn't change the semantics of R; variable x is a ghost variable, to be used only in the proof.
>
> Now the variant function f is: $x + \sum_c f_c$, where the summation ranges over all cycles. Indeed, each rule in a cycle, c say, decreases f_c and does not increase any other summand of f. And, each rule not contained in any cycle decreases the value of x while not increasing any other summand of f. So, the value of f is decreased in each step of the repetition.

Concluding remarks At various points there were opportunities to build in more, or less, intelligence: some constructions may involve only syntax or may take semantics into account. Thus the constraint maintenance scheme discussed here can be realised in a variety of ways, one more efficient or with more machine assistance than the other.

As one example of more intelligence, consider the notion of transactions as we have defined it: "The only aspect of transactions that we use is: the set of operations occurring in it." Clearly, if two operations in a transaction are executed in sequence, and the second one undoes the effect of the first one, we may consider both operations not to occur in the transaction. This may happen often in practice, with such operations like: *insert* a tuple and *delete* the tuple.

There is one thing that we haven't mentioned so far: rules may prioritized, giving one rule priority over another. Prioritization can be modelled by extra guards in the rules: if R_i has priority over R_j, then take the negation of R_i's guard as extra condition in R_j's guard. For example consider a repetition of two rules:

$$\textbf{do } A \to S \parallel B \to T \textbf{ od}$$

Priority of rule $A \to S$ over $B \to T$ can be modelled by:

$$\textbf{do } A \to S \parallel \neg A \wedge B \to T \textbf{ od}$$

Semantically, but not operationally, the two repetitions are equivalent. So, prioritization does not affect the correctness and termination arguments given above, but may help the user in the construction of the compensation actions and the termination proof.

References

[1] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th VLDB Conference*, pages 566–577, 1990.