Test Data Selection Using
the RELAY Model of Error Detection

Debra J. Richardson
Margaret C. Thompson

COINS Technical Report 87-78
August 1987

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

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 criterion for test data selection. The model is applied by choosing a fault classification, instantiating the revealing 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. In this paper, we summarize the RELAY model of error detection and its instantiation. RELAY is then applied to a sample program to illustrate its use as a test data selection criterion.

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*.

The next section summarizes the RELAY model and briefly describes its instantiation to develop revealing conditions for a class of faults; more detail is provided in a related paper[RT86b]. In the third section, we present an example of the application of RELAY as a test data selection criteria.

---

[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.

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. 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

---

[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).

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 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

---

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

[5] For simplicity, the granularity of the context oracle is assumed to be the same as that of the control flow graph,

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 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. The

---

although this is not necessary.

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

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 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
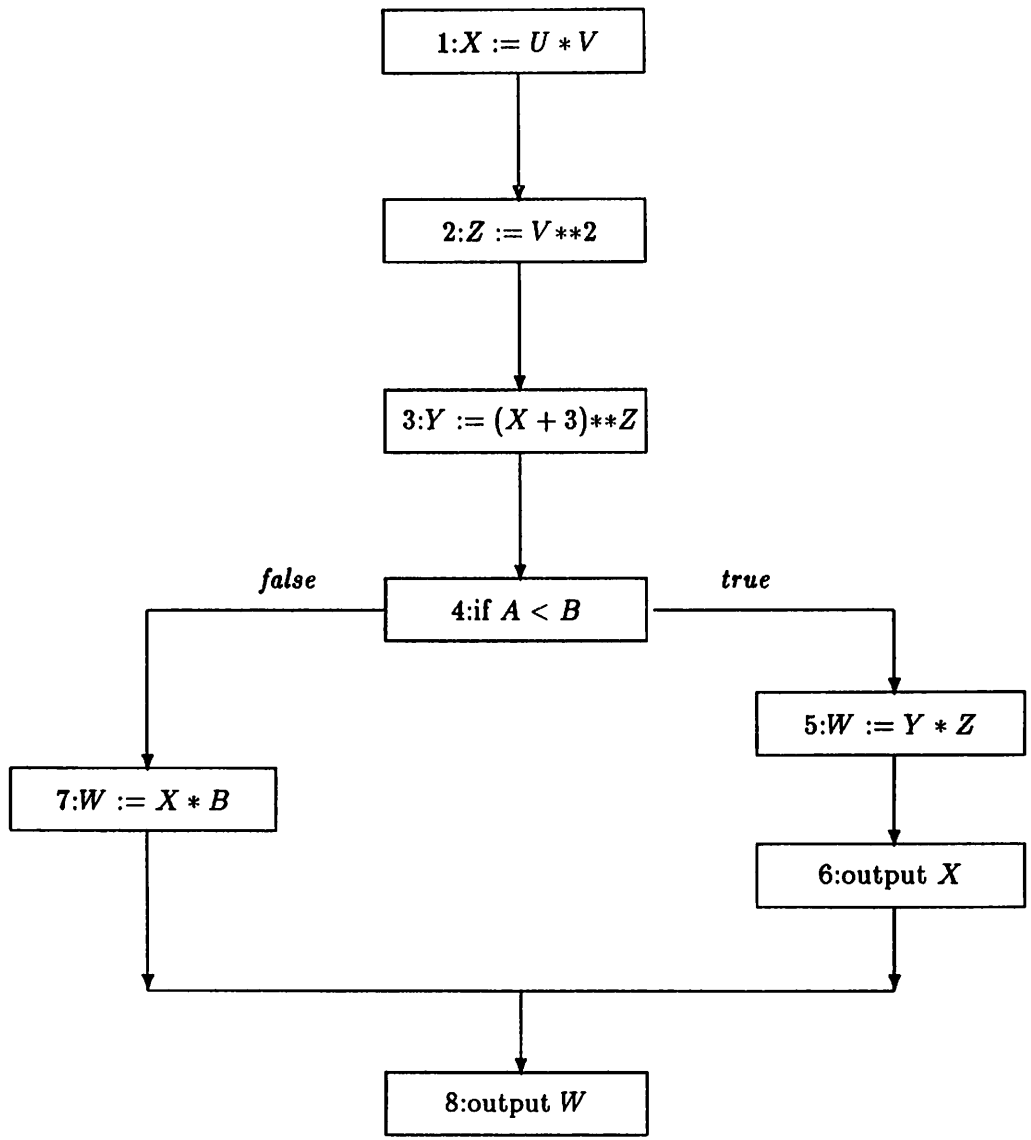
5

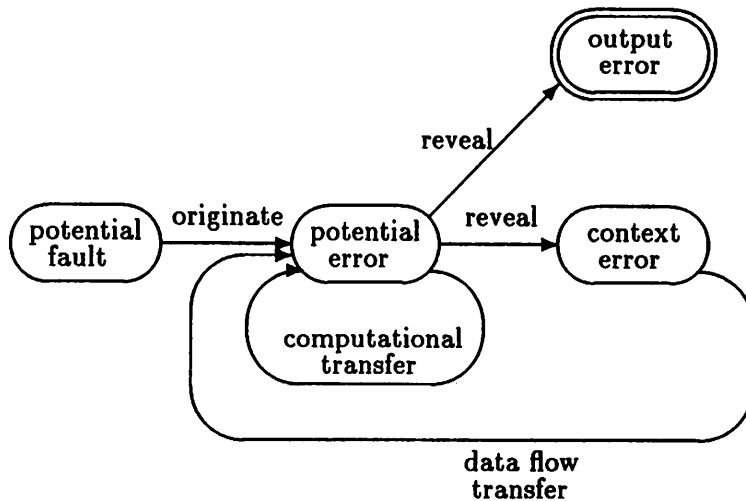**Figure 1: Module for Test Data Selection**

Figure 2: RELAY Model of Error Detection

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 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
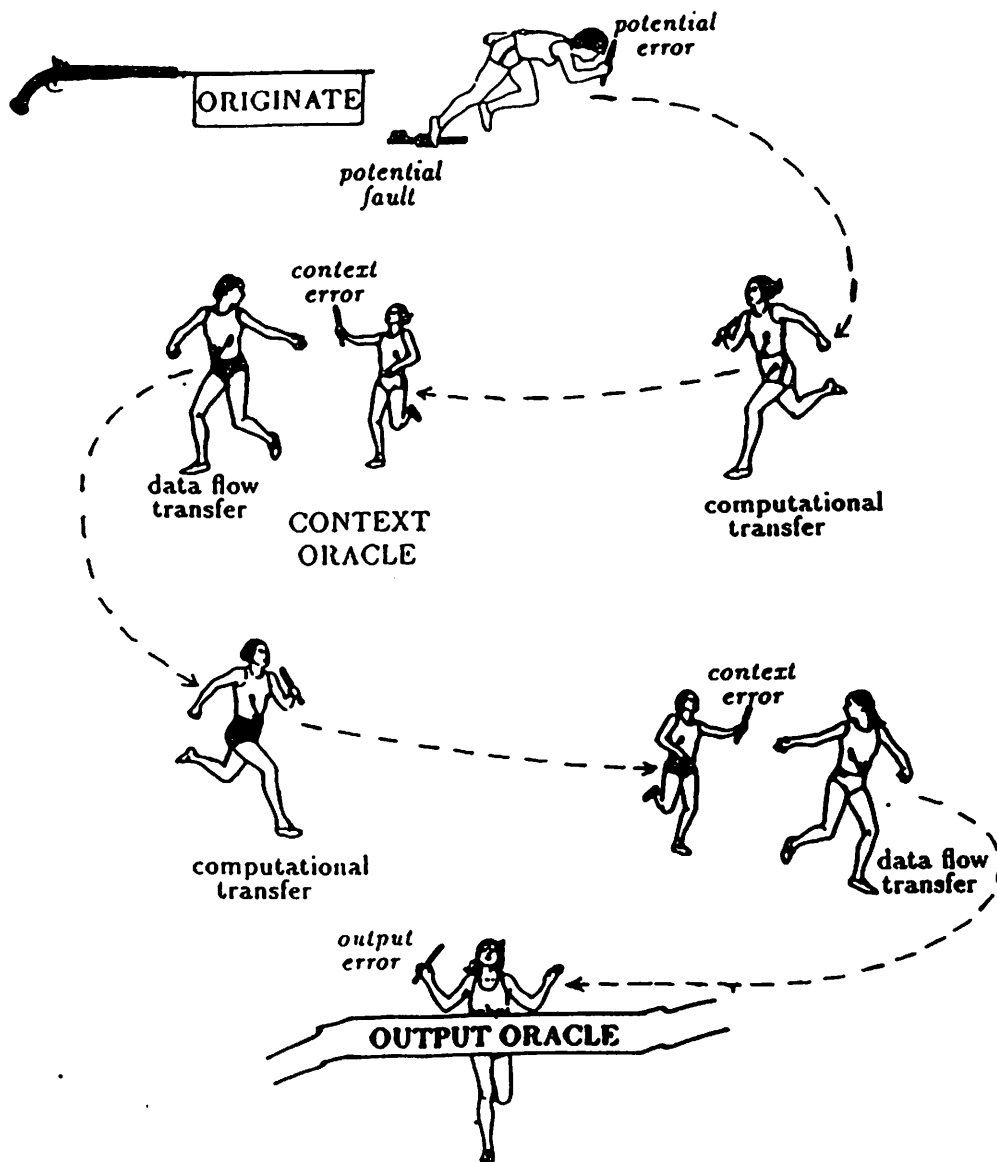
ORIGINATE

potential error

potential fault

context error

data flow transfer

CONTEXT ORACLE

computational transfer

computational transfer

context error

data flow transfer

output error

OUTPUT ORACLE

Figure 3: The Testing Relay

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

9

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

11

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, $\bar{U}$ [7], T contains a test datum where the value of U

---

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

12

is different from the value of $\bar{U}$ 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 $\{[u \neq \bar{u}] \mid \bar{U} \in \{A, B, V, W, X, Y, Z\}\}$. 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 $[(a \geq b) \text{ and } (b \neq 0)]$. Thus, the set $\{[(u \neq \bar{u} \text{ and } (v \neq 0) \text{ and } (a \geq b) \text{ and } (b \neq 0)] \mid \bar{U} \in \{A, B, V, W, X, Y, Z\}\}$ 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.

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. We have instantiated the model for six classes of faults: constant reference fault, variable reference fault, variable definition fault, boolean operator fault, relational operator fault, and arithmetic operator fault [RT86b]. 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 the next section.

3   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 arithmetic operator fault. These classes are three of six for which RELAY has been instantiated thus

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

Table 1: Origination Condition Set for Variable Reference Fault

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.

3.1 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 1. When evaluated for the reference to U at node 1 [8] , the origination condition set is $\{[u \neq \bar{u}] \mid \bar{u} \in \{A, B, V, W, X, Y, Z\}\}$. The transfer condition for multiplication is shown in Table 2; when evaluated for node 1, this requires $(v \neq 0)$.

Thus, the context error revealing condition set is $\{[(u \neq \bar{u}$ and $(v \neq 0)] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$ (as constructed in the previous section). A similar condition set must be satisfied to reveal a context error for a potentially incorrect reference to V; it is $\{[(v \neq \bar{v}$ 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

---

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

14

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

Table 2: Transfer Conditions through Arithmetic Operators

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 3: Origination Condition Set for $*$ Operator Fault

multiplication operator, where the alternative operators are $+, -, /, **$, is shown in Table 3. 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 $\{[(u*v) \neq (u+v)], [(u*v) \neq (u-v)], [(u*v) \neq (u/v)], [(u*v) \neq (u**v)]\}$. 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 4. Again assuming all eight variables are of the same type, the origination condition set is $\{[(x \neq \overline{x}$ or $(u*v \neq x)] \mid \overline{X} \in \{A, B, U, V, W, Y, Z\}\}$. 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.

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

Table 4: Origination Condition Set for Variable Definition Fault

3.2 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 the previous section) is $[(a \geq b) \text{ and } (b \neq 0)]$.

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 $(a < b)$ and no other node transfer conditions are 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 2. 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 $(\mathbf{odd}(z)))$ [9] . 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 $(a < b)]$.

3.3 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.

---

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

incorrect reference to U	$\{\,[\,(u \neq \bar{u})$ and $(v \neq 0)$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,]$ $\mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\,\}$
incorrect reference to V	$\{\,[\,(v \neq \ddot{v})$ and $(u \neq 0)$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,]$ $\mid \overline{V} \in \{A, B, U, W, X, Y, Z\}\,\}$
incorrect arithmetic operator	$\{\,[\,((u{*}v) \neq (u{+}v))$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,],$ 　$[\,((u{*}v) \neq (u{-}v))$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,],$ 　$[\,((u{*}v) \neq (u/v))$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,],$ 　$[\,((u{*}v) \neq (u{**}v))$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,]\,\}$
incorrect variable definition to X	$\{\,[\,(x \neq \ddot{x})$ or $(u{*}v \neq x)$ and 　　$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or 　　$((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))\,]$ $\mid \ddot{X} \in \{A, B, U, V, W, Y, Z\}\,\}$

3.4 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

$\forall\ VAR \in \{A, B, W, Y, Z\} \mid\ [\,(u \neq v \neq x \neq var)$ and $(u \neq v \neq 0)$ and

$((a \geq b)$ and $(b \neq 0))$ or $(a < b)$ or $((a < b)$ and $(z \neq 0)$ and $\mathbf{odd}(z))]$[10]

---

[10]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.

4 Conclusion

In this paper, we present the RELAY model of error detection 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, these origination condition sets are then evaluated for the potential faults. To develop the revealing condition sets, the applicable computational and data flow transfer conditions are added to each origination condition in the origination condition sets. Test data selected to satisfy these revealing condition sets is capable of guaranteeing detection of an error for 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.

RELAY has also been used to evaluate the error detection capabilities of other testing techniques[RT86a]. This analysis demonstrated 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.

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.

[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.

[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.

[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.

[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.