A safe regression test selection technique for database–driven applications

David Willmor and Suzanne M. Embury
School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, United Kingdom
d.willmor|s.m.embury@cs.man.ac.uk

Abstract

Regression testing is a widely-used method for checking whether modifications to software systems have adversely affected the overall functionality. This is potentially an expensive process, since test suites can be large and time-consuming to execute. The overall costs can be reduced if tests that cannot possibly be affected by the modifications are ignored. Various techniques for selecting subsets of tests for re-execution have been proposed, as well as methods for proving that particular test selection criteria do not omit relevant tests. However, current selection techniques are focussed on identifying the impact of modifications on program state. They assume that the only factor that can change the result of a test case is the set of input values given for it, while all other influences on the behaviour of the program (such as external interrupts or hardware faults) will be constant for each re-execution of the test.

This assumption is impractical in the case of an important class of software system, i.e. systems which make use of an external persistent state, such as a database management system, to share information between application invocations. If applied naively to such systems, existing regression test selection algorithms will omit certain test cases which could in fact be affected by the modifications to the code. In this paper, we show why this is the case, and propose a new definition of safety for regression test selection that takes into account the interactions of the program with a database state. We also present an algorithm and associated tool that safely performs test selection for database–driven applications, and (since efficiency is an important concern for test selection algorithms) we propose a variant that defines safety in terms of database state alone. This latter form of safety allows more efficient regression testing to be performed for applications in which program state is used only as a temporary holding space for data from the database. The claims of increased efficiency of both forms of safety are supported by the results of an empirical comparison with existing techniques.

1. Introduction

Regression testing is a well known technique for verifying the behaviour of a modified program [4]. Briefly, it involves the execution of tests subsequent to change to the program in an attempt to determine whether the behaviour of the program has changed in ways outside the scope of the intended modification. Whilst regression testing is important, it is also an expensive process (in terms of the cost of time and resources) and a number of methods have been proposed to improve its efficiency. In particular, *regression test selection* algorithms [1, 2, 5, 6, 7, 8, 9, 11, 12, 13] aim to increase efficiency by focussing only on those test cases that can possibly be affected by the modifications that have been made to the system. This is done by analysing the source code of the program before and after modification, and then determining (according to some criteria) which tests need to be re-executed. Many forms of regression test selection have been proposed; the differences between them lie in the way the source code is analysed and the criteria that determine what tests will be affected by a modification.

A regression test selection algorithm is deemed *safe* if it selects, from the original test suite, every test case that may reveal a fault in the modified program [11]. Traditional safe selection techniques focus upon the effect a modification has on statements that manipulate program state. The manipulation of program state, however, is not the only means by which programs achieve their intended effects; other forms of state may also have an important role to play. For example, many modern business programs operate over a database, with program state being used merely as a temporary holding site for data that has been retrieved from the database or that will be written back to it. Such programs manipulate two kinds of state at the same time (the program state and the persistent state), often in closely inter-related ways. Traditional test selection algorithms (and the proofs of test selection safety on which they are based) do not account for this form of program behaviour and instead as-

sume that the database state will remain fixed (controlled) during episodes of regression testing. If this assumption cannot be met and the database is allowed to evolve (as is necessary for most real systems) then there is no guarantee that all necessary tests will be executed and some faults may go undetected.

The key reason for this is that database state is not a simple variant of program state. It is fundamentally different in several ways. Whereas program state is organised as a collection of named locations where individual data values can be stored, database state is organised relative to the concepts provided by the data model (such as the relational data model). Large segments of the state can be accessed or modified through the execution of a single program statement. Moreover, changes to the state are controlled by a transaction mechanism and may be undone by some later statement. Finally, database state is persistent, so that changes made by a statement in one program may have an effect on the behaviour of other statements in completely separate programs.

In this paper, we discuss the ramifications of these differences for the process of regression test selection (Section 3.1), and show how existing definitions of selection safety are inappropriate in the context of database–driven applications. We present a new definition of safety tailored for exactly this context (Section 4) and derive from it an algorithm that performs database–state safe selection of test cases. We also introduce the notion of test selection that is guaranteed to be safe only in terms of the manipulation of database state, and that allows more efficient regression test selection in cases where the manipulation of program state is secondary to the manipulation of the external persistent state. We have implemented both these algorithms and have used the resulting testing tool to empirically determine their relative efficiencies in comparison with existing approaches (Section 6). Finally, we present concluding remarks and possible directions for future work (Section 7).

2. Background

Regression test selection has been shown to improve the efficiency of regression testing [12]. Its aim is, given an existing program P, a test suite T associated with P and a modified version of the program P', to identify a set of tests $T' \subseteq T$ that verifies all changes present in P'. This is accomplished by first analysing P and P' in order to determine the changes that have be made. These changes are referred to as the *dangerous entity set*, and may consist of a collection of statements, nodes of a control flow graph or some other granularity of system component. Once established, the dangerous entity set is then used by the selection algorithm to determine which tests might be affected by the modifications (and therefore should be re-executed) and which cannot be affected (and therefore can be omitted).

A number of regression test selection criteria have been proposed [1, 2, 7, 8][1] The efficacy of any given criteria is determined based on the notion of the set of fault-revealing tests $F \subseteq T$ that consists of all test cases in T that may reveal a fault in P'. Given this, a test selection algorithm is evaluated in terms of its: (a) *inclusiveness*, i.e. the extent to which all necessary fault-revealing tests are included in T', (b) *precision*, i.e. the extent to which non-fault-revealing tests are excluded from T'; (c) *efficiency*, i.e. the time required to perform test selection[2]; and (d) *generality*, i.e. the suitability of both the algorithm and selection criterion for different situations and types of application [11].

The only existing work on regression test selection for database–driven applications is that reported by Haraty *et al.* [6], in which a regression testing approach for stored procedures in databases was proposed. In many ways, stored procedures are similar to application programs, although they are typically much smaller and simpler. Haraty *et al.* based their work on the firewall approach of Leung and White [8], which has since been proved to be an unsafe selection technique (relative to program state) by Rothermel and Harrold [11]. None of the adaptations proposed by Haraty *et al.* correct this and therefore their technique may omit test cases that have the potential to reveal a fault.

2.1. Safe regression test selection

A key requirement for any regression test criterion and algorithm is that they should be *safe*; that is, the set of tests selected by the algorithm (T') should include all the fault-revealing tests F that are present in the original test suite (i.e. $F \subseteq T'$). This notion has been formalised by Rothermel and Harrold [12] to provide a mechanism by which the safety of a given selection technique can be proved. Since our own work builds on such proofs, we will present a summary of the technique, as applied to a control-flow based selection technique, also proposed by Rothermel and Harrold [12].

In this technique, a control flow graph (CFG) is generated for both the original program P and modified version P', and test traces (i.e., the set of edges of the CFG that are covered by a given test) are determined for the original program and its associated test suite. The following defines a control flow graph:

Definition 1 *A Control Flow Graph G_v for a method m_v is a pair $\langle N_v, E_v \rangle$ where:*

1 A full evaluation of each different technique is beyond the scope of this paper, but can be found elsewhere [11].

2 A test selection algorithm is deemed to be efficient if the time required to select and execute T' is less than is required to execute T.

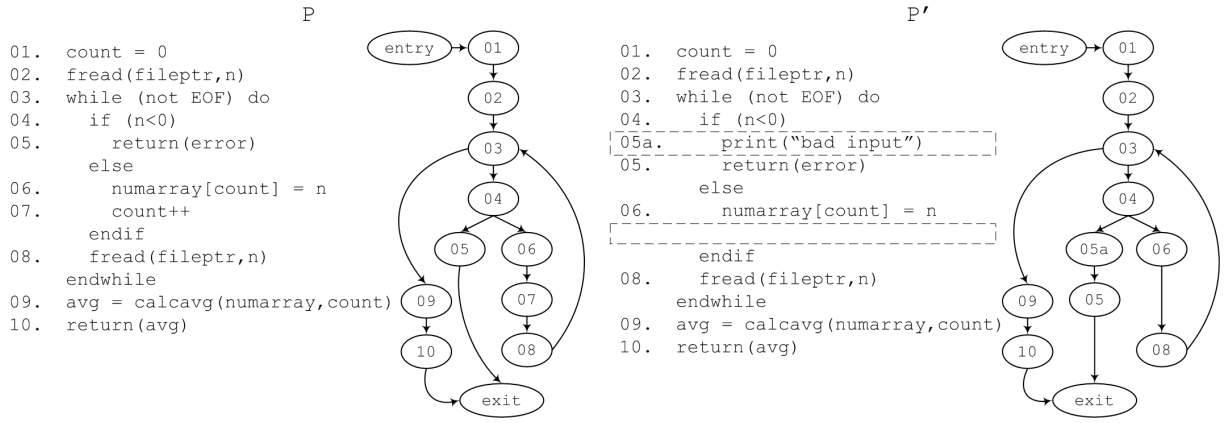

Figure 1. An example of P and P' [12]

- *N_v is a set of nodes such that each $n \in N_v$ represents a statement in method m_v.*
- *E_v is a set of edges such that $e \in E_v$ represents a transfer of control in method m_v.*

Rothermel and Harrold's depth-first algorithm then traverses both CFGs simultaneously. When a pair of nodes, n from P and n' from P', are discovered that are lexicographically different, the algorithm includes n's incoming edge into the dangerous entity set. This set determines which edges must be covered by T' in order to safely verify the modifications.

Using the approach of Rothermel and Harrold [12], T' is determined by selecting every test case $t \in T$ that, according to its test trace, covers (i.e. executes) an element of the dangerous entity set. Figure 1 provides an example of the source code and control flow graph for a program P and a modified version P' [12]. In this example, two modifications have been made to the original program, P: (1) a new statement has been inserted between statements 4 and 5 (called 5a); and (2) statement 7 has been deleted. This gives a dangerous entity set containing $\{(4,5),\ (6,7)\}$. Test suite T' is modification-traversing (and therefore fault-revealing) if it contains all test cases in T which cover these edges.

2.2. Rothermel and Harrold's proof

The steps by which Rothermel and Harrold proved that their technique is safe for controlled regression testing are summarised below [10, 12], beginning with their definition of a test case:

Definition 2 *"A test is a triple $t = \langle n, i, S(i)\rangle$ where n is an identifier for the test, i is the input for the execution of the program, and $S(i)$ is the output from the program for a specific input." [10] (p. 17). In regression testing, an obsolete test (denoted* Obsolete(t)*) is one that no longer exercises the components which it was originally intended for [10] (p. 18).*

The goal of regression test selection is to determine the subset of T that will reveal a fault in P'; this is called the set of fault–revealing tests [10] (p. 19). However, faults in software testing are typically defined in terms of a mismatch with the specification of the program. In regression testing, the aim is to compare the behaviour of two programs (P and P'), rather than a program with its specification, and therefore the notion of fault cannot be reused naively. Instead, Rothermel and Harrold base their proof on the assumption that P halts with the correct output for all tests in T. More formally, the P-Correct-for-T Assumption states that:

"$(\forall t = \langle n, i, S(i)\rangle \in T)\ [(P(i) = S(i)) \wedge P$ *halts on input* $i]$" [10] (p. 21)

Under this assumption, since every $t \in T$ is correct for P then any t that gives a different result for P' must reveal a fault. In practice, of course, many different factors could contribute to a change in the result from a test. External interrupts may prevent a subcomponent from completing its processing, for example, as may contention for external resources and hardware faults. Rothermel and Harrold therefore place a further assumption on their proof, which is that all such external influences on the behaviour of test cases will be held fixed between regression testing episodes. This is called *proper* (or *controlled*) regression testing. Under this assumption, any change in the result returned from a test case must be solely due to the code changes that produced P'. Such a test is said to be *modification–revealing* [10] (p. 21).

The test selection problem now reduced to one of determining the set of all modification–revealing tests in T. Unfortunately, this is also unsolvable. Instead, Rothermel and Harrold approximate this notion by searching for the set of *modification traversing* tests, based on the execution

traces of each test. For a program P "the execution trace $ET(P(i))$ is the sequence of statements $\langle s_1, s_2, \ldots, s_n \rangle$ in P that are executed for input i" [10] (p. 22). A test which results in different execution traces for P and for P' must be producing this different behaviour because of the modifications to P', i.e. it is modification–traversing.

Definition 3 *"a test $t = \langle n, i, S(i) \rangle \in T$ is modification–traversing for P and P' if and only if $\neg Obsolete(t) \wedge ET(P(i)) \not\equiv ET(P'(i))$." [10] (p. 24).*

Definition 4 *"a non–obsolete test $t = \langle n, i, S(i) \rangle \in T$ has been properly regression tested on P' if and only if $P(i) \neq P'(i) \Rightarrow ET(P(i)) \not\equiv ET(P'(i))$." [10] (p. 24)*

From the two previous definitions, we can see that all modification–revealing tests must by necessity also be modification-traversing:

Proposition 1 *"If $t = \langle n, i, S(i) \rangle \in T$ has been properly regression tested on P', and t is modification–revealing for P and P', then t is modification–traversing for P and P'." [10] (p. 24)*

Proof 1 *"Because t is modification–revealing for P and P', t is non-obsolete, and $P(i) \neq P'(i)$. Because t has been properly regression tested on P, $P(i) \neq P'(i) \Rightarrow ET(P(i)) \not\equiv ET(P'(i))$. Thus $ET(P(i)) \not\equiv ET(P'(i))$: t is modification–traversing for P and P'." [10] (p. 24)*

The consequence of this proof is that the set of modification–traversing tests can provide us with an accurate, though not minimal, approximation of the set of all modification-revealing tests, and therefore of the set of fault–revealing tests. Thus, Rothermel and Harrold's test selection technique is shown to be safe.

3. Regression test selection for database–driven applications

In order to adhere to the requirement for *proper* regression testing in the context of database–driven applications, it would be necessary to ensure that the same database state was used to execute each test case as was used on the unmodified program P. Not every test case can be executed sensibly in every database, however. For example, a test case which verified the deletion of customers who are deemed to be inactive would only make sense if executed against a database state that contained some such customers. One would also probably wish to include some customers that do not meet the constraints for inactivity, in order to check that they are not deleted by the program under investigation. Another test case might require that no customers of this kind be present in the database system, in which case the testing team would need to provide two initial database states, one for each test case. Although some test cases could share initial database states, in general one would be faced with the problem of creating and maintaining a collection of several database states that could only be used for regression testing purposes. Alternatively, it would be possible to recreate the required databases dynamically, before each test case is executed. This would allow us to have as many initial database states as needed without requiring unrealistic amounts of storage space, but it also adds significantly to the cost of regression testing, which is delayed while the necessary database states are prepared [3].

There is a more serious problem, however. The presence of a secondary, persistent form of state in a software system means that the programs it consists of can no longer be considered as independent units. Typically, such programs use the persistent state to communicate with one another, so that one program reads data that has been written to the state by another. In this context, a fault that is introduced in one program may not become visible to the testing system until a second program (e.g. that reads the faulty data) is executed. In fact, we cannot even consider individual complete paths through a single program to be independent, since a fault that is introduced in one path may be picked up and propagated to the output by execution of another path through the same program.

Because of this, it is not sufficient in the context of database applications to consider regression test selection in terms of the execution traces of individual programs with fixed initial database states. Instead, we must take into account the interactions between program executions if we are to locate the full set of modification–revealing tests. In order to explain this point in more detail, we will discuss the particular characteristics of database systems and show how they impact on the technique of selecting modification–traversing tests for regression testing.

3.1. Characteristics of database–driven applications

A database–driven application is a type of computer system in which the required behaviour is achieved by the combined manipulation of a persistent external state (the database state) and an ephemeral internal state (the program state). In general, each system will consist of multiple application programs (possibly implemented in a variety of programming languages), each of which may access multiple external states (possibly based on different storage technologies). However, for simplicity of presentation, we will view all the external state as a single entity, structured according to a single data model and accessed through a single interface.

We also assume that the application programs are object-oriented in nature, although our approach is applicable in principle to procedural programs as well. Thus each pro-

gram consists of a set of classes, each of which contains a number of methods. This leads to the following definition of a database–driven application:

Definition 5 *A database–driven application consists of a program P and a database D structured according to a schema Σ. A program P is a pair $\langle C_P, I \rangle$[3]:*

- *C_P is a set of control flow graphs, where each $c_n \in C_P$ is the CFG corresponding to method m_n of program P.*
- *I is an interclass relation graph (IRG) representing the relationships between the classes (and their methods) in P [7].*[4]

In the above definition D and Σ represent the database state, which takes a rather different form to program state. Rather than being a collection of simple variables, all data (D) are organised as complex data structures with well-developed and often explicit semantics (as defined by the schema Σ). In order to model the effect a statement has upon database state, in a way that is analogous to the define-use model that has proved to be so successful for program state, we utilise an approach we defined in previous work [14] in which each operation is categorised in terms of three properties: read, add and delete. In the relational data model, for example, data is structured as tables, with columns and rows (tuples), and program statements manipulate a subset of the full dataset (e.g. a SELECT operation reads all tuples of some table that match a given condition). In more modern forms of database–driven application, database operations are embedded into program statements as declarative descriptions of the behaviour required. This makes it possible to automatically extract an intensional description of the elements of database state that are accessed by a given statement. The data manipulation undertaken by each node in a program P can be described by the sets $\{P_{def}, P_{ref}, D_{add}, D_{del}, D_{read}\}$:

- P_{def} the subset of program state defined by n
- P_{ref} the subset of program state referenced by n
- D_{add} the subset of database state that is updated by n
- D_{del} the subset of database state that is deleted by n
- D_{read} the subset of database state that is read by n

For example, given a program with embedded SQL commands, we can extract relational algebra expressions describing the sets D_{add}, D_{del}, and D_{read} for each statement.

3 In this definition we describe program P in terms of an object-oriented program, however, the principles are applicable to both procedural and interprocedural programs.

4 The IRG models the complexities associated with object-oriented languages, including variable and object type information; internal and external methods; interprocedural interactions; inheritance, polymorphism, dynamic binding; and exception handling.

3.2. The limitations of controlled regression testing of database–driven applications

A fundamental type of regression fault that is the *definition–use fault*. This fault occurs when a modification to a definition of some variable adversely affects a later use of it. Rothermel and Harrold's selection algorithm [12] will choose tests that can determine the presence of this type of fault for all variables within program state, by selecting every test which covers a use of a variable subsequent to a modified definition of that same variable. This approach succeeds for applications that manipulate only program state because the use of a variable cannot precede its definition in any given paths through a program. With database–driven applications, however, this is not the case. It is perfectly possible for a program statement to write some data to the database that will later be read by a program statement that precedes the first statement in some execution path. Another possibility is that the two statements appear in completely separate paths, while still participating in a very real define-use relationship.

Given this, it is straightforward to locate counter-examples that show that when the database is allowed to evolve between program executions, Rothermel and Harrold's proposition that all fault–revealing tests are also modification–traversing is no longer applicable. Suppose we have a database–driven application program P_{DB} that includes at least two complete paths, π_1 and π_2, which are the execution traces for tests t_1 and t_2 respectively. Path π_1 includes a database operation which adds new tuples to a table that stores details of orders for a particular product. Path π_2, on the other hand, reads from the order table and produces some result based on the data that is retrieved. In this case, a modification to the update statement in π_1 could cause a fault to be introduced into the database state. The fault is not propagated to the output of test t_1, but it is detectable in the output of test t_2, even though this test is not modification–traversing.

More formally, we can say that, if s_1 is the update statement in π_1 and s_2 the read statement in π_2 then if $s_2.D_{read} \cap (s_1.D_{add} \cup s_1.D_{del}) \neq \emptyset$ there is a define-use relationship between the two statements that must be exercised by the selected tests.

4. A safe selection algorithm for database–driven applications

If the identification of modification–traversing tests is not sufficient in the context of database–driven applications, what additional selection criteria should we use? The missing tests are those which exercise both ends of all definition-use relationships where the definition statement has been

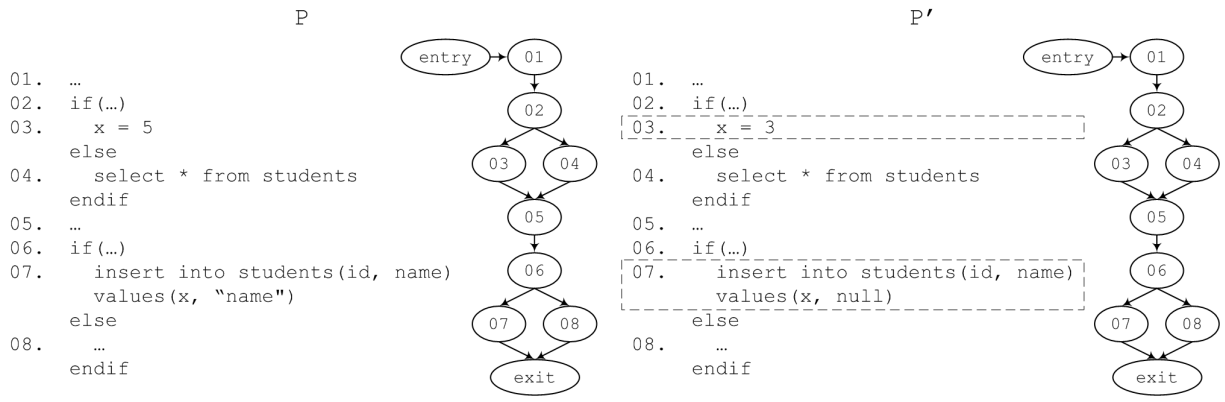

Figure 2. An example database–driven application

modified. We call a test case which is included in the regression test suite by this criterion a database–dependent test. In order to select all such tests for a modification, we need to identify:

1. the set M_δ of all statements that cause an update to the database state and that have been modified in P' in such a way that the portion of the database state that they affect has changed, and
2. the set U_δ of all statements, in any program, that are use-dependent upon the statements in M_δ.

To calculate M_δ, we must be able to determine when a modification to a statement may cause it to affect a different portion of the database state. This can be done by comparing the intensional descriptions of the parts of the database state that are added and deleted by the statements:

Definition 6 *Two database operations δ and δ' have an equivalent effect on the database state iff* $(\delta'.D_{del} \equiv \delta.D_{del}) \wedge (\delta'.D_{add} \equiv \delta.D_{add})$.

In the general case, this requires a satisfiability test between arbitrary logical expressions. Since we are actually dealing with relational algebra expressions, it is possible to implement a more efficient test that covers the most commonly occurring forms of expression.

To calculate the set U_δ, we must be able to determine the statements that are involved in definition-use relationships by means of the database state. In fact, previous work on program slicing in the presence of database state already provides a definition of these dependencies and a means of computing them [14]:

Definition 7 *A statement δ_1 is database state dependent on a statement δ_2 iff:*

1. $\delta_1.D_{read} \cap (\delta_2.D_{add} \cup \delta_2.D_{del}) \neq \emptyset$*, and*
2. *there is a rollback-free execution path p between δ_2 and δ_1 such that:*
   - $\delta_2.D_{add} \setminus p.D_{del} \neq \emptyset$ *or*
   - $\delta_2.D_{del} \setminus p.D_{add} \neq \emptyset$

We use the notation $\delta_1 \dashrightarrow \delta_2$ to denote the fact that δ_1 is database state dependent on δ_2. The definition states that this relationship is true for any pair of statements where the portion of the database state read by the dependent statement overlaps with the portion of the state updated by the other, and where no intervening statement acts to completely cancel out the overlap (either by causing a transaction rollback or by making compensating updates to the database state).

The required set of dependencies for a given modified statement can be computed by means of the following steps: (a) index into the graph to find all database statements; (b) determine which of these is potentially dependent on the modified statement; and (c) determine whether there exists a path between the modified node and the candidate dependent node.

In order to identify a dependency, we need to be able to perform tests for non-empty set intersections and non-empty set differences on the intensional set descriptions that we have extracted from the program code for the two statements in question. The first of these translates into a test to determine whether the two intensional set descriptions are satisfiable, to which the same comments given previously apply. The second corresponds to the question of whether one statement completely reverses the effect of the other, and is most easily specified on a case-by-case basis:

- Any update is completely cancelled by a rollback command
- An insertion to a relation r of a set of tuples which satisfy the condition c_i is completely cancelled by a deletion from r with condition c_d iff $c_d \Rightarrow c_i$.
- A deletion from a relation r of a set of tuples which satisfy the condition c_d is completely cancelled by an insertion to r with condition c_i iff $c_i \Rightarrow c_d$.

- A modification to a relation r where the set of tuples to be modified satisfy the condition c_m and the new tuples satisfy the condition c'_m is completely cancelled by a deletion from r with condition c_d and an insertion to r with condition c_i iff $c_i \Rightarrow c_m$ and $c_d \Rightarrow c'_m$.

From this we can compute the set of statements that are database state dependent on a given statement and therefore also the set of database–dependent tests.

Rather than computing the set of modification–traversing tests and the set of database–dependent test separately, it is more efficient if both can be computed together. We can achieve this by augmenting the original dangerous entity set with the statements that are database state dependent on some modified operation. This is trivial for statements which are present in P (the incoming edge of the statement will be added to the dangerous entity set). However, it is more difficult for statements that have been newly inserted into P'. In this case we must traverse the graph from the statement to the start selecting the first edge that exists in both P and P' in each path from the statement.

5. Optimising selection for database state

5.1. Avoiding redundant testing

The expansion of the dangerous entity set of Rothermel and Harrold [12] to include extra dependencies relative to the presence of database state represents a sacrifice of efficiency for safety. However, the well-defined and rich semantics of database access open up some possibilities for optimising the selection of test cases. To give a simple example, consider the case in which a statement that adds new tuples to a table is modified so that it is capable of adding only a subset of the tuples that could be added by the original statement. In this case, no new faults can be introduced by the modification, since any updates performed by the new program could also have been performed by the original version. Therefore, no regression testing is required for the database–aspects of this modification.

In general, we can categorise each modification of a database state as resulting in one of three effects:

- $\delta'.D_x$ is equal to $\delta.D_x$ iff $\delta'.D_x = \delta.D_x$.
- $\delta'.D_x$ is covered by $\delta.D_x$ iff $\delta'.D_x \subset \delta.D_x$
- $\delta'.D_x$ arbitrarily modifies $\delta.D_x$ iff $\delta'.D_x \not\subseteq \delta.D_x$

where x is either $read$, del or add. Based on these categories, for each modification we can determine whether or not reverification is required as follows:

	$\delta'.D_x = \delta.D_x$	$\delta'.D_x \subseteq \delta.D_x$	$\delta'.D_x \not\subseteq \delta.D_x$
$read$	$\times$	$\times$	$\surd$
del	$\times$	$\surd$	$\surd$
add	$\times$	$\times$	$\surd$

Composite operations (such as SQL UPDATE statements that both add and delete data from the database) must be reverified whenever any one of their component operations requires it.

By this means, we can reduce the number of tests selected for regression testing without reducing the safety of the selection algorithm.

5.2. Safety in only database state

A further means to increase the efficiency of the selection algorithm is available for applications in which the program state is used only as a temporary holding space for data that has been retrieved from, or is about to be written to, the database state. This is a very common style of programming in information systems. In such cases, safety of test selection relative to program state can be seen as an expensive luxury; the only form of safety that is critical to us is that relative to database state.

In order to improve the efficiency of regression test selection, therefore, we can focus the analysis solely on the operations that interact with the database state. The obvious way to accomplish this is to use only the set of tests selected due to their inclusion in database state dependencies, ignoring those selected due to modifications to statements that manipulate only program state. However, database operations are themselves dependent upon other statements within the system. For example, a statement may define a variable in program state that holds the value that a database operation later inserts into the database. This raises the question of whether faults can occur in the database state due to a modification of a statement defining a variable that stores a value read from or written to the database. For example, suppose we have a database–driven application P_{DB} that includes a database operation that inserts a variable v_1 of java datatype int into a column bound to the SQL datatype $integer$. The conversion between these java and SQL datatypes is valid and will not result in any errors. In program version P'_{DB} variable v'_1 is converted to the java datatype $double$. With JDBC, when a loss of precision occurs) a DataTruncation runtime exception is thrown by the program and the current transaction is aborted. This exception will result in a faulty database state which in turn may affect subsequent uses of that portion of database state.

Therefore, in order to select the tests necessary for safety in only database state we must determine if any statement that a database operation is dependent upon has been modified. This is accomplished using the concept of program–database dependencies [14] summarised as follows:

Definition 8 *The following dependency exists between program and database states:*

- *A database statement δ is dependent on a statement s if there exists some variable v such that:*

– *v is defined by s,*
– *v is used in δ, and*
– *there is a v-definition free path from s to δ.*

Using this criterion it is possible to determine the set of statements that the database operations of a program are dependent upon. For each database operation in P', we select every statement the operation is dependent upon that has either been newly introduced into P' or is a modified version of a statement in P. In the case in which the statement has been newly introduced (or for modified statements for which it is impractical to determine the original statement in P) we traverse the graph from the statement to the start selecting the first edge that exists in both P and P' in each path from the statement.

Therefore, the dangerous entity set for database state safety will contain:

- Every database operation that has been modified in P',
- Every database operation that is dependent upon a modified operation, and
- Every statement that has been modified in P' and is depended upon by a database operation.

Thus, by removing the statements that only influence program state we can create a more efficient test selection algorithm for cases where the expense of test selection relative to program state is unjustified.

6. Empirical study

In order to empirically determine the effectiveness of the techniques presented in this paper when applied to real world software systems, we have implemented them within a prototype tool, *DOT-select*. This tool is part of a larger Data–Oriented–Testing framework that is under development at the University of Manchester. We have used DOT-select in three empirical studies, following a multiple case study design. Here, we present the goals, design and results of the study.

6.1. Study goals and design

Our study utilises three open source systems: *Compiere*, *James*, and *Mp3CD Browser*. Each system is implemented in Java using JDBC for database access. MySQL version 4.1.12 was used as the database management system. The experiments were carried out on an AMD Athlon 2.4Ghz with 512Mb. Each test suite is of a realistic size with a high level of both statement and method coverage. Compiere (*http://www.compiere.org/*) is a software system for small-to-medium size enterprises that provides customer management, supply chain and accountancy functions created by ComPiere Ltd and released as open source. Compiere has a number of modules for performing various business functions as well as several interface components. James (The Apache Java Enterprise Mail Server) (*http://james.apache.org/*) is a Java based SMTP and POP3 Mail Server and NNTP News Server. James uses JDBC to process and store email and news messages. Mp3cd Browser (*http://mp3cdbrowser.sourceforge.net/*) is an application designed to manage an mp3 collection. Each version of the systems undertest were considered major releases by their associated developers (in comparison to minor bug fixes).

We consider four test selection techniques in this study: (1) *RetestAll*, this technique selects all test cases in T to be executed upon P'. We use this as the control technique; (2) *ProgramSafety*, this technique uses the approach of Rothermel *et al.* to select all test cases in T that are modification traversing in P'. We use this technique in order to determine the cost of attaining safety; (3) *CombinedSafety*, this technique implements the approach described in Section 4 to perform test case selection that is safe in both program and database state; and (4) *DatabaseSafety*, this technique implements the approach described in Section 5.2 to perform test case selection that is safe only in database state.

Similarly to existing empirical studies into regression test selection [7, 9, 11, 12] the research question we wish to answer in connection with both the CombinedSafety and DatabaseSafety criteria is: what is the effectiveness of the technique in terms of savings in testing effort? We utilise two measures of effectiveness: (1) the reduction in test-suite size and (2) the reduction of test suite (and analysis) execution time.

6.2. Study results

Figure 3 shows the test suite sizes that were produced in the experiment. In the graph, the horizontal axis shows the versions of each program in each case study, and the vertical axis shows the percentage of test cases selected by DOT-select. These results show that on average each technique results in a reduction of test suite size compared with the full test suite. As predicted, CombinedSafety always selected at least as many tests (and often more) than ProgramSafety — on average it showed a 10 percent increase in the number of test cases selected. Therefore, attaining safety of database state in addition to program state adds extra costs to regression testing. DatabaseSafety selected fewer test cases on average than the other three approaches: 42 percent compared to RetestAll, 59 percent compared to ProgramSafety, and 52 for CombinedSafety. Therefore, in situations in which database state is critical, DatabaseSafety can provide a significant saving in regression testing effort.

The benefits of a smaller test suite, however, will fail to materialise if the overall overall cost of selecting and then

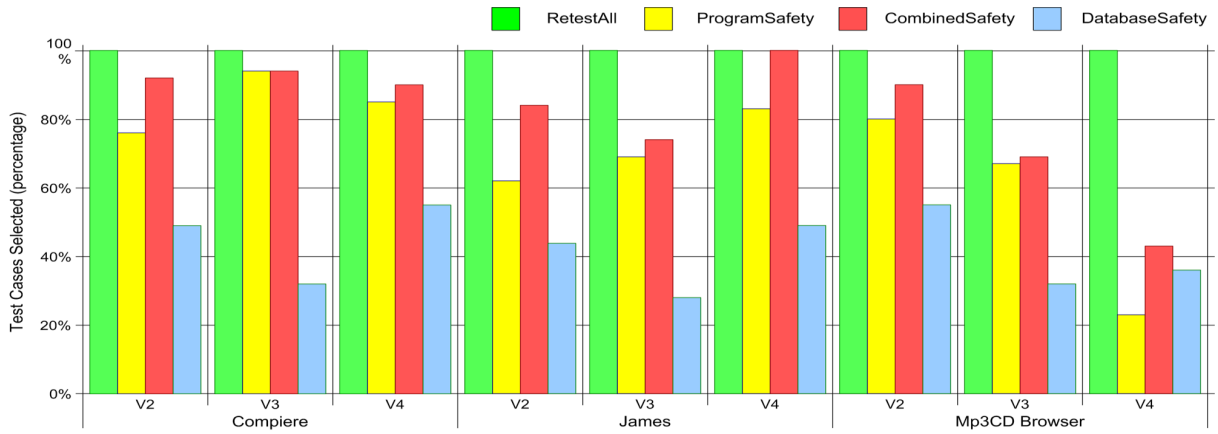

Figure 3. Percentage of test cases selected for each program version

executing the tests is higher than for the simple execution of all the test cases. Figure 4 shows the execution costs of each selection technique. In the graph, the vertical axis shows the percentage of the total running time for the three test selection algorithms as compared to RetestAll. These results show that on average each technique results in a reduction of execution time compared with RetestAll. CombinedSafety resulted in a longer execution time, when compared to ProgramSafety, due to the extra test cases selected and the extra analysis step. CombinedSafety's 23 percent saving for Mp3CD Browser shows that in certain situations it can be a beneficial technique. However, the -2 percent saving for Compiere shows that it can also be expensive. Therefore, in a general sense it can be observed that the savings possible with CombinedSafety are dependent on the types of program and changes made. However, DatabaseSafety produced significant savings when compared to the other techniques due to the smaller number of test cases selected.

7. Conclusions

Database–driven applications make up a very large proportion of the source code that is currently being maintained throughout the world. However, traditional regression test selection techniques are not tailored to the particular challenges of these systems. In order for safety of selection to be attained database state must be controlled. This may result in a number of test cases being omitted from selection that would expose possible faults that may exist as a result of the complex interactions possible between program and database states.

In this paper, we have presented the first safe regression selection algorithm for database–driven applications. This CombinedSafety approach utilises the explicit semantics present in database–driven applications to reason about the program's interactions with database state, and from this selects all modification–revealing test cases in terms of both program and database state. It allows us to determine possible points within the system that may be impacted by a modification to the source–code, even when they are not directly involved in the modifications themselves. We have also presented a secondary algorithm (DatabaseSafety) that performs selection based solely on the interactions with database state and so is more efficient but only safe in terms of database state.

We have implemented a tool *DOT-select* that performs both types of selection on systems designed in Java using JDBC for interfacing to the database state. The empirical evaluation we carried out using this tool has shown that both the proposed techniques result in a reduction in the number of test cases selected in comparison to the RetestAll approach. As expected, the CombinedSafety algorithm selects more test cases than the traditional technique we used as a benchmark, since it includes tests which take into account the persistent nature of the database state and the additional interactions this creates within and between programs. The compromise DatabaseSafety algorithm selects fewer test cases than the benchmark technique, thus allowing more efficient selection to be performed in the situations in which program state is only a temporary holding space for the manipulation of data.

Our work has presented a number of possible avenues for future work, of which the most urgent is the question of ordering of test cases within suites, in order to take into account the additional dependencies between tests that result from the persistent nature of the database state. Our approach has selected test cases that may be fault–revealing for a particular program, provided they are executed in the correct sequence. We are examining methods by which appropriate and efficient sequences of tests can be automatically selected. A further possibility is to make use of dynamic analysis in performing test case selection, both in im-

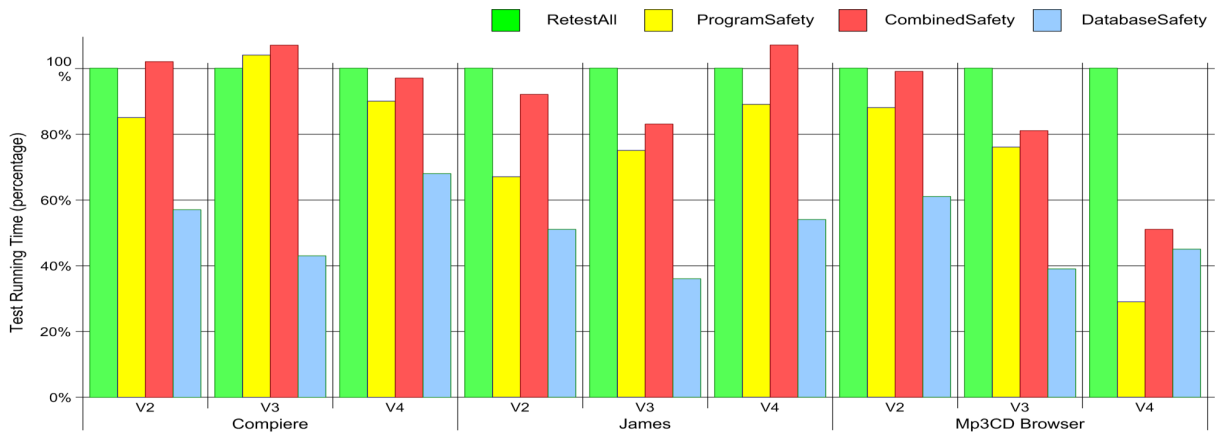

Figure 4. Overall time for regression testing (including both test suite execution and analysis time)

proving the efficiency of the complex set-based operations involved and in furthering the forms of optimisation that can be carried out, possibly during test suite execution. Finally, the additional complications of the presence of a changing database state in analysing and diagnosing the results of regression testing must also be addressed.

Acknowledgements

Thanks are due to Leonardo Mariani and the anonymous reviewers for their valuable comments and suggestions on earlier drafts of this paper. David Willmor is supported by a studentship from the EPSRC.

References

[1] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 384–396, 1993.

[2] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[3] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–157, August 2000.

[4] S. G. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification and Reliability*, 13(2):65–83, 2003.

[5] R. A. Haraty, N. Mansour, and B. Daou. Regression testing of database applications. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC), March 11-14, 2001, Las Vegas, NV, USA*, pages 285–289. ACM, 2001.

[6] R. A. Haraty, N. Mansour, and B. Daou. Regression test selection for database applications. In K. Siau, editor, *Advanced Topics in Database Research*, volume 3, pages 141–165. Idea Group, 2004.

[7] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 312–326. ACM, October 2001.

[8] H. Leung and L. White. A study of integraton testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290–300, 1990.

[9] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE)*, pages 241–251. ACM, October 2004.

[10] G. Rothermel. *Efficient, effective regression testing using safe test selection techniques*. PhD thesis, December 1995.

[11] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[12] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology*, 6(2):173–210, 1997.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[14] D. Willmor, S. M. Embury, and J. Shao. Program slicing in the presence of a database state. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 448–452. IEEE Computer Society, September 2004.