The RELAY of Fault-based Testing

(Technical Report 89-17)

Debra J. Richardson†
Margaret C. Thompson‡

March 1989

†Information and Computer Science
University of California
Irvine, California 92717

‡Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Fault-based testing focuses on the detection of particular classes of faults. RELAY is a fault-based testing technique whose model resembles a relay race. Analysis has shown that RELAY overcomes the weaknesses of other fault-based testing techniques.

RELAY defines *revealing conditions* that guarantee that a fault *originates* an error during execution and that the error *transfers* through computations and data flow until a failure is *revealed*. This model of error detection provides a fault-based criterion for test data selection. The model is applied by choosing a fault classification, instantiating the conditions for the classes of faults, and evaluating them for the program being tested. Such an application guarantees the detection of errors caused by any fault of the chosen classes. As a formal model of error detection, RELAY provides the basis for an automated testing tool. This paper presents the concepts behind RELAY, discusses how RELAY could be used as the foundation for a test ng system, and compares RELAY to other fault-based testing techniques.

This work was supported by grants DCR-8 8404217 from the National Science Foundation, CCR-8704311 (ARPA o.) and CCR-8704478 (Arpa Order No. 6104) from the National Science Foundation with cooperation from the Defense Advanced Research Projects Agency, 84M103 from Control Data Corporation, and F30602-86-C-0006 from the Rome Air Development Center.

1 Introduction

Testing is intended to reveal failures in program execution or to provide confidence that failures do not occur. This is typically done by selecting test data expected to cause erroneous execution should faults exist in the program. A testing technique can be classified as "error-based" or "fault-based". An error-based technique is geared toward revealing specific error types, where an error is an erroneous result produced by program execution. Fault-based testing selects test data by focusing on detecting particular fault types, where a fault is a particular mistake in the source code. Fault-based testing is capable of detecting many of the subtle errors of commission that are revealed only for very specific data, although it does not, in general, detect errors of omission. Moreover, fault-based testing facilitates debugging.

Fault-based testing, when done accurately, can guarantee that faults are detected or do not exist. We analyzed several fault-based test data selection techniques, however, and demonstrated two major failings of these techniques [RT86a]. First, most techniques do not consider the conditions required to guarantee that a fault actually reveals observable erroneous behavior. Instead, the "fault-specific rules" that comprise these techniques may introduce an erroneous intermediate value caused by a corresponding fault, but do not guarantee that such a value affects the output. Second, in most cases, these rules are merely sufficient (but not necessary and sufficient) to introduce an error. When such a fault-specific rule is unsatisfiable, a corresponding fault will not necessarily cause erroneous execution, and thus may remain undetected.

This paper reports on the RELAY model of testing, which remedies the weaknesses that remain in other fault-based testing techniques. The RELAY model provides a framework for describing faults in software and a mechanism for developing conditions that guarantee their detection. As such, RELAY is a basis for a fault-based test data selection tool.

The next section surveys related works in fault-based testing and indicates their major weaknesses. The third section summarizes the RELAY model; a more detailed presentation appears in other papers [RT86b, RT88b]. The fourth section outlines how the model is used to select test data and describes a RELAY-based testing tool. In conclusion, we highlight how RELAY overcomes the weaknesses in other fault-based testing techniques, discuss RELAY's status, and outline our future research directions.

2 Related Fault-based Testing Work

Fault-based testing techniques consist, in some sense, of "fault-specific rules", each intended to detect a particular fault type. Fault-based heuristics have been used by testers since the dawn of programming. Such heuristics are employed by examining the source code and selecting test data sensitive to commonly-introduced faults. Myers outlines many such heuristics [Mye78].

The attempts to formalize fault-based testing 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. In its various forms, this assumption essentially implies that the hypothetical correct program is in the "neighborhood" of the test program. Differences between the two can be detected by considering the alternative programs in that neighborhood that are associated with the faults the technique considers. If the class of faults is broad enough — that is, the neighborhood is large enough — we gain confidence in the tested program.

Formal fault-based testing approaches fall into two categories: those that evaluate preselected test data adequacy and those that guide test data selection. In what follows, we first discuss several fault-based test data evaluation techniques and then describe several fault-based test data selection techniques. It is beyond the scope of this paper to fully compare these techniques. A more thorough survey of fault-based testing appears elsewhere [RT88a].

The earliest formalized fault-based testing techniques were introduced independently by Hamlet and by DeMillo, Lipton and Sayward. Both techniques *seed* or substitute particular types of faults into the test program and evaluate a user-selected set of test data adequacy in terms of its ability to detect the seeded faults. Hamlet's *testing with the aid of a compiler* [Ham77], seeds faults that are alternative expressions "smaller" than the original expression in the source code. An extended compiler instruments the code to compare the values computed by each alternate and the corresponding original expression. *Mutation analysis* [DLS78], introduced by DeMillo, Lipton, and Sayward, seeds simple, single-token faults into the source code to produce "mutant" programs. The system then executes the

original and mutant programs on the pre-selected test data and determines which mutants are "killed" — that is, which produce different output results from the original for at least one test datum. In each of these approaches, the tester augments the test data set iteratively to eliminate the seeded faults. The philosophy behind these approaches is that the process of finding all seeded faults also eliminates real faults in the source code.

These two approaches require explicit construction and execution (or at best partial interpretation) of many alternate programs. Two more recent evaluation techniques take a more mathematical approach. Rather than evaluate a pre-selected test data set through execution, Zeil's and Morell's techniques analyze the test data set and the program and determine faults that could exist in the program and would remain undetected by execution on the test data. Zeil's *perturbation testing* [Zei83, Zei84] provides a functional description of a "perturbation" class that would not be detected by the pre-selected test data set. Morell described a fault-based testing model [Mor84] that introduces two concepts: "creating" an initial error for a fault, and "propagating" it to the output. *Symbolic fault-based testing* [Mor88] uses the model to symbolically represent faults that would not be detected by a particular execution.

These fault-based test data evaluation approaches do not provide much guidance as to how to select test data that eliminate the faults considered. Several fault-based testing techniques more directly guide the test data selection process. Foster introduced the idea of conditions under which a fault manifests itself as an erroneous value [Fos80]. Foster's *error-sensitive test case analysis* consists of conditions sufficient to distinguish expressions that may contain a fault from the correct expression for several fault classes. In *weak mutation testing* [How82] (more recently called *fault-based functional testing*), Howden refined these conditions and introduced others. Weak mutation testing is applied to the low level "functions" (e.g., statements) in a program. Functional testing [How85, How87] augments this low-level testing by test selection rules applicable to the synthesis of functions from component (already tested) functions.

Two extensions to mutation analysis are oriented toward test data selection. In his mutation testing suite, Budd includes a component called *error-sensitive test monitoring* [Bud83] with conditions that must minimally be satisfied to detect some of the mutant classes in expressions containing them. Offutt described *constraint-based testing* [DGK+88] as a part of the MOTHRA mutation analysis system. This approach explicitly selects test data to detect

mutants at the statement containing them, and then program execution on such test data is compared with mutant program execution to determine if the mutant has been killed. If it has not been killed, additional test data is tried.

These condition-based approaches have three major weaknesses. First and foremost, they are not easily extensible; they provide specific rules rather than defining a general framework within which test data selection rules can be defined for particular faults. Second, these techniques focus only on introducing a potential error, either at the fault location or at the statement containing the fault; there is no guarantee that a failure is produced. Third, many of the rules that comprise these techniques are sufficient but not necessary to introduce a potential error; if a rule is unsatisfiable, therefore, faults of the associated class may not be detected.

The RELAY model differs significantly from each of the fault-based testing techniques described here. The RELAY model is most similar to Morell's work. We introduce concepts similar to Morell's creation and propagation; our *origination* and *transfer*[1] refer to the first erroneous evaluation and the persistence of that erroneous evaluation, respectively. We refine his theory by more precisely defining origination and by differentiating between the transfer of an error through computations and its transfer through data flow. This differentiation facilitates defining fault-based rules for test data selection, whereas Morell's model is used for test data evaluation. In what follows, we outline the RELAY model and describe its use for test data selection. In the conclusion, we discuss the benefits that distinguish RELAY from other fault-based testing techniques.

3 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 erroneous value and that this error is *transferred* through computations and data flow until a failure is revealed. Note that a *fault* is a syntactic discrepancy in the source code,

[1] We have chosen the term "originate" rather than "create" or "introduce", because we feel it better connotes the first location at which an erroneous evaluation occurs and does not imply the mistake a programmer makes while coding. We have chosen the term "transfer" over "propagate" so as to avoid the connotation of an "increase in numbers" and instead of "persist" so as not to conflict with Glass's notion [Gla81], where an error is persistent if it escapes detection until late in development.

an *error* is an incorrect intermediate value, and a *failure* is observable incorrect behavior. As currently formulated, RELAY is limited to the detection of failures resulting from a single fault.

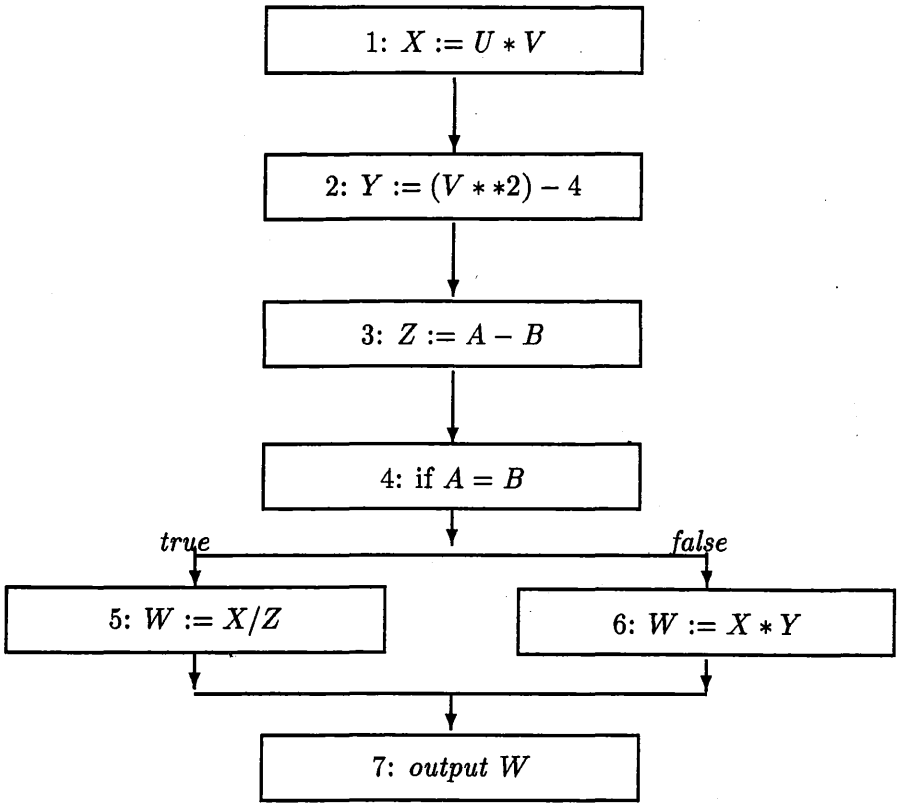

Figure 1: Test Module

To see that the process of revealing a failure is more complicated than it might appear, let us consider the module whose control flow graph is shown in Figure 1. Suppose that the reference to variable U at statement 1 should be a reference to B — that is, statement 1 should be $X := B * V$. The fault in this case is an incorrect variable reference. Let's walk through the test data set shown in Table 1. This table shows the values for the test data in the first column and the values of intermediate computations in the remaining columns[2]. All

[2]In the table and the discussion that follows, variable names are indicated by upper case while symbolic variable values are indicated by lower case

	test data	VAR	var	$\ldots * V$	$X := \ldots$	$W := X * Y$	output W
1	$a = 1, b = 1$	U	1				
	$u = 1, v = 1$	B	1				
2	$a = 1, b = 2$	U	1	0			
	$u = 1, v = 0$	B	2	0			
3	$a = 1, b = 2$	U	1	2	2	0	
	$u = 1, v = 2$	B	2	4	4	0	
4	$a = 1, b = 2$	U	1	3	3	15	15
	$u = 1, v = 3$	B	2	6	6	30	30

Table 1: Test Data Set for Module in Figure 1

shown test data execute the faulty statement, yet only one test datum reveals a failure.

Test datum 1: The values of b and u are the same; a failure cannot possibly result because no erroneous value is introduced.

Test datum 2: b and u have different values, but this error is masked upon multiplication by v, which has the value zero (both $U * V$ and $B * V$ evaluate to zero). Hence, no failure could be revealed.

Test datum 3: An error is manifested in different values for X, and that value is used at statement 6. In the computation at statement 6, however, the error is masked by multiplication by Y, which has the value zero.

Test datum 4: An error in X used at statement 6 is not masked, and W evaluates erroneously. A failure is revealed when W is output at statement 7.

As seen in the above discussion, to guarantee a particular fault's detection by revealing a failure, careful attention must be paid to introducing intermediate erroneous values and to carrying intermediate errors throughout module execution. These ideas are captured in the RELAY model.

RELAY starts with a source code expression that contains a discrepancy from some correct module[3]. As demonstrated, it is possible to mask such a discrepancy during execution. Module execution may mask the discrepancy for some, but not all, test data; output appears correct but just by coincidence of the test data selected. This is often referred to as *coincidental correctness.* It is also possible that a discrepancy is masked by all test data that

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

may execute the discrepancy. In this case, although a discrepancy exists between the test module and some correct module, the two are equivalent. In sum, we do not know whether a discrepancy can cause a failure, and thus it is only potentially a fault. A **potential fault** is a syntactic discrepancy between the test module and some correct module. Potential fault execution may introduce incorrect intermediate values, but coincidental correctness may still occur; later computations may mask the incorrect intermediate value. An incorrect value is thus referred to as a **potential error.**

The RELAY model determines test data requirements that must be satisfied for a potential fault to introduce a potential error, carry it throughout execution, and eventually produce a failure. Given a potential fault, RELAY first considers the requirement to introduce a potential error. We say that a potential error **originates** if the smallest subexpression containing the potential fault evaluates differently from the corresponding subexpression in the correct module. After a potential error originates, it must be carried through module execution to affect subsequent computations and eventually the output; this is called **transfer.** There are two types of transfer. **Computational transfer** refers to transfer of an erroneous result through a statement that uses potentially erroneous values. **Data flow transfer** refers to transfer from an erroneous variable assignment to a use of that variable. When a potential error transfers to the outermost expression in a statement, a **context error** results. For an assignment statement, for instance, an erroneous value assigned to a variable manifests a context error. For a conditional statement, the selection of an incorrect branch[4] manifests a context error. A context error results after a potential error originates and transfers through all computations in the potentially faulty statement. Context errors may also result at subsequent statements where at least one referenced variable still holds an erroneous value due to the potential fault. When a potential error transfers through all computations and data flow to reach an output, a **failure** results[5].

Let us look again at the test data shown in Table 1 and see how each fits into the RELAY model.

[4]To simplify the discussion in this paper, we assume that context errors in conditional statements are sufficient to reveal failure, since an incorrect path is traversed. We concentrate here on the transfer of context errors in assigned variables.

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

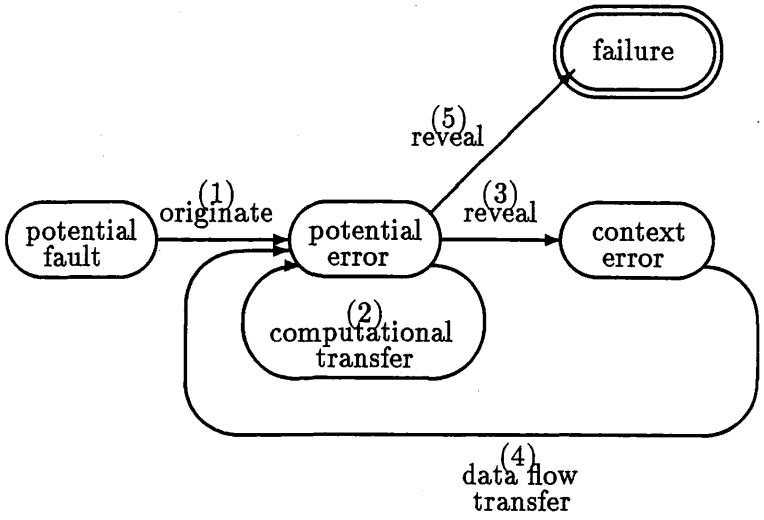

Figure 2: The RELAY Model

Test datum 1: A potential error fails to originate.

Test datum 2: A potential error originates but computational transfer does not occur through the multiplication.

Test datum 3: A context error occurs after statement 1, and data flow transfer occurs to the use of X at statement 6; the potential error fails, however, on computational transfer through the multiplication.

Test datum 4: A potential error originates and transfers through all computations and along data flow to cause a failure. The potential fault is indeed a fault.

The RELAY model is summarized in Figure 2. As shown in the figure, we start with a potential fault. The potential fault must first originate a potential error (1). The potential error must computationally transfer through all ancestor operators in the statement containing the potential fault (2), after which a context error is revealed (3). This context error, which is manifested as an erroneous value for some variable, is transferred by data flow to some use of this variable (4), resulting in a potential error at the statement where the use occurs. Again, the potential error must transfer through all computations in the statement. This process of transferring through all computations at a statement to produce a context error followed

by transfer of the context error through data flow to some other statement (2,3,4) continues until a statement is reached where the potential error 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. How then can this model be used to guide testing a module? Certainly, if we knew ahead of time where and what the faults were, we would not be testing for them but would simply fix them. 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 every statement in the code 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's task is to select test data that guarantees the test module and the hypothetical correct module behave differently. Given such test data, If the test module produces a failure for such test data, then it actually contains the potential fault that is hypothesized (since the correct module containing the alternate behaves correctly and the test module and the correct module behave differently). If the test module does not cause a failure for such test data, then the hypothesized potential fault is not a fault. RELAY's application does not require constructing the hypothetical correct module containing the alternate to determine the existence or nonexistence of the potential fault that was hypothesized, but only requires executing the test module on the selected test data. 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.

The first step is to guarantee that a potential error originates and produces a context error. 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. A potential error originating at the smallest subexpression containing a potential fault must transfer to affect evaluation of the entire statement by assigning an erroneous intermediate value. Thus, the potential error must transfer through each operator that is an ancestor of the subexpression in which the potential error originates. The **computational transfer condition** guarantees that a potential error transfers through all ancestor operators by distinguishing each ancestor expression referencing a potential error from the

The RELAY model of faults, errors, and failures is analogous to a relay race, as shown in Figure 3, hence its given name. In this analogy, the starting blocks correspond to the fault location. The take off of the first runner, as the gun sounds the beginning of the race, is analogous to the origination of a potential error. The runner carrying the baton through the first leg of the race illustrates the computational transfer of the potential error through that first statement. The successful completion of a leg of the race has a parallel in revealing a context error, and the passing of the baton from one runner to the next is analogous to the data flow transfer of the context error from one statement to another. Each succeeding leg of the race corresponds to the computational transfer through another statement. The relay team completes the race when the finish line is crossed, which is analogous to revealing a failure. Here, this failure is revealed as erroneous output with an output oracle.

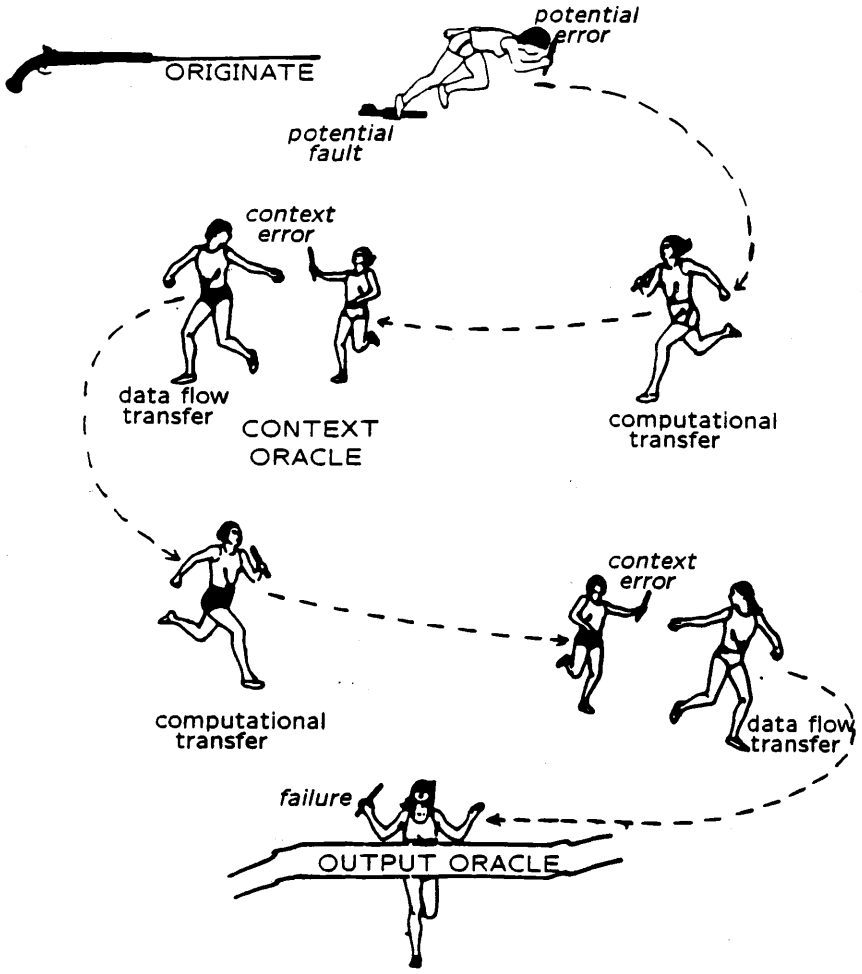

Figure 3: The Testing Relay

ancestor expression referencing the evaluation of the correct subexpression. The **context error condition** at the originating statement, is the conjunction of the origination condition and the computational transfer condition.

From here, the context error must transfer to some output statement where a failure is demonstrated. There may be many routes along which the potential error 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 error from a definition in the chain to the next use (e.g., execution of a def-use pair). To transfer a potential error along a selected chain, the data flow transfer conditions for all def-use pairs in the chain must be satisfied. In addition, at each use in the chain, the computational transfer condition to transfer the use of the potential error to 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 error 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 not 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 error 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. This is the failure condition. If this disjunction is unsatisfiable, then it is not possible to transfer the potential error along any of the routes. This means that the potential fault and the alternate are equivalent, and the potential fault is not a fault.

Turning back to the example, we hypothesized that the reference to U at statement 1 should be a reference to B. The 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 error) is not zero-valued. When applied to this statement, the computational transfer condition is

$(v \neq 0)$. Thus, the context error 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.

Let us first consider the chain consisting of the use of X at statement 5, where W is defined, followed by the output of W at statement 7. Here, the data flow transfer condition is $(a = b)$. Next, the computational transfer at statement 5 must be considered. The potential error in X must transfer through the division by Z. The transfer condition through division requires that the non-erroneous operand have a non-zero value; evaluation of $(z \neq 0)$ 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 because $(a = b)$ and $(a - b \neq 0)$ are contradictory. The context error 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 through the multiplication at node 6 requires that the variable Y have a non-zero value. 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.

4 A RELAY Testing Tool

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 needs fault specific origination and transfer conditions. As an example of a fault specific origination condition, consider a multiplication operator mistakenly replaced by an addition operator. The origination condition for this potential fault type has the form $(exp_1 * exp_2 \neq exp_1 + exp_2)$, which simplifies as ($exp_1 \neq 0$ or $exp_2 \neq 0$). As an example of a fault specific computational transfer condition, consider transfer through a multiplication operator. The transfer condition has the form $exp_1 * exp_2 \neq \overline{exp_1} * exp_2$, where $\overline{exp_1}$ is the operand containing the potential error, which simplifies as $exp_2 \neq 0$.

In deriving the fault specific conditions, we group faults into classes based on a common characteristic transformation. For example, all faults that involve replacement of an arithmetic operator by another are grouped into the more general class of arithmetic operator faults. We then instantiate fault class origination conditions for each fault class. The origination 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]). Given a particular fault class, computational transfer conditions must also be instantiated for each operator whose operands may be a potential error caused by a fault in the class. For an arithmetic operator, for example, an originated potential error might need to transfer through arithmetic operators, relational operators, and/or boolean operators, and the transfer condition must be instantiated for each to consider this fault class. Note that, in general, a transfer condition is applicable to many fault classes. The computational transfer conditions have been instantiated for the following operator types: boolean operator, relational operator, arithmetic operator, assignment operator (see [RT88a])[6]. 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

[6] These operators are applicable to those faults for which origination conditions have been developed

class differ only in origination condition, sharing identical transfer conditions (computational as well as data flow).

A prototype RELAY testing tool is currently being developed. The initial instantiation is for the six fault classes mentioned above. The tool contains the fault class origination and computational transfer conditions for these six classes in tables. It is easy, therefore, to extend the tool to cover new fault classes by adding new fault class conditions to the relevant tables.

Testing with the RELAY tool consists of four steps: 1) determining the RELAY requirements, 2) generating the context error condition, 3) generating the chain transfer condition, 4) creating the failure condition, and 5) evaluating and/or selecting test data. These steps are outlined in Figure 4 and discussed more completely below.

When testing with the RELAY tool, the user specifies a RELAY criterion, which dictates testing for some fault class(es) and 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. To develop the revealing condition for a particular potential fault, the appropriate fault class conditions are evaluated as determined by the fault class, location, and transfer chain.

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. Derivation of the context error 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 error 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 error conditions.

For each potential fault, the tool first derives the context error 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 error conditions for the class of potential faults. This is done for all potential faults for which we are testing at a selected

Step 1 Determine RELAY Requirements

1. select RELAY criterion for which to test
2. identify fault class(es)
3. identify potential faults (class and location)

Step 2 Context Error Condition at Originating Statement
(for each potential fault)

1. evaluate origination condition for potential fault
2. evaluate the computational transfer condition for each ancestor operator at statement
3. conjoin to form context error condition

Step 3 Chain Transfer Condition
(for an originating statement)

1. select a chain of alternating definitions and uses from originating statement to output
2. determine data flow transfer condition for each def-use pair and computational transfer condition for each use
3. conjoin to form chain transfer condition

Step 4 Failure Condition
(for all potential faults at an originating statement)

1. conjoin chain transfer condition to context error condition to form failure condition

Step 5 Test Data Selection/Evaluation
(for each failure condition)

1. check existing test data for satisfaction of failure condition
2. select test data to satisfy failure condition (if necessary)

Figure 4: Application of RELAY

statement.

Then, the tool derives the chain transfer condition for a selected chain of alternating definitions and uses from the originating statement to output. A chain is selected by analyzing a flow graph annotated with def-use pairs. For each def-use pair along the chain, 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 error condition at the originating statement to provide a failure condition.

Note that the failure conditions are constructed incrementally and may at any time become infeasible — that is, the new transfer condition may be inconsistent with another condition in the conjunction. The tool checks feasibility incrementally so as not to waste valuable computation time extending an already infeasible condition.

Finally, the RELAY tool is used to evaluate pre-selected test data and/or to select test data. Since the RELAY model of error 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 environment for testing, evaluation, and analysis of software. The TEAM environment provides the essential building blocks, through generic component technology, for easily constructing new tools. Within TEAM, testing tools are built upon generic analysis components providing capabilities that seem to be required by most testing techniques. One important TEAM component is ARIES [ZE88], a generic interpretation tool that provides symbolic evaluation capabilities. Symbolic evaluation assigns symbolic names to input values and interprets a path, maintaining variable and condition values in terms of the symbolic input values. Our implementation relies heavily upon the symbolic capabilities provided by ARIES. Symbolic evaluation is

necessary for 1) interpreting a path up to a potential fault, 2) interpreting a path covering a selected chain, 3) evaluating the fault class origination and transfer conditions in terms of symbolic values, and 4) evaluating the data flow transfer conditions along a chain in terms of symbolic values. Another essential component in ARIES provides formal reasoning capabilities, which are used for determining feasibility of the failure conditions and solving the failure conditions to provide test data. TEAM also provides low-level facilities such as language processing and object management. The TEAM environment enables us to build a RELAY testing tool more rapidly and with lower development costs than could be achieved by independent implementation.

5 Conclusion

In this paper, we present the RELAY model of fault detection and demonstrate its use as a test data selection technique. RELAY models detection of a fault by origination of an erroneous value that transfers through execution until a failure is revealed. The model also defines model origination and transfer conditions, which provide the general requirements for fault detection. The model conditions are instantiated for various fault classes to produce program-independent fault class origination and transfer conditions, which define guaranteed detection of faults in the classes. 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 by evaluating the appropriate fault class conditions in the context of the program. Implementation of a testing tool based on the RELAY model is currently underway. This tool is part of the TEAM environment [CRZ88] for testing, evaluation and analysis and makes use of the generic analysis components provided in that environment.

Related works in fault-based test data evaluation and test data selection are described. A more comprehensive survey and comparison of fault-based testing techniques including their relationships to RELAY is reported in [RT88a], while a formal analysis of several fault-based test data selection techniques is found in [RT86a]. RELAY provides several advantages over other fault-based testing techniques:

1. RELAY actually provides a mechanism for test data selection and does not merely eval-

uate the adequacy of user-selected test data, as do the fault-based test data evaluation techniques.

2. RELAY recognizes the need to produce an observable failure for a given fault; most other fault-based test data selection techniques just introduce an intermediate erroneous value.

3. RELAY develops conditions that are both necessary and sufficient to detect faults, whereas most of the fault-based test data selection techniques consist of rules that are only sufficient for introducing intermediate erroneous results.

4. RELAY distinguishes between origination of a potential error for a fault class and computational transfer of that potential error, which facilitates extension to additional fault classes.

5. RELAY provides a specific framework in which all these components fit and which is applicable for test data selection.

We believe that RELAY provides a cleaner, clearer view of fault-based testing than other approaches to date and that it is a significantly more powerful approach.

We continue to extend the RELAY model of error 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 erroneous variable and those that cover looping constructs. 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 error across a procedure invocation. In addition, we are considering the use of the RELAY model as a specification-based testing technique.

Our evaluation of the RELAY model of error detection has thus far been of an analytical nature through which its error detection capabilities have been compared to those of other fault-based techniques [RT86a]. While this has provided considerable insight, and we expect that further analysis will prove useful, there is a clear need for empirical evidence of the model's worth. Empirical studies are particularly important in testing research since often the worst case analysis can lead to very different conclusions than experimental studies of typical operational performance. The TEAM environment will enable us to experiment with other

fault-based techniques and not only RELAY, thus providing empirical evidence of RELAY's merits relative to similar techniques.

References

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

[CR85] Lori A. Clarke and Debra J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, January 1985.

[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.

[Gla81] Robert L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, SE-7(2):162–168, March 1981.

[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.

[How85] William E. Howden. The theory and practice of functional testing. *IEEE Software*, 2(5):6–17, September 1985.

[How87] William E. Howden. *Functional Program Testing and Analysis.* Series in Software Engineering and Technology. McGraw-Hill, 1987.

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

[Mor88] Larry J. Morell. Theoretical insights into fault-based testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

[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. Technical report, Information and Computer Science, University of California at Irvine, May 1989.

[RC85] Debra J. Richardson and Lori A. Clarke. Testing techniques based on symbolic evaluation. In T. Anderson, editor, *Software: Requirements, Specification and Testing*, pages 93–110. Blackwell Scientific Publications Ltd., 1985.

[RT86a] 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.

[RT86b] Debra J. Richardson and Margaret C. Thompson. A new model of error detection. Technical Report 86-64, 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.

[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.

[Zei83] Steven J. Zeil. Testing for perturbations of program statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.

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