Testing Based on the RELAY Model of Error Detection

Debra J. Richardson†
Margaret C. Thompson‡

COINS Technical Report 87-119
September 1987

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

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

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

Keywords: software testing, test data selection, fault-based testing, criteria evaluation

This research was supported by grants DCR-83-18776 and DCR-84-04217 from the National Science Foundation, 84M103 from Control Data Corporation, and by the Rome Air Development Center grant F30602-86-C-0006

Abstract

RELAY, a model for error detection, defines *revealing conditions* that guarantee that a fault *originates* an error during execution and that the error *transfers* through computations and data flow until it 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 applying them to 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, describes why it is better than other fault-based testing criteria, and discusses how RELAY could be used as the foundation for a testing system.

1 Introduction

The goal of testing a program is the detection of errors. This is typically done by attempting to select test data for which execution of the program produces erroneous results. Many testing techniques [Bud81,Bud83,Ham77,How82,How85,Mor84,Zei83,Wey81] are directed toward the detection of errors that might result from commonly occurring faults in software. These "fault-based" testing techniques assume that the program being tested is "almost correct", which might be determined by successfully passing some high-level functional testing phase. If a "comprehensive" set of test data is selected by a fault-based technique and the program executes correctly, then the tester gains confidence that the program does not contain specific types of faults.

This paper reports on a new model of error detection called RELAY, which provides a fault-based criterion for test data selection. The RELAY model builds upon the testing theory introduced by Morrell [MH81,Mor84], where an error is "created" when an incorrect state is introduced at some fault location, and it is "propagated" if it persists to the output. We refine this theory by more precisely defining the notion of when an error is introduced and by differentiating between the persistence of an error through computations and its persistence through data flow operations. We introduce similar concepts, *origination* and *transfer*[1], as the first erroneous evaluation and the persistence of that erroneous evaluation, respectively. The RELAY model defines *revealing conditions* that guarantee that a fault *originates* an error during execution and that the error *transfers* through all affected computations until it is *revealed*.

Analysis of other fault-based testing criteria has shown that RELAY provides a more precise model of error detection. We have analyzed three test data selection criteria, each of which attempt to detect faults in six fault classes. This analysis demonstrated that none of these criteria thoroughly considers the potential unsatisfiability of their rules; each criterion includes rules that are sufficient to cause errors for some fault classes, yet when such rules are unsatisfiable, many errors may remain undetected. Moreover, the criteria fail to integrate their proposed rules; although a criterion may

[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.

cause a subexpression to take on an erroneous value, there is no effort made to guarantee that the enclosing expression evaluates incorrectly. RELAY overcomes these two common weaknesses because the test data selected by RELAY satisfies precise origination conditions that are coupled with transfer conditions. Moreover, these conditions are necessary and sufficient for revealing an error.

For the most part, testing research is directed either at the problem of determining the paths that must be tested or at the problem of selecting test data for the chosen paths. RELAY shows that this usual separation just doesn't work. RELAY approaches test data selection in conjunction with path selection by selecting test data that not only originates an error but also transfers that error to affect the output.

The next section summarizes the RELAY model; more detail is provided in a related paper[RT86b]. In the third section, we describe the instantiation of RELAY to develop revealing conditions for a class of faults, while section four analyzes three test data selection criteria in terms of their ability to detect errors caused by faults in this class. This analysis shows the advantages of the RELAY model of error detection. In the fifth section, we present an example of the application of RELAY as a test data selection criterion and discuss how this application might be automated. In summary, we discuss the implications of the model and our future plans for RELAY.

2 RELAY: A Model of Error Detection

The primary use for the RELAY model of error detection is as a test data selection criterion. RELAY is capable of guaranteeing the detection of errors that result from some chosen class or classes of faults. In addition, given test data that has been selected by another criterion, RELAY can be used as a measurement technique for determining whether that test data detects such errors. RELAY can also be used to analyze the error detection capabilities of other criteria. Before describing the RELAY model of error detection, we first introduce a terminology within which we define the RELAY model.

2.1 Terminology

We consider the testing of a *module* M that implements some function $F_M : X_M \rightarrow Z_M$. A module M can be represented by a *control flow graph* $G_M = (N, E)$, where N is a (finite) set of nodes and $E \subseteq N \times N$ is the set of edges. N includes a start node n_{start} and a final node n_{final}; each other node in N represents a simple statement[2] or the predicate of a conditional statement in M. A *subpath* in a control flow graph $G_M = (N, E)$ is a finite, possibly empty, sequence of edges $p = [(n_1, n_2), ..., (n_{|p|}, n_{|p|+1})]$ in E; note that the last node $n_{|p|+1}$ has been selected by virtue of its inclusion in the last edge but is not visited in the subpath traversal. An *initial subpath* p is a subpath whose first node is n_{start}. A *path* P[3] is an initial subpath whose last node is n_{final}.

A *test datum* t for a module M is a sequence of values input along some initial subpath. For any node n in G_M, $DOMAIN(n)$ is the set of all test data t for which n can be executed. A test datum t may be an incomplete sequence of input values in the sense that it cannot execute a complete path. This may be because: 1) additional input is needed to complete execution of some path; or 2) the initial input t may cause the module to terminate abnormally (before n_{final}) or possibly never to terminate. The possibility of incomplete input sequences allows for testing criteria that consider invalid test data, which are not in the domain of M but for which M may initiate execution. The *test data domain* D_M for $G_M = (N, E)$ is the domain of inputs from which test data can be selected. [4] $D_M = \{t \mid \exists\, n \in N, t \in DOMAIN(n)\}$.

A testing method typically specifies some subset of the test data domain for execution. A *test data set* T_M for a module M is a finite subset of the test data domain, $T_M \subseteq D_M$. A *test data selection criterion* S, is a set of rules for determining whether a test data set T_M satisfies selection requirements for module M.

To reveal errors by testing, there is usually some test oracle that specifies correct execution of the module [How78,Wey82]. A test oracle might be a functional representation, formal specification, or correct version of the module or simply a tester who knows the module's correct output. In any

[2]Single statements are considered here for the clarity of the presentation; a simple extension allows nodes to represent a group of simple statements.

[3]Where the distinction between a subpath and a path is important, we will use an upper case letter (P) to signify a path and a lower case letter (p) for a subpath (or initial subpath).

[4]D_M does not include data that would result in an invalid call of the module.

case, an *oracle* $O(X_O, Z_O)$ is a relation, $O = \{(x, z)\} \subset X_O \times Z_O$, where X_O and Z_O are the domain and range, respectively, of the oracle. When $(x, z) \in O$, z is an acceptable output for x. Execution of a module M on input x reveals an *output error* when $(x, M(x)) \notin O$.

A "standard" oracle judges the correctness of the module's output for valid input data. Testers often have a concept of the "correct" behavior of a module, however, and not just its correct output. Rather than waiting until output is produced to find errors, the tester might check the computation of the module at some intermediate point, as one does when using a run-time debugger. This approach to testing can be performed with an oracle that includes information about intermediate values that should be computed by the module; this information might be derived from some correct module, an axiomatic specification, run-time traces [How78], or monitoring of assertions. Let us associate with the execution of an initial subpath p on some test datum t a *context* $C_{p(t)}$, which contains the values of all variables after execution of $p(t)$. A *context oracle* O_C is a relation $O_C = \{((t, p), C_{p(t)})\}$, that relates a test datum and an initial subpath (t, p) to one or more contexts $C_{p(t)}$ that are acceptable after execution of p on t[5]. Execution of a path p on test datum t reveals a *context error* when $((t, p), C_{p(t)}) \notin O_C$.

2.2 RELAY

The errors considered within the RELAY model are those caused by some chosen class or classes of *faults* in the module's source code. The fault-based approach to testing relies on an assumption that the module being tested bears a strong resemblance to some hypothetically-correct module. Such a module need not actually exist, but we assume that the tester is capable of producing a correct module from the given module and knowledge of the errors detected. As currently formulated, RELAY is limited to the detection of errors resulting from a single fault.

A node containing a fault may be executed yet not reveal an error; the module appears correct, but just by coincidence of the test data selected. It is also possible that the tested module produces correct output for all input despite a discrepancy between it and the hypothetically-correct module. In this case, the module is not merely coincidentally correct, it is correct. Thus, a discrepancy is

[5]For simplicity, the granularity of the context oracle is assumed to be the same as that of the control flow graph, although this is not necessary.

only "potentially" a fault. Likewise, incorrect evaluation of an expression is only "potentially" an error since the erroneous value may be masked out by later computations before an erroneous value is output. A **potential fault**, denoted f_n, is a discrepancy between a node n in the tested module M and the corresponding node n^* in the hypothetically-correct module M^* — that is, $n \neq n^*$. The evaluation of some expression EXP[6] in M, which contains a potential fault, and the corresponding expression EXP^* in M^* results in a **potential error** when $exp \neq exp^*$. To discover a potential fault, erroneous results must appear for some test datum as a context error or as an output error, depending on the type of oracle used.

RELAY is a model that describes the ways in which a potential fault manifests itself as an error. Given some potential fault, a potential error **originates** if the smallest subexpression of the node containing the potential fault evaluates incorrectly. Consider the module in Figure 1 for example. Suppose that the statement $X := U*V$ at node 1 contains a variable reference fault and should be $X := B*V$. A potential error originates in the smallest expression containing the potential fault, which is the reference to U, whenever the value of U differs from the value of B.

It is not only important to originate an error but also to ensure that it is not masked out by later computations. A potential error in some expression **transfers** to a "super"–expression that references the erroneous expression if the evaluation of the "super"–expression is also incorrect. Take another look at Figure 1; if V holds the value zero, the potential error in U that originates in node 1 does not transfer to affect the assignment to X; the potential error transfers, on the other hand, whenever V is nonzero. To reveal a context error, a potential fault must originate a potential error that transfers through all computations in the node thereby causing an incorrect context. This is termed **computational transfer**. To reveal an output error, a potential fault must cause a context error that transfers from node to node until an incorrect output results. This transfer includes **data flow transfer**, whereby a potential error reaches another node — that is, the potential error is reflected in the value of some variable that is referenced at another node — as well as computational transfer within the nodes that an erroneous value reaches. Using the example of Figure 1 again, the potential error in X must transfer through data flow to a use, say at node 7, transfer through the computations at node 7 to produce an error in W, and then transfer to

[6]Upper case [EXP] is used here to denote the source-code expression, while lower case [exp] denotes the evaluated expression.

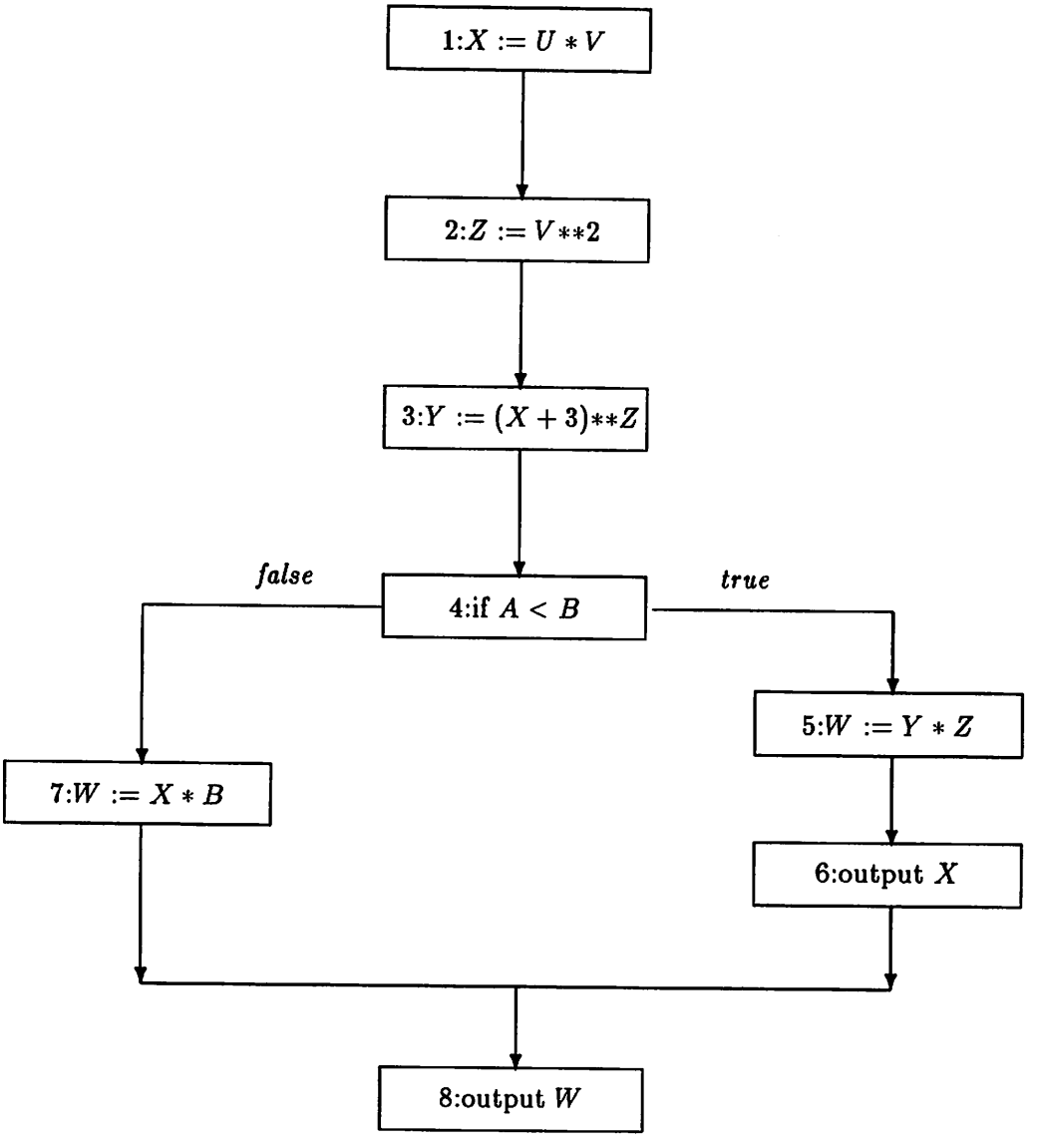

Figure 1: Module for Test Data Selection

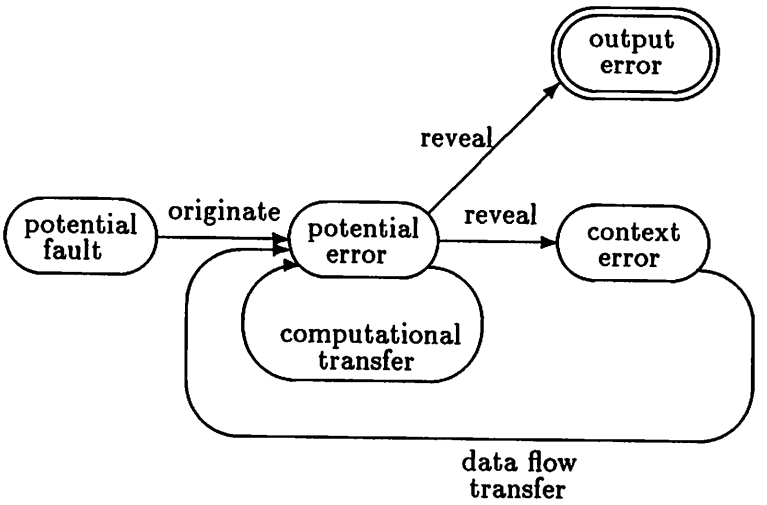

Figure 2: RELAY Model of Error Detection

the output of W at node 8. We know unequivocally that the module is incorrect only if an output error is revealed. Thus, a potential fault is a **fault** only if it produces incorrect output for some test datum.

Figure 2 illustrates the RELAY model of error detection and how this model provides for the discovery of a fault. The conditions under which a fault is detected are 1) origination of a potential error in the smallest containing subexpression; 2) computational transfer of that potential error through each operator in the node, thereby revealing a context error; 3) data flow transfer of that context error to another node on the path that references the incorrect context; 4) cycle through (2) and (3) until a potential error is output. If there is no single test datum for which a potential error originates and this "total" transfer occurs, then the potential fault is not a fault, and the module containing the potential fault is equivalent to some hypothetically-correct module.

As shown in Figure 3, the RELAY view of error detection has an illustrative analogy in a relay race, hence the name of our model. 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 is the computational transfer of the 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 error from one statement to

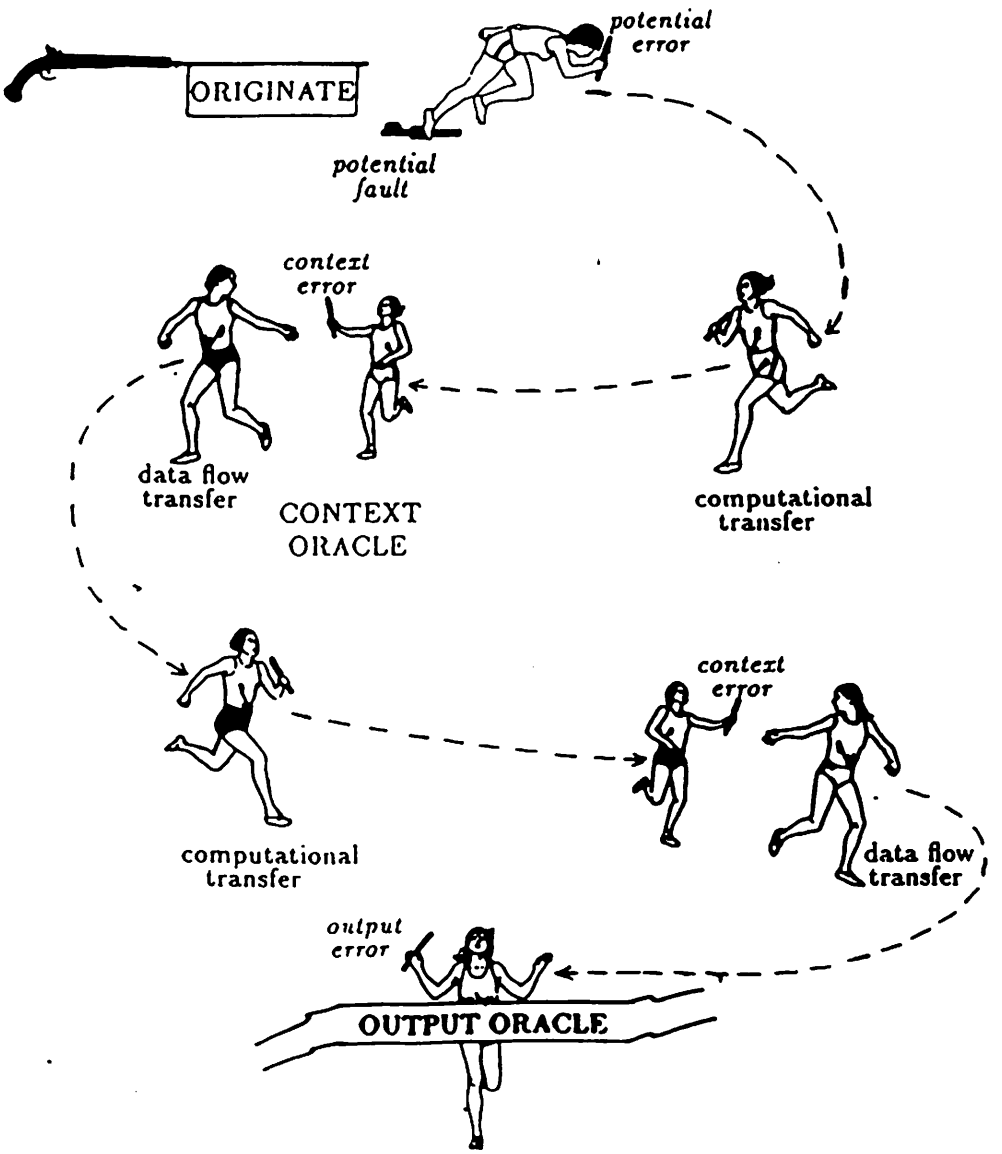

Figure 3: The Testing Relay

another. Each succeeding leg of the race corresponds to the computational transfer through another statement. The race goes on until the finish line is crossed, which is analogous to the test oracle revealing an output error.

Our goal, of course, is to complete the relay race — that is, to detect errors. To this end, the RELAY model proposes the selection of test data that originates an error for any potential fault of some type and transfers that error until it is revealed. RELAY uses the concepts of origination and transfer to define conditions that are necessary and sufficient to guarantee error detection. When these conditions are instantiated for a particular type of fault, they provide a criterion by which test data can be selected for a program so as to guarantee the detection of an error caused by any fault of that type.

Given an oracle and a module M with $G_M = (N, E)$ that contains a potential fault f_n at node $n \in N$, a test data selection criterion S is said to **guarantee detection** of a fault f_n if for all test data sets T_M that satisfy S, there exists $t \in T_M$ such that f_n originates an error for $M(t)$ that transfers until it is detected by the oracle. If a context oracle exists, the potential fault must reveal a context error for some test datum in every test data set. Note that guaranteeing detection of a context error does not necessarily mean that an output error will result for this execution, since it is possible that the context error is masked out by later statements and not transferred to the output. If error detection is done by a standard output oracle, then a context error revealed by the fault must also transfer to the output for some test datum in every test data set.

Here, we define *origination*, *transfer*, and *revealing conditions* that are necessary and sufficient to guarantee that an error is revealed. Sufficient means that if the module is executed on data that satisfies the conditions and the node is faulty, then an error is revealed. Necessary, on the other hand, means that if an error is revealed then the module must have been executed on data that satisfies the condition and the node is faulty.

These conditions are defined for a potential fault independent of where the node occurs in the module. The test data selected, however, must execute the node within the context of the entire module. Thus, for a potential fault at node n, such test data are restricted to $DOMAIN(n)$. Because these conditions are both necessary and sufficient, if the conditions are *infeasible* within $DOMAIN(n)$, then no error can be revealed and the potential fault is not a fault. Although, in general, the feasibility problem is undecidable, in practice, it can usually be solved.

First, suppose that we are attempting to detect a particular fault f_n in a node n. This is somewhat unrealistic, since if one explicitly knew the location of a fault, one would fix it. We will address this issue in a moment, after some groundwork is laid.

To reveal an output error, we must first generate a context error at the node containing the fault; thus, let us first consider the conditions required to guarantee the detection of a context error. By requiring test data to distinguish the faulty subexpression from the correct one, the **origination condition** for a potential fault f_n guarantees that the smallest subexpression containing f_n originates a potential error. A potential error originating at the smallest subexpression containing a potential fault must transfer to affect evaluation of the entire node. By requiring test data that distinguishes the parent expression referencing a potential error from the parent expression referencing the correct subexpression, the **computational transfer condition** guarantees that a potential error transfers through a parent operator. To affect the evaluation of a node, test data must satisfy the computational transfer condition for each operator that is an ancestor of the subexpression in which the potential error originates thereby producing a context error. The **node transfer condition** is the conjunction of all such computational transfer conditions. To guarantee a fault's detection through revealing a context error, a single test datum must satisfy both the origination and node transfer conditions. The **revealing condition** for a context error resulting from a potential fault f_n occurring in node n is the conjunction of the origination condition and the node transfer condition for f_n and n.

As an example of these conditions for error detection, consider again the module in Figure 1. If the statement $X := U * V$ at node 1 should be $X := B * V$, then the origination condition is $[u \neq b]$. This originated potential error must transfer through the multiplication by V; the corresponding computational transfer condition is $(u * v \neq b * v)$, which simplifies to $(v \neq 0)$. This value must then transfer through the assignment to X, which is trivial. Thus, the revealing condition for a context error resulting from this potential fault is $[(u \neq b)$ and $(v \neq 0)]$.

Testing is primarily concerned with the generation of an output error as the manifestation of a fault and not only with incorrect values at intermediate points in the module. Thus, we must guarantee that a context error transfers to affect execution of the module as a whole. A context error is evidenced through a potential error in at least one variable. By requiring test data that causes the execution of a statement referencing a variable that contains a potential error and

that causes the smallest subexpression containing that reference to result in a potential error, a **data flow transfer condition** describes the requirements for transfer of a context error from one statement to another. To reveal an output error, we must execute a def-use chain that begins with the node containing the potential fault and ends with the output of a variable. A *def-use chain* is a chain of alternating definitions and uses of variables, where each definition reaches the next use in the chain and that use defines the next variable in the chain. Satisfaction of the data flow transfer conditions will force execution of such a chain. In addition, the subsequent node transfer conditions for the references forced by data flow as well as the context error revealing condition at the location of the fault must be satisfied. A **chain transfer condition** for a def-use chain is the conjunction of the data flow transfer conditions for all pairs in the def-use chain and the node transfer conditions for all uses in the def-use chain. The **revealing condition** for an output error is the conjunction of the context error revealing condition and the chain transfer condition for the def-use chain from the fault location to the output.

Consider again the potential variable reference fault at node 1 in Figure 1. One def-use chain from the fault location to an output consists of the definition of X at node 1, followed by a use of X at node 7, where W is defined, followed by a use of W in the output statement at node 8. The potential error in X transfers through data flow to node 7 whenever the false branch of the conditional at node 4 is taken; thus, the data flow transfer condition is $(a \geq b)$. Reference to the potential error in X must transfer through the multiplication by B to the assignment of W at node 7, which entails a node transfer condition of $(b \neq 0)$. Thus, for this def-use chain, the chain transfer condition is $[(a \geq b)$ and $(b \neq 0)]$. Recall that the context error revealing condition is $[(u \neq b)$ and $(v \neq 0)]$, creating an output error revealing condition for this chosen def-use chain of $[(u \neq b)$ and $(v \neq 0)$ and $(a \geq b)$ and $(b \neq 0)]$.

As currently defined, derivation of revealing conditions is dependent on knowledge of the correct node. Since this is unlikely, an alternative approach is to assume that any node, in fact any subexpression of any node, might be incorrect and consider the potential ways in which that expression might be faulty. By grouping these potential faults into classes based on some common characteristic of the transformation, we define conditions that guarantee origination of a potential error for any potential fault of that class. A class of potential faults determines a set of alternative expressions, which must contain the correct expression if the original expression indeed contains a

fault of that class. To guarantee origination of a potential error for a class, the potentially faulty expression must be distinguished from each expression in this alternate set. For each alternative expression, then, our model defines an origination condition, which guarantees origination of a potential error if the corresponding alternate were indeed the correct expression. For an expression and fault class, we define the **origination condition set**, which guarantees that a potential error originates in that expression if the expression contains a fault of this class. The origination condition set contains the origination condition for each alternative expression.

For each alternative expression, a potential error that originates must also transfer through each operator in the node to reveal a context error and through data flow and subsequent computations to reveal an output error. The transfer conditions, which are determined by these subsequent manipulations of the data, are independent of the particular alternate. Thus, for a fault class, each alternate defines a revealing condition, which is the conjunction of the origination condition and the transfer conditions. The **revealing condition set** contains a revealing condition for each alternate in the alternate set and is necessary and sufficient to guarantee that a potential fault of a particular class reveals an output error.

Once again, consider the module in Figure 1 and the statement $X := U * V$, but now suppose that the reference to U might be faulty but we do not know what variable should be referenced. To guarantee origination of a potential error for an incorrect reference to U, we must select test data such that for each alternative variable, $\overline{U}$ [7] , T contains a test datum where the value of U is different from the value of $\overline{U}$ at node 1. The possible alternates depend on what other variables may be substituted for U without violating the language syntax. If we assume that all variables referenced in this module are of the same type, then there are seven alternates and hence seven origination conditions. The origination condition set is $\{[u \neq \overline{u}] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$. The node transfer condition for node 1 is $[v \neq 0]$. The chain transfer condition for the use of X to define W at node 7 and the output of W at node 8 is $[(a \geq b)$ and $(b \neq 0)]$. Thus, the set $\{[(u \neq \overline{u}$ and $(v \neq 0)$ and $(a \geq b)$ and $(b \neq 0)] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$ is a sufficient revealing condition set for this potential fault. This set is sufficient but not necessary because all def-use chains are not considered.

[7] We use the bar notation to denote an alternate.

The RELAY model of error detection is based on the generic revealing condition sets just defined. The model is applied by first selecting a fault classification. Given a particular class of faults, the generic origination and transfer conditions are instantiated to provide conditions specific to that class. An example instantiation is provided in the next section. For a module being tested, the instantiated origination and transfer conditions are evaluated for the nodes in a module's control flow graph to determine the actual revealing condition sets. Satisfaction of these sets guarantees the detection of an error for any fault in the chosen classification. The actual revealing conditions for a module can be used to measure the effectiveness of a pre-selected set of test data and/or to select a set of test data. A simple example of RELAY as a test data selection criterion is presented in Section 5. The origination and transfer conditions for a fault classification can be used to evaluate the effectiveness of another test data selection technique for classes of faults; Section 4 illustrates this application of RELAY. This analysis demonstrates the flaws inherent in most techniques and shows the advantages provided by the use of RELAY for test data selection.

3 Instantiation of RELAY

In this section, we discuss the instantiation of the RELAY model for a class of faults. The development of the revealing condition set for a class of faults consists of the development of the origination condition set and of any applicable transfer conditions. In [RT86b], RELAY is instantiated for six classes of faults: constant reference fault, variable reference fault, variable definition fault, boolean operator fault, relational operator fault, arithmetic operator fault. This instantiation process is illustrated here for the class of variable definition faults. We derive the origination condition set for this class. This class requires no computational transfer conditions to reveal a context error. To reveal an output error, however, a potential error in a defined variable must often transfer through an arithmetic operator [8] at the statement where the variable is referenced, so we illustrate instantiation of transfer conditions by considering computational transfer through arithmetic operators.

[8] The potential error may also transfer through other operations; these have been considered but are not illustrated here.

assignment	origination condition set
$V := EXP$	$\{[(\overline{v} \neq v)$ or $(exp \neq v) \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$.

Table 1: Origination Condition Set for Variable Definition Fault

3.1 Origination of Variable Definition Fault

An origination condition guarantees that the smallest expression containing a potential fault produces a potential error. Thus, given the smallest expression $SEXP$ containing a potential fault and an alternative expression $\overline{SEXP}$, the origination condition guarantees that $sexp \neq \overline{sexp}$. The origination condition set contains the origination condition for each alternate.

Consider here the class of variable definition faults, where a potential error may result when the name of a defined variable is mistakenly replaced by another valid variable name. Given a definition of a variable $V := EXP$ [9], if V is a faulty variable name, then the correct definition must be in the alternate set $\{[\overline{V} := EXP \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$.

The origination condition for an alternate $\overline{V}$ distinguishes between the assignments $V := EXP$ and $\overline{V} := EXP$. To distinguish these assignments and originate a potential error, either the two variables, V and $\overline{V}$, must have different values immediately before execution of the assignment or the value assigned to the variable must differ from its value immediately before execution of the assignment. The origination condition, therefore, is $[(\overline{v} \neq v)$ or $(exp \neq v)]$. The origination condition set consists of the set of origination conditions for each alternative variable, as shown in Table 1.

To demonstrate that the origination condition is both necessary and sufficient to originate a potential error, see Table 2, which enumerates all combinations of the values of pertinent variables and expressions for both the original and the alternate before and after evaluation of the statement

[9]Here we use the assignment operator := in the general sense to include all types of expressions that may result in a variable definition (e.g., procedure call).

	Values Before	Values After Assignment Evaluation	
		Original $V := EXP$	Alternate $\overline{V} := EXP$
i	$(\overline{v} \neq v)$ $(exp = v)$	$v' = v$ $\overline{v'} \neq v$	$v' = v$ $\overline{v'} = v$
ii	$(\overline{v} = v)$ $(exp \neq v)$	$v' \neq v$ $\overline{v'} = v$	$v' = v$ $\overline{v'} \neq v$
iii	$(\overline{v} \neq v)$ $(exp \neq v)$	$v' \neq v$ $\overline{v'} = v$	$v' = v$ $\overline{v'} \neq v$
iv	$(\overline{v} = v)$ $(exp = v)$	$v' = v$ $\overline{v'} = v$	$v' = v$ $\overline{v'} = v$

Table 2: Variable Definition

[10] . For cases i,ii, and iii, the values of V, $\overline{V}$, and EXP before evaluation of the statement satisfy the origination condition, and evaluation of the assignment statement originates a potential error. In each of cases i and iii, evaluation of the original and alternative statements result in different values for $\overline{V}$. In cases ii and iii, evaluation of the original and alternate result in different values for V. Thus, the origination condition is sufficient to originate a potential error for a variable definition fault. To see that the condition $[(exp \neq v)$ or $(\overline{v} \neq v)]$ is necessary, consider case iv for which the origination condition is not satisfied. Evaluation of the original and the alternative statements result in the same values for the variables; hence no potential error originates.

Note that each origination condition in the origination condition set includes the condition $(exp \neq v)$. If this single condition is satisfied, the origination condition set is satisfied. Thus, $(exp \neq v)$ is sufficient to guarantee origination of a potential error for a variable definition fault, and the set $\{[exp \neq v]\}$ represents a sufficient origination condition set. When the condition $(exp \neq v)$ is infeasible, however, the the set $\{[(\overline{v} \neq v) \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$ must be satisfied to guarantee origination of a potential error for a variable definition fault.

[10] In the table, v is the value of V before evaluation of the statement, and v' is the value after evaluation

3.1.1 Transfer through Arithmetic Operators

The transfer conditions through arithmetic operators guarantee that EXP_1 **aop** EXP_2 is distinguished from $\overline{EXP_1}$ **aop** EXP_2 or that EXP_2 **aop** EXP_1 is distinguished from EXP_2 **aop** $\overline{EXP_1}$, when EXP_1 and $\overline{EXP_1}$ are distinguished. Since addition and multiplication are commutative, the two cases need not be considered separately for these operators. The arithmetic transfer conditions depend upon the arithmetic operator and are derived by determining the complement of the conditions under which exp_1 **aop** $exp_2 = \overline{exp_1}$ **aop** exp_2. The transfer conditions derived here assume that both operands are of the same type, and that there is no round off error. Transfer conditions through the following arithmetic operators are considered: $+$, $-$, $*$, / (real operands), and $**$.

For the arithmetic operator $+$, there are no values exp_1, $\overline{exp_1}$, and exp_2 (assuming that $exp_1 \neq \overline{exp_1}$) for which $exp_1 + exp_2 = \overline{exp_1} + exp_2$; thus, for all values of exp_1, $\overline{exp_1}$, and exp_2 a potential error between exp_1 and $\overline{exp_1}$ will transfer through addition in an arithmetic expression. The same argument holds for subtraction $(-)$.

For the arithmetic operators $*$ and /, $(exp_1 * exp_2 = \overline{exp_1} * exp_2)$ and $(exp_1/exp_2 = \overline{exp_1}/exp_2)$ and $(exp_2/exp_1 = exp_2/\overline{exp_1})$ only when $exp_2 = 0$. Thus to guarantee transfer through a multiplication or division operator, exp_2 must be nonzero.

For the exponentiation operator $**$, we must consider the order of the operands. When EXP_1 and $\overline{EXP_1}$ are the base raised to the power EXP_2, we must examine when $(exp_1 ** exp_2 = \overline{exp_1} ** exp_2)$. This is true only when $(exp_2 = 0)$ or $(exp_1 = -\overline{exp_1}$ and exp_2 is even). Thus, the transfer conditions for an exponential expression when the potential fault is contained in the base operand are $(exp_2 \neq 0)$ and $(exp_1 \neq -\overline{exp_1}$ or $exp_2 \bmod 2 \neq 0)$. To determine the transfer conditions when the potential fault is within the exponent, we must examine the conditions where $(exp_2 ** exp_1 = exp_2 ** \overline{exp_1})$. This is true when $(exp_2 = 0)$ or $(exp_2 = 1)$ or when $(exp_2 = -1$ and exp_1 and $\overline{exp_1}$ are both even or both odd). Thus, the transfer conditions are $(exp_2 \neq 0)$ and $(exp_2 \neq 1)$ and $(exp_2 \neq -1$ or $exp_1 \bmod 2 \neq \overline{exp_1} \bmod 2)$.

The transfer conditions for arithmetic operators are summarized in Table 3 [11] .

[11] In the table, exp_1 contains a potential error, $\overline{exp_1}$ contains the alternate, and hence $exp_1 \neq \overline{exp_1}$.

operator	expression	transfer conditions
+	$exp_1 + exp_2 \neq \overline{exp_1} + exp_2$	*true*
−	$exp_1 - exp_2 \neq \overline{exp_1} - exp_2$	*true*
−	$exp_2 - exp_1 \neq exp_2 - \overline{exp_1}$	*true*
*	$exp_1 * exp_2 \neq \overline{exp_1} * exp_2$	$exp_2 \neq 0$
/	$exp_1/exp_2 \neq \overline{exp_1}/exp_2$	$exp_2 \neq 0$
/	$exp_2/exp_1 \neq exp_2/\overline{exp_1}$	*true*
**	$exp_1{**}exp_2 \neq \overline{exp_1}{**}exp_2$	$(exp_2 \neq 0)$ and $(exp_1 \neq -\overline{exp_1}$ or $exp_2 \bmod 2 \neq 0)$
**	$exp_2{**}exp_1 \neq exp_2{**}\overline{exp_1}$	$(exp_2 \neq 0)$ and $(exp_2 \neq 1)$ and $(exp2 \neq -1$ or $exp_1 \bmod 2 \neq \overline{exp_1} \bmod 2)$

Table 3: Transfer Conditions for Arithmetic Expressions

4 RELAY Versus Related Test Data Selection Criteria

RELAY provides a sound method for analyzing the error detection capabilities of another test data selection criterion in terms of its ability to guarantee detection of an error for some chosen class or classes of faults. A test data selection criterion is usually expressed as a set of rules that test data must satisfy. Our analysis approach evaluates a criterion in terms of the relationship between its rules and the revealing conditions defined by RELAY for the six fault classes. The revealing conditions are both necessary and sufficient to guarantee error detection, so this is an unbiased means of analysis. A rule or combination of rules is judged either to be insufficient to reveal an error, to be sufficient to reveal an error, or to guarantee that an error is revealed. This analysis is completely program independent.

In this section, we illustrate our analysis approach by using the origination condition set for variable definition fault and the transfer conditions for arithmetic operator to analyze the error detection capabilities of three fault-based test data selection criteria — Budd's *Error-Sensitive Test Monitoring (Estimate)* [Bud81,Bud83], Howden's *Weak Mutation Testing (WMT)* [How82,How85], and Foster's *Error Sensitive Test Case Analysis (ESTCA)* [Fos80,Fos83,Fos84,Fos85]. In [RT86b], we analyze these criteria for six fault classes. Each of these criteria was selected because its author claims that it is geared toward detection of faults of these six classes. Our analysis shows, however, that none of the criteria guarantees detection of these types of faults. The analysis also points

out two weaknesses that are common to all three criteria and provides insight into how RELAY rectifies these common problems.

4.1 Origination of Variable Definition Fault

The origination conditions set for variable definition fault appears in Table 1.

Howden's *WMT* and Budd's *Estimate* include identical rules that are directed toward the detection of variable definition faults. To satisfy their criteria, a test data set T must satisfy the following rule.

> For each each assignment $V := EXP$ at each node n, T contains a test datum $t \in DOMAIN\ (n)$ such that $exp \neq v$.

A test datum that satisfies this rule fulfills the origination condition set. The origination condition set (as developed in the last section), however, contains another condition, $(\overline{v} \neq v)$, that must be satisfied whenever $(exp \neq v)$ is unsatisfiable. Neither *Estimate* nor *WMT* satisfy this other condition, and thus a potential error caused by a variable definition fault may remain undetected by these test data selection criteria. Consider the following example:

```
1  read A, B, C;
2  if C = A+B then
3     C := A+B;
       ⋮
```

The condition $(a + b \neq c)$, which is the evaluation of $(exp \neq v)$, is unsatisfiable at node 3. It is possible, in fact quite likely, however, that the definition at node 3 should be to a variable other than C, such as to D. To detect such a variable definition fault, the values of C and D must differ before execution of node 3, a condition not required by these other criteria. Thus, *Estimate* and *WMT* are sufficient to originate a potential error for a variable definition fault, but do not guarantee origination for this class of faults.

Foster's *ESTCA* contains no rules that approach origination of an error for variable definition fault.

4.2 Transfer through Arithmetic Operators

The transfer conditions through arithmetic operators appear in Table 3

Estimate includes two rules that apply to arithmetic expressions.

> For each binary arithmetic expression EXP_1 aop EXP_2 at each node n, T contains a test datum $t \in DOMAIN\ (n)$ such that $exp_1 > 2$ and $exp_2 > 2$.
>
> For each binary arithmetic expression EXP_1 aop C or C aop EXP_1 (where C is a constant) at each node n, T contains a test datum $t \in DOMAIN(n)$ such that $exp_1 > 2$.

A test datum satisfying these rules of *Estimate* satisfies transfer conditions through all arithmetic operators but exponentiation. These rules, however, are more restrictive than necessary; unsatisfiability of the rules does not guarantee absence of a fault. Consider the arithmetic expression in the following:

```
1  read Y, Z;
2  X:=1;
3  if Y ≤ X then
4     A := X*Y;
       ⋮
```

where X contains a potential error at node 4. No test datum satisfies these rules for this node. However, a test datum such that $y \neq 0$ transfers any potential error in X. Thus, *Estimate* is sufficient to transfer through most but not all arithmetic operators but does not guarantee transfer.

WMT also includes two rules that apply to arithmetic expressions.

> For each arithmetic expression EXP at node n, T contains test data $t_a, t_b \in DOMAIN\ (n)$ such that:
>
> a. the expression EXP is executed;
>
> b. $exp \neq 0$.
>
> For each arithmetic expression EXP at node n, where k is an upper bound on the exponent in the expression, T contains test data $t_1, t_2, ...t_{k+1} \in DOMAIN\ (n)$ such that $\{t_1, t_2, ...t_{k+1}\}$ is any cascade set of degree $k+1$ in $DOMAIN\ (n)$.

The first rule satisfies the transfer conditions for all arithmetic operators but the exponentiation operator. Consider the following module fragment:

```
1  read X, Y;
2  A := X**Y;
   :
```

where a potential error originates in Y. The test datum (1,2) satisfies this rule; however, a potential error in Y does not transfer through the exponentiation operator with $x = 1$. Because the degree of the expression in node 2 is unknown, a proper cascade set cannot be selected, and thus, the second rule cited does not apply. In sum therefore, *WMT* only partially guarantees transfer through arithmetic operators.

Foster prescribes one rule in *ESTCA* that deals with arithmetic operations.

> For each assignment $V := EXP$ at node n and for each variable W referenced in EXP, T contains a test datum $t \in DOMAIN\ (n)$ such that:
>
> a. w has a measurable effect on the sign and magnitude of *exp*.

This rule attempts to disallow the effect of a variable or subexpression to be masked out by other operations in the statement. While the specifics of how this rule is applied are unclear, one might interpret this as requiring transfer of a potential error through arithmetic operators. Under the broadest interpretation, therefore, *ESTCA* guarantees transfer through arithmetic operators.

4.3 Summary of Analysis

All three test data selection criteria fail in their ability to guarantee revelation of a context error for variable definition fault. Foster's *ESTCA* is completely insufficient even to reveal a context error. Budd's *Estimate* and Howden's *WMT* are sufficient to reveal a context error, but if their rules are unsatisfiable, a variable definition fault may exist but not reveal a context error.

In terms of transfer, each of the criteria approaches transfer through arithmetic operators. None, however, contain any prescription for how rules should be combined and thus each fails to force a context error to transfer to a statement where the erroneous-valued variable is referenced. If these fault-based criteria were used in conjunction with a data flow path selection technique [CPRZ85,CPRZ86,LK83,Nta84,RW85], the context error might reach a reference through the selection of paths covering pairs of definitions and uses. This combination of techniques, however, does not guarantee that the potential error transfers to affect the outcome of the statement reached for a definition and use pair.

Our analysis for six fault classes showed that this failure to integrate rules is pervasive to all three criteria. In general, a major weakness of each criterion is that it does not require data that satisfy origination conditions to also satisfy transfer conditions, and thus transfer of an originated potential error is not guaranteed. In fact, these criteria do not guarantee that origination and computational transfer throughout the node containing the fault are satisfied by the same test datum; hence, revelation of a context error is not even guaranteed for most fault classes.

The analysis of the three test data selection criteria for revelation of context errors is summarized in Table 4. The entry insufficient means that the criterion does not include a rule that satisfies the condition. The entry sufficient means that the criterion includes a rule that when satisfied fulfills the condition. The entry partially sufficient means that the criterion includes a rule that is sufficient to distinguish many but not all of the alternates or transfer through many but not all of the operators. The entry guarantees means that the criterion includes a rule that satisfies the conditions when the conditions are feasible, while partially guarantees means the criterion includes a rule that satisfies many but not all of the conditions when feasible.

The analysis presented in this table is not intended to be a complete review of the error detection capabilities of these criteria. Only six fault classes are considered, whereas a full analysis would require a broader classification of faults. In addition, the analysis presented in this paper does not represent full discussion of the capability of these criteria to reveal an output error for variable definition fault. We have only considered transfer through arithmetic operators while transfer through other operators, such as relational and boolean, may be applicable. Because these criteria fail to integrate rules that satisfy origination with those that satisfy transfer, consideration of other applicable transfer yields no better results. The analysis presented in this section, however, provides insight into the use of the RELAY model as an analysis tool. Moreover, it demonstrates how a test data selection criterion based on the RELAY model provides better error detection capabilities.

5 Use of RELAY for Test Data Selection

As an example of how RELAY can be used to select test data, consider the potential faults at node 1 in the example shown in Figure 1. If we assume that the module is "almost correct", then the statement at node 1 might have a variable reference fault, a variable definition fault, or an

	Budd's *Estimate*	Howden's *WMT*	Foster's *ESTCA*
Origination			
1. Constant Reference Fault	guarantees	guarantees	guarantees
2. Variable Reference Fault	insufficient	sufficient	insufficient
3. Variable Definition Fault	sufficient	sufficient	insufficient
4. Boolean Operator Fault	guarantees	guarantees	guarantees
5. Relational Operator Fault	guarantees	guarantees	sufficient
6. Arithmetic Operator Fault	partially sufficient	partially guarantees	insufficient
Transfer			
1. Assignment Operator	guarantees	guarantees	guarantees
2. Boolean Operator	guarantees	guarantees	guarantees
3. Relational Operator	insufficient	insufficient	insufficient
4. Arithmetic Operator	partially sufficient	partially guarantees	guarantees
Revelation of a Context Error			
1. Constant Reference Fault	insufficient	insufficient	insufficient
2. Variable Reference Fault	insufficient	insufficient	insufficient
3. Variable Definition Fault	sufficient	sufficient	insufficient
4. Boolean Operator Fault	insufficient	insufficient	guarantees
5. Relational Operator Fault	insufficient	insufficient	insufficient
6. Arithmetic Operator Fault	insufficient	insufficient	insufficient

Table 4: Analysis Summary

arithmetic operator fault. These classes are three of six for which RELAY has been instantiated thus far. Both the origination conditions for these three classes as well as the applicable computational transfer conditions are reported in this paper. Let us first consider the conditions that must be satisfied to guarantee that a context error is revealed at node 1 for these three classes of faults. Then, we construct the chain transfer conditions necessary to reveal an output error. Finally, we append each transfer condition to each origination condition to provide the output error revealing condition.

variable referenced	origination condition set
V	$\{[\overline{v} \neq v \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$

Table 5: Origination Condition Set for Variable Reference Fault

expression	origination condition set
$(exp_1 * exp_2)$	$\{[(exp_1 * exp_2) \neq (exp_1 + exp_2)]$, $[(exp_1 * exp_2) \neq (exp_1 - exp_2)]$, $[(exp_1 * exp_2) \neq (exp_1 / exp_2)$ or $[(exp_1 * exp_2) \neq (exp_1 ** exp_2)]\}$.

Table 6: Origination Condition Set for $*$ Operator Fault

5.1 Context Error Revealing Conditions

Consider first a potential variable reference fault at node 1; either the reference to U or the reference to V could be incorrect. The origination condition set for this class of faults is shown in Table 5. When evaluated for the reference to U at node 1 [12] , the origination condition set is $\{[u \neq \overline{u}] \mid \overline{u} \in \{A, B, V, W, X, Y, Z\}\}$. The transfer condition for multiplication is shown in Table 3; when evaluated for node 1, this requires $(v \neq 0)$. Thus, the context error revealing condition set is $\{[(u \neq \overline{u}$ and $(v \neq 0)] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$ (as constructed in section 2). A similar condition set must be satisfied to reveal a context error for a potentially incorrect reference to V; it is $\{[(v \neq \overline{v}$ and $(u \neq 0)] \mid \overline{V} \in \{A, B, U, W, X, Y, Z\}\}$.

Next, consider a potential arithmetic operator fault. The multiplication operator at node 1 could potentially be any other arithmetic operator. The origination condition set for an incorrect multiplication operator, where the alternative operators are $+, -, /, **$, is shown in Table 6. Thus, to guarantee origination of a potential error for a potential arithmetic operator fault in this node, a test data set must satisfy the origination condition set $\{[(u*v) \neq (u+v)], [(u*v) \neq (u-v)], [(u*v) \neq$

[12] These conditions refer to values before execution of the node since they are requirements on test data selected for the node.

$(u/v)], [(u*v) \neq (u**v)]\}$. The potential error originated by a potential arithmetic operator fault in this statement must transfer through the assignment operator in order to reveal a context error; this requires no additional conditions. The context error revealing condition set is, therefore, the same as the origination condition set.

Now, consider a potential variable definition fault. The origination condition set for this class of faults is shown in Table 1. Again assuming all eight variables are of the same type, the origination condition set is $\{[(x \neq \bar{x}$ **or** $(u*v \neq x)] \mid \bar{X} \in \{A, B, U, V, W, Y, Z\}\}$. There are no transfer conditions required to reveal a context error for this potential fault once a potential error is originated. Thus, the context error revealing condition set for this potential fault is the same as the origination condition set.

5.2 Chain Transfer Conditions

We are now in a position to consider the additional requirements necessary to guarantee revelation of an output error for the potential faults discussed above. There are two paths in this module, and for each path, we must construct the chain transfer condition for each def-use chain, where the definition at node 1 reaches an output.

Let us first consider the path $[(1,2),(2,3),(3,4),(4,7),(7,8)]$, where the false branch of the condition at node 4 is taken. Along this path, the only def-use chain consists of the definition of X at node 1, the use of X at node 7 to define W, and the output of W at node 8. The corresponding chain transfer condition (as described in section 2) is $[(a \geq b)$ and $(b \neq 0)]$.

Now, consider the second path $[(1,2)(2,3)(3,4),(4,5),(5,6),(6,8)]$, along which there are two def-use chains beginning with the fault location and ending with an output statement. One such def-use pair consists of the definition of X at node 1 followed by the use of X in the output statement at node 6. The data flow transfer condition to force this def-use pair is $(a < b)$ with no other node transfer conditions required. Thus, the chain transfer condition for this def-use chain is $[(a < b)]$.

The other def-use chain along this path consists of the definition of X at node 1, followed by the use of X at node 3 where Y is defined, followed by the use of Y at node 5 to define W, followed by the use of W in the output statement of node 8. The transfer from node 1 to node 3 is sequential, so there is no data flow transfer condition required. At node 3, the potential error in X must transfer through the addition and the exponentiation to the assignment of Y; this is defined by

the node transfer condition for node 3. The transfer conditions for addition and exponentiation are shown in Table 3. The transfer condition for + is trivial — that is, $(true)$. There are several possible ways, however, in which exponentiation could mask out a potential error in the expression $(X + 3)$. The most obvious way is if the value of Z is zero; thus one transfer condition is $(z \neq 0)$. Another possibility is if Z is even and the potential error expression containing X is the negation of the potentially correct expression. It is sufficient, but not necessary, to simply constrain Z to be odd. Thus, a sufficient node transfer condition for node 3 is $((z \neq 0)$ and $(\mathbf{odd}(z)))$ [13] . The potential error must also transfer from this definition of Y at node 3 to the use of Y at node 6, which is defined by the data flow transfer condition $(a < b)$. Within node 6, the potential error in Y must transfer through the multiplication by Z to the definition of W, which requires satisfaction of the node transfer condition $(z \neq 0)$. Thus, revelation of an output error for this def-use chain is achieved by satisfying the following chain transfer condition — $[(z \neq 0)$ and $(\mathbf{odd}(z))$ and $(a < b)]$.

5.3 Output Error Revealing Conditions

We can now construct the output error revealing condition sets for the potential faults of node 1. A context error at node 1 may transfer to an output error through any of the def-use chains discussed above. This is described by the disjunction of the chain transfer conditions for each def-use chain — $[((a \geq b)$ and $(b \neq 0))]$ or $[(a < b)]$ or $[((z \neq 0)$ and $\mathbf{odd}(z))$ and $(a < b)]$. To construct the revealing condition set for a particular fault class, we conjoin each condition in the context error revealing condition set with this disjunction of the chain transfer conditions. For the fault classes considered above, this results in the following output error revealing condition sets.

[13] For simplicity, we will continue to use this sufficient constraint.

incorrect reference to U

$$\{\,[\,(u \neq \bar{u}) \text{ and } (v \neq 0) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,] \mid \bar{U} \in \{A, B, V, W, X, Y, Z\}\,\}$$

incorrect reference to V

$$\{\,[\,(v \neq \bar{v}) \text{ and } (u \neq 0) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,] \mid \bar{V} \in \{A, B, U, W, X, Y, Z\}\,\}$$

incorrect arithmetic operator

$$\{\,[\,((u{*}v) \neq (u{+}v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,],\ [\,((u{*}v) \neq (u{-}v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,],\ [\,((u{*}v) \neq (u/v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,],\ [\,((u{*}v) \neq (u{**}v)) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,]\,\}$$

incorrect variable definition to X

$$\{\,[\,(x \neq \bar{x}) \text{ or } (u{*}v \neq x) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))\,] \mid \bar{X} \in \{A, B, U, V, W, Y, Z\}\,\}$$

5.4 Test Data Selection

If we blindly select test data to satisfy these revealing condition sets, then we might potentially select 25 test data. There is much overlap, however, between these condition sets. In fact, selection of a single test datum such that

$$\forall\ VAR \in \{A, B, W, Y, Z\} \mid [\,(u \neq v \neq x \neq var) \text{ and } (u \neq v \neq 0) \text{ and } ((a \geq b) \text{ and } (b \neq 0)) \text{ or } (a < b) \text{ or } ((a < b) \text{ and } (z \neq 0) \text{ and } \mathbf{odd}(z))]$$[14]

[14]Because this is the first node in the module, no additional conditions restrict the domain from which the test data are selected.

satisfies the revealing condition sets for both potential variable reference faults and the variable definition fault. The single test datum $(a = 2, b = 3, u = 4, v = 5, w = 6, x = 7, y = 8, z = 9)$ satisfies this sufficient output error revealing condition for these two fault classes and transfers an error along the chain through the definitions of X at node 1, Y at node 3, and W at node 5. Upon examination of the revealing condition set for an arithmetic operator fault, we see that this test datum satisfies each condition in that set as well. This single test datum, therefore, is sufficient to guarantee detection of any fault in node 1 of these classes. This means that if execution of the module on this test datum provides correct results, we know that node 1 does not contain any fault of these classes. While we could have selected test data that executes all def-use chains, execution of such data would not provide any additional information about the presence or absence of faults at node 1 for the chosen fault classes. Under the assumption that the module is "almost correct", we have guaranteed that node 1 is correct with respect to the fault classes considered. This testing process must, of course, be undergone for all nodes in the module to guarantee detection of any such fault in the module.

5.5 A Testing Tool

As a formal model of error detection, RELAY provides the basis for a testing tool. Such a tool would begin with the instantiated origination condition sets and transfer conditions for chosen classes of faults. This information is module independent and serves to specialize the testing tool for the detection of specific types of faults.

For a given module, the RELAY tool would develop the revealing condition sets for the nodes of interest, based on the chosen fault classification. Construction of these conditions would rely heavily on symbolic techniques [CR85,RC85]. The construction of path conditions, as done by symbolic evaluation, is a useful technique for determining the domains of the nodes as well as the data flow transfer conditions. The symbolic values for the variables, constructed as the path computation by symbolic evaluation, are required for evaluation of the origination condition sets and the computational transfer conditions. This reliance on symbolic evaluation could easily be supported by the incremental use of a flexible symbolic evaluation tool such as ARIES [EZ86], which is being developed as part of the ARCADIA software development environment [TCO*86].

Determining the revealing condition sets for each fault at each node independently is clearly

computationally expensive. As seen in the previous section, however, the tool can cut the expense by considering all fault classes that may occur at a node and determining the chain transfer conditions for all faults at that node. Furthermore, the construction of the chain transfer conditions can be performed simultaneously in an incremental fashion using global symbolic evaluation techniques [CR85,CHT79].

Once the revealing condition sets have been constructed, the RELAY tool can be used for test data selection and/or evaluation. Since the RELAY model of error detection assumes that the module being tested is almost correct, the module has presumably passed some high level testing phase. The tool, therefore, should first determine what revealing condition sets are satisfied by the test data selected during this previous testing phase. Test data must then be selected for any revealing condition set not yet satisfied. Since some of the revealing conditions may already be satisfied, the tool is more efficient because determining that a condition is satisfied is less costly than solving that condition and retesting. We believe an automated testing tool based on RELAY that guides in the selection of test data that is sensitive to a chosen fault classification is feasible.

6 Conclusion

In this paper, we present the RELAY model of error detection, discusses how RELAY represents an improvement to fault based testing, and demonstrate its use as a criterion for test data selection. The model itself defines generic origination and transfer conditions that must be satisfied to guarantee the detection of an error. To use RELAY as a test data selection criterion, a fault classification is chosen and the origination condition sets and the applicable transfer conditions are instantiated for that fault classification. For each node in a module, the origination condition sets are evaluated for the potential faults and the applicable computational and data flow transfer conditions are added to each origination condition in the origination condition sets. This develops revealing condition sets whose satisfaction by test data is capable of guaranteeing detection of any fault in the chosen fault classification.

This paper provides an example of the selection of test data for a chosen fault classification. Test data is selected for one node, which potentially has faults in three classes. The application of the origination condition sets is straightforward, and it is relatively easy to determine the data

flow and computational transfer conditions for all def-use chains from the potential fault to output. In general, finding a sufficient output error revealing condition set is not difficult since it requires determining only a single def-use chain from the potential fault to an output. When test data is selected to satisfy a sufficient revealing condition set for some fault class, correct execution guarantees that the module does not contain a fault of the class. If that set is not satisfiable, however, absence of a fault in the class is not guaranteed; the complete revealing condition set must be considered. Determining the complete, necessary condition set, however, may be more difficult since it requires determining all possible def-use chains from the potential fault to output. This is particularly complex when a potential error transfers through a looping construct.

This paper also shows how RELAY can be used to evaluate the error detection capabilities of other testing techniques. This analysis[RT86a] demonstrates how the rules of a test data selection criterion must be carefully designed and tightly integrated to reveal an error for any potential fault by showing how other techniques have failed to accomplish this precision. Without this precise analysis, it is easy to arrive at test data selection rules that do not guarantee the detection of a fault and may not even be sufficient to do so. Using RELAY, we have evaluated where previous criteria have failed in this regard. Furthermore, RELAY points out the flaws in the independent selection of paths and test data, which is common to so many testing approaches. RELAY requires test data that not only originates an error but also transfers that error along a path to output.

We continue to extend the RELAY model of error detection, to evaluate its capabilities by instantiating it for other classes of faults, and to analyze other testing criteria using RELAY. In addition, we are investigating the implementation of a tool to automate the selection of test data based on the RELAY model.

REFERENCES

[Bud81] Timothy A. Budd. Mutation analysis: ideas, examples, problems and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148, North-Holland, 1981.

[Bud83] Timothy A. Budd. *The Portable Mutation Testing Suite.* Technical Report TR 83-8, University of Arizona, March 1983.

[CHT79] Thomas E. Cheatham Jr., Glenn H. Holloway, and Judy A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, SE-5(4),

July 1979.

[CPRZ85] Lori A. Clarke, A. Podgursky, D.J. Richardson, and S.J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London, England, August 1985.

[CPRZ86] Lori A. Clarke, A. Podgursky, D.J. Richardson, and S.J. Zeil. An investigation of data flow path selection criteria. In *Proceedings of the ACM SIGSOFT/IEEE Workshop on Software Testing*, pages 23–32, Banff, Canada, July 1986.

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

[EZ86] Ed C. Epp and Steven J. Zeil. *ARIES: A Multi-Lingual Interpreter for a Tool-Fragment Environment.* Technical Report 57, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

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

[Fos83] Kenneth A. Foster. Comment on the application of error-sensitive testing strategies to debugging. *ACM Software Engineering Notes*, 8(5):40–42, October 1983.

[Fos84] Kenneth A. Foster. Sensitive test data for logical expressions. *ACM Software Engineering Notes*, 9(3), July 1984.

[Fos85] Kenneth A. Foster. Revision of an error sensitive test rule. *ACM Software Engineering Notes*, 10(1), January 1985.

[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.

[How78] William E. Howden. Introduction to the theory of testing. In Edward Miller and William E. Howden, editors, *Tutorial: Software Testing and Validation Techniques*, pages 16–19, IEEE, New York, 1978.

[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.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[MH81] Larry J. Morrell and Richard G. Hamlet. *Error Propagation and Elimination in Computer Programs.* Technical Report 1065, University of Maryland, July 1981.

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

[Nta84] Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[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.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[TCO*86] R.N. Taylor, L.A. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young. Arcadia: a software development environment research project. In *Proceedings of IEEE Computer Society Second International Conference on Ada Applications and Environments*, April 1986.

[Wey81] Elaine J. Weyuker. *An Error-based Testing Strategy.* Technical Report 027, Computer Science, Institute of Mathematical Sciences, New York University, January 1981.

[Wey82] Elaine J. Weyuker. On testing nontestable programs. *The Computer Journal*, 25(4), 1982.

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