Analysing the Impact of Adding Integrity Constraints to Information Systems

Suzanne M. Embury[1] and Jianhua Shao[2]

[1] Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, England, United Kingdom
SEmbury@cs.man.ac.uk

[2] Department of Computer Science, Cardiff University, P.O. Box 916, Cardiff, CF24 3XF, Wales, United Kingdom
J.Shao@cs.cf.ac.uk

Abstract. The ability of a business to change its working practices, in order to gain or retain competitive edge, is closely aligned to its ability to change the business rules implemented by its information systems. Unfortunately, adding a new business rule to an existing system is both time-consuming and error-prone. It is all too easy, for example, for the programmer to overlook some program elements that are affected by the addition of the new rule, with the result that it is not enforced uniformly by the system as a whole. When this happens, the information system can begin to behave in confusing and anomalous ways.

In this paper, we describe an impact analysis technique that aims to support the programmer in the difficult task of implementing an important class of business rules, namely, integrity constraints. We have adapted techniques from database integrity maintenance to allow us to identify whether a program is likely to be affected by the addition of a new constraint, and to pinpoint the specific program statements that must be guarded against the possibility of constraint violation. Our technique can also be used to provide guidance to the programmer as to the conditions that must be included in any new guards.

1 Introduction

The ability of a business to gain and retain competitive edge is closely aligned to its ability to change the *business rules* which govern its day-to-day behaviour. A business rule is a statement that "defines or constrains some aspect of a business" [10]. For example, one such rule might define the criteria that make a customer eligible for a certain kind of discount; another might state the circumstances under which credit can be given. Such rules are typically very volatile. New business rules are regularly imposed on organisations by changes to statutory regulations, government policy and economic/market conditions. However, in addition to these enforced changes, many businesses also choose to modify their business rules frequently as they fight to maximise revenues, minimise churn and capture new market share.

One of the key costs for any organisation wishing to make changes to its business rules arises from the need to evolve the supporting information systems so that they too enforce the new rules. Adding new business rules to a software system (or modifying existing ones) is typically both time-consuming and error-prone. The reason for this is that there is no obvious correspondence between a business rule and the collection of software artefacts that implement it. Real world information systems typically consist of a great many programs, each of which has a slightly different effect on the system state and each of which may potentially have a role in enforcing a new business rule. Deciding which programs (or which parts of a program) might violate the new rule, and exactly what changes have to be made to them to prevent the violations, is an extremely challenging task.

Current software maintenance tools offer little help with this task, and it is usually necessary to manually inspect a significant proportion of the program code. Not only is this costly in terms of programmer time, but it is also highly error prone. It is extremely easy to overlook some programs that really ought to be modified or to add redundant checks to programs that cannot in fact cause a violation of the rule. And yet much of this inspection work is highly repetitive and mundane, and is therefore amenable to automation.

In this paper, we present a technique that partially automates the process of adding a particular kind of business rule to an existing information system. The business rules that we focus on are those that act as database integrity constraints [14]. Of course, evolution is straightforward for those integrity constraints that are simple enough to be implemented centrally by the database management system (DBMS). However, it is still the case that many of the more complex forms of business rule must be implemented in the application programs, either because of limitations in the capabilities of the DBMS or because of the need for better performance than can be achieved using a centralised approach. Tool support for such modifications is therefore still required, even though DBMS facilities have improved significantly over the last decade.

In the remainder of this paper, we describe our technique for determining the impact of adding a new integrity constraint to a software system. We begin (Section 2) by outlining the steps involved in implementing a new integrity constraint, and consider how far existing software tools can support each of these steps. We then go on to show how techniques from the field of database integrity maintenance (Section 3) can be adapted to provide key elements of the information necessary for performing a detailed impact analysis for this form of software maintenance (Section 4). Finally, we illustrate the behaviour of our technique on some examples (Section 5) and conclude (Section 6).

2 Implementing New Integrity Constraints

Constraint-style business rules begin life as high-level conditions that the organisational state must adhere to [10]. These conditions can typically be violated by several different actions. Consider, for example, the following constraint:

All customers with a negative balance must have a valid authorisation for their overdraft.

This rule can potentially be broken by several different operations: lowering the balance of a customer's account, or cancelling an overdraft authorisation for some customer, for example. The supporting software systems must ensure that, whenever one of these "dangerous" actions occurs, the constraint is not violated by the action. The exact processing steps required to detect a specific type of violation will, in general, depend on the action which causes it. For example, if the customer's balance is lowered, then we need to check whether it has fallen below zero and if so whether an overdraft authorisation exists. If, on the other hand, a program results in the cancellation of an overdraft request, it is only necessary to check whether that customer's balance is above zero at the time of the cancellation.

To the best of our knowledge, there is no widely accepted methodology for the implementation of business rules[1]. However, by examining the requirements for the accurate and complete implementation of constraint-type rules, we can infer something about what is involved in this evolution task and derive from it the potential for automation.

The first requirement is to understand how the new constraint may be invalidated by various forms of state change. Since we are focussing on business rules that are equivalent to constraints over a database state, each such rule can be expressed as a first order logic expression over predicates corresponding to the schema elements in the database (and the standard inequality predicates). For example, the informal constraint given earlier can be expressed in first order logic (FOL) as follows:

$$(\forall x, y, z)\ \mathit{customer}(x, y, z) \land z < 0 \Rightarrow \mathit{overdraftAuth}(x)$$

For any finite FOL expression, there is a finite set of atomic updates that can violate it. For example, the insertion of a new record into the *customer* table might invalidate our example constraint, but deletion of a *customer* record can never have this effect. The full set of violating updates can be discovered through analysis of the form of the constraint (as we shall describe in Section 3).

Once we know which updates can trigger a constraint violation, the next requirement is to examine each of the programs within the system, to see which of them have the capacity to effect one of these dangerous state changes. Each such program must be modified in order to prevent (or flag) violations of the rule. The most common strategy is to insert a pre- or post-condition into the program, as a guard on the offending update or any associated transaction commit operations.

[1] The nearest approximation to such a methodology is perhaps the externalisation of business rules using rule engine technology, such as that provided by ILOG JRules [11]. However, rule engines are not a universal solution to the problem of business rule implementation and they cannot easily be grafted onto an existing information system. In this paper, therefore, we concentrate on the problems faced by the maintainers of systems which have been implemented in a more traditional manner, and where the rules are internalised within the code of the application programs.

From these basic requirements, we can derive the following skeleton process for modifying an information system so that it enforces a new constraint:

1. Analyse the new constraint and determine from it which state change operations can potentially cause it to be violated.
2. Locate the set of program statements which perform the state change operations identified in step 1. From this, we can also identify the set of programs which are affected by the new constraint.
3. For each potentially dangerous program statement identified in step 2, determine whether it really does have the capability to violate the constraint (i.e. check that it does not occur under conditions which would make a violation of the rule impossible).
4. For each program statement remaining after step 3, determine how we will detect whether it violates the constraint and how such violations will be prevented (i.e. choose a pre- or post-condition approach).
5. For each program statement remaining after step 3, determine the condition that must be checked by the program in order to prevent violations in the manner selected in step 4.

Each of these steps presents a potential challenge to the maintenance programmer, either because of the complexity of the reasoning task involved or because of the sheer volume of cases that must be considered. Despite this, few software tools exist which can support the programmer in this task. Step 1, for example, must be carried out manually at present, although (as we shall later demonstrate) techniques do exist for assisting in this task.

Step 2 corresponds closely to the traditional notion of impact analysis, in that it essentially involves the identification of the set of software components (in this case, programs) that are affected by the proposed change. However, most current impact analysis tools [1] discover impacts by tracking *dependencies* of various kinds between software artefacts. While certain forms of dependencies (e.g. data dependencies) might be of use in determining which programs are impacted by the addition of a new constraint, they do not provide us with a complete solution to the problem. In fact, the kind of functionality required to support step 2 is provided to a large extent by modern data dictionary systems. For example, Predict (a data dictionary for use with the ADABAS/NATURAL development environment [17]) maintains links to program statements which contain database update commands, and thus allows easy identification of the (super)set of programs which can potentially violate a given constraint.

For steps 3 to 5, we require a tool which is able to reason over both the semantics of the programming language *and* the semantics of the data manipulation commands. Presently, very few impact analysis techniques incorporate knowledge of database semantics. There are a few commercial tools (e.g. CAST Envision [16] and Quest's SQL Impact [13]) and some limited research proposals [4,15] which perform impact analysis for database schema evolution. However, their scope is limited to changes to table and attribute structure, and they cannot identify the impacts of more complex forms of schema evolution (such as changes to integrity constraints).

3 Identifying Potentially Violating Updates

The key to automating the process of constraint implementation lies in the first of the steps mentioned in the previous section. If we can discover which updates can potentially violate the new constraint, then we can use this information, alongside conventional source code analysis techniques, to provide support for all of the remaining steps (i.e. steps 2 to 5).

As it happens, this is a problem that has been well-studied for many decades, and a variety of solutions have been proposed in the literature [12,3,9]. In general, these solutions are derived from the notion that, given any arbitrary database update, we can classify its effect on the truth value of a condition over the database state as belonging to one of the following three categories [5]:

– The update has no effect on the validity of the constraint, regardless of the circumstances under which it occurs. In this case, the update is said to be *trivially satisfying.*
– The update will always result in a violation of the constraint, regardless of the circumstances under which it occurs. In this case, the update is said to be *trivially violating.*
– The update will cause a violation of the constraint in some circumstances but not in others. Updates of this kind are said to be *potentially violating.*

As an illustration of these three categories, consider the following simple constraint:

$$ic_1 \;\equiv\; (\forall c, a, b)\ cust(c, a, b) \wedge managed(c) \Rightarrow gold(c)$$

Informally, this rule expresses the company policy that, for the purposes of the company's reward programme, all managed customers are considered automatically to have "gold" status. We will now present some example updates, and classify them according to their effect on this constraint. We will use the notation $+tablename(x_1, \ldots, x_n)$ to indicate the insertion of a new tuple $< x_1, \ldots, x_n >$ into the table called *tablename*, and the notation $-tablename(x_1, \ldots, x_n)$ to indicate the deletion of the tuple $< x_1, \ldots, x_n >$ from *tablename.* Modifications to existing tuples are expressed as combination of a deletion and an insertion.

We illustrate the different categories of update using the following examples:

– Since our example constraint is range-restricted [12], any update to a schema element which does not appear in the constraint expression can be classified as being trivially satisfying. For instance, the update

$$-account(x, y, z)$$

is trivially satisfying relative to our example constraint for any value of x, y and z.
– Some updates involving schema elements which *do* appear in a constraint may also be trivially satisfying, as in the case of the following update:

$$-cust(x, y, z)$$

Deleting a customer from the database can never violate constraint ic_1. Such an update can only have the effect of falsifying the left hand side of the implication for a given set of values, and thus the implication must remain true for those values.

– There is only one update that is trivially violating for constraint ic_1. It is the composite update:

$$+cust(x, y, z),\ +managed(x),\ -gold(x)$$

Since all these updates refer to the same customer identifier (x), we know that a violation must always result if all three are executed successfully.

– Finally, we present an example of an update that is potentially violating for our example constraint:

$$-gold(x)$$

In this case, we cannot tell from the form of the update whether it will cause a violation or not. In order to make this decision, we need to interrogate the database to discover more about the context of the update. For instance, in this case, we need to know whether the customer whose gold status is being deleted is a managed customer or not. If he or she is, then this update *will* cause a violation, but if the customer is a normal domestic customer then we can freely delete his or her gold status without breaching this particular business rule.

When we add a new integrity constraint to an information system, we need to identify the set of all updates which are trivially or potentially violating for that constraint[2]. Any program capable of effecting a state change that includes such an update is potentially impacted by the addition of the new constraint.

To identify this set of updates, we can make use of techniques from the field of database integrity checking [12,2,5,9]. Various approaches have been proposed, each of which operates on a slightly different class of constraints. Here, we will describe the method we have implemented in our prototype impact analysis tool, which is adapted from a technique proposed for use in identifying database repairs [8]. Due to space restrictions, we will present a cut-down version of the method that operates over universally quantified expressions only. The reader is referred to the literature on integrity maintenance for details of techniques which can operate on constraints involving existentially quantified variables [9].

The method we describe here uses a truth table representation of the constraint to derive the different kinds of update that can cause it to become false. For example, consider the following truth table for constraint ic_1:

[2] For brevity, in the remainder of this paper, we will use the term *potentially violating updates* to indicate the set of both trivially *and* potentially violating updates.

	$cust(c, a, b)$	$managed(c)$	$gold(c)$	ic_1
1	F	F	F	T
2	F	F	T	T
3	F	T	F	T
4	F	T	T	T
5	T	F	F	T
6	T	F	T	T
7	T	T	F	F
8	T	T	T	T

The truth table shows us that a violation of ic_1 occurs iff a substitution θ_1 of values for the variables a, b and c exists, such that:

$$cust(c/\theta_1, a/\theta_1, b/\theta_1) \wedge managed(c/\theta_1) \wedge \neg gold(c/\theta_1)$$

is satisfied by the database state[3]. By examining the differences between pairs of true and false rows in the truth table, we can discover what state changes could cause such a substitution to be brought into existence. For example, consider the differences between rows 1 and 7 of the table. Suppose we have a database state *ds* which satisfies the row 1 expression, i.e. some substitution θ_2 exists for which the expression:

$$\neg cust(c/\theta_2, a/\theta_2, b/\theta_2) \wedge \neg managed(c/\theta_2) \wedge \neg gold(c/\theta_2)$$

is true in *ds*. The columns which have a different value in the two rows being compared tell us what changes we need to make to *ds* in order to cause it to satisfy the false row expression for θ_2 — that is, to cause a violation to be introduced. These changes are:

$$+cust(c/\theta_2, a/\theta_2, b/\theta_2), +managed(c/\theta_2)$$

The columns where the value is the same in both rows indicate the necessary and sufficient condition for determining whether a violation will definitely occur as a result of these updates. In the case of our example, the condition to be tested is: $gold(c/\theta_2)$. In other words, if we are adding a new managed customer to the database, we need to check whether that customer has gold status or not, in order to determine whether a violation with result from the update.

By considering the differences between all possible pairs of true and false rows in this way, we can extract the complete set of potentially violating updates (and their associated conditions) for this rule:

$1 \rightarrow 7$	*+cust(c, a, b), +managed(c)*	*if* $\neg$*gold(c)*
$2 \rightarrow 7$	*+cust(c, a, b), +managed(c), -gold(c)*	*if true*
$3 \rightarrow 7$	*+cust(c, a, b)*	*if managed(c)* $\wedge \neg$*gold(c)*
$4 \rightarrow 7$	*+cust(c, a, b), -gold(c)*	*if managed(c)*
$5 \rightarrow 7$	*+managed(c)*	*if* $(\exists a, b)$ *cust(c, a, b)* $\wedge \neg$*gold(c)*
$6 \rightarrow 7$	*+managed(c), -gold(c)*	*if cust(c, a, b)*
$8 \rightarrow 7$	*-gold(c)*	*if cust(c, a, b)* $\wedge$ *managed(c)*

[3] The proof of this is given elsewhere [8]

Any program fragment which has the potential to bring about one of these state changes can also potentially violate the constraint. For example, the state change described by the transition between row 5 and row 7 can be effected by insertion of a new *cust* tuple, or by updating an existing one.

By analysing the form of the constraint that we wish to add to the system, therefore, we can identify the set of "symptoms" that indicate where a change to source code might be required. In the following section, we will describe how we can make use of this information in analysing the impact of adding new integrity constraints to existing software systems.

4 Impact Analysis for the Addition of Constraints

The ability to determine the complete set of updates that can violate a constraint opens up a number of possibilities for developing techniques to support programmers in maintaining and evolving integrity constraints. For example, if we consider the steps involved in implementing a new constraint (as outlined in Section 2), the following forms of support can be envisaged:

1. If we know which state changes can potentially cause a violation of the new constraint, we can use this information to identify those programs which have the potential to bring such state changes about (i.e. to support steps 2 and 3). This corresponds to the set of programs which need to be modified in order to implement the new constraint. In theory, it should be possible to identify this set exactly; in practice, a more efficient technique which identifies a close superset may be just as useful. It is important, however, that any more efficient but less accurate technique returns a true superset of the actual results. We can live with the programmer having to examine a handful of programs unnecessarily, but we want to be confident that no potentially violating programs have been omitted.
2. Once we have determined that a particular program requires modification, it should be possible for a software tool to indicate to the programmer exactly which of the database update statements need to be guarded against violation of the constraint (i.e. to support step 4). Again, in practice, we can be satisfied with a technique which occasionally points out update statements that cannot violate the constraint, provided that it does not omit any statements that can.
3. Since we can derive the necessary and sufficient condition for violation along with each potentially violating update, we can also provide some guidance to the programmer as to the form of the guard condition that is required in each case (i.e. to support step 5). Of course, the actual changes to the code must be chosen by the programmer, but the advice provided by the software tool could at least help to ensure that awkward and complex cases are not overlooked or interpreted incorrectly.

Each of the above cases involves the use of information derived by the analysis of a constraint in determining the impact of adding that constraint to a given set of programs. Such impacts can be identified at a variety of granularities, ranging from identification of impacted programs (point 1 in the above list), through identification of impacted program statements (point 2), to guidance as to the form of the new code that is required (point 3).

Given the capability to determine impacted program statements, it might be thought that the ability to determine impacted programs is unnecessary. After all, the set of impacted programs should be exactly the set of programs which contain impacted statements. However, since analysis of a program to discover the set of impacted statements is a potentially expensive task, there is still some value in a "quick and dirty" filtering step, which can remove programs which are not impacted from further consideration. Only programs which pass through this filtering step are subjected to the more comprehensive impact analysis step.

Before we describe the operation of these two steps in more detail, we first present some definitions and assumptions. We use the name ic_n to refer to the integrity constraint that is to be added to the system, while Ps denotes the set of programs that are to be analysed for the impacts of this change. We assume the existence of a function called *gcu* which maps constraints onto the set of guarded complex updates which can potentially violate them. A *guarded complex update* is a pair $< Us, C >$, where Us is a set of database updates and C is the necessary and sufficient condition (i.e. the guard) that determines whether the updates in Us result in a rule violation or not. For example, at the end of Section 3, we listed the seven GCUs that are potentially violating for constraint ic_1.

Each member of Us is a triple $< UT, TN, V >$, where UT is the update type ($UT \in \{`i', `d', `m'\}$, representing the standard "insert", "delete" and "modify" update types), TN is the name of the table that is updated and V is a tuple of variables representing the updated tuple in the named table.

We further assume the existence of a function called *dbs*, which maps a program p ($p \in Ps$) onto a set of triples $< UT, TN, S >$, where UT is the update type (as defined earlier), TN is the name of the table that is updated, and S is the identifier of a statement in the program that corresponds to an update of the given type to the given table. This function provides us with a convenient means of locating program statements that correspond to particular database update commands.

4.1 Identifying Programs Likely to Be Impacted

Once the set of potentially violating updates (PVUs) has been identified for the new constraint, we can use this information to quickly determine which of the application programs are likely to be capable of violating it. Informally, we can state the criterion used to filter the set of programs as follows:

> A program is potentially impacted by the addition of the new constraint iff there is some PVU for that constraint such that every component update of the PVU is performed by some part of the program.

In the case of example constraint ic_1, this criterion implies that any program which performs a deletion from the *gold* table is impacted by the addition of ic_1.

Programs which contain no deletion or modification operations on the *gold* table and no insertions or modifications to either *cust* or *managed* are not impacted.

More formally, for a new constraint ic_n, we can define the set of potentially impacted programs IP ($IP \subseteq Ps$) as:

$$IP = \{\, p \mid (\exists u, c)\; p \in Ps \wedge < u, c > \in gcu(ic_n) \wedge$$

$$((\forall ut, tn, v) < ut, tn, v > \in u \;\Rightarrow (\exists s) < ut, tn, s > \in p) \,\}$$

The advantage of this filtering criterion is that it is relatively cheap to compute (especially if an index has been created that provides quick access to the database update statements performed by the program [17]). It also meets our requirements in that it can eliminate a high proportion of the non-impacted programs without eliminating any impacted programs.

It is not, however, a completely accurate determiner of impact, and some non-impacted programs will pass successfully through our filter. This can happen when the updates within a PVU all occur in a program but in such a way that it is impossible for them to be executed together in one run of the program.

Another situation in which the filter will give inaccurate results is when the PVU updates occurring in the program do so within a context where the guard condition can never be satisfied. For example, if we have the following PVU:

$$+cust(c, a, b) \;\; \textit{if} \;\; b \leq 0$$

then a COBOL program p containing the following fragment would be considered to be potentially violating according to our filtering criterion:

```
IF WS-BALANCE > 0
   MOVE WS-CNO TO DB-CUST-CNO
   MOVE WS-CUST-ADDR TO DB-CUST-ADDR
   MOVE WS-BALANCE TO DB-CUST-BAL
   STORE CUST.
```

However, if this is the only update to the *cust* table in p, then p should not be included in the set of impacted programs. This is because the update specified in the PVU occurs in a context in which the associated condition can never be satisfied; the enclosing if-statement is already guarding the update against this particular kind of violation.

4.2 Identifying the Impacts within a Program

Having determined that a particular program is likely to be impacted by the addition of a new constraint, the next step is to perform a more detailed analysis of that program, to discover the ways in which it is impacted more precisely. In order to do this, we must identify the subset of the PVUs of the new constraint that may be performed by the program being analysed. This allows us to pinpoint the exact source code statements which are involved in each PVU, so that the programmer can be given some guidance on where new guard conditions must

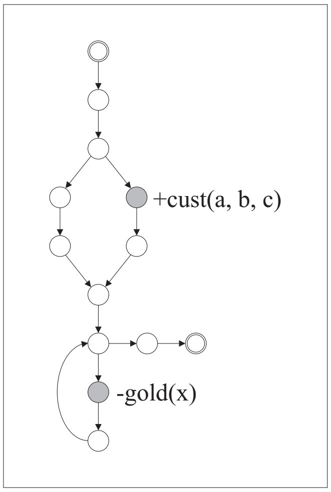

Fig. 1. Example CFG with update nodes distinguished

be added to the program. In general, of course, a given program will be capable of performing several different PVUs, and may even be able to perform the same PVU in several different ways.

Conceptually, the problem of identifying the set of program statements that can perform a PVU corresponds to the identification of subgraphs in the program's control flow graph (CFG) which contain all the updates in the PVU in sequence. For example, Figure 1 shows the basic structure of a control flow graph for a very simple program. The nodes of the graph correspond to the statements of the program and the edges between the nodes indicate the control flow paths that are possible between statements. Nodes corresponding to database update commands are shown in grey and are labelled with a description of the state change operation the command performs.

If we match this example control flow graph against the PVU set identified from constraint ic_1 then we discover that the following three PVUs can be performed by this program:

+cust(c, a, b)	*if managed(c)* $\wedge \neg$ *gold(c)*
+cust(c, a, b), -gold(c)	*if managed(c)*
-gold(c)	*if cust(c, a, b)* $\wedge$ *managed(c)*

In order to compute such matches, we first construct a "trace" of the database update commands that can be performed by the program being analysed, and then match each PVU update set against this trace[4]. The trace is computed

[4] In our prototype implementation, we actually compute the trace and perform the matching at the same time, as this is more efficient than computing the trace separately from matching. However, we present them as two separate steps here, in order to simplify the presentation.

by walking the abstract syntax tree of the program, adding the identifiers of update statements as they are encountered. Effectively, it provides a (finite) representation of the sequences of update commands that may be executed by the program.

Since programs may contain conditional branches and loops, it is not possible to use a simple list structure to represent this trace. Instead, we use a form of AND-OR tree, where AND nodes represent sequential execution of traces and OR nodes represent alternative execution of traces. Traces formed from the bodies of loops are tagged with a special label, which indicates that the updates in the trace may match against multiple elements of the PVU set.

The algorithm for constructing update traces follows the usual syntax-directed pattern for analysing programs. Here, we give its definition for several representative kinds of programming language construct. In each case, the inputs to the *trace* function are the current statement in the abstract syntax tree, and the trace constructed so far. The result is the trace describing the behaviour of the program immediately after the execution of the statement given as input.

A sequence of statements: in this case, the resulting trace is that produced by execution of the second part of the sequence, in the context of the trace produced by the first.

$$\text{trace}([\![S1; S2]\!], \text{tr}) = \text{trace}(S2, \text{trace}(S1, \text{tr}))$$

A conditional statement: for this kind of statement, we process each branch of the conditional in the context of the given trace, and then combine the resulting traces together using an OR node.

$$\text{trace}([\![\textit{if } C \textit{ then } S1 \textit{ else } S2]\!], \text{tr}) = \text{or}(\text{trace}(S1, \text{tr}), \text{trace}(S2, \text{tr}))$$

A loop statement: most loop constructs actually include a conditional element, representing the exit condition. For example, consider the definition of the *trace* function for a standard *while-do* loop:

$$\text{trace}([\![\textit{ while } C \textit{ do } S]\!], \text{tr}) = \text{or}(\text{tr}, \text{tag}(\text{trace}(S, \text{tr}))$$

If the loop condition C is false, then this loop will execute zero times, in which case the resulting trace is simply that produced after execution of the preceding statement (i.e. the input trace). Otherwise, the final trace is that produced by executing the loop body an arbitrary number of times.

How do we produce a trace representing the effects of an arbitrary (and possibly infinite) number of iterations of the loop body? In fact, if we could be sure that every update type to every database table would appear at most once in each PVU, the answer to this question would be exceedingly straightforward, since we would obtain the same set of matches from the trace of a single execution of the loop body as we would from an arbitrary number of executions. Unfortunately, there is no reason why a PVU cannot contain multiple examples of the same kind of update. For example:

+cust(A, B, C), +cust(B, D, E) if ...

This example PVU matches with a program that contains a single insertion to the *cust* table only if the insertion command appears within the body of a loop. This is because, effectively, updates which occur with loops may match with updates in a PVU an arbitrary number of times, while updates which do not occur within a loop may be matched with only one PVU update. We can therefore model the behaviour of loops by means of the simple expedient of tagging updates which occur within loops, so that the matching algorithm is aware that they can be treated specially.

A simple non-update command: simple commands which are not database updates are essentially ignored by the trace generation algorithm, e.g.

$$\text{trace}([\![\textit{print } T]\!], \text{tr}) = \text{tr}$$

A database update command: if we encounter an update command, we must add the details of the update (represented as a triple of update type, table name and statement identifier) to the current trace. For example:

$$\text{trace}(\ [\![\ \textit{store tn}\ ]\!], \text{tr}) = \text{and}(\ \text{tr}, < `i', tn, s >\ )$$
$$\text{where s} = \text{the identifier of the current statement}$$

Having computed the trace of updates produced by a program, we can then attempt to match each of the PVUs against it. Informally, we say that a PVU $< us, c >$ matches a trace t iff all the updates in the set us occur as a sequence somewhere within t. More formally, a set of updates us matches t in the following cases:

– If $us = \emptyset$ then us matches t.
– If $us \neq \emptyset$ and $t = and(t_1, t_2)$ then us matches t iff us can be partitioned into two disjoint subsets u_1 and u_2, such that u_1 matches t_1 and u_2 matches t_2.
– If $us \neq \emptyset$ and $t = or(t_1, t_2)$ then us matches t iff us matches t_1 or t_2.
– If $us \neq \emptyset$ and $t = tagged(and(t_1, t_2))$ then us matches t iff us can be partitioned into two disjoint subsets u_1 and u_2, such that u_1 matches $and(t_1, t_2)$ and u_2 matches t.
– If $us \neq \emptyset$ and $t = tagged(or(t_1, t_2))$ then us matches t iff us can be partitioned into two disjoint subsets u_1 and u_2 such that:
  • u_1 matches t_1 and u_2 matches t, or
  • u_2 matches t_2 and u_1 matches t.
– If $us = \{< `i', tn, v >\}$ then $us$ matches $t$ iff $t = \{< `i', tn, s >\}$ or $t = < `m', tn, s >$.
– If $us = \{<' d', tn, v >\}$ then us matches t iff $t = \{< `d', tn, s >\}$ or $t = \{< `m', tn, s >\}$.

For each match against a given trace, we record the identifier of the PVU which has been matched and a list of the update statements responsible for it. We also store a "contextualised" form of the necessary and sufficient condition associated with the matched PVU. A contextualised condition is one in which some of the original variables have been replaced with variables from the program code. It describes a constraint over the program variables that are involved in the update commands, rather than over the "disembodied" logical variables of the original integrity constraint.

Details of all possible matches are recorded (and shown to the programmer), since each one represents a potential cause of violations within the program that must be guarded against. In some cases, this may mean that a single program statement matches with several different PVUs. It is important that the programmer is aware of all the ways in which a program statement may lead to a violation, before he or she decides on the guard conditions that will be added to the program.

5 An Illustrative Example

In order to illustrate our impact analysis technique in action, we will present a small (and rather artificial) example of the kinds of result it can produce. Assume that we have an information system which consists of a customer database (*db*) and three applications programs (p_1, p_2 and p_3). We wish to add constraint ic_1 to the system. Recall that ic_1 is defined as follows:

$$ic_1 \equiv (\forall c, a, b)\ \mathit{cust}(c, a, b) \wedge \mathit{managed}(c) \Rightarrow \mathit{gold}(c)$$

Program p_1 performs a query over the database and produces a report. It does not make any updates to the database, and would therefore be filtered out by our program during phase 1 of the analysis.

Program p_2, on the other hand, is used to delete customers and their associated records. All the database update commands it contains are included in the following fragment of code, which appears in the program immediately after a section of code which attempts to retrieve the customer record to be deleted. The variable `DB-CUST-CNO` identifies this record.

```
100 IF DB-REC-EXISTS
       ...
160    MOVE DB-CUST-CNO TO DB-GOLD-CNO
170    DELETE GOLD
       ...
230    MOVE DB-CUST-CNO TO DB-MANAGED-CNO
240    DELETE MANAGED
250    DELETE CUST
260    COMMIT.
```

This program meets our criterion for impacted programs, since it contains all the updates included within one of the PVUs derived from ic_1:

$$8 \rightarrow 7 \qquad -\mathit{gold}(c) \quad \mathit{if}\ \ \mathit{cust}(c, a, b) \wedge \mathit{managed}(c)$$

During the second phase of impact analysis, the PVUs are matched against the structure of the program in more detail, and statement 170 is found to match with the PVU shown above. The contextualised condition that is attached to the record of this match is:

$$(\exists a, b)\ \mathit{cust(DB\text{-}CUST\text{-}CNO,\ a,\ b)} \land \mathit{managed(DB\text{-}CUST\text{-}CNO)}$$

The programmer browses the set of matches, and the statements to which they are attached, and attempts to determine how much of the contextualised condition is already implemented by the program. In fact, since the programmer knows that the *customer number* attribute is a key attribute of all three of the tables updated by this program, he or she can also deduce that the PVU condition can never be satisfied at the point when the `COMMIT` operation is invoked (line 260). This is because any *cust* and *managed* tuples which join with the deleted *gold* tuple are also deleted by the program, before the effect of this PVU is committed to the database. The programmer therefore decides that no change is required to program p_2 (except perhaps for the addition of a comment to explain why the constraint cannot be violated by this sequence of updates).

Program p_3 contains a more complex sequence of updates. The parts of the program that are relevant to our analysis (i.e. those that involve database updates) are shown below:

```
    ...
120 IF NOT DB-REC-EXISTS
130     MOVE WS-CNO TO DB-CUST-CNO
140     MOVE WS-ADDR TO DB-CUST-ADDR
150     MOVE WS-BALANCE TO DB-CUST-BAL
160     STORE CUST
170 ELSE
    ...
210 END-IF.
    ...
340 IF WS-CTYPE = "M"
350     MOVE WS-CNO TO DB-MANAGED-CNO
360     STORE MANAGED
370 ELSE
    ...
420 END-IF.
    ...
780 COMMIT.
    ...
```

This program passes successfully through the phase 1 filter, and is then matched against the PVUs. As a result, three records are created, to record the following PVU matches:

– The program may execute PVU $1 \rightarrow 7$, through update statements [160, 360]. The contextualised condition is:

$$\neg\ \mathit{gold(DB\text{-}CUST\text{-}CNO)}$$

- The program may execute PVU 3 → 7, through update statement [160]. The contextualised condition for determining violation is:

$$\mathit{managed(DB\text{-}CUST\text{-}CNO)} \wedge \neg\ \mathit{gold(DB\text{-}CUST\text{-}CNO)}$$

- The program may execute PVU 5 → 7, through update statement [360]. The contextualised condition is:

$$(\exists a, b)\ \mathit{cust(DB\text{-}CUST\text{-}CNO,\ a,\ b)} \wedge \neg\ \mathit{gold(DB\text{-}CUST\text{-}CNO)}$$

At first sight, the first and third of these matches might be thought to be redundant, as their update sequences are both subsets of that of the second match. However, since it is possible for the flow of control to pass through one of the identified statements (i.e. 160 and 360) without necessarily passing through them both, the program actually has the capability to violate the constraint in all these three ways. The programmer must be made aware of all these possibilities, so that the most appropriate guard (or guards) can be designed and inserted into the program.

In our example, since the conditions of the first and third matches are both stronger than the condition associated with the second, the programmer might decide to add pre-condition guards to each of the individual updates. Alternatively, he or she might decide to place a single, more complex guard before the commit operation, that tests for all three of the contextualised conditions at once.

6 Conclusions

Modifying an information system so that it enforces new business rules is a challenging and error prone task. If any of the necessary modifications are omitted or carried out incorrectly, then the business rule will only be enforced by part of the system and inconsistent behaviour will result [6]. In this paper, we have described how techniques from database integrity maintenance can be used as the basis of a technique which can help make the implementation of certain kinds of business rule a less painful and risky procedure. The technique, which has been prototyped in Prolog, helps the programmer by identifying the set of programs which have the potential to cause violations and by indicating which lines of code are responsible for this ability. It is also able to provide guidance as to the form of guard that should be inserted into the program to prevent violations from occurring at run-time.

We expect that the principal practical benefits of our technique will be in reducing the amount of time programmers have to spend examining program code that is not related to the new constraint; and in reducing the chance of errors of omission by helping to ensure that all the possibilities for violations, however complex or obscure, are considered by the programmer. It is also possible that this technique might have a role in validation of software, i.e. in helping to determine whether a software system correctly implements a given set of database constraints, or whether it has the capacity to violate some of them. In addition, knowledge of the source code statements implicated in the implementation of a constraint can also help in generating white-box test cases aimed at detecting defects in the implementation, or for use in regression testing.

However, further evaluation of the technique is required before these claims can be verified. In particular, we need to discover whether the algorithms described in this paper are capable of processing real-scale constraints and programs in realistic timescales. The time required to compute impacts by our method is dependent on two factors: the number of potentially violating updates that are derived from the new constraint, and the number of distinct sequences of database update commands that can be performed by the programs being analysed.

Unfortunately, the number of PVUs derivable from a constraint is 2^n in the average case and 2^{2n-2} in the worst case, where n is the number of distinct atomic predicates appearing in the rule. Similarly, a program containing m database update statements will be capable of performing up to $2^m - 1$ distinct update sequences, all of which must be matched against the set of PVUs. The efficiency of this technique in practice is therefore clearly a matter of concern. Our experience with extraction of business rules from legacy systems suggests that, for many rules, n will be small (less than 10, for example) [7]. We cannot be so sanguine in the case of database update statements - it does not seem unreasonable to expect a typical database application program to contain tens of update statements. However, we can reasonably expect the average number of update sequences to be much lower than that predicted by our worst case analysis.

The next stage in our research, therefore, will be to test our technique on some real application programs, for the implementation of constraints of realistic complexity. If the tool is efficient enough to be practically usable, then we can attempt to gauge its success in reducing effort and errors in integrity constraint implementation. If the algorithm's complexity is found to be a stumbling block to exploitation of the tool, we shall seek a less accurate but more efficient version of the algorithms that can at least give some help to the programmer in dealing with the complexity of the rule implementation task.

References

1. R. Arnold and S. Bohner, editors. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.
2. S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In G. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of 17th International Conference on Very Large Databases*, pages 577–589, Barcelona, 1991. Morgan Kaufmann Publishers, Inc.
3. S. Ceri and J. Widom. Production Rules in Parallel and Distributed Database Environments. In L.-Y. Yuan, editor, *Proceedings of 18th International Conference on Very Large Databases*, pages 339–351, Vancouver, August 1992. Morgan Kaufmann Publishers, Inc.

4. L. Deruelle, M. Bouneffa, N. Melab, and H. Basson. A change propagation model and platform for multi-database applications. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 42–51, Florence, Italy, November 2001. IEEE Computer Society Press.
5. O. Díaz. Deriving Rules for Constraint Maintenance in an Object-Oriented Database. In A.M. Toja and I. Ramos, editors, *DEXA 92 Conference*, pages 332–337. Springer-Verlag, 1992.
6. A.B. Earls, S.M. Embury, and N.J. Turner. A Method for the Manual Extraction of Business Rules from Legacy Source Code. *BT Technology Journal*, 2002. To Appear.
7. Andrew B. Earls. A Method to Extract the Business Rules from a Large Legacy System. Master's thesis, Dept. of Computer Science, Cardiff University, 2001.
8. S.M. Embury, S.M. Brandt, J.S. Robinson, I. Sutherland, F.A. Bisby, W.A. Gray, A.C. Jones, and R.J. White. Adapting Integrity Enforcement Techniques for Data Reconciliation. *Information Systems*, 26(8):657–689, December 2001.
9. S.M. Embury and P.M.D. Gray. Compiling a Declarative, High-Level Language for Semantic Integrity Constraints. In R. Meersman and L. Mark, editors, *Proceedings of 6th IFIP TC-2 Working Conference on Data Semantics*, pages 188–226, Atlanta, USA, May 1997. Chapman and Hall.
10. D. Hay and K.A. Healy. Defining Business Rules — What are they Really? Technical report, The Business Rules Group, Revision 1.3, July 2000. Available on-line at `http://www.essentialstrategies.com/ publications/businessrules/`.
11. ILOG, Inc. JRules Business Rule Engine. `http://www.ilog.com/products/rules/ engines/jrules/`, 2002.
12. J.-M. Nicolas. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, 18:227–253, 1982.
13. Quest Software. Effective Application Source Code SQL Analysis for Change Impact, Performance and Quality. White paper, available at `http://www.quest.com/whitepapers/`, 2002.
14. J. Shao and C. Pound. Reverse Engineering Business Rules from Legacy Systems. *BT Technology Journal*, 17(4):179–186, 1999.
15. D.I.K. Sjöberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, 1993.
16. CAST Software. Envision, October 2002. `http://www.castsoftware.com/`.
17. Software AG. Adabase High Performance Database System. `http://www.softwareag.com/adabas/`, 2002.