Fault-Based Regression Testing in a Reactive Environment

Debra J. Richardson

July 1989

Technical Report No.90-30

Information and Computer Science
University of California
Irvine, California 92717

Abstract

Regression testing is the process of retesting software after modification. Regression testing is a major factor contributing to the high cost of software maintenance. To control this cost, regression testing must be accomplished efficiently through effective reuse of test cases and judicious generation of new test cases.

Fault-based testing focuses on the detection of particular classes of faults. RELAY is a fault-based testing technique that guarantees the detection of errors caused by any fault in a chosen fault classification. RELAY can be used as a regression testing technique to generate the test cases required to demonstrate that a modification is properly made. In addition, the information related to a test case chosen to detect a potential fault guides in choosing previously-selected test cases that should be reused for a given modification.

This paper presents the concepts behind RELAY and discusses how RELAY could be used as a regression testing technique. It also describes a testing environment that supports reactive regression testing as well as testing throughout the development lifecycle, which is based on integrating the RELAY model with other testing techniques.

This work was supported in part by the National Science Foundation under grants CCR-8704311, CCR-8704478, and CCR-8996102, with cooperation from the Defense Advanced Research Projects Agency (ARPA Orders 6100, 6104, and 6108, Program Code 7T10).

1 Introduction

Regression testing is "selective testing to verify that modifications have not caused unintended adverse side effects or to verify that a modified system still meets requirements" [IEE83]. This process contributes heavily to the high cost of maintaining large scale software systems. Personnel charged with the task of regression testing are typically faced with the prospect of rerunning all test cases or making haphazard, if well intentioned, guesses about what test cases should be rerun. Moreover, reuse of existing test cases is seldom sufficient, and little guidance is provided as to what new test cases must be generated to test a modification. Regression testing technology is long overdue.

Cost effective regression testing involves selection of an appropriate subset of the current test case set as well as astute generation of new test cases to exercise the modification. To determine appropriate test case reuse, we must retain test cases in a form that facilitates identifying those that exercise software constructs affected by the modification. To further reduce costs, we should retain and reuse analysis results from the original [previous] validation process to reanalyze the modified software and select new test cases.

Regression testing is a time-consuming activity that must be automated and should be triggered when a modification is complete. A software engineering environment for large scale software should be both proactive and reactive — that is, it should automatically perform certain activities for software developers, especially those that do not require developer interaction, and it should react to software developers' activities by initiating other activities. Regression testing should be an automated, reactive process that is initiated after software modification.

Fault-based testing selects test data geared toward detecting particular fault types, where a fault is a specific mistake in the source code. Fault-based testing techniques have a common underlying theme: distinguishing the test program from alternatives in a set of related programs that differ by defined fault types. Fault-based testing is capable of detecting most faults resulting from subtle errors of commission, which are typically revealed only for very specific data, although it does not, in general, detect errors of omission.

The RELAY model provides a fault-based testing technique based on a framework for describing faults in software and a mechanism for developing conditions that guarantee their detection. RELAY can guarantee that faults in a selected fault classification are detected or do not exist. The only way to guarantee fault detection, however, is to target all possible fault classes. In real situations, this is impractical, acn faults will persist after what many would consider "pretty thorough" testing. Maintenance is bound to be required as software failures are revealed after delivery.

RELAY provides an effective regression testing technique. With knowledge of source code modification, RELAY can identify test cases to reuse because they test software constructs potentially affected by the modification. In addition, RELAY develops conditions to determine whether the software has been modified properly. Both steps are more cost effective by retaining appropriate artifacts of previous applications of RELAY to the software being maintained. We are in the process of implementing a fault-based test data selection tool based on the RELAY model and intend to incorporate appropriate regression testing technology.

The next section summarizes the RELAY model and outlines how RELAY selects test data; a more detailed presentation appears elsewhere [RT88b]. The third section describes how RELAY provides an effective regression testing technique. In conclusion, we discuss RELAY's status and future research directions.

2 RELAY as a Fault-Based Testing Technique

Fault-based testing techniques consist, in some sense, of "fault-specific rules", each intended to detect a particular fault type. Myers outlines many fault-based heuristics [Mye78].

More formal fault-based techniques have a common underlying theme: distinguishing the test program from alternatives in a set of related programs that differ by defined fault types. These techniques assume the test program is "almost correct" and differs from some hypothetical correct program by at most some definable faults (the *competent programmer hypothesis* [DLS78]). This near correctness might be determined by successfully passing some high-level functional testing phase or by satisfying some structural testing criterion. Several fault-based testing techniques have been proposed [Bud83, DLS78, Fos80, Ham77, How82, How86, Mor84, DGK$^+$88, Zei84, Zei89]. A survey of fault-based testing techniques is beyond the scope of this paper (see [RT89]). Analysis has demonstrated that fault-based testing based on the RELAY model is more effective than other fault-based test data selection techniques [RT86].

2.1 The RELAY Model

RELAY is a fault-based testing technique that generates test data guaranteed to detect specific classes of faults. It does so by developing revealing conditions that guarantee that a fault *originates* an incorrect intermediate value that is *transferred* through computations and data flow until a failure is revealed. Note that a *fault* is a syntactic discrepancy in the source code and a *failure* is observable incorrect behavior. These ideas are captured in the RELAY model.

RELAY starts with a source code expression that contains a discrepancy from some correct module[1]. It is possible to mask such a discrepancy during execution; output appears correct but just by coincidence of the test data selected (often referred to as *coincidental correctness*). It is also possible that although a discrepancy exists between the test module and some correct module, the two are equivalent. In general, we do not know whether a discrepancy can cause a failure, and thus it is only potentially a fault. A **potential fault** is

[1]As we shall see later, the actual fault need not be known in advance.

a syntactic discrepancy between the test module and some correct module. Potential fault execution may introduce incorrect intermediate values, but later computations may mask the incorrect intermediate value before it is observed. An incorrect intermediate value is thus referred to as a **potential failure**.

The RELAY model determines test data requirements that must be satisfied for a potential fault to introduce a potential failure, carry it throughout execution, and eventually produce a failure. Given a potential fault, a potential failure **originates** if the smallest subexpression containing the potential fault evaluates differently from the corresponding subexpression in the correct module. After a potential failure originates, it must **transfer** through module execution to affect subsequent computations and eventually the output. There are two types of transfer. **Computational transfer** refers to transfer of an incorrect result through a statement that uses incorrect intermediate values. **Data flow transfer** refers to transfer from an incorrect variable assignment to a use of that variable. When a potential failure transfers to the outermost expression in a statement, a **context failure** results. Context failures result both at the originating statement and at subsequent statements along a chain of definitions and uses to the output. When a potential failure transfers through all computations and data flow to reach an output, a **failure** results[2].

The RELAY model, so-named becuase it resembles a relay race, is summarized in Figure 1. As shown in the figure, we start with a potential fault. The potential fault must first originate a potential failure (1). The potential failure must computationally transfer through all ancestor operators in the statement containing the potential fault (2), after which a context failure is revealed (3). This context failure, which is manifested as an incorrect value for some variable, is transferred by data flow to some use of this variable (4), resulting in a potential failure at the statement where the use occurs. Again, the potential failure must transfer through all computations in the statement. This process of trans-

[2]Other failure types include fatal run-time failures and deadlock. We are currently concentrating on revealing output failures.

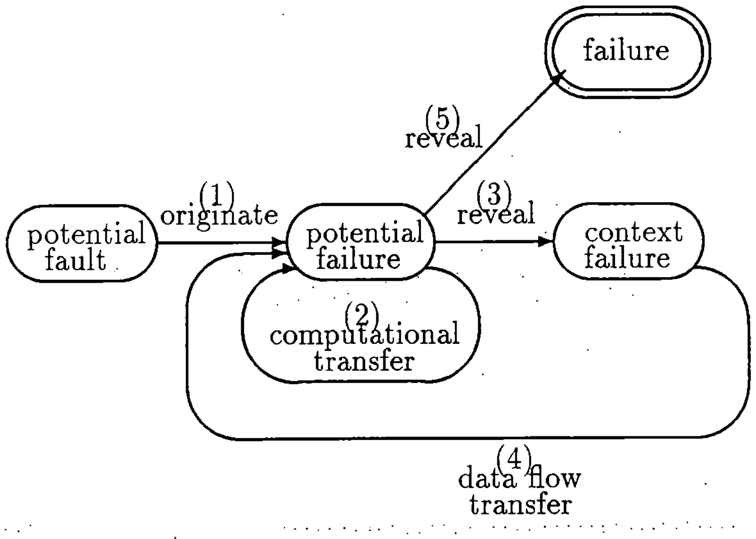

Figure 1: The RELAY Model

ferring through all computations at a statement to produce a context failure followed by transfer of the context failure through data flow to some other statement (2,3,4) continues until a statement is reached where the potential failure is revealed as a failure (5).

The model described above details how a particular fault causes a failure to be revealed, which seems to require prior knowledge of the existence of a fault. On the contrary, RELAY selects test data that produces failures for specific classes of faults that might occur in the code. RELAY assumes that the test module is "almost correct" and considers how it might differ from a hypothetical correct module. RELAY considers that any statement is potentially faulty and hypothesizes what potential faults could exist in the code. Hypothesizing that a module contains a potential fault in some expression means that a hypothetical correct module contains an alternate expression that is correct. RELAY selects test data that guarantees the test module and the hypothetical correct module behave differently. Based on the ideas of origination and transfer, RELAY constructs the conditions that are necessary and sufficient to guarantee a failure occurs if the fault exists.

First, a potential failure must originate, and then computationally transfer to affect evaluation of the originating statement producing a context failure. The **origination condition** is the necessary and sufficient condition to guarantee that the smallest subexpression containing the potential fault and the alternate subexpression evaluate differently. The **computational transfer condition** guarantees that a potential failure transfers through all ancestor operators in the statement by distinguishing each ancestor expression referencing a potential failure from the ancestor expression referencing the evaluation of the alternate subexpression. The **context failure condition** at the originating statement, is the conjunction of the origination condition and the computational transfer condition.

From here, the context failure must transfer to some output statement where a failure is demonstrated. There may be many routes along which the potential failure may transfer. Each route is defined by a chain of alternating definitions and uses, *def-use pairs*, where each definition reaches the next use in the chain and that use partially defines the next variable in the chain. A **data flow transfer condition** describes the requirements for transfer of a context failure from a definition in the chain to the next use (e.g., execution of a def-use pair). To transfer a potential failure along a selected chain, the data flow transfer conditions for all def-use pairs in the chain must be satisfied and, at each use in the chain, the computational transfer condition to guarantee the use of the potential failure affects the next definition must also be satisfied. The **chain transfer condition** for a selected chain is the conjunction of the data flow transfer conditions for the def-use pairs along the chain and the computational transfer conditions required by each use in the chain.

The conjunction of the context failure condition at the originating statement along with the chain transfer condition forms a sufficient **failure condition**. If test data can be selected to satisfy this failure condition and the module executes correctly, then the potential fault is <u>not</u> a fault. If test data that satisfies the condition produces a failure, then the module contains the hypothesized fault. If we are unable to satisfy this failure condition, then we must consider other routes along which the potential failure could transfer to output. The disjunction of the sufficient failure conditions for all chains from the originating statement to a failure is both necessary and sufficient to reveal a failure due to this potential fault. If this disjunction is unsatisfiable, then it is not possible to transfer the potential failure along any of the routes. This means that the potential fault and the alternate are equivalent, and the potential fault is not a fault.

Consider the module shown in Figure 2, and suppose that the reference to U at statement 1 should be a reference to B — that is, statement 1 should be $X := B * V$. The

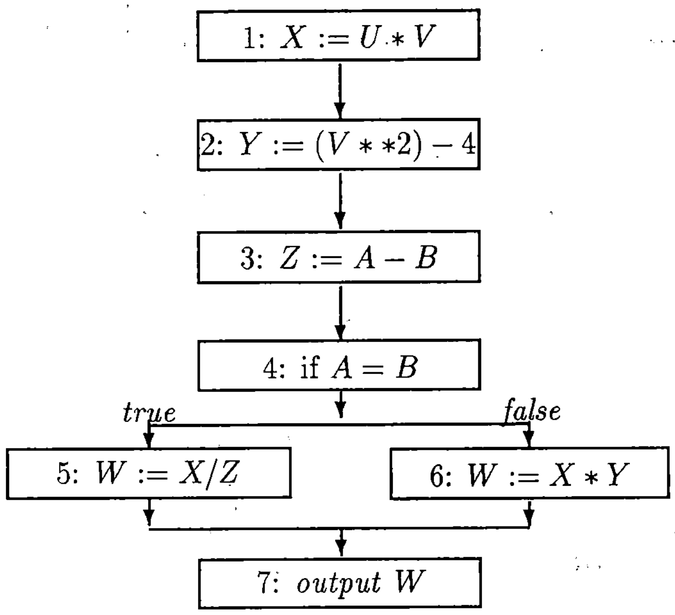

Figure 2: Test Module

origination condition for this potential fault is $(u \neq b)$. The only computation at this statement is multiplication by V. The transfer condition through multiplication is that the other operand (the one that does not contain a potential failure) is not zero-valued.

When applied to this statement, the computational transfer condition is $(v \neq 0)$. Thus, the context failure condition at the originating statement resulting from this potential fault is the conjunction of these conditions — $(u \neq b)$ and $(v \neq 0)$. From here, we consider the chains to output that use X's value defined at statement 1. There are two such chains. Consider first the chain consisting of the use of X at statement 5, where W is defined, followed by the output of W at statement 7. The data flow transfer condition is $(a = b)$. The computational transfer at statement 5 requires that Z be nonzero, which results in the computational transfer condition $(a - b \neq 0)$, since $z = (a - b)$. Thus, the chain transfer condition is $(a = b)$ and $(a - b \neq 0)$, which is infeasible. The context failure cannot transfer along this chain, therefore, and another chain must be considered. The second chain consists of the use of X at statement 6, where W is defined, followed by the output of w at statement 7. For this chain, the data flow transfer condition is $(a \neq b)$. Computational transfer at node 6 requires that Y be non-zero. Since $y = (v{*}{*}2) - 4$, the computational transfer condition is $((v{*}{*}2) - 4 \neq 0)$, which simplifies to $v \neq \pm 2$. Thus, the chain transfer condition is $(a \neq b)$ and $(v \neq \pm 2)$. The sufficient failure condition for this chain is

$$(u \neq b) \text{ and } (v \neq 0) \text{ and } (a \neq b) \text{ and } (v \neq \pm 2)$$

This condition is satisfied by the test datum $(a = 1, b = 2, u = 1, v = 3)$, which would reveal a failure caused by the hypothesized fault.

2.2 RELAY Application

Using the RELAY model to select test data may seem time and resource consumptive, but we do not intend for it to be applied blindly to test for all fault classes at all locations. We are designing a user model based on process programming [Ost87, RAO89], whereby a user can choose a RELAY criterion, which specifies a group of source code locations and class(es) of potential faults that may occur at those locations. For example, one such

RELAY criterion is variable reference faults for the entire program; another is conditional operator faults in loop conditions. The RELAY tool derives the appropriate revealing conditions for a given criterion and selects test data to satisfy those conditions. In this section, we discuss the implementation of a RELAY tool and describe the steps involved in applying RELAY.

The RELAY model, itself, is generic — that is, it describes generic model conditions for origination and transfer that are instantiated for specific faults. To derive revealing conditions, the RELAY tool employs fault specific origination and transfer conditions. In deriving the fault specific conditions, we group faults into classes based on a common characteristic transformation. We then instantiate fault class origination conditions for each fault class as well as computational transfer conditions for each operator whose operands may be a potential failure caused by a fault in the class. The origination conditions and computational transfer conditions have been instantiated for the following fault classes: boolean operator fault, relational operator fault, arithmetic operator fault, variable reference fault, constant reference fault, and variable definition fault (see [RT88a]). Fault classification is useful, because there is often substantial overlap amongst the origination conditions for the potential faults in a class. Hence, the generation of origination conditions for each fault in a class is similar, and a single test datum often satisfies multiple origination conditions. Moreover, the failure conditions for the faults in the class differ only in origination condition, sharing identical transfer conditions (computational as well as data flow).

The RELAY tool contains the fault class origination and computational transfer conditions. When testing with the RELAY tool, the user specifies a RELAY criterion, which dictates testing for some fault class(es) at some source code locations. The tool identifies each potential fault, which consists of a particular fault class and an applicable location, specified by the chosen criterion. Note that a RELAY criterion may specify a number of fault classes and locations at a single statement. The RELAY tool reduces costs by isolating those parts of the revealing conditions that are independent of a potential fault and reuses these artifacts for other potential faults (this will also prove useful in regression testing). Derivation of the context failure condition (both origination condition and computational transfer conditions) at the originating statement is specific to a particular fault class and location. Contrarily, the transfer of a context failure from the originating statement along a chain to produce a failure is independent of a particular fault; each fault at the statement may transfer along the same chain. Thus, chain transfer conditions are developed independently of the context failure conditions.

For each potential fault, the tool must select and interpret an initial path from the starting node to the potential fault. At this point, the tool derives the context failure conditions. This requires evaluating the appropriate fault class origination conditions at the fault location to provide the actual origination conditions, and then evaluating the applicable transfer conditions for each ancestor operator in the originating statement. The computational transfer condition is conjoined to each origination condition to create context failure conditions for the class of potential faults.

Next, the tool derives the chain transfer condition for a selected transfer route — a chain of alternating definitions and uses from the originating statement to output. In addition, an activating path that traverses these def-use pairs is selected[3] and will be interpreted incrementally as the chain transfer condition is constructed. A chain is selected by analyzing a flow graph annotated with def-use pairs. For each def-use pair along the route, the data flow transfer condition is determined by evaluating the required conditional transfers between the definition and the use, and the computational transfer condition is determined by evaluating the applicable transfer conditions for the ancestor operators of the use. The same chain transfer condition is conjoined to each context failure condition

[3]Note that the activating path lists all nodes on a path from the originating node to output while the transfer route lists only those where transfer occurs.

at the originating statement to provide a failure condition.

Finally, the RELAY tool is used to evaluate pre-selected test data and/or to select test data. Since the RELAY model of fault detection assumes that the module being tested is almost correct, the module should have passed some other testing phase. This may simply be user-selected test data. We have also been investigating the integration of RELAY with other automated testing techniques [RAO89]. The tool, therefore, first determines what failure conditions are satisfied by any pre-selected test data. Test data is then selected for any failure conditions not yet satisfied. Augmenting a pre-selected test data set is more efficient, because determining that a condition is satisfied is less costly than solving that condition and retesting.

The RELAY testing tool is one of the inhabitants of TEAM [CRZ88], a support framework for extensible integration of testing, evaluation, and analysis techniques. The TEAM framework provides the essential building blocks, through generic component technology, for easily constructing new tools. Two important TEAM components are ARIES [ZE88], a generic interpretation tool that provides symbolic evaluation capabilities for evaluating the revealing conditions, and a formal reasoning component, which is used for determining feasibility of the revealing conditions and solving the failure conditions to provide test data. Effective coordination of components within TEAM depends on support from the ARCADIA environment architecture [TBC+88]. ARCADIA provides object management facilities, which enable RELAY to retain the artifacts it develops. Another essential feature of the ARCADIA environment is process programming [Ost87]. The operators of a process program are tool components and the operands are the artifacts created by those tools and the users. We have designed RELAY as a process program, which facilitates integration of RELAY with other techniques and provides for easy modifications to the process based on our experience [RAO89]. This approach also allows extension to the process to support such activities as regression testing.

3 RELAY Regression Testing

RELAY can be used as a fault-based regression testing technique applied reactively after software modification. After a modification, RELAY must provide the same (or better) confidence in software reliability as would be achieved if we "re-RELAY'ed" with the chosen RELAY criterion. With appropriate retention and reuse of the artifacts generated in previous applications, RELAY can effectively validate a software modification without need for reanalyzing the entire software. This section describes how RELAY can be used for regression testing and how the RELAY artifacts can be reused in this process.

Maintenance and modification differ in response to arising needs for change [Swa76]. *Corrective maintenance* entails changing software to correct failures discovered after system delivery. Regression testing must determine that the modification is correct and that there are no adverse side effects. *Adaptive maintenance* involves modifying software in response to changing requirements. Regression testing must validate that new (and remaining) requirements are met, much the same as is done in testing during development. *Perfective maintenance* covers enhancements to improve software; it does not involve change to functional requirements but may modify non-functional requirements. Regression testing must ensure that functional requirements are still met and enhancement requirements are satisfied.

In as much as the implementation is changed for each maintenance type, RELAY has regression testing applications for all three types. After any modification, RELAY can be employed to identify the test cases that exercise those portions of the source code that are affected by the modification. In its current stage, RELAY is especially useful to select new test cases after corrective maintenance since these modifications tend to "fix bugs", much like faults in the classes for which RELAY is currently applied. RELAY is also useful after simple perfective maintenance to ensure that functionality remains the same. We are working on extended application of RELAY that would cover more abstract faults and specification-based testing, which would be readily applicable for extensive perfective and adaptive maintenance.

To effectively accomplish both the choice of test cases to reuse and the selection of new test cases, RELAY retains artifacts created during its application for reuse[4]. This is accomplished through an object management system in the environment architecture that supports persistence of typed objects and relationships between these objects [TBC+88].

The basic program structure used by RELAY is a control flow graph whose nodes are represented by a syntax-tree-like internal representation. This structure is augmented with relationships required for RELAY's analysis and relationships that retain the RELAY artifacts. The list below outlines these relationships, where $\rightarrow$ indicates a one-to-one relationship, $\Rightarrow$ indicates a many-to-one relationship. Note that some information is redundant and is not actually stored but rather derived from other relationships.

- def-use pair $\Rightarrow$ node representing a definition
- potential fault $\rightarrow$ (node, internal-rep location, fault class) as indicated by RELAY criterion
- potential fault $\Rightarrow$ node
- origination condition $\rightarrow$ potential fault
- computational transfer condition $\rightarrow$ potential fault
- context failure condition $\rightarrow$ potential fault
- initial path $\rightarrow$ potential fault
- transfer route $\rightarrow$ potential fault
- def-use pairs $\Rightarrow$ transfer route
- activating path $\rightarrow$ transfer route
- potential failure $\rightarrow$ (node, internal-rep location)
- computational transfer condition $\rightarrow$ potential failure

[4]These artifacts are also reused to benefit the development validation process.

- data flow transfer condition $\rightarrow$ def-use pair
- chain transfer condition $\rightarrow$ transfer route
- failure condition $\rightarrow$ (potential fault, transfer route)
- test case $\Rightarrow$ failure condition[5]
- test case $\rightarrow$ (input data, expected output, ...)[6]

After software modification, the RELAY regression testing process, which is triggered by the maintenance process, consists of:

1. Retest all retained test cases that remain valid and are affected by the modification;
2. Generate and test a new test case that distinguishes the modified software from the previous version;
3. Generate and test new test cases for potential faults whose test cases were invalidated.

Each step involves extensive reuse of RELAY artifacts. Naturally, all newly developed artifacts are retained in the same manner as described above.

When a software modification is made, we must first determine what RELAY artifacts are no longer valid. This is more complicated than it might appear. Software modifications cause changes to the control flow graph structure or to the intermediate representation of the statements. A change to a statement, insertion of a new statement, or deletion of a statement anywhere along the initial path or activating path may invalidate a RELAY artifact and not only a change to the originating statement or subsequent statements on the transfer route. This is because modification of a statement along either path may

[5]One test case may satisfy multiple failure conditions, although this must be determined per condition. RELAY avoids redundant test cases.

[6]We are not so concerned here with what constitutes a test case but rather its association to revealing conditions.

change a variable's value that is used in computational or data flow transfer condition, change the def-use pairs, or change the path condition.

The test cases that remain valid (i.e., none of the artifacts leading up to the derivation of the test case are made invalid) are candidates for reuse. A test case that also exercises some construct in the software that is potentially affected by the modification should be rerun to determine that there are no adverse side effects of the change. The originating statement and the subsequent statements on the transfer route associated with a test case identify the constructs covered by the test. Any modification to one of these statements may adversely affect the other statements along the transfer route. Thus, RELAY uses the originating statement and transfer route to identify test cases that should be reused.

The retention of RELAY artifacts also facilitates the generation of new test cases for regression testing a modification. RELAY explicitly generates a test case to differentiate the modified software from the previous version by generating a failure condition for the modification. If we are doing corrective maintenance, this test case should produce a failure for the previous version but correct results for the modified software. On the other hand, if we are doing perfective maintenance, we should not be able to differentiate the two versions as they should be equivalent functionally.

To generate a failure condition, RELAY may reuse many of the retained artifacts. An initial path leading up to the modification location may be retained, or one requiring additional statements be appended may exist; such an initial path may be reusable. An origination condition must be generated for the smallest subexpression containing the modification[7] The computational transfer condition at the originating statement must also be derived. If the originating statement lies along some retained transfer route or a trans-

[7]Note that had this modification been tested as a potential fault, we would not be in a maintenance situation, thus there is no origination condition to reuse. It may not have been chosen in the RELAY criterion, or it may be a fault classification that RELAY for which RELAY does not apply (this does not hinder the application here).

fer route emanating from it merges with a retained transfer route, then the retained chain transfer condition may be reusable in part or in whole.

In addition to generating a new test case to distinguish the modified software from the previous version, the RELAY artifacts are used to generate new test cases to test constructs that are affected by the modification. We want to assure the same confidence in the software as would be provided by a complete retesting based on the chosen RELAY criterion. If the modification occurs along some activating path and invalidates the failure condition for a potential fault, this potential fault must be retested. What was not a fault in the previous version may now be a fault. RELAY generates a failure condition to guarantee that no adverse affects have occurred. For this potential fault, the context failure condition and the chain transfer condition preceding the modification may be reused as might some of the data flow and computational transfer conditions following the modification.

As an example, consider again the module in Figure 2. Suppose that our RELAY criterion was all variable reference faults. Suppose we now modify the source by replacing the multiplication operator in statement 1 by an addition operator; we did not explicitly tested for this fault. We showed the failure condition for one potential fault — U replaced by B at statement 1 — in the previous section. The test case associated with the potential fault mentioned above is invalidated because the computational transfer required at statement 1 has changed. The computational transfer condition for the modification is *true*, and the chain transfer condition can be reused, so the failure condition for this potential fault is

$$(u \neq b) \text{ and } \mathit{true} \text{ and } (a \neq b) \text{ and } (v \neq \pm 2)$$

This condition is still satisfied by the test datum $(a = 1, b = 2, u = 1, v = 3)$, which would reveal a failure caused by the hypothesized fault. We must rerun this test case to determine whether the modification has adverse effects. We would follow this same process for any other potential faults affected by this change.

We must also create a context failure condition for this modification, which is $(u * v \neq$

$u + v$), but can use the same chain transfer condition as described for the other potential fault at statement 1. Thus, the failure condition for the modification is

$$(u * v \neq u + v) \text{ and } (v \neq 0) \text{ and } (a \neq b) \text{ and } (v \neq \pm 2)$$

This condition is satisfied by the test datum ($a = 1, b = 2, u = 1, v = 3$), which will distinguish the modified software from the previous version. Note that this test datum is the same as the one selected above and will be associated with both failure conditions.

The separation of origination and transfer in the RELAY model facilitates the reuse of RELAY artifacts in the regression testing (as well as in development testing). The details of these regression testing selection mechanisms are beyond the scope of this paper and are being described formally in terms of the RELAY model. Moreover, their implementations are being designed as we progress with the development of a RELAY testing tool. Finally, we are designing the regression testing capabilities to be triggered by the maintenance process program when modification is complete. There is very little, if any, maintainer interaction required in the process described above.

4 Conclusion

In this paper, we demonstrate the use of the RELAY model of fault detection as a regression testing technique. RELAY models detection of a fault by origination of an incorrect intermediate value that transfers through execution until a failure is revealed. To test a program, a RELAY criterion is selected, which specifies fault classes and potential fault locations in the source. RELAY provides revealing conditions that guarantee detection of the faults identified by the RELAY criterion. The regression testing process based on RELAY reuses many of the artifacts developed during previous applications of RELAY for the previous version of the software. The process consists of 1) retesting all test cases affected by the modification that still satisfy the RELAY criterion, 2) generating a new test case and testing to distinguish the modified software from the previous version, and 3) generating new test cases and testing for potential faults identified by the RELAY criterion but are not covered by (1). This process guarantees the same confidence in software reliability as would a complete retesting based on the RELAY criterion.

We continue to extend the RELAY model of fault detection. We are evaluating its generality by instantiating it for other classes of faults, including more complex and higher level faults. Our current investigation of data flow transfer focuses on more complex def-use chains: those that include a statement that uses more than one potentially incorrect variable and those that cover looping constructs.

Implementation of a testing tool based on the RELAY model is currently underway. This tool is part of the TEAM framework [CRZ88] for testing, evaluation and analysis and makes use of the generic analysis components provided in that framework as well as basic components in the ARCADIA environment [TBC$^+$88]. Within the TEAM framework, we are designing an overall methodology for testing that is based on integrating RELAY with other testing techniques. We have designed the integration of RELAY with data flow testing through the use of process programming as it is supported in the ARCADIA environment [RAO89]. We intend to support all phases of the software lifecycle. This paper shows how RELAY can be used to support regression testing. An approach to doing incremental data flow testing after software modification has been proposed [HS88]; we intend to evaluate how this process can be integrated in the same fashion as data flow testing and RELAY were integrated. We are also examining the application of RELAY within an integration testing paradigm by considering the conditions that must be satisfied to guarantee transfer of a potential failure across a procedure invocation. In addition, we are considering the use of the RELAY model as a specification-based testing technique [ROT89]. Specification-based testing is useful in regression testing when the requirements are modified and in integration testing. In all of these efforts, we are stressing reuse of previous analysis and artifacts as well as reuse of generic component tools.

References

[Bud83] Timothy A. Budd. The portable mutation testing suite. Technical Report TR 83-8, University of Arizona, March 1983.

[CRZ88] Lori A. Clarke, D.J. Richardson, and S.J. Zeil. Team: A support environment for testing, evaluation, and analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, Massachusetts, November 1988.

[DGK+88] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. An extended overview of the mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

[DLS78] Richard DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 4(11), April 1978.

[Fos80] Kenneth A. Foster. Error sensitive test case analysis (estca). *IEEE Transactions on Software Engineering*, SE-6(3):258–264, May 1980.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[How82] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(2):371–379, July 1982.

[How86] William E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, October 1986.

[HS88] M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance 1988*, 1988.

[IEE83] Software Engineering Technical Committee of the IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology, Standard 729-1983*, 1983.

[Mor84] Larry J. Morell. *A Theory of Error-Based Testing.* PhD thesis, University of Maryland, April 1984.

[Mye78] Glenford J. Myers. *The Art of Software Testing.* Wiley-Interscience, 1978.

[Ost87] Leon Osterweil. Software processes are software too. *9th International Conference on Software Engineering*, 1987.

[RAO89] Debra Richardson, Stephanie Leif Aha, and Leon Osterweil. Integrating testing techniques through process programming. In *Proceedings of the ACM SIGSOFT '89: Third Symposium on Testing, Verification, and Analysis*, page (to appear), Key West, FLorida, December 1989. ACM Press.

[ROT89] Debra Richardson, Owen O'Malley, and Cindy Tittle. Approahces to specification-based testing. In *Proceedings of the ACM SIGSOFT '89: Third Symposium on Testing, Verification, and Analysis*, page (to appear), Key West, FLorida, December 1989. ACM Press.

[RT86] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the relay model of error detection. Technical Report 86-65, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[RT88a] Debra J. Richardson and Margaret C. Thompson. Relay: A model of fault detection. Technical Report 88-125, Computer and Information Science, University of Massachusetts, Amherst, December 1988.

[RT88b] Debra J. Richardson and Margaret C. Thompson. The relay model of error detection and its application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

[RT89] Debra J. Richardson and Margaret C. Thompson. The RELAY of fault-based testing. Technical Report 89-17, Information and Computer Science, University of California, Irvine, March 1989.

[Swa76] E. Swanson. The dimensions of maintenance. In *Proceedings of the Second International Conference on Software Engineering*, 1976.

[TBC+88] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the arcadia environment architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, Massachusetts, November 1988.

[ZE88] Steven J. Zeil and Ed. C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the Tenth International Conference on Software Engineering*, April 1988.

[Zei84] S.J. Zeil. Perturbations testing for computation errors. In *Proceedings of the Seventh International Conference on Software Engineering*, March 1984.

[Zei89] Steven J. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, SE-15(6):737–746, June 1989.