Title
An analysis of test data selection criteria using the RELAY model of error detection

Authors
Richardson, Debra J.
Thompson, Margaret C.

Publication Date
1987

Peer reviewed

An Analysis of Test Data Selection Criteria Using the RELAY Model of Error Detection

(Technical Report 88-05)

Debra J. Richardson†
Margaret C. Thompson‡

July 1987

†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 work was supported by grants DCR-8404217 from the National Science Foundation, CCR-8704478 from the National Science Foundation with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6104), CCR-8704311 from the National Science Foundation with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6108), F30602-86-C-0006 from the Rome Air Development Center.

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 framework within which the capabilities of other testing criteria can be evaluated. In this paper, we analyze three test data selection criteria that attempt to detect faults in six fault classes. This analysis shows that none of these criteria is capable of guaranteeing error detection for these fault classes and points out two major weaknesses of these criteria. The first weakness is that the criteria do not consider 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. Their second weakness is failure to integrate their proposed rules; although a criterion may cause a subexpression to take on an erroneous value, there is no effort made to guarantee that the enclosing expression evaluates incorrectly. This paper shows how the test data selection criterion defined by RELAY overcomes these weaknesses.

1 Introduction

Many testing techniques [Bud83,Fos80,Ham77,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 are often sufficient to select data that cause the computation of erroneous values for particular faults but do not guarantee that these erroneous results are reflected in the output. Instead, the erroneous intermediate values are often masked out by later computations. This extremely common occurence is a type of "coincidental correctness," which is the bane of testing. Coincidental correctness occurs when no error is detected, even though a statement containing a fault has been executed; thus the effort put into selecting the data and the associated execution is for naught.

This paper reports on a study that analyzes several "fault-based" testing techniques in terms of their abilities to actually reveal errors. This analysis is based on the RELAY model of error detection, which formalizes a fault-based approach to testing. RELAY 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; more detail is provided in a related paper[RT86]. The third section briefly describes the instantiation of the model to develop revealing conditions for a particular class of faults. The origination and transfer conditions for six fault classes are found in Appendix A. In the fourth section, we present an evaluation of the error detection capabilities of three proposed test data selection criteria for these six fault classes using the model. In summary, we discuss the implications of the analysis and our future plans for RELAY.

2 RELAY: A Model of Error Detection

The RELAY model has three principal uses. First, it is a test data selection criterion that when used to test a program is capable of guaranteeing the detection of errors that result from some chosen class or classes of faults. Second, 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. Third, RELAY provides a method for analyzing the ability of other test data selection criteria to guarantee detection of errors for classes of faults. It is this third application that is the focus of this paper.

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 $n^\star$ in the hypothetically-correct module $M^\star$ — that is, $n \neq n^\star$. The evaluation of some expression EXP[1] in M, which contains a potential fault, and the corresponding expression $EXP^\star$ in $M^\star$ results in a **potential error** when $exp \neq exp^\star$. To discover a potential fault, erroneous results must appear for some test datum which requires the use of 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* is a relation that specifies acceptable output for any input. Execution of a module reveals an **output error** when the input-output pair is not in the oracle relation. 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, in addition to its correct output. Rather than waiting until output is produced to find errors, the tester might check the computation of the module at

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

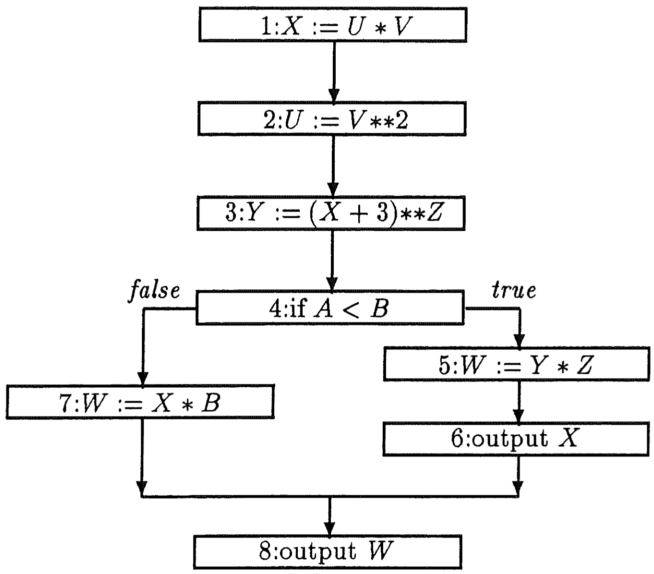

Figure 1: Module for Application of RELAY

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, monitoring of assertions, or run-time traces [How78]. Let us associate with the initial execution of a module a *context*, which contains the values of all variables after that execution. A *context oracle* is a relation that relates an initial execution to one or more acceptable contexts. Execution reveals a **context error** when the context is not accepted by the context oracle.

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 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

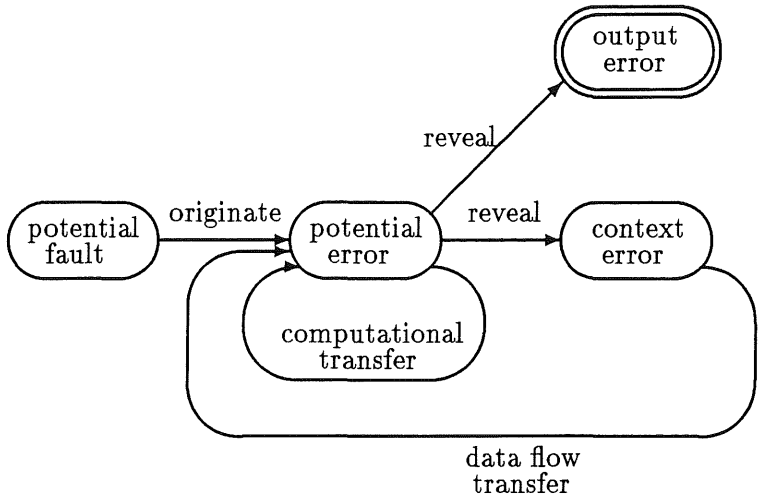

Figure 2: RELAY Model of Error Detection

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. Using the concepts of origination and transfer, RELAY develops *revealing conditions* that are necessary and sufficient to *guarantee* error detection — that is, any test data set that satisfies these conditions contains some test datum for which a potential error originates *and* transfers until it is detected by the oracle. 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. 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

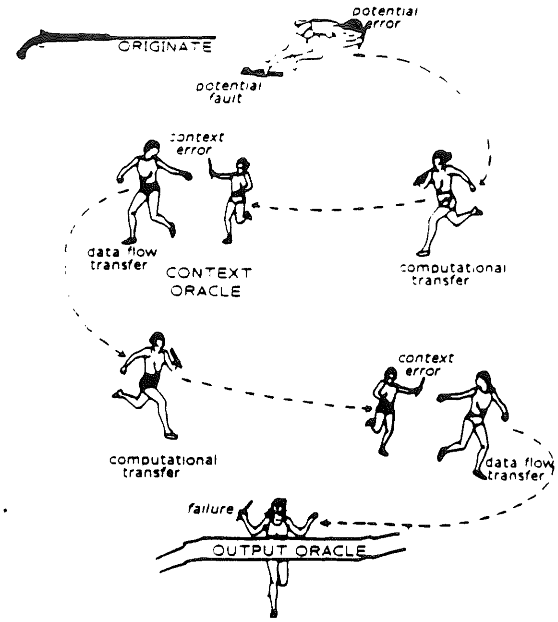

Figure 3: The Testing Relay

of that type.

Revealing 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 the domain of n, which is defined by the union of the domains of all initial paths ending at n. Because these conditions are both necessary and sufficient, if the conditions are *infeasible* within that domain, 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, $\overline{U}$ [2], 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.

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. The next section summarizes the instantiation of RELAY for six classes of faults. The instantiated origination and transfer conditions can be evaluated for the nodes in a module's control flow graph to provide the actual revealing condition sets that must be satisfied to guarantee the detection of any fault in the chosen classification. The actual revealing conditions for a module

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

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 at the end of the next section. The instances of the origination and transfer conditions can also be used to evaluate the ability of another test data selection criterion to guarantee detection of an error caused by the chosen classes of faults. RELAY is applied in this fashion to analyze three test data selection criteria for the six fault classes in section four. 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. This instantiation process is illustrated for the class of relational operator faults. We derive the origination condition set for this class and the computational transfer conditions through boolean operators since a relational expression may be contained within boolean expressions.

The class of relational operator faults is one of six for which RELAY is instantiated in [RT86]. The six classes are constant reference fault, variable reference fault, variable definition fault, boolean operator fault, relational operator fault, arithmetic operator fault. These six classes were selected because of their relevance to a number of test data selection criteria, which include those criteria analyzed here. The application presented provides revealing conditions for context errors for single statements potentially containing a fault in one of the six classes. Each of the six classes is a class of atomic faults, where a (potential) fault f_n is *atomic* if the node n differs from the correct node n^* by a single token. Moreover, the restriction to context errors means that only computational transfer need be considered at this time.

To determine the revealing conditions for a class of potential faults, we must instantiate the origination condition set for the class as well as the applicable computational transfer conditions. Thus, for the six fault classes, we derive origination conditions for each class as well as transfer conditions through all operators applicable to these faults — that is, assignment operator, boolean

	test data relation		
expression evaluated	$(exp_1 < exp_2)$	$(exp_1 = exp_2)$	$(exp_1 > exp_2)$
$(EXP_1 \leq EXP_2)$	*true*	*true*	*false*
$(EXP_1 < EXP_2)$	*true*	*false*	*false*
$(EXP_1 = EXP_2)$	*false*	*true*	*false*
$(EXP_1 \neq EXP_2)$	*true*	*false*	*true*
$(EXP_1 > EXP_2)$	*false*	*false*	*true*
$(EXP_1 \geq EXP_2)$	*false*	*true*	*true*

Table 1: Relational Operator Evaluation

operator, arithmetic operator, and relational operator.

3.1 Origination Conditions

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 the class of relational operator faults, where a potential error may result when a relational operator is mistakenly replaced with another relational operator. We consider six relational operators: $<, \leq, =, \neq, \geq, >$. Given a relational expression $(EXP_1 \mathbf{rop} EXP_2)$, if the relational operator **rop** is faulty, then the correct expression must be in the alternate set $\{(EXP_1 \overline{\mathbf{rop}} EXP_2) \mid \overline{\mathbf{rop}}$ is a relational operator other than **rop** $\}$.

As an example, let us construct the origination condition set for the relational operator $<$. We must determine the origination condition that distinguishes $(EXP_1 < EXP_2)$ from each alternate $(EXP_1 \, \overline{\mathbf{rop}} \, EXP_2)$. For any relational expression, there are three possible relations for which test data may be selected — $(exp_1 < exp_2)$, $(exp_1 = exp_2)$, $(exp_1 > exp_2)$. Table 1 enumerates the evaluation of any relational expression with data satisfying these three relations. For illustration, let us construct the origination condition for alternative operator $=$. As seen from Table 1 the original expression, $(EXP_1 < EXP_2)$, and alternative expression, $(EXP_1 = EXP_2)$, evaluate differently for any test datum satisfying either the relation $(exp_1 < exp_2)$ or the relation $(exp_1 = exp_2)$;

thus the condition $(exp_1 \leq exp_2)$ is sufficient for origination of a potential error. For a test datum satisfying the third possible relation, $(exp_1 > exp_2)$ the original and alternate expressions evaluate the same; hence, the condition $(exp_1 \leq exp_2)$ is also necessary for origination of a potential error. The origination condition to distinguish between $EXP_1 < EXP_2$ and $EXP_1 = EXP_2$, therefore, is $[exp_1 \leq exp_2]$. The origination conditions for the other alternative operators are derived similarly. The origination conditions for relational operator faults are summarized in Table 2.

operators	unsimplified origination condition	origination condition
$<, \leq$	$[exp_1 = exp_2]$	$[exp_1 = exp_2]$
$<, =$	$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)]$	$[exp_1 \leq exp_2]$
$<, \neq$	$[exp_1 > exp_2]$	$[exp_1 > exp_2]$
$<, \geq$	$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)$ or $(exp_1 > exp_2)]$	[*true*]
$<, >$	$[(exp_1 < exp_2)$ or $(exp_1 > exp_2)]$	$[exp_1 \neq exp_2]$
$\leq, =$	$[exp_1 < exp_2]$	$[exp_1 < exp_2]$
$\leq, \neq$	$[(exp_1 = exp_2)$ or $(exp_1 > exp_2)]$	$[(exp_1 \geq exp_2]$
$\leq, \geq$	$[(exp_1 < exp_2)$ or $(exp_1 > exp_2)]$	$[exp_1 \neq exp_2]$
$\leq, >$	$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)$ or $(exp_1 > exp_2)]$	[*true*]
$=, \neq$	$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)$ or $(exp_1 > exp_2)]$	[*true*]
$=, \geq$	$[exp_1 > exp_2]$	$[exp_1 > exp_2]$
$=, >$	$[(exp_1 = exp_2)$ or $(exp_1 > exp_2)]$	$[exp_1 \geq exp_2]$
$\neq, \geq$	$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)]$	$[exp_1 \leq exp_2]$
$\neq, >$	$[(exp_1 < exp_2)]$	$[exp_1 < exp_2]$
$\geq, >$	$[exp_1 = exp_2]$	$[exp_1 = exp_2]$

Table 2: Origination Conditions for Relational Operator Faults

The origination condition set for the class of relational operator faults for a particular operator is the set of all origination conditions that distinguish that original operator from some alternate. For a less than ($<$) fault, for instance, the origination condition set can be derived from Table 2 as $\{[exp_1 = exp_2], [exp_1 \leq exp_2], [exp_1 > exp_2], [true], [exp_1 \neq exp_2]\}$. The origination condition sets for other relational operator faults are derived similarly and stated in Table 3.

3.2 Transfer Conditions

A computational transfer condition guarantees that a potential error in an operand of an expression is not masked out by the computation of a parent operator. Thus, given an expression

operator	origination condition set
$<$	$\{[exp_1 = exp_2]$, $[exp_1 \leq exp_2]$, $[true]$, $[(exp_1 > exp_2)]$, $[exp_1 \neq exp_2]\}$
$\leq$	$\{[exp_1 = exp_2]$, $[exp_1 < exp_2]$, $[true]$ $[exp_1 \geq exp_2]$, $[exp_1 \neq exp_2]\}$
$=$	$\{[exp_1 \leq exp_2]$, $[exp_1 < exp_2]$, $[true]$, $[exp_1 > exp_2]$, $[exp_1 \geq exp_2]\}$
$\neq$	$\{[exp_1 > exp_2]$, $[exp_1 \geq exp_2]$, $[true]$, $[exp_1 \leq exp_2]$, $[exp_1 < exp_2]\}$
$\geq$	$[true]$, $[exp_1 \neq exp_2]$, $[exp_1 > exp_2]$, $\{[exp_1 \leq exp_2]$, $\{[exp_1 = exp_2]\}$
$>$	$\{[exp_1 \neq exp_2]$, $[true]$, $[exp_1 < exp_2]$, $[exp_1 \geq exp_2]$, $[exp_1 = exp_2]\}$

Table 3: Origination Condition Sets for Relational Operator Faults

$\mathbf{op}(\ldots, EXP, \ldots)$, where a potential error exists in EXP, the transfer condition guarantees that $\mathbf{op}(\ldots, exp, \ldots)$ also produces a potential error. More specifically, given EXP containing a potential fault and $\overline{EXP}$ an alternate, the existence of a potential error in exp implies that $exp \neq \overline{exp}$, and the transfer condition guarantees that $\mathbf{op}(\ldots, exp, \ldots) \neq \mathbf{op}(\ldots, \overline{exp}, \ldots)$.

Let us now continue with our illustration for relational operator faults. A relational expression may be contained within a boolean expression; thus, in order to develop revealing condition sets for the class of relational operator faults, we must also develop transfer conditions through boolean operators and must consider both unary and binary boolean operators.

Consider first transfer through a unary boolean operator. The unary boolean transfer condition guarantees that **not** (EXP_1) is distinguished from **not** $(\overline{EXP_1})$, where EXP_1 and $\overline{EXP_1}$ are distinguished. From Table 4, we see that no additional conditions are necessary for transfer of a potential error in a unary boolean expression because **not** $(exp_1) \neq$ **not** $(\overline{exp_1})$ if and only if $exp_1 \neq \overline{exp_1}$.

The binary boolean transfer conditions guarantee both that $(EXP_1\, \mathbf{bop}\, EXP_2)$ is distinguished from $(\overline{EXP_1}\ \mathbf{bop}\ EXP_2)$ and that $(EXP_2\ \mathbf{bop}\ EXP_1)$ is distinguished from $(EXP_2\ \mathbf{bop}\ \overline{EXP_1})$, whenever EXP_1 and $\overline{EXP_1}$ are distinguished. Since the binary boolean operators are commu-

exp_1	$\overline{exp_1}$	exp_2	exp_1 **and** exp_2	$\overline{exp_1}$ **and** exp_2	exp_1 **or** exp_2	$\overline{exp_1}$ **or** exp_2
true	*false*	*true*	*true*	*false*	*true*	*true*
true	*false*	*false*	*false*	*false*	*true*	*false*
false	*true*	*true*	*false*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*false*	*true*

Table 4: Boolean Expression Evaluation

tative, we need not develop separately the transfer conditions for a potential error in the right operand. The binary boolean transfer conditions depend upon the boolean operator. For the boolean operator **and**, we see from Table 4 that (exp_1 **and** exp_2) $\neq$ ($\overline{exp_1}$ **and** exp_2) only when $exp_2 = true$. Thus, exp_2 must be *true* to guarantee that a potential error in exp_1 transfers through the boolean operator **and**. For the boolean operator **or**, notice that (exp_1 **or** exp_2) $\neq$ ($\overline{exp_1}$ **or** exp_2) only when $exp_2 = false$. Hence, exp_2 must be *false* to guarantee transfer of the potential error in exp_1 through the boolean operator **or**.

The transfer conditions for boolean operators are summarized in Table 5. The transfer conditions through the operators applicable to the six fault classes are summarized in Appendix A.

operator	expression	transfer condition
not	**not**(exp_1) $\neq$ **not**($\overline{exp_1}$)	*true*
and	exp_1 **and** exp_2 $\neq$ $\overline{exp_1}$ **and** exp_2	exp_2 =*true*
or	exp_1 **or** exp_2 $\neq$ $\overline{exp_1}$ **or** exp_2	exp_2 =*false*

Table 5: Transfer Conditions for Boolean Operators

3.3 Revealing Conditions

In this section, we illustrate the formation of context error revealing conditions for the class of relational operator faults and demonstrate how these conditions can be used to select test data. Consider the module fragment and that portion of the control flow graph shown in Figure 4.

The relational operator at node 2 is potentially faulty. The origination condition set for the class of relational operator faults for $<$ in node 2 is $\{[x*y = z], [x*y > z], [x*y \leq z], [x*y \neq z], [$*true*

X, Y, Z : integer
B, C : boolean
1 input X, Y, Z, B, C
2 if $(X * Y < Z$ **or** $B)$ **and** C then

$\vdots$

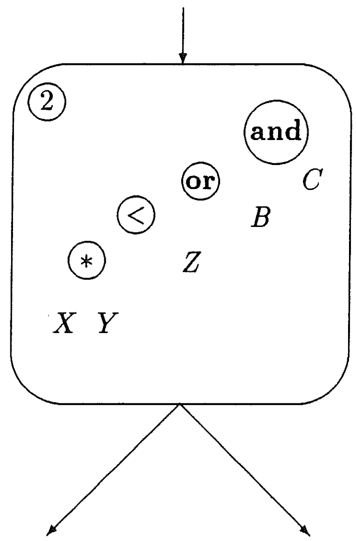

Figure 4: Module Fragment

]}. Examination shows that the origination conditions $[x * y = z]$ and $[x * y > z]$ are sufficient to satisfy the entire set. Thus, a sufficient origination condition set is $\{[x * y = z], [x * y > z]\}$. A potential error resulting from the $<$ in node 2 must transfer through the boolean operators **or** and **and**. The node transfer condition is simply

$(b = \mathit{false})$ and $(c = \mathit{true})$.

The origination condition set combines with the node transfer condition to form the following revealing condition set

$$\{[(x * y = z) \text{ and } (b = \mathit{false}) \text{ and } (c = \mathit{true})], \\ [(x * y > z) \text{ and } (b = \mathit{false}) \text{ and } (c = \mathit{true})]\ \}.$$

We are now in a position to select a test data set that satisfies the revealing condition set. A test datum that satisfies a revealing condition must be selected within the domain of the module; further it must be selected such that the revealing conditions are satisfied before execution of the node. Because the node for which we have developed revealing conditions is one of the first nodes of the module, selection of test data that satisfies the conditions is relatively easy. There are many possible test data sets that satisfy the revealing conditions developed for this example. Consider a test data set that contains the following two data $(1, 2, 2, \mathit{false}, \mathit{true})$ and $(1, 3, 2, \mathit{false}, \mathit{true})$. The first datum satisfies the first revealing condition in the revealing condition set, and the second datum satisfies the second revealing condition. If the $<$ operator should have been some other relational operator, then execution for these two test data will reveal a context error. If no context error is revealed, then the $<$ operator is correct.

4 Analysis of Related Test Data Selection Criteria

RELAY provides a sound method for analyzing the error detection capabilities of a 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 use the origination and transfer conditions for the six fault classes (provided in Appendix A) to analyze the error detection capabilities of three fault-based test data selection criteria — Budd's *Error-Sensitive Test Monitoring* [Bud81,Bud83], Howden's *Fault-based Functional Testing* [How82,How85,How87], and Foster's *Error Sensitive Test Case Analysis* [Fos80,Fos83,Fos84,Fos85]. Each of these criteria was selected because its author claims that it is geared toward detection of faults of the six classes previously discussed. 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 demonstrates how RELAY rectifies these common problems.

As noted, the application of RELAY discussed in this paper is limited to revealing context errors. Thus, the revealing condition set is necessary for the detection of a fault (as opposed to a potential fault), but not sufficient. This is because the context error introduced by satisfaction of these conditions may still be masked out by later computations on the path and thus not transfer to produce an output error. To describe the conditions under which a criterion guarantees the detection of a fault of a particular class through an output error, the revealing condition set must be augmented to include data flow transfer conditions. The analysis of the test data selection criteria to follow does not consider whether or not these criteria guarantee the detection of a fault through revealing an output error. As we shall see, however, this limitation is of little consequence, since for the most part, the criteria do not guarantee the revealing of a context error.

For each criterion, we first define it in the terminology provided in Section 2. Next, we examine the criterion's ability to satisfy the origination condition sets for each class of faults and also its ability to satisfy transfer conditions through the applicable operators. Then, for each class of faults, we discuss the circumstances in which the criterion will guarantee revelation of a context error, which requires that a single test datum must be selected to satisfy both a specific origination condition in the origination condition set and the node transfer condition. Thus, although a criterion may include rules that satisfy the origination condition set and the applicable transfer conditions, if the criterion does not explicitly force all such transfer conditions to be satisfied by the same data that satisfies the origination condition sets for a class of faults, revelation is not guaranteed for that class. In the case where only origination is guaranteed, revelation of a context error is guaranteed only when the potential fault is in the outermost expression of the statement or is contained only within expressions for which transfer conditions are trivial (e.g., unary boolean). Furthermore, recall that the test data selected for a particular node n must be in $DOMAIN(n)$. If no such data exists to satisfy the application of a particular rule in a criterion, then the rule is *unsatisfiable* for n. When no alternative selection guidelines are proposed, we do not assume the selection of any test data for an unsatisfiable rule.

In the discussion that follows, we provide counter examples to demonstrate when a criterion does not guarantee origination or transfer. When it is obvious that a criterion guarantees origination or transfer (e.g., a rule of a criterion is equivalent to an origination or transfer condition), we merely state this fact. Several of the conditions are trivially met by any criterion that satisfies statement coverage, these include origination of a constant reference fault and transfer through assignment operator. Since each of the three criteria analyzed here direct their selection of test data to each statement in a module, we will not belabor the satisfaction of these trivial conditions.

The following is not intended to be a complete analysis of the error detection capabilities of these criteria. Only those faults discussed in Section 4 are included in the discussion. A complete analysis must consider a more complete classification of faults. The analysis presented in this paper, however, provides insight into how our model of error detection can be used to analyze the strengths and weaknesses of testing criteria.

4.1 Budd's Estimate

Budd's *Error-Sensitive Test Monitoring (Estimate)* [Bud81,Bud83] is the first stage of Budd's Mutation Testing suite. For the most part, the testing suite is directed toward the evaluation of a test data set, but the first stage also provides a criterion that aids in the selection of test data. A test data set satisfying Budd's *Estimate* executes components in the program (e.g., variables, operators, statements, control flow structures) over a variety of inputs. The rules below outline test data that must be selected to pass *Estimate*.

Rule 1 For each variable V, T contains test data t_a, t_b, t_c, for some node n_a, n_b, n_c such that:

a. $t_a \in DOMAIN(n_a)$ and $v = 0$;
b. $t_b \in DOMAIN(n_b)$ and $v < 0$;
c. $t_c \in DOMAIN(n_c)$ and $v > 0$.

Rule 2 For each each assignment $V := EXP$ at each node n, T contains a test datum $t_a \in DOMAIN\ (n)$ such that:

a. $exp \neq v$.

Rule 3 For each binary logical expression, EXP_1 **bop** EXP_2 at each node n, T contains test data $t_a, t_b \in DOMAIN\ (n)$ such that:

a. $exp_1 = true$ and $exp_2 = false$;
b. $exp_1 = false$ and $exp_2 = true$.

Rule 4 For each edge $(n, n') \in E$, where $BP(n, n')$ is the branch predicate, T contains a test datum t_a such that:

a. $t_a \in DOMAIN(n)$ and $bp(n, n') = true$.

Rule 5 For each relational expression, EXP_1 **rop** EXP_2, at each node n, T contains test data $t_a, t_b, t_c, t_d \in DOMAIN(n)$ such that:

a. $exp_1 - exp_2 = 0$;

b. $exp_1 - exp_2 > 0$;

c. $exp_1 - exp_2 < 0$;

d. $exp_1 - exp_2 = -\epsilon$ or $+\epsilon$ (where ϵ is a "small" value).

Rule 6 For each binary arithmetic expression $EXP_1 \, \mathbf{aop} \, EXP_2$ at each node n, T contains a test datum $t_a \in DOMAIN\ (n)$ such that:

a. $exp_1 > 2$ and $exp_2 > 2$.

Rule 7 For each binary arithmetic expression $EXP_1\mathbf{aop}C$ ($C\mathbf{aop}EXP_1$), (where C is a constant), at each node n, T contains a test datum $t_a \in DOMAIN(n)$ such that:

a. $exp_1 > 2$.

First, let us consider *Estimate*'s ability to originate potential errors for the six fault classes. Rule 3 satisfies the origination condition set for boolean operator faults, and rule 5 satisfies the origination condition set for relational operator faults. Thus, *Estimate* guarantees origination of a potential error for boolean and relational operator faults.

Rule 1 appears to be concerned with forcing variables to take on a variety of values, which is one requirement for detection of variable reference faults. Consider the following segment of code:

```
1  read A, B;
2  X := 2*A;
      ⋮
```

The three test data (0,0), (3,3), and (-10,-10) satisfy rule 1, for variables A and B, but would not distinguish a reference to A from a reference to B at node 2. *Estimate* is not sufficient, therefore, to originate a potential error for a variable reference fault.

Estimate's rule 2 is directed toward the detection of variable definition faults. A test datum that satisfies this rule fulfills the origination condition set. The origination condition set, however, contains another condition, ($\overline{v} \neq v$), that must be satisfied if ($exp \neq v$) is infeasible. *Estimate* does not satisfy this other condition, and thus a potential error caused by a variable definition fault may remain undetected by *Estimate*. 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 *Estimate*. Thus, *Estimate* is sufficient to originate a potential error for a variable definition fault, but it does <u>not</u> guarantee origination for this class of faults.

Rule 6 is specifically concerned with arithmetic operator faults. Budd notes that test data satisfying this rule distinguishes between an arithmetic expression and an alternate formed by replacing the arithmetic operator by another arithmetic operator except for an addition or a subtraction operator replaced by a division operator (or vice versa). We agree that *Estimate* originates a potential error for any potential arithmetic operator fault in all but the four exceptions just cited. *Estimate*, however, is more stringent than necessary. When this rule is unsatisfiable — that is, no test datum exists such that $(exp_1 > 2)$ and $(exp_2 > 2)$ – there may exist an undetected potential error due to an arithmetic operator fault. For instance, consider the following code segment:

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

Note that at node 3, X and Y are restricted to values less than or equal to 2. In this case, *Estimate*'s rule is unsatisfiable, and no data must be selected to satisfy rule 6 for this statement. The expression $A := X{+}Y$ is an alternate that is not equivalent; there are data within the domain of the statement for which the two expressions evaluate differently — (e.g., $x = 2$ and $y = 1$). Thus, *Estimate* is only sufficient to originate a potential error for arithmetic operator faults except for the four noted exceptions, where *Estimate* is insufficient. *Estimate*, however, does not guarantee origination of a potential error for any arithmetic operator fault.

Let us now consider how *Estimate* does with transfer conditions. Note first that rule 3 fulfills and guarantees the transfer conditions through boolean operators.

Estimate's rule 5 is similar to one of the general sufficient transfer conditions shown in Appendix A, although *Estimate* does not consider the assumptions noted there. Even if these assumptions were taken into account, one of these sufficient conditions is not by itself sufficient to guarantee transfer through a relational operator. Consider the relational expression in the following:

```
read X,Y;
if X*Y ≥ 10 then
  ⋮
```

(where X and Y are of type integer). Suppose $X * Y$ should be $X + Y$. Test datum (11,1) would originate a potential error (since $11+1 \neq 11*1$), and satisfies rule 5 (since $X * Y$ differs from 10 by a small amount). However, the potential error is not transferred through the relational operators since both $11+1$ and $11*1$ are ≥ 10. Thus, *Estimate* is not sufficient to transfer through relational operators.

A test datum satisfying *Estimate*'s rule 6 satisfies transfer conditions through all arithmetic operators but the exponentiation operators. Rule 6, however, is more restrictive than necessary; when unsatisfiable, it does not guarantee absence of a fault. Consider the arithmetic expression in the following:

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

where a potential error originates in x at node 3. No test datum satisfies rule 6 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.

We are now in a position to determine the ability of *Estimate* to guarantee revelation of a context error for the six fault classes. In general, *Estimate* does not require data that satisfy origination conditions to also satisfy transfer conditions, and thus transfer of an originated potential error is not guaranteed. This is because *Estimate* does not prescribe any integration of the application of its rules.

	value of variable		
datum	a	b	z
i	1	3	*true*
ii	3	1	*true*
iii	2	2	*true*
iv	1	2	*false*
v	2	1	*true*
vi	3	1	*false*

Table 6: Sample Test Data Selected by *Estimate* for $(A < B)$ **or** Z

When two or more rules are applicable to an expression, *Estimate* does not dictate any way in which these two rules should interact. As an example, consider revelation of a context error for a potential relational operator fault in the expression $(A < B)$ **or** Z (assume for simplicity that A and B are of type integer) in the following:

```
1  read A, B, Z;
2  if (A < B) or Z then
     ⋮
```

The test data shown in Table 6 satisfies *Estimate*'s rules 3, 4 and 5 for this expression. Test data i, ii, and iii satisfy rule 5 for the relational expression containing the operator $<$. If this relational operator should have been any other relational operator, this test data would originate a potential error; for these test data, however, $z =$*true*, which will not transfer any potential error. Test data iii and iv satisfy rule 3 for the outer boolean expression containing **or**. Data v and vi satisfy rule 4 for the conditional statement. Test data iv and vi are the only data that would transfer any potential error originated in the relational expression; these data alone, however, are insufficient to guarantee origination of a potential error for the potential relational operator fault. If, for example, the $<$ should be $\leq$, no selected datum both originates and transfers a potential error caused by this fault. Thus, *Estimate* does not guarantee revelation of a context error for this potential relational operator fault.

The prescription of rule integration is lacking even in the repeated use of a single rule, as illustrated in the application of rule 3 to the boolean expression $(X$ **and** $Y)$ **or** Z in the following

	value of variable		
datum	x	y	z
i	*true*	*false*	*true*
ii	*false*	*true*	*true*
iii	*true*	*true*	*false*
iv	*false*	*false*	*true*

Table 7: Sample Test Data Selected by *Estimate* for $(X$ **and** $Y)$ **or** Z

code:

```
read X, Y, Z;
if (X and Y) or Z then
   ⋮
```

The test data shown in Table 7 satisfies *Estimate*'s rule 3 for the conditional expression in this example. Test data i and ii satisfy rule 3 for the inner boolean expression containing the operator **and**. Test data iii and iv satisfy rule 3 for the outer boolean expression containing **or**. If the inner operator should have been an **or**, test data i and ii would originate a potential error. For these test data, however, $z = true$, which will not transfer any potential error. Test data iii and v are the only data that would transfer a potential error originated at the inner expression, but for these test data, the values x and y would not originate a potential error. Thus, *Estimate* does not guarantee revelation of a context error for a potential boolean operator fault.

When origination of a potential error is guaranteed for a class of potential faults, revelation of a context error is guaranteed by *Estimate* only when the transfer conditions are trivial. In general, this occurs when the smallest expression containing the potential fault is the outermost expression in the node. The transfer conditions are always trivial for a variable definition fault. Since *Estimate* is sufficient to originate a potential error for this class, it is also sufficient to reveal a context error. Recall, however, that *Estimate* does not guarantee origination for this class.

4.2 Howden's *Fault-based Functional Testing*

Howden's *Fault-based Functional Testing* (*FFT*) [How82,How85,How87] is a test data selection criterion whereby test data is selected to distinguish between a component and alternative components generated by application of component transformations— e.g., substitution of one variable for another. Howden considers six transformations, which may be applied to various program components, and includes test data selection rules geared toward the detection of these transformations. Although Howden's transformations are presented quite differently than the six fault classes, each of these transformations result in one of the faults classes. The rules below specify test data intended to distinguish between a program component and alternatives generated by the transformations. These rules must be met by a test data set T to satisfy Howden's *FFT*.

Rule 1 For each reference to a variable V at node n, T contains a single test datum $t_a \in$ *DOMAIN* (n) such that for each other variable $\overline{V}$

a. $v \neq \overline{v}$ [3] .

Rule 2 For each assignment $V := EXP$ at node n, T contains a test datum $t_a \in$ *DOMAIN* (n) such that:

a. $v \neq exp$.

Rule 3 For each boolean expression $\mathbf{bop}(EXP_1, EXP_2, \ldots, EXP_i)$ at each node n, T contains test data $t_1, t_2, \ldots, t_{2^i} \in$ *DOMAIN* (n) such that $\{t_1, t_2, \ldots, t_{2^i}\}$ covers all possible combinations of *true* and *false* values for the subexpressions $EXP_1, EXP_2, \ldots, EXP_n$.

Rule 4 For each relational expression EXP_1 *rop* EXP_2, at each node n, T contains test data $t_a, t_b, t_c \in$ *DOMAIN* (n) such that:

a. $exp_1 - exp_2 = -\epsilon$ (where $-\epsilon$ is the negative difference of smallest satisfiable magnitude);

b. $exp_1 - exp_2 = 0$;

c. $exp_1 - exp_2 = +\epsilon$ (where ϵ is the positive difference of smallest satisfiable magnitude).

Rule 5 For each arithmetic expression EXP at node n, T contains test data $t_a, t_b \in$ *DOMAIN* (n) such that:

a. the expression is executed;

b. $exp \neq 0$;

[3] Howden proposes a more restrictive rule that is specifically concerned with array references. Since this rule is subsumed by rule 1, it does not provide any additional error detection capabilities and we do not include it here.

Rule 6 For each arithmetic expression EXP, where k is an upper bound on the exponent in the *exp*, at node n, 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).

Howden's *FFT* guarantees origination of a potential error for boolean and relational operator faults. Rule 3 satisfies the origination condition set for boolean operator fault, and rule 4 satisfies the origination condition set for relational operator fault.

Rule 1 is obviously directed toward detection of variable reference faults, and a test datum that satisfies this rule does satisfy the origination condition set. This rule, however, is more restrictive than required for this class of faults; it requires a single test datum to distinguish between the potentially incorrect variable reference and all other variable references. This rule may not be satisfiable although the origination condition set is feasible. In this case, a non-equivalent alternate may not be distinguished. Consider, for example, the reference to X at node 3 in the following module fragment:

```
1  read X, Y, Z;
2  if (X = Y) or (X = Z) then
3     A := 2 * X;
```

The origination condition set requires that a test set T contains a test datum such that $x \neq y$ and a test datum such that $x \neq z$ to distinguish an incorrect reference to X at this statement from possibly correct references to Y or Z. *FFT*, on the other hand, requires a single test datum such that $x \neq y$ and $x \neq z$. In this example, it is possible to satisfy the origination condition set with two test data, such as (1,1,2) and (1,2,1), but it is not possible to satisfy the *FFT* requirement which requires a single test datum. In this case, *FFT* will not necessarily distinguish a reference to Y or a reference to Z from a reference to X, although neither reference is equivalent. *FFT* is sufficient to originate a potential error, therefore, but does not guarantee origination for variable reference faults.

FFT's rule 2 is the same as *Estimate*'s rule 2, which is directed toward detection of variable definition faults. As noted in the discussion of *Estimate*, a test datum satisfying this rule will originate a potential error for a variable definition fault. This rule alone is incomplete, however, since it does not guarantee absence of a potential fault when it is unsatisfiable. Thus, *FFT* is sufficient but does not guarantee origination for this class.

Rule 5 and 6 are the only rules specifically directed toward exercising arithmetic expressions. For a potential error for a potential arithmetic operator fault that exchanges an addition operator for a subtraction operator (and vice versa), rule 5 will guarantee origination of a potential arithmetic operator fault. For other arithmetic operator faults, this rule is insufficient. Rule 6 is insufficient to guarantee origination of a potential error due to a potential arithmetic operator fault. This is because such a fault may change the degree of the arithmetic expression. Consider the arithmetic expression in node 2 of the following:

```
1  read X, Y;
2  A := X + Y;
   ⋮
```

Rule 6 requires a cascade set of degree 2 for this expression. One such set is $\{(0,0),(2,2)\}$. This set of test data, however, does not distinguish the expression $X+Y$ from the alternate $X*Y$.

Next, consider the ability of *FFT* to transfer a potential error. Rule 3 selects data that satisfies the boolean transfer condition and guarantees transfer through boolean operators.

FFT's rule 4 is similar to the sufficient transfer conditions for relational operators. For these transfer conditions to be sufficient, the two assumptions noted in the table in Appendix A must also hold. *FFT* does not consider these assumptions. Hence, even when *FFT*'s relational operator rule is satisfied, a potential error may not transfer through a relational operator. Consider transfer of a potential error in the arithmetic expression in node 2 through the relational operator $\geq$ in the following.

```
1  read X, Y;
2  if X * Y > 10 then
   ⋮
```

Suppose $X*Y$ should be $X+Y$, where X and Y are integers. The test data (3,3), (2,5), and (11,1) satisfy *FFT* rule 4. In all three cases, while a potential error originates, the potential error and the potentially correct expression share the same relationship to the right-hand-side of the relational expression, and no potential error transfers. Thus *FFT* is insufficient to transfer a potential error through a relational operator.

Rule 5 satisfies the transfer conditions for all arithmetic operators but the exponentiation operator. Consider the following module fragment:

$$\begin{array}{ll} 1 & \text{read } X, Y; \\ 2 & A := X{*}{*}Y; \\ & \vdots \end{array}$$

where a potential error originates in Y. The test datum (1,2) satisfies *FFT's* rule 5; however, a potential error in Y does not transfer through the exponentiation operator with $x = 1$. Rule 6 does not apply because a proper cascade set cannot be selected when the degree of the expression is unknown. *FFT*, therefore, only partially guarantees transfer through arithmetic operators.

As with *Estimate*, *FFT* does not require that a rule that satisfies origination be related to a rule that satisfies transfer. Thus, origination and transfer are not guaranteed to be satisfied by the same test datum, and hence revelation of a context error is not guaranteed. As with *Estimate*, this may happen both when the same rule applies for origination as for transfer and when different rules apply. Consider the same example expressions as in the discussion of Budd's *Estimate*.

Consider the relational expression $(A < B)$ **or** Z (where A and B are of type integer). The test data shown in Table 8 satisfies *FFT* for the relational expression as well as the boolean expression in this example. Test data i, ii, and iii satisfy rule 4, while test data iv, v, vi, and vii satisfy rule 3. Test data v and vii are the only data that could transfer a potential error originated in the relational expression; these two data alone, however, are insufficient to guarantee origination of a potential error for relational operator fault. If, for example, the $<$ operator is incorrect and should be $\leq$, no datum in the set both originates as well as transfers a potential error caused by the potential relational operator fault. Thus, *FFT* does not reveal a context error for this potential relational operator fault.

Now consider the boolean expression $((X$ **and** $Y)$ **or** $Z)$ which would be written by *FFT* as **or**(**and**$(X, Y), Z)$. The test data in Table 9 satisfies *FFT* for both boolean expressions contained in the example expression. Test data i, ii, iii, and iv satisfy rule 3 for the expression **and** (X, Y). Test data v, vi, vii, and viii satisfy rule 3 for the expression **or** (EXP, Z), where $EXP =$ **and**(X, Y). Test data ii and iii would originate a potential error if the **and** should be **or**, but for these test data, $z =$*true*, and any potential error does not transfer through the outer **or**. Test data vi and

	value of variable		
datum	a	b	z
i	1	2	*true*
ii	2	1	*true*
iii	2	2	*true*
iv	1	3	*true*
v	1	3	*false*
vi	3	1	*true*
vii	3	1	*false*

Table 8: Sample Test Data Selected by FFT for $(A < B)$ **or** Z

	variable value		
datum	x	y	z
i	*true*	*true*	*true*
ii	*true*	*false*	*true*
iii	*false*	*true*	*true*
iv	*false*	*false*	*true*
v	*true*	*true*	*true*
vi	*true*	*true*	*false*
vii	*false*	*false*	*true*
viii	*false*	*false*	*false*

Table 9: Sample Test Data Selected by FFT's for $(X$ **and** $Y)$ **or** Z

viii would transfer a potential error in X **and** Y since $z = false$. Neither of these test data, however, satisfies the origination condition set for the nested expression. Thus, *FFT* criterion does not guarantee revelation of a context error for a potential boolean operator fault.

In sum, Howden's *FFT* guarantees revelation of a context error when origination of a potential error is guaranteed for a class of potential faults and the transfer conditions are trivial. Only for variable definition fault are the transfer conditions always trivial. *FFT* is sufficient to originate a potential error for this class and hence is sufficient to reveal a context error.

4.3 Foster's *Error-Sensitive Test Case Analysis*

Foster's *error-sensitive test case analysis ESTCA* [Fos80,Fos83,Fos84,Fos85] adapts ideas and techniques from hardware failure analysis such as "stuck-at-one, stuck-at-zero" to software. He has presented his rules in a number of articles. Where there is inconsistency, we will evaluate the most recently published applicable rules. A test data set T satisfies Foster's *ESTCA* if the rules outlined below are satisfied.

Rule 1 For each variable V input at node n_v, and for each variable W input at node n_w, T contains test datum, $t_a \in DOMAIN(n_{final})$ such that:

a. the value input for V is not equal to the value input for W

Rule 2 For each variable V input at node n and some edge(n, n'), T contains test data $t_a, t_b \in$ *DOMAIN* (n') such that the value input for V at node n is:

a. $v_a > 0$;

b. $v_b < 0$.

where v_a and v_b have different magnitude (if v is restricted to only positive or negative values, v_a and v_b need only be of different magnitude).

Rule 3 For each logical unit L [4] of each boolean expression $EXP = (\ldots L \ldots)$ at node n, let $EXP' = (\ldots \neg L \ldots)$, T contains test data $t_a, t_b \in$ *DOMAIN* (n) such that:

a. $l = true$ and $exp' = \neg exp$ [5];

b. $l = false$ and $exp' = \neg exp$.

Rule 4 For each relational expression $EXP_1 \; rop \; EXP_2$ at each node n, T contains test data $t_a, t_b, t_c \in$ *DOMAIN* (n) such that:

[4] A logical unit is either a logical variable, a relational expression or the complement of a logical unit.

[5] that is, substituting ¬L in EXP complements the value of EXP.

a. $exp_1 - exp_2 = -\epsilon$ (where $-\epsilon$ is the negative number of smallest magnitude representable for the type of $exp_1 - exp_2$);

b. $exp_1 - exp_2 = 0$;

c. $exp_1 - exp_2 = +\epsilon$ (where ϵ is the positive number of smallest magnitude representable for the type of $exp_1 - exp_2$).

Rule 5 For each assignment $V := EXP$ at node n and for each variable W referenced in EXP, T contains a test datum $t_a \in DOMAIN\ (n)$ such that:

a. w has a measurable effect on the sign and magnitude of exp.

Foster's *ESTCA* contain no rules that approach the origination condition sets for either a potential variable reference fault or a potential variable definition fault.

Foster's *ESTCA* guarantees origination of a boolean operator fault. Rule 3 considers a boolean expression in terms of logical units. A logical unit is a variable or relational expression that is one of the operands or is a subexpression of one of the operands of a boolean expression (EXP_1 **bop** EXP_2). *ESTCA* requires selection of test data such that each such logical unit takes on the value *true* (and the value *false*) and complementing the logical unit complements the entire boolean expression. This rule satisfies the origination condition sets for boolean operator faults. To see this, notice that for any boolean expression EXP_1 **bop** EXP_2, three test data are selected, $(exp_1, exp_2) =$ (T,F), (F,T), and (T,T) if **bop** is **and**, or (F,F) if **bop** is **or**. This test data satisfies origination condition sets for a boolean operator fault. Thus, *ESTCA* guarantees origination of a potential error for the class of boolean operator faults.

Consider now the class of relational operator faults. When satisfiable, *ESTCA*'s rule 4 results in data such that $exp_1 > exp_2$, $exp_1 = exp_2$, $exp_1 < exp_2$. Thus, test data satisfying this rule will originate a potential error for potential relational operator faults. This rule, however, is more stringent than required and may be unsatisfiable while the origination condition set is feasible. Consider the relational expression in node 4 in the following code segment:

```
1  read X,Y;
2  if X mod 2 = 0 and Y mod 2 = 0 then
       ⋮
3      if X > Y then
       ⋮
4      endif
5  endif
```

ESTCA's rule 4 is unsatisfiable at node 3 since the values of X and Y must differ by at least 2. There is data within the domain of node 4, however, that would satisfy the origination condition set for the relational operator and originate a potential error. Thus, *ESTCA* is sufficient to originate a potential error for relational operator faults but does not guarantee origination of a potential error for relational operator faults.

In an attempt to detect faults in arithmetic expressions, *ESTCA*'s rule 5 requires selection of test data such that variables in arithmetic expressions have a measurable effect on the sign and magnitude of the result. Although the meaning of this rule is ambiguous, it clearly does not imply the origination of a potential error for an arithmetic operator fault. It is possible for variables in an arithmetic expression to have a measurable effect on the sign and magnitude of the result yet still evaluate the same for alternate arithmetic operators in the expression. *ESTCA* does not, we conclude, guarantee origination of a potential error for arithmetic operator faults.

Let us now consider the satisfaction of transfer conditions. *ESTCA*'s rule 3 satisfies transfer conditions through boolean operators. The requirement that complementing the logical unit complements the entire expression is equivalent to selecting test data that satisfies the transfer conditions.

Rule 4 is similar to the general sufficient transfer conditions through relational operators. Like Howden, however, Foster does not consider the assumptions that must hold for these conditions to be sufficient for transfer. Moreover, rather than specifying ϵ to be the smallest satisfiable difference, Foster fixes ϵ at the smallest representable magnitude. As a result, the ability of *ESTCA* to transfer a potential error through a relational operator is further limited. Consider, for example, the relational expression in the module fragment below, where a potential error originates within the arithmetic expression in node 3.

```
1  read X, Y, Z;
2  if X mod 2 = 0 and Y mod 2 = 0 then
       ⋮
3      if 2 * X > Y then
       ⋮
4      endif
5  endif
```

Again, the condition at node 2 causes rule 4 to be unsatisfiable at node 3, and hence, no data need be selected that satisfies rule 4. There is data in the domain of node 3, however, that could transfer a potential error originated within the arithmetic expression. Suppose the reference to X at line 3 should reference Z. The test datum $(4, 4, 1)$ originates a potential error for this potential fault and transfers the potential error through the relational operator. Thus, *ESTCA* is insufficient to transfer a potential error through a relational operator.

Rule 5 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.

As with the other criteria, Foster fails to prescribe integration between *ESTCA* rules that satisfy origination and those that satisfy transfer. Rule 3, however, does guarantee revelation of a context error for boolean operator faults. As seen above, this rule satisfies the origination and transfer conditions for relational operator faults. In addition, when applied to the outermost boolean expression, this rule selects a single datum for each nested binary boolean expression that originates a potential error due to a potential fault in the associated boolean operator and transfers that potential error to the outermost expression. To see this, consider any expression $EXP = EXP_1$ **bop** EXP_2. Some test datum selected for logical units within EXP_1 fulfills the origination condition for potential boolean operator faults in EXP_1. Complementing a test datum selected for a logical unit that is a subexpression of EXP_1 must complement the value *exp*. To force this, if *bop* = **and** then exp_2 =*true*, or if *bop* = **or** then exp_2 =*false*. Thus, for any test datum selected for a logical unit that is a subexpression of EXP_1, EXP_2 will take on a value that will transfer any potential error originated within EXP_1 to the outer expression EXP. Therefore, *ESTCA*'s boolean operator rule satisfies origination as well as transfer conditions simultaneously and hence guarantees revelation of a context error for boolean operator faults.

4.4 Summary of Analysis

Table 10 summarizes the analysis of the three test data selection criteria. 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 when satisfied. 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.

We have analyzed the ability of three test data selection criteria to guarantee revelation of a context error for six classes of faults. Our analysis shows that none of these criteria is adequate for this fault classification and indicates two major weaknesses of the criteria. First, each criterion includes rules that are sufficient but not necessary to originate or transfer an error. When such a rule is not satisfiable, an undetected fault in the class may remain even though test data has been selected to satisfy the criterion. Hence, these techniques do not guarantee detection of these faults. This weakness is primarily due to the creation of rules that are too narrow and the failure of the authors to consider what data is necessary when these restrictive rules are not satisfiable. Second, the authors failed to propose ways in which their rules should be integrated. Each criterion includes rules that guarantee origination of potential errors for some classes of faults and rules that guarantee computational transfer of potential errors through some operators, yet no criterion explicitly forces the rules guaranteeing transfer to be satisfied by the data selected for the rule that guarantees origination. Thus, in most cases, none of the criteria guarantee that a context error is revealed for any of the six classes of faults. The one exception is *ESTCA*, which guarantees detection of any boolean operator fault.

	Budd's *Estimate*	Howden's *FFT*	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			
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 10: Analysis Summary

5 Conclusion

In this paper, we use the RELAY model of error detection to evaluate the error detection capabilities of other testing techniques. This analysis 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.

We feel that this analysis of other techniques demonstrates four points that distinguish RELAY from other work:

1. Relay develops conditions that are both necessary and sufficient to reveal an error;
2. Relay distinguishes between origination of a potential error in the smallest subexpression that contains a potential fault and the computational transfer of that potential error to parent expressions;
3. RELAY acknowledges the need to transfer a potential error through data flow to reveal an output error;
4. RELAY provides a specific framework in which all these components fit.

Let us address the significance of each of these points in turn.

RELAY specifically directs the determination of conditions that are both necessary and sufficient to reveal an error. As shown by the analysis, many other fault-based testing techniques select test data that are sufficient to originate a potential error for some fault classes. When these techniques are not satisfiable, however, an undetected fault in the class may remain. Hence, these techniques do not guarantee detection of these faults. Because RELAY considers both the necessary and sufficient conditions, it does guarantee detection. When a revealing condition for a fault class is not satisfiable, in the RELAY model, we know that a potential fault in the class is not a fault but rather is an "equivalent discrepancy".

RELAY determines origination conditions for the smallest subexpression containing a fault. It then considers additional computational transfer conditions necessary to reveal a potential error in parent expressions. Some researchers, such as Foster, have presented techniques that are capable of originating an error in the smallest subexpression, but have not considered the additional conditions necessary to cause a larger expression to evaluate incorrectly. Other researchers, such as Budd, have recognized the need for a larger expression containing a fault to evaluate incorrectly. They, however, have not detailed specifically the conditions necessary to cause such transfer, nor have they defined the relationship of origination to transfer. RELAY specifically defines such a relationship and details general transfer rules. Other researchers, such as Howden, have examined conditions required to reveal faults in larger expression. The problem here is that the rules developed are specific for certain classes of expressions, e.g., constant reference fault in polynomial expressions. As a result, although a constant reference fault can occur in a variety of types of expressions, the rule is not generally applicable. Further, RELAY's separation of origination and transfer conditions provides a framework for error detection that is easily extended. When a new fault class is considered, RELAY requires that the origination condition set for the class be developed. Applicable transfer conditions from other classes are applied independently, however, and thus require no changes. Techniques that consider larger expressions must develop the "revealing" condition for that entire expression class. We feel that proving properties about origination conditions of a fault class is less complicated than proving properties about the revealing conditions for expression classes.

A third major distinction of RELAY is its consideration of data flow transfer. While some techniques that consider classes of faults that may occur in larger expressions may select test data that is capable of producing a context error, they do not (for the most part) consider what is required for a context error to transfer to output. Hence, these techniques do not guarantee generation of an output error. Techniques that are directed toward the detection of faults in larger expressions effectively achieve data flow transfer by applying their rules to path expressions developed through symbolic evaluation. This approach, however, is only applicable to faults on paths that produce particular expression classes; this limitation is discussed above. Our data flow work is still in preliminary stages and several problems, such as data flow through loops, are left to solve. We intend to elaborate on the data flow transfer conditions in future papers.

The final significant contribution of RELAY is that it provides a general yet applicable framework that describes how a potential fault introduces a potential error and then how it can transfer through a module. We believe that RELAY provides a cleaner, clearer view of fault-based testing than other approaches to date and that it is a sufficiently more powerful approach. This is clearly demonstrated in our analysis which indicates that none of the previously proposed techniques examined is capable of guaranteeing detection of a context error for the selected fault classes.

We continue to extend our model of error detection and to evaluate its capabilities by instantiating it for other classes of faults. In addition, we are applying this analysis method to other testing criteria. One direction of future research is the RELAY analysis of error-based (rather than fault-based) testing techniques, such as Cohen's and White's *Domain Testing* [WC80,CHR82], and path selection techniques, such as the variety of *Data Flow Path Selection* techniques [RW85,Nta84,LK83,CPRZ86]. We expect that this will provide us with insight into the relationship of faults and errors in programs. Moreover, we hope to address the strengths and weaknesses of the two very different approaches to testing.

Appendix A

A.1 Origination Conditions

constant referenced	origination condition set
C	*true*

Table A-1: Origination Condition Set for Constant Reference Fault

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 A-2: Origination Condition Set for Variable Reference Fault

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 A-3: Origination Condition Set for Variable Definition Fault

operator	origination condition set
not	$\{ [true] \}$
null	$\{ [true] \}$
and	$\{[exp_1 \neq exp_2]\}$
or	$\{[exp_1 \neq exp_2]\}$

Table A-4: Origination Condition Sets for Boolean Operator Faults

operator	origination condition set	sufficient condition set
$<$	$\{[exp_1 = exp_2], [exp_1 > exp_2], [exp_1 \leq exp_2], [exp_1 \neq exp_2]\}$	$\{[exp_1 = exp_2], [exp_1 > exp_2]\}$
$\leq$	$\{[exp_1 = exp_2], [exp_1 < exp_2], [exp_1 \geq exp_2], [exp_1 \neq exp_2]\}$	$\{[exp_1 < exp_2], [exp_1 = exp_2]\}$
$\neq$	$\{[exp_1 > exp_2], [exp_1 \geq exp_2], [exp_1 \leq exp_2], [exp_1 < exp_2]\}$	$\{[exp_1 < exp_2], [exp_1 > exp_2]\}$
$=$	$\{[exp_1 \leq exp_2], [exp_1 < exp_2], [exp_1 > exp_2], [exp_1 \geq exp_2]\}$	$\{[exp_1 < exp_2], [exp_1 > exp_2]\}$
$\geq$	$\{[exp_1 \neq exp_2], [exp_1 > exp_2], [exp_1 \leq exp_2], [exp_1 = exp_2]\}$	$\{[exp_1 = exp_2], [exp_1 > exp_2]\}$
$>$	$\{[exp_1 \neq exp_2], [exp_1 \geq exp_2], [exp_1 < exp_2], [exp_1 = exp_2]\}$	$\{[exp_1 < exp_2], [exp_1 = exp_2]\}$

Table A-5: Origination Condition Sets for Relational Operator Faults

operator	origination condition set
$+$	$\{[(exp_1 + exp_2) \neq (exp_1\ \mathbf{op}\ exp_2)] \mid \mathbf{op} =, *, /, \mathbf{div}, **\}$
$-$	$\{[(exp_1 - exp_2) \neq (exp_1\ \mathbf{op}\ exp_2)] \mid \mathbf{op} = +, *, /, \mathbf{div}, **\}$
$*$	$\{[(exp_1 * exp_2) \neq (exp_1\ \mathbf{op}\ exp_2)] \mid \mathbf{op} = +, -, /, \mathbf{div}, **\}$
$/$	$\{[(exp_1 / exp_2) \neq (exp_1\ \mathbf{op}\ exp_2)] \mid \mathbf{op} = +, -, *, \mathbf{div}, **\}$
div	$\{[(exp_1\ \mathbf{div}\ exp_2) \neq (exp_1\ \mathbf{op}\ exp_2)[\mid \mathbf{op} = +, -, *, /, **\}$
$**$	$\{[(exp_1 ** exp_2) \neq (exp_1\ \mathbf{op}\ exp_2)] \mid \mathbf{op} = +, -, *, /, \mathbf{div}\}$

Table A-6: Origination Condition Sets for Arithmetic Operator Fault

A.2 Transfer Conditions

operator	expression	transfer condition
:=	$V := EXP \neq V := \overline{EXP}$	*true*

Table A-7: Transfer Condition Through Assignment Operator

operator	expression	transfer condition
not	$\mathbf{not}(exp_1) \neq \mathbf{not}(\overline{exp'_1})$	*true*
and	exp_1 **and** $exp_2 \neq \overline{exp_1}$ **and** exp_2	$exp_2 = true$
or	exp_1 **or** $exp_2 \neq \overline{exp_1}$ **or** exp_2	$exp_2 = false$

Table A-8: Transfer Condition Through Boolean Operators

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 **mod** $2 \neq 0)$
**	$exp_2{**}exp_1 \neq exp_2{**}\overline{exp_1}$	$(exp_2 \neq 0)$ and $(exp_2 \neq 1)$ and $(exp2 \neq -1$ or exp_1 **mod** $2 \neq \overline{exp_1}$ **mod** $2)$

Table A-9: Transfer Conditions Through Arithmetic Operators

operator	expression	transfer conditions
$<$	$exp_1<exp_2 \neq \overline{exp_1}<exp_2)$	$(exp_1<exp_2$ and $\overline{exp_1}\geq exp_2)$ or $(exp_1\geq exp_2$ and $\overline{exp_1}<exp_2)$
$\leq$	$exp_1\leq exp_2 \neq \overline{exp_1}\leq exp_2)$	$(exp_1\leq exp_2$ and $\overline{exp_1}>exp_2)$ or $(exp_1>exp_2$ and $\overline{exp_1}\leq exp_2)$
$=$	$exp_1=exp_2 \neq \overline{exp_1}=exp_2)$	$(exp_1 = exp_2$ and $\overline{exp_1} \neq exp_2)$ or $(exp_1 \neq exp_2$ and $\overline{exp_1} = exp_2)$
$\neq$	$exp_1\neq exp_2 \neq \overline{exp_1}\neq exp_2)$	$(exp_1 \neq exp_2$ and $\overline{exp_1} = exp_2)$ or $(exp_1 = exp_2$ and $\overline{exp_1} \neq exp_2)$
$>$	$exp_1>exp_2 \neq \overline{exp_1}>exp_2)$	$(exp_1>exp_2$ and $\overline{exp_1}\leq exp_2)$ or $(exp_1\leq exp_2$ and $\overline{exp_1}>exp_2)$
$\geq$	$exp_1\geq exp_2 \neq \overline{exp_1}\geq exp_2)$	$(exp_1\geq exp_2$ and $\overline{exp_1}<exp_2)$ or $(exp_1<exp_2$ and $\overline{exp_1}\geq exp_2)$

Table A-10: Transfer Conditions Through Relational Operators

operators	sufficient transfer conditions
$<, \leq, =, \neq, >, \geq$	$exp_2 - exp_1 = \epsilon$, $exp_2 - exp_1 = -\epsilon$, $exp_2 - exp_1 = 0$

Table A-11: General Sufficient[6] Transfer Conditions Through Relational Operators

[6]For sufficient transfer conditions through relational operators, ϵ is the smallest magnitude positive difference between exp_2 and exp_1 and $-\epsilon$ is the smallest magnitude negative difference; note that $+\epsilon$ and $-\epsilon$ may be of different magnitude. In addition, these conditions are only sufficient under the assumption that the relation between exp_1 and $\overline{exp_1}$ is the same for each of the three test data selected to satisfy all three $\epsilon-$conditions listed in the table. In addition, these conditions are not sufficient unless ϵ is the smallest positive difference between exp_1 and exp_2 and is no greater than the smallest positive difference between $\overline{exp_1}$ and exp_2. If any of these $\epsilon-$ conditions is infeasible, absence of a fault is not guaranteed by satisfaction of the remaining ϵ-conditions.

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.

[CHR82] Lori A. Clarke, Johnette Hassell, and Debra J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, SE-8(4):380–390, July 1982.

[CPRZ86] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven 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.

[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.

[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.

[How87] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, 1987.

[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.

[Mor84] Larry J. Morell. *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.

[RT86] 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.

[WC80] L.J. White and E.I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, May 1980.

[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.