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

(Technical Report 92-38)

Debra J. Richardson†
Margaret C. Thompson‡

April 1992

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

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

To appear in *Transactions on Software Engineering.*

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

This material is based upon work sponsored by the National Science Foundation under grants DCR-8404217, CCR-8704478, and CCR-8704311 with cooperation from the Defense Advanced Research Projects Agency (ARPA Orders 6104 and 6108), the Defense Advanced Research Projects Agency under grants MDA972-91-J-1009 and MDA972-91-J-1012 and the University of California MICRO program and Hughes Aircraft Company under grant 91-131. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

Abstract

RELAY is a model of faults and failures that defines *failure conditions*, which describe test data for which execution will guarantee that a fault *originates* erroneous behavior that also *transfers* through computations and information flow until a failure is *revealed*. This model of fault detection provides a framework within which other testing criteria's capabilities 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 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 potential failures for some fault classes, yet when such rules are unsatisfiable, many faults 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 intermediate values cause observable, erroneous behavior. This paper shows how the RELAY model overcomes these weaknesses.

1 Introduction

Testing is intended to reveal failures or to provide confidence that failures do not occur, where a failure is the observable result of erroneous program behavior. This is typically done by selecting test data, executing the program on that data, and comparing the results to some test oracle, which determines whether the results are correct or erroneous. Many testing criteria [Bud81, DGK+88, Fos80, Ham77, How86, Mor88, Wey81, Zei83] select test data focused on detecting failures caused by particular fault types, where a fault is a syntactic defect in the source code. This "fault-based testing" approach is capable of detecting many of the subtle errors of commission that are revealed only for very specific data, although it can not, except by chance, detect errors of omission[1].

In the context of certain assumptions, fault-based testing can guarantee that particular faults are detected or do not exist. This paper reports on a study that analyzes several fault-based testing criteria in terms of their abilities to actually reveal failures for particular fault types. This analysis is based on the RELAY model of faults, failures, and fault detection. The RELAY model defines how a fault *originates* a potential failure in an evaluated expression containing the fault and how that potential failure must *transfer* through computations to produce a *state failure* and through information flow until it is revealed as an external (or observable) *failure*. The model provides a mechanism for developing *failure conditions* that guarantee fault detection. We use these conditions to analyze the fault detection capabilities of three fault-based testing criteria; in particular, this paper examines the ability of these criteria to reveal state failures at the statement containing a fault.

Through this analysis, we demonstrate two major failings of these criteria. First, in most cases, the "fault-specific rules" that comprise these criteria are merely sufficient (i.e., not necessary and sufficient) to introduce a potential failure. When such a fault-specific rule is unsatisfiable, a corresponding fault will not necessarily cause erroneous execution and thus may remain undetected. Second, in all cases, the criteria do not consider the conditions

[1] In general, a testing criterion would have to take requirements and/or specifications into account to do so.

required to guarantee that erroneous behavior is observable and a failure is revealed. Instead, the rules may introduce an erroneous intermediate value caused by a corresponding fault, but do not guarantee that such a value affects the output or external environment (in fact, we show that in most cases they do not even guarantee effect on the intermediate state of the program variables). The erroneous intermediate values are often masked out by later computations. This extremely common occurrence is *coincidental correctness*, which is the bane of testing. Coincidental correctness occurs when no failure is detected, even though a fault has been executed; thus the effort put into selecting the data and the associated execution is for naught.

Section 2 surveys related works in fault-based testing and compares them to our work on RELAY. Section 3 defines terminology and presents notation used later to describe the RELAY model and the analyzed testing criteria. Section 4 summarizes the RELAY model and developing failure conditions that guarantee fault detection. We present the detail necessary to understand the analysis in this paper and only briefly describe other aspects of the model (more detailed presentations appear in other papers [RT86c, RT88, Tho91]). Section 5 describes an application of the model to develop failure conditions for fault classes and illustrates that application for one fault class. The failure conditions for six fault classes are developed in [RT86b] and provided in the appendix. In section 6, we use the model and these failure conditions to analyze the fault detection capabilities of three fault-based test data selection criteria for these six fault classes. In conclusion, we discuss the implications of the analysis and our future plans for RELAY.

2 Related Fault-based Testing Work

Fault-based testing criteria consist, in some sense, of "fault-specific rules" intended to detect particular fault types. Fault-based heuristics have been used by testers since the dawn of programming. Such heuristics are employed by examining the source code and selecting test data sensitive to commonly occurring faults. Myers outlines many such heuristics [Mye79].

The attempts to formalize fault-based testing have a common underlying theme: distinguishing the test program from alternatives in a set of related programs. This approach assumes the test program is "almost correct" and differs from some hypothetical correct program by at most some definable faults (the *competent programmer hypothesis* [DLS78, DLS79]). This near correctness might be determined by successfully passing some high-level functional testing phase or by satisfying some structural testing criterion. In its various forms, this assumption is taken to mean that the hypothetical correct program is in the "neighborhood" of the test program. Budd and Angluin formalize the notion of "program correctness within a neighborhood of alternate programs" [BA80]; assuming the correct program is within the neighborhood of the test program, then a test set that distinguishes the test program from each alternate program in the neighborhood is reliable [How76, GG75] for the test program. A fault-based testing criterion defines a neighborhood by the class of faults that it considers. The broader the class of faults considered, and hence the larger the neighborhood, the more confidence we gain in the testing activity.

Formal fault-based testing criteria use one of two techniques: either they measure the adequacy of pre-selected test data or they guide test data selection. In what follows, we first discuss several fault-based test data measurement criteria and then describe several fault-based test data selection criteria. It is beyond the scope of this paper to fully compare these criteria. We provide slightly more detail on those criteria that are most similar to the RELAY model; more thorough surveys of fault-based testing and their relation to RELAY appear elsewhere [RT86a, RT86c, Tho91].

The earliest formalized fault-based testing criteria were introduced independently by Hamlet and by DeMillo, Lipton and Sayward. Both criteria *seed* particular types of faults into the test program and measure the adequacy of a set of test data selected by some other means in terms of its ability to detect the seeded faults. Hamlet's *testing with the aid of a compiler* [Ham77] seeds faults as alternative expressions that are "simpler" than the original expression in the source code. An extended compiler instruments the code to compare the values computed by each alternate and the corresponding original expression for the pre-selected test data and reports those alternates that are not distinguished. *Mutation analysis* [DLS78], introduced by DeMillo, Lipton, and Sayward, seeds simple, single-token faults into the source code to produce "mutant" programs. The system then executes the original and mutant programs on the pre-selected test data and determines which mutants are "killed" — that is, which produce different output results from the original for at least one test datum. For both these criteria, the tester augments the test data set iteratively to eliminate the seeded faults that have not yet been distinguished and that are determined not to be equivalent to the original code. The underlying philosophy is that in the process of finding all seeded faults any actual, possibly more complex, faults in the source code will also be eliminated (which is founded on belief in the *coupling effect* [DLS78, DLS79]).

These two criteria require explicit construction and execution (or at best partial interpretation) of many alternate programs. Two more recent fault-based measurement criteria, developed independently by Morell and Zeil, are more analytically-based. Rather than measure a pre-selected test data set through execution, both criteria analyze the test data set and the program to determine faults that could exist in the program that would remain undetected by execution on the test data. Morell's criterion is based on a fault-based testing model [Mor84] that introduces two concepts: "creation" of an initial erroneous state after the statement containing a fault, and its "propagation" to the output. Creation and propagation conditions are described that are sufficient for a fault to create an erroneous state that propagates to the output. Morell's model provided the basis for our initial work on the

RELAY model[2]. In *Symbolic fault-based testing* [Mor88, Mor90], Morell uses his model to symbolically represent faults that would not be detected by execution on a pre-selected test data set. Zeil's criterion describes functional descriptions of "perturbations" that correspond to fault classes [Zei83]. *Perturbation testing* [Zei84, Zei89] thereby identifies faults of a particular functional class that would not introduce an incorrect state and hence would not be detected by the pre-selected test data set. Perturbation testing also determines if the output is partially dependent on the perturbation, thus checking to see if it could produce a failure, but does not explicitly describe this dependence.

Fault-based test data measurement does not provide much guidance as to how to select test data to eliminate the faults considered. Several fault-based testing criteria more directly guide the test data selection process. Foster introduced the idea of conditions under which a fault manifests itself as an erroneous value [Fos80]. Foster's *error-sensitive test case analysis* consists of conditions sufficient to distinguish expressions that may contain a fault from the correct expression for several fault classes. In *weak mutation testing* [How78], more recently called *fault-based functional testing* [How85], Howden refined these conditions and introduced others. Weak mutation testing is applied to the low level "functions" (e.g., statements) in a program. Functional testing [How85, How87] augments this low-level testing by test selection rules applicable to the synthesis of functions from component functions, which have already been tested.

Two extensions to mutation analysis are oriented toward test data selection to assist in satisfying mutation testing. In his mutation testing suite, Budd included the *Estimate* component (for *error-sensitive test monitoring*) [Bud83], which has conditions that must minimally be satisfied to detect some of the mutant classes in expressions containing them. Offutt described *constraint-based testing* [DGK+88] as a part of the MOTHRA mutation analysis system. This criterion defines constraints on a test data set required for the set to be mutation adequate. There are three types of constraints: "reachability" conditions guarantee

[2]We highlight the significant differences at the end of this survey and in the conclusion.

that a mutant is executed; "necessary" conditions guarantee that a mutant is detected at the statement containing it; and "sufficiency" conditions guarantee that the mutant affects the output. The MOTHRA system explicitly selects test data to satisfy the reachability and necessary conditions. Program execution on such test data is compared with mutant program execution to determine if the mutant has been killed; if it has not been killed, additional test data is tried in an effort to also satisfy the sufficiency conditions. Offutt thus recognizes the need to affect the output but provides no guidance in developing the sufficiency conditions or selecting data to satisfy them.

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

The RELAY model differs significantly from each of the fault-based testing criteria described here. The RELAY model is most similar to Morell's work [Mor88]. We introduce concepts similar to Morell's creation and propagation; our *origination* and *transfer*[3] refer to the first erroneous evaluation and the persistence of that erroneous behavior, respectively. We refine Morell's theory by more precisely defining origination and by differentiating between the transfer of a potential failure through computations and its transfer through information flow. This refinement facilitates defining fault-based rules for test data selection, whereas Morell's model is used for test data measurement. Moreover, RELAY considers information

[3]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.

flow transfer through both data dependence and control dependence, whereas Morell's model does not consider propagation through control dependence. In what follows, we outline the RELAY model and describe its potential use for test data selection. In the conclusion, we return to the features that distinguish RELAY from other fault-based testing criteria.

3 A Framework for Testing

A number of test data selection criteria have been proposed throughout the years. These criteria, however, have been defined imprecisely. Here, we outline a representation of programs, execution, and testing, which provides a framework within which test data selection criteria can be formally defined. This formality results in greater precision in defining the criteria as well as a consistent base for evaluating and comparing the criteria. The full definition of this framework is provided in [RT86a], along with the complete, formal definitions of the three fault-based testing criteria defined and analyzed in section 6. This framework also serves as the foundation for the RELAY model, which is described in section 4.

We consider the testing of a **module**, where a module is a procedure or function with a single entry point. A module M implements some function F_M, which maps an input vector x in a domain X_M to an output vector $z = M(x)$ in a range Z_M, $F_M : X_M \rightarrow Z_M$. A module implementation M can be represented by a **control flow graph** G_M that describes the possible flow of control through the module — $G_M = (N, E)$, where N is a (finite) set of nodes and $E \subseteq N \times N$ is the set of edges. N includes a unique start node n_{start} and a unique final node n_{final}. Each other node in N represents a simple statement, a group of simple statements, or the predicate of a conditional statement in M. Associated with each edge (n_k, n_l) is a branch predicate, $BP(n_k, n_l)$, which is the condition that must hold to allow control to pass directly from node n_k to node n_l. If a node has a single successor node, then the branch predicate associated with the edge leaving the node is simply *true*.

The control flow graph defines the paths within a module. A **subpath** in a control flow graph $G_M = (N, E)$ is a finite, possibly empty, sequence of nodes $p = [n_{i_1}, n_{i_2}, ..., n_{i_{|p|}}]$ such that for all i, $1 \leq i < |p|$, $(n_{i_j}, n_{i_{j+1}}) \in E$. An **initial path** p is a subpath whose first node is n_{start}. For any node $n \in N$, the set $INIT(n)$ contains all initial paths in G_M whose last node is n. A **path** P [4] is an initial path whose last node is n_{final}. The set of all paths in

[4] 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 path).

G_M is denoted by $PATHS(G_M)$; note that $PATHS(G_M) = INIT(n_{final})$. The graph G_M is well-formed if and only if every node in N occurs along some path in $PATHS(G_M)$; in our analysis, we consider only modules with well-formed control flow graphs.

An initial path p of M may be executed on some input x; this execution is denoted $p(x)$. Associated with execution of an initial path p on input x is a **state** $S_{p(x)}$, which defines the state of the computation. $S_{p(x)}$ is a vector of values for all variables and the value of the last branch predicate (denoted by the dummy variable BP) after execution of $p(x)$. When we are not particular about what initial path was executed but only a state at node n, we denote that state $S_{n(x)}$. When we are not concerned about a particular input, we denote that state S_n.

Each node in the control flow graph can be represented as an expression tree, where the leaf nodes represent data objects and the internal nodes represent operators. A subexpression of a statement is then represented by a subtree of the node's expression tree. To denote a source code expression, EXP (upper case) is used. An expression evaluated over the module's state S_n is denoted exp (lower case). The expression for an m-ary operator may be represented **OP** $(EXP_1, EXP_2, ..., EXP_m)$; for convenience, a binary expression may be written EXP_1 **OP** EXP_2.

A **test datum** t for a module M with control flow graph $G_M = (N, E)$ is a sequence of values input along some initial path — that is, $t = [t_1,...,t_s]$. The **domain** of an initial path p, denoted $dom(p)$, is the set of test data t for which p can be executed. For any node n in G_M, the set $dom(n)$ [5] is the set of all test data t for which n can be executed —

$$dom(n) = \bigcup_{p \in INIT(n)} dom(p).$$

Note that $dom(n_{final}) = X_M$. A test datum t may be a complete sequence of input values — that is, $\exists P \in PATHS(G_M)$, $t \in dom(P)$ — or incomplete — that is, $\forall P \in PATHS(G_M)$,

[5] We overload the *dom* notation, but there should be no confusion between application to nodes and application to paths.

$t \notin dom(P)$. A test datum t may be incomplete simply because after executing some initial path p, additional input is needed to complete execution of some path, or there may not be any additional data to complete t, because the initial input t may cause the module to terminate abnormally (before n_{final}) or possibly never to terminate. This allows for evaluation of testing criteria that consider invalid inputs, 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, $D_M = \{t \mid \exists\, n \in N, t \in dom(n)\}$. Note that D_M is not merely the domain of M, since neither invalid input values nor initial test data are in X_M — in fact, $D_M = dom(n_{start})$ [6].

Testing 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**, or simply a **criterion**, C is a relation between modules and test data sets such that if $(M, T_M) \in C$, then the T_M satisfies C for M. A criterion, then, is a set of rules for determining whether a test data set satisfies selection requirements for a particular module.

Execution of a module on test data does little good unless the resulting behavior is judged. A *test oracle* [How78, HE78, Wey82] is a means of recognizing (un)acceptable, or (in)correct, behavior of a module. More formally, an **oracle** $O(X_O, Z_O)$ is a relation on $X_O \times Z_O$, $O = \{(x, z)\} \subset X_O \times Z_O$[7]; X_O is the domain of the oracle and Z_O is the range of the oracle. When $(x, z) \in O$, z represents acceptable behavior for x. Ideally, the oracle domain is the module's test data domain so that for any possible test, the oracle will judge the module's behavior[8]. An "external" oracle verifies the module's external behavior, or output[9], for input

[6] If the run-time system does not disable initiation of a module on any invalid input, then the test data domain D_M is the universe of all possible input sequences

[7] Note that an oracle relation allows nondeterminism, where multiple acceptable outputs are specified for an input, and also allows incompleteness, where "don't care" cases can be specified.

[8] This allows the oracle to evaluate *robustness* (reasonable behavior on unexpected inputs) as well as *correctness* (specified behavior on valid inputs)

[9] We will often refer to an output when we mean any external behavior. Note also, that an external oracle may require additional information, such as timing, to enable it to verify correct behavior.

data. Thus, for an external oracle O, $(x, z) \in O$ means z is an acceptable output for x. A test oracle might specify acceptable behavior by a functional representation, correct version of the module (a "gold program"), input/output pairs, simply a tester who can accurately evaluate the module's behavior, or be derived from a formal specification [RAO92]. A module M is **correct** with respect to an oracle O if the module produces acceptable behavior for all valid inputs — $\forall x \in X_M$, $(x, M(x)) \in O$.

A tester often has a concept of the "correct" intermediate behavior in addition to its correct output. Rather than waiting until output is produced to judge behavior, the tester might check the computation of the module at intermediate points, as one does when using a run-time debugger. This approach to testing is supported by an oracle that includes information about intermediate values that should be computed by the module. Such information might be derived from some correct module, an axiomatic specification, self-checking assertions [LvH85], run-time traces [How78], or simply a tester who evaluates intermediate behavior. A **state oracle** O_S is a relation $O_S = \{((t,p), A_{p(t)})\}$, that relates a test datum and an initial path (t,p) to one or more acceptance states $A_{p(t)}$, whic specify an acceptable vector of variable values and the last branch predicate value after execution of $p(t)$. If for any test datum t, satisfaction of the state oracle for execution of all initial paths on t implies the external oracle is satisfied — $\forall t \, \forall p : t \in dom(p), ((t,p), S_{p(t)}) \in O_S \Rightarrow (t, M(t)) \in O$ — then the state oracle and the external oracle for a module are **consistent**. Note that the reverse is not true — that is, the external oracle may be satisfied while the state oracle was violated at some point along the path.

4 RELAY: A Model of Fault Detection

The RELAY model has two principal uses. First, it provides testing criteria that under certain assumptions are capable of guaranteeing fault detection for chosen fault classes. The RELAY testing criteria can be used to select test data or to measure the adequacy of test data selected by another criterion to detect such faults. This use is described in [RT88], and the underlying assumptions are evaluated in [TRC92]. Second, RELAY provides a means of analyzing test data selection criteria's fault detection capabilities. It is this second application that is the focus of this paper. This section defines the RELAY model, but only to the extent required for the analysis presented in sections that follow. More formal definitions of the model and its terminology can be found elsewhere [Tho91].

The failures considered within the RELAY model are those caused by *faults* in the module's source code. The fault-based testing approach relies on two basic assumptions, as does RELAY. The first assumption is that the module being tested is "almost correct". This assumption is similar to the *competent programmer hypothesis* [DLS78], which states that the module being tested bears a strong resemblance to some hypothetical, correct module or differs from the correct module by some small set of faults. 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 faults detected. In the application described here, RELAY is limited to faults that do not change the program schema although the model supports extensions to more complex faults. Second, we assume either that there is a single fault in the module or that multiple faults do not interact to mask each other. This is called the *non-masking faults* assumption and is similar to an assumption based on the *coupling effect* [DLS78], which states that detection of single, simple faults is sufficient to detect multiple or complex faults. The RELAY model addresses faults independently in the formulation presented in this paper. Although these assumptions may seem overly restrictive, the RELAY model allows us evaluate the implications of these assumptions and with further development may allow us to tone them

down a bit [Tho91, TRC92].

Development of the RELAY model was motivated by studying the problems of *coincidental correctness*, where a node containing a fault may be executed yet not reveal a failure; thus, 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 (not just the selected data) despite a discrepancy between it and the hypothetical, correct module. In this case, the module is actually correct, not merely coincidentally correct. Recall that a fault is a syntactic defect in the source code and a failure is observable incorrect behavior. A *potential failure* is an intermediate incorrect result (which may potentially lead to a failure). For a fault to cause a failure, a potential failure must *originate* at the faulty node and *transfer* through computations and along information flow to a failure. Subsection 4.1 describes the RELAY model of faults and failures. One application of the RELAY model is the construction of failure conditions that guarantee fault detection; this application in outlined in subsection 4.2.

4.1 The RELAY Model

The RELAY model describes how a fault causes a failure to occur on execution for some test datum[10]. A failure occurs when execution of a module on some test datum causes an observable incorrect behavior, which most commonly takes the form of incorrect output. Revealing a failure by testing necessitates an oracle to verify the module's correct externally observable behavior.

> *A* **failure** *is an unacceptable result of execution of M on some test datum t — that is, $M(t)$ such that $(t, M(t)) \notin O$.*[11]

A failure is caused by one or more faults in a module. A fault may be thought of as a transformation applied to some expression in the source code that would correct the fault and

[10] In all definitions that follow, we use the notation introduced in section 3: M is the given module being tested; $G_M = (N, E)$ is the control flow graph of M; M^* is the hypothetical, correct module; t is a test datum, while $M(t)$ is the execution of M on t.

[11] We assume that if a module has not terminated after some finite time period, this is incorrect behavior for which the oracle reveals a failure. Thus, the oracle may require information other than the expected output values.

produce a correct module.

> *A* **fault** *f is a transformed expression $f(EXP)$ in M such that $f(EXP) = EXP^{\star}$, where $EXP^{\star}$ is the corresponding expression in the hypothetical, correct module $M^{\star}$, and execution of EXP reveals a failure for some test datum.*

For a fault to cause a failure, execution must first introduce a potential failure, which is later reflected in the execution state, and is eventually externally observable.

> *A* **potential failure** *is the incorrect evaluation exp of some expression EXP[12] in M on some test datum t when $exp \neq exp^{\star}$, where $EXP^{\star}$ is the corresponding expression in $M^{\star}$.*

In the context of a state oracle, a potential failure may be observed in the module's state, which is termed a state failure.

> *A* **state [potential] failure** *is an incorrect state revealed when partial execution of M on test datum t for initial path p is not accepted by the state oracle O_S — $((t,p), S_{p(t)}) \notin O_S$.*

A state failure exists after execution of an initial path when a variable is assigned an incorrect value or when the last branch predicate evaluates incorrectly.

The RELAY model describes the ways in which a potential fault manifests itself as a failure. Consider first how a potential failure is introduced. Some fault transformations affect code that cannot by itself be evaluated (such as an operator), thus we consider introduction of a potential failure in the smallest valued expression that contains the fault. Introduction of the first potential failure is termed origination.

> *A potential failure* **originates** *for some test datum t executing a fault f in M in the smallest evaluable expression EXP containing f at node n when $exp \neq exp^{\star}$ over $S_{n(t)}$, where $f(EXP) = EXP^{\star}$ and $EXP^{\star}$ is the corresponding expression in $M^{\star}$.*

The first potential failure, which occurs at origination, is termed the *original* potential failure.

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 failure

[12] Recall that upper case, EXP, is used here to denote the source-code expression, while lower case, exp, denotes the expression evaluated over the module's state.

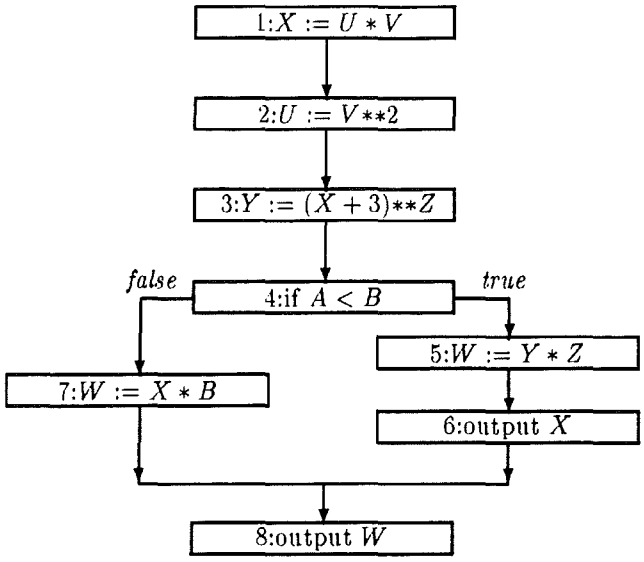

Figure 1: Module for Explanation of RELAY

originates in the smallest evaluable expression containing the fault, which is the reference to U, whenever the value of U differs from the value of B — $u \neq b$. On the other hand, suppose that node 1 contains an arithmetic operator fault and should be $X := U+V$. Then, the smallest evaluable expression is $U*V$ (since $*$ cannot be evaluated), which originates a potential failure whenever the value of $U*V$ differs from the value of $U+V$ — $u*v \neq u+v$.

Once a potential failure originates, it must not be masked out by computations at the faulty node so that it causes a state failure and also must not be masked out later before a failure is revealed. When a potential failure in some expression is not masked out but rather causes a "super"–expression that references it to evaluate incorrectly, we say the potential failure *transfers*. The RELAY model defines three types of transfer: computational transfer, data dependence transfer, and control dependence transfer.

Within a node, a potential failure must transfer through all parent operators in that node to affect evaluation of the entire node.

A potential failure EXP in M for some test datum t **computationally transfers** *to a parent expression $OP(...EXP...)$ when $op(...exp...) \neq op(...exp^{\star}...)$ over $S_{n(t)}$, where $exp \neq exp^{\star}$ over $S_{n(t)}$ and $exp^{\star}$ is the corresponding expression in $M^{\star}$.*

Take another look at Figure 1. If V holds the value zero, the original potential failure in U in node 1 does not transfer to affect the assignment to X; the original potential failure transfers, on the other hand, whenever V is nonzero.

Incorrect evaluation of a node requires computational transfer through all parent operators of the original potential failure. This results in a state failure, which may be reflected in a variable with an incorrect value or the incorrect selection of a branch. The first state failure, which occurs when the node containing a fault evaluates incorrectly, is termed the original state failure.

While it is true that no failure can be revealed if an original state failure is not first introduced, it is also the case that an original state failure may be revealed only when a state oracle of some sort is available. If only an external oracle is available, the original state failure must transfer to affect subsequent nodes until a failure (incorrect output) is produced. **Information flow transfer**, whereby a state failure affects a subsequent node, is based on the concept of information flow [DD77, FOW87, HPR88] and the program dependence relations discussed in [Pod89, PC90]. Information flow transfer occurs when the state failure, which is reflected in the value of some variable, is used at a subsequent node either 1) to incorrectly define a variable (e.g., in an assignment statement) or to incorrectly defined the branch predicate (e.g., in a conditional predicate statement), or 2) to define a variable on an incorrectly selected branch differently than if the correct branch had been selected. These are termed **data dependence transfer** and **control dependence transfer**, respectively. For a fault to cause a failure, the original state failure must transfer along some information flow chain(s) from the faulty node to a failure node. An information flow chain is a sequence of nodes such

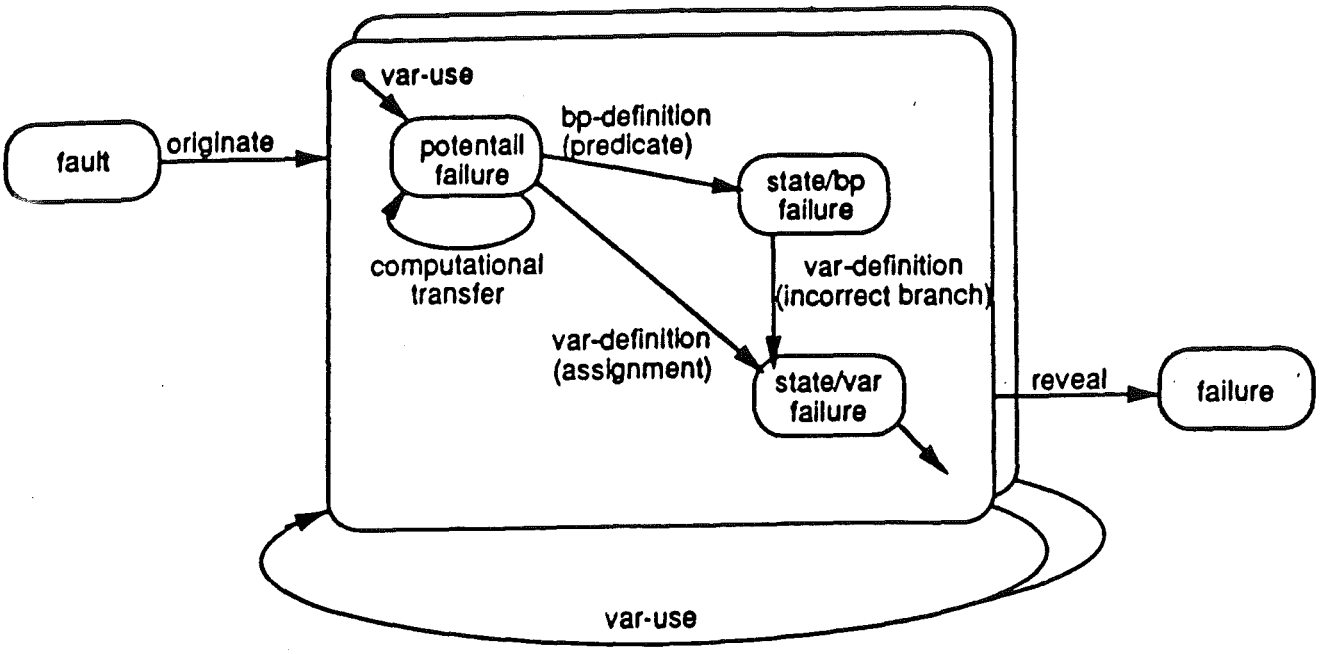

Figure 2: RELAY Model of Fault Detection

that each node is either data dependent[13] or control dependent[14] on the previous node in the chain. Transfer along an information flow chain requires data or control dependence transfer at each link in the chain. Using the example of Figure 1 again, the potential failure in X transfers through data dependence to a use, say at node 7, where it transfers through the computations to produce a state failure in W, and then transfers to the output of W at node 8. The RELAY model of information flow transfer includes a framework within which the components of data and control dependence transfer fit and which identifies the interaction between multiple information flow chains. It is beyond the scope of this paper to present the full details of information flow transfer; moreover, they are not critical to the analysis presented in this paper. Precise definitions of information flow transfer and that aspect of the model may be found elsewhere [Tho91, TRC92].

Figure 2 illustrates the RELAY model of fault detection and how this model provides for the discovery of a fault. The conditions under which a fault is detected are (1) origination of

[13] on a node n_i when a variable V defined at n_i is used at n_j and there is a def-clear path with respect to V from n_i to n_j.

[14] *control dependent* on a node n_i if n_i determines whether n_j is executed.

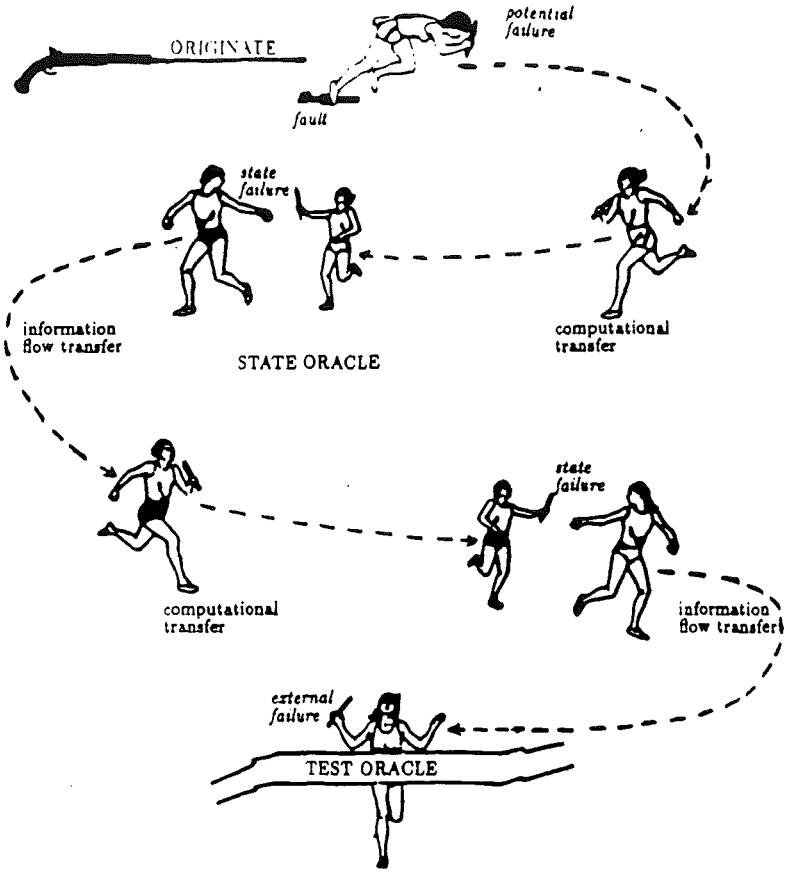

Figure 3: The Testing Relay

a potential failure in the smallest valued expression containing the fault; (2) computational transfer of that potential failure through each operator in the node, thereby revealing a original state failure by a state oracle; (3) information flow transfer of the state failure to an assignment or branch predicate node on the path that references the incorrect state; either (4) computational transfer through the assignment node producing a state failure variable, or (5) computational transfer through the branch predicate node producing a state failure *BP* and assignment of a variable on the selected branch differently than on the correct branch thereby producing a state failure variable; and (6) cycle through (3) and (4 or 5) until a failure is revealed by an external oracle[15].

[15]Note that data dependence transfer is a sequence of var-use, repeated computational transfer, and var-definition (at an assignment) transitions, while control dependence transfer is a sequence of var-use, repeated computational transfer, bp-definition, and var-definition (on the incorrect branch) transitions.

The RELAY view of fault detection has an illustrative analogy in a relay race, as shown in Figure 3, 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 failure. A runner carrying the baton through one leg of the race corresponds to the computational transfer of the failure through a statement. The successful completion of a leg of the race has a parallel in revealing a state failure, and the passing of the baton from one runner to the next is analogous to information flow transfer of the failure from one statement to another. The race goes on until the finish line is crossed, which is analogous to the test oracle revealing a failure.

Our goal, of course, is to complete the relay race and analogously to detect faults. To this end, the RELAY model forms the basis for conditions that define how to guarantee that a fault originates a potential failure and transfer occurs until a failure is revealed. This application of the RELAY model is outlined in the next subsection.

4.2 Failure Conditions

Using the concepts of origination and transfer, RELAY also models *failure conditions* that are necessary and sufficient to *guarantee* fault detection — that is, satisfaction of these conditions for a fault means that a potential failure originates and transfers until a failure can be revealed by the oracle. Sufficient means that if the module is executed on data that satisfies the conditions and the node is faulty, then a failure is revealed. Necessary, on the other hand, means that if a failure is revealed then the module must have been executed on data that satisfies the condition and the node is faulty[16]. Thus, the failure conditions are the unique conditions to guarantee fault detection.

The RELAY model describes how a particular fault causes a failure and is thus dependent on knowledge of the fault. Since this is unlikely (otherwise one would simply fix the fault), application of RELAY hypothesizes that a node is faulty and considers how such a hypothetical

[16]This holds only in the context of a single fault

fault causes a failure, if indeed it is a fault.

> *A* **hypothetical fault** *f is a transformation to some expression EXP in M such that $f(EXP) = EXP'$, where EXP' is an alternative expression in the hypothetically correct module M', which is identical to M except for EXP and is correct if f is a fault.*

Note that we now talk of a hypothetically correct module, since we can easily describe this module.

The failure condition to detect a hypothetical fault guarantees an original state failure is introduced and is transferred along an information flow chain to output. The analysis presented in this paper is concerned only with guaranteeing original state failures, so we focus on the *original state failure condition*, which consists of an *origination condition* and *computational transfer conditions.*

The failure conditions are developed below for a hypothetical fault f independent of where the faulty node n occurs in the module; the conditions are constraints on the module's state before execution of n. To guarantee fault detection, the failure conditions must be true when evaluated over this state. In conjunction with the domain of the faulty node ($dom(n)$), the failure condition describes a test data set, where execution of any single test datum in the set would execute the faulty node and reveal a failure[17]. Because the failure conditions are necessary, if the conditions are *infeasible* within $dom(n)$, then no failure can be revealed and the hypothetical fault is not a fault. Although, in general, the feasibility problem is undecidable, in practice, it can often be solved.

The origination condition guarantees that the smallest valued expression containing a hypothetical fault originates a potential failure (hence that the hypothetically faulty expression evaluates differently than the hypothetically correct one).

> *The* **origination condition** *for a hypothetical fault f in M in the smallest evaluable expression EXP containing f at node n is $[exp \neq exp']$ evaluated over S_n, where $f(EXP) = EXP'$ and EXP' is the corresponding expression in M'.*

[17] A state failure would be revealed for an original state failure condition, and an external failure would be revealed if information flow transfer conditions are added.

When the origination condition is infeasible, the hypothetically faulty expression is equivalent to the alternate, and no such fault exists.

The original potential failure for a hypothetical fault must transfer to affect evaluation of the entire node. A computational transfer condition guarantees that a potential failure in an operand transfers through a parent operator so that the parent expression is a potential failure (hence that the parent expression referencing the hypothetical fault evaluates differently than the hypothetically correct parent expression).

> *The* **computational transfer condition** *for an expression* $\mathbf{OP}(..., EXP, ...)$ *containing a potential failure exp in M at node n is* $[op(...exp...) \neq op(...exp'...)]$ *evaluated over* S_n*, where* $exp \neq exp'$ *over* S_n *and* EXP' *is the corresponding expression in* M'*.*

When a computational transfer condition is infeasible, the potential failure cannot transfer to affect the parent expression; as above, the hypothetically faulty expression is equivalent to the alternate one and no such fault exists.

The conjunction of the computational transfer conditions for each ancestor operator in the node of the originating expression guarantees transfer to affect the entire node and produce an original state failure. To guarantee a fault's detection by revealing an original state failure, the origination and the computational transfer conditions for all ancestor operators in the node must be jointly satisfied.

> *The* **original state failure condition** *for a hypothetical fault f in M at node n is the conjunction of the origination condition for f and all computational transfer conditions for f and n.*

As an example of an original state failure condition, consider again the module in Figure 1. Hypothesize that statement $X := U * V$ at node 1 should be $X := B * V$, then the origination condition is $[u \neq b]$. This original potential failure 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 original state failure condition resulting from this hypothetical fault is $[(u \neq b)$ and $(v \neq 0)]$.

For the analysis presented in this paper, we consider only the original state failure conditions, because, as we will show, most fault-based testing criteria do not even satisfy these. Typically, however, testing is primarily concerned with revealing an output failure as the manifestation of a fault (and not only incorrect intermediate values). To address this, the RELAY model provides a framework for developing failure conditions to guarantee that a state failure transfers to affect module execution as a whole. It does so by extending the failure condition to also guarantee that the original state failure transfers along some information flow chain(s) from the faulty node to a failure node. The information flow transfer conditions guarantee that data and/or control dependence transfer occurs at each link in the information flow chain(s). Consider again the hypothetical variable reference fault at node 1 in Figure 1. One information flow 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 failure in X transfers through information flow to node 7 whenever the false branch of the conditional at node 4 is taken. Reference to the potential failure in X must transfer through the multiplication by B to the assignment of W at node 7. Thus, for this information flow chain, the transfer condition is $[(a \geq b)$ and $(b \neq 0)]$. Recall that the original state failure condition is $[(u \neq b)$ and $(v \neq 0)]$, creating a failure condition for this information flow chain of $[(u \neq b)$ and $(v \neq 0)$ and $(a \geq b)$ and $(b \neq 0)]$. The development of information flow transfer conditions and failure conditions is fully defined in [Tho91], as are the details of data dependence transfer, control dependence transfer, and complex computational transfer (in the context of interacting potential failures). As mentioned, these are not required for the analysis presented in this paper, so we have merely provided a general description to portray the full nature of the RELAY model.

The failure conditions describe what is required to guarantee that a fault produces a failure. Thus, they define test data that must be executed to reveal a failure for a hypothetical fault. This means that if a failure is not revealed for data in the domain of the hypothetically faulty node and satisfying the failure condition, then the hypothetical fault is not a fault (for

any test data that could execute the node) and hence the hypothetically correct module is not correct. On the other hand, revealing a failure for such data indicates that the module contains the hypothetical fault. In the case of the original state failure condition, we only know that a state failure has been produced, and the additional information flow transfer conditions must be satisfied to reveal an external failure. Moreover, a failure condition that is infeasible within the domain of the hypothetically faulty node implies that the hypothetically faulty module and the hypothetically correct module are equivalent[18].

One possible application of the RELAY model is to hypothesize faults in a module, actually construct failure conditions that guarantee fault detection of the hypothesized faults, and select test data to satisfy these failure conditions. Although this application provides a fault-based test data selection criterion, we are not suggesting that it is feasible or practical[19] Rather, our model shows what is required to guarantee fault detection and demonstrates the complexity of the problem. The insight provided by the failure conditions, however, are useful for analyzing the fault detection capabilities of test data selection criteria. This analysis is the focus of section 6.

As currently defined, a failure condition is derived for any hypothesized fault independently, although many faults are similar and much of the transfer requirements are independent of a particular hypothetical fault. The application described in the next section leverages this fact by grouping hypothetical faults into classes based on some common characteristic of the transformation and defines original state failure conditions for all hypothetical faults of a class. When these conditions are instantiated for a particular fault class, they provide conditions that guarantee introducing a state failure caused by any fault of that class. In the next section, we discuss the original state failure conditions for six fault classes. A simple example of test data satisfying a specific original state failure condition is presented at the

[18] An infeasible external failure condition means the failure conditions must be infeasible for all information flow chains; again this is described more completely in [Tho91].

[19] It is well-known that selection of data to satisfy any condition is undecidable; it is not our intention to address the equivalence problem with failure conditions.

end of the next section. These conditions can be used to evaluate the ability of test data selection criteria to guarantee detection of faults in chosen classes. RELAY is applied in this fashion to analyze three test data selection criteria for the six fault classes in section 6.

5 Application of RELAY for Fault Classes

The previous section described the RELAY model and how it defines a failure condition for a hypothetical fault, which guarantees origination of a potential failure, computational transfer to produce an original state failure at the faulty node, and information flow transfer to a failure node. Here, we describe how we can hypothesize many potential ways in which a node might be faulty and develop failure condition sets that apply to a class of hypothetical faults. This technique for applying the RELAY model takes advantage of two facts. First, similar hypothetical faults (such as transformation to alternative arithmetic operators) have similar origination conditions. Second, all hypothetical faults in a particular expression must basically transfer through the same computations and information flow to a failure. Thus, although the origination conditions may differ, the transfer conditions are basically the same. This section describes RELAY's application for fault classes, demonstrates the instantiation of the original state failure conditions for one fault class, and illustrates by example what these mean for test data.

Any syntactic expression in a module's source code may be faulty, but only in ways that retain the module's semantic correctness (compilability). Thus, for any expression, we can hypothesize limited classes of faults that might occur. By grouping these hypothetical faults into classes based on some common characteristic of the transformation, we can define failure conditions that guarantee origination of a potential failure for any hypothetical fault of that class. Moreover, we can consider the ancestor operators that reference such an expression and define the computational transfer conditions that apply to a fault class and are required to transfer the original potential failure to produce an original state failure; and likewise for information flow transfer.

For an expression in a module, a hypothetical fault class 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 failure for a class, the

hypothetically faulty expression must be distinguished from each expression in this alternate set. For each alternative expression, the RELAY model defines an origination condition, which guarantees origination of a potential failure if the corresponding alternate were indeed the correct expression. For an expression and fault class, we define the **origination condition set** as the set of origination conditions for each alternative expression transformed by the fault class. The origination condition set guarantees that a potential failure originates in that expression if the expression contains a fault of this class.

For each alternative expression, a potential failure that originates must also transfer through each operator in the node to reveal a state failure. The computational transfer conditions, which are determined by these subsequent manipulations of the data, are independent of the particular alternate. Thus, for a fault class, an original state failure condition is defined for each alternate, which is the conjunction of the origination condition and the computational transfer conditions. The **original state failure condition set** contains an original state failure condition for each alternate in the alternate set. It is a necessary and sufficient set of conditions to guarantee that a hypothetical fault of a particular class reveals an original state failure.

Likewise, the original state failure for each alternate must transfer through information flow to reveal an external failure. And, likewise, these transfer conditions are independent of the alternate and can be conjoined to each original state failure condition in the set. The failure condition set contains a failure condition for each alternate and guarantees that a hypothetical fault of the class reveals a failure.

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 failure for an incorrect reference to U, the value of each alternative variable $\overline{U}$[20] must be distinguished from the value of U at node 1. The possible alternates depend on what other variables may be substituted for U

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

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 $\{[u \neq \overline{u}] \mid \overline{U} \in \{A, B, V, W, X, Y, Z\}\}$. Recall that the computational transfer condition for node 1 is $[v \neq 0]$. For the information flow chain where X is used to define W at node 7 and W is output at node 8, recall that the transfer condition 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 transfer condition set for this hypothetical fault. This set is sufficient but not necessary because all information flow chains are not considered.

Thus, the RELAY model can be applied for a chosen fault classification. Hypothesizing a particular fault class, the origination and transfer conditions are instantiated to provide conditions specific to that class. The next subsection summarizes the instantiation of RELAY for fault classes. The instantiated origination and transfer conditions can then be evaluated for selected (applicable) locations in a module to provide the specific failure condition sets that must be satisfied to guarantee the detection of any fault in the chosen classification at the selected locations. The specific transfer conditions for a module can be used to measure the effectiveness of a pre-selected set of test data and/or to select test data. A simple example of constructing an original state failure condition set and of test data satisfying it is presented at the end of this section. The instantiated origination and transfer conditions can also be used to evaluate the ability of test data selection criteria to guarantee fault detection for chosen fault classes. RELAY is applied in this fashion to analyze three test data selection criteria for six fault classes in section 6. This analysis demonstrates the flaws inherent in most criteria and the advantage of a complete model of faults and failures.

5.1 Instantiation of RELAY

In this section, we discuss the instantiation of the RELAY model for a fault class. The application presented provides original state failure conditions for statements hypothetically containing a fault in one of six classes. The restriction to original state failures means that

only computational transfer need be considered at this time. Developing original state failure conditions for a fault class consists of developing the origination conditions for the fault class and also developing any applicable computational transfer conditions. This instantiation process is illustrated for the class of relational operator faults. We derive the origination conditions for this class and the computational transfer conditions through boolean operators since a relational expression may be contained within boolean expressions.

RELAY is instantiated for six fault classes in [RT86b]. 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. Each of the six classes is a class of atomic faults, where a (hypothetical) fault f is *atomic* if the node n differs from the hypothetically correct node n' by a single token.

To determine the original state failure conditions for a class of hypothetical faults, we must instantiate the applicable computational transfer conditions as well as the origination condition for the class. Thus, for the six fault classes, in [RT86b] we derive origination conditions for each class as well as transfer conditions through all operators applicable to these faults — that is, assignment operator, boolean operators, arithmetic operators, and relational operators. The origination conditions for the six fault classes along with the computational transfer conditions through the four applicable operators are summarized in the appendix.

5.1.1 Origination Conditions for Relational Operator Faults

An origination condition guarantees that the smallest valued expression containing a hypothetical fault produces a potential failure. Thus, given the smallest evaluable expression EXP containing a hypothetical fault and an alternative expression $\overline{EXP}$, the origination condition guarantees that $exp \neq \overline{exp}$.

Consider the class of relational operator faults, where a potential failure 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 a set of alternates $\{(EXP_1\ \overline{\mathbf{ROP}}\ EXP_2) \mid \overline{\mathbf{ROP}}$ is a relational operator other than **ROP**$\}$.

As an example, let us construct the origination condition for the relational operator $<$ and an alternative operator $=$. We must determine the origination condition that distinguishes $(EXP_1 < EXP_2)$ from $(EXP_1 = EXP_2)$. For any relational expression, there are three possible relations for which test data may be selected — $(exp_1 < exp_2)$, $(exp_1 = exp_2)$, $(exp_1 > exp_2)$. The origination condition to distinguish between $EXP_1 < EXP_2$ and $EXP_1 = EXP_2$ is $[exp_1 \leq exp_2]$. The original expression, $(EXP_1 < EXP_2)$, and alternative expression, $(EXP_1 = EXP_2)$, evaluate differently whenever either the relation $(exp_1 < exp_2)$ or the relation $(exp_1 = exp_2)$ is satisfied; thus the condition $(exp_1 \leq exp_2)$ is sufficient for origination of a potential failure. When the third possible relation is satisfied, $(exp_1 > exp_2)$, the original and alternate expressions evaluate the same; hence, the condition $(exp_1 \leq exp_2)$ is also necessary for origination of a potential failure. The origination conditions for the other alternative operators are derived similarly; this derivation is detailed in [RT86b]. The origination conditions for all relational operator faults are summarized in Table 1. The origination condition set for a given relational operator and the relational operator fault class is the set of all origination conditions that distinguish the given operator from some alternate. Thus, for a hypothetically faulty , operator, the origination condition set is $\{[exp_1 = exp_2], [exp_1 \leq exp_2], [exp_1 > exp_2], [true], [exp_1 \neq exp_2]]\}$. Often an origination condition set can be reduced to a sufficient condition set due to the overlap between conditions. If this set is feasible, then it's satisfaction implies origination. On the other hand, if it is infeasible, the more specific origination conditions in the full set must be considered. The sufficient origination condition set for $<$ is $\{[exp_1 = exp_2], [exp_1 > exp_2]\}$. Similar sufficient condition sets are developed for the other fault classes in [RT86b].

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 1: Origination Conditions for Relational Operator Faults

5.1.2 Computational Transfer Conditions for Boolean Operators

A computational transfer condition guarantees that a potential failure in an operand of an expression is not masked out by the computation of a parent operator. Thus, given an expression $\mathbf{OP}(\ldots, EXP, \ldots)$, where a potential failure exists in EXP, the transfer condition guarantees that $\mathbf{op}(\ldots, exp, \ldots)$ also produces a potential failure. More specifically, given EXP containing a hypothetical fault and $\overline{EXP}$ an alternate, the existence of a potential failure 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, 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. No additional conditions are necessary for transfer of a potential failure 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 $(EXP_1$ **BOP** $EXP_2)$ is distinguished from $(\overline{EXP_1}$ **BOP** $EXP_2)$ and that $(EXP_2$ **BOP** $EXP_1)$ is distinguished from $(EXP_2$ **BOP** $\overline{EXP_1})$, whenever EXP_1 and $\overline{EXP_1}$ are distinguished. Since the binary boolean operators are commutative, we need not develop separately the transfer conditions for a potential failure in the right operand. The binary boolean transfer conditions depend upon the boolean operator. For the boolean operator **and**, $(exp_1$ **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 failure in exp_1 transfers through the boolean operator **and**. For the boolean operator **or**, $(exp_1$ **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 failure in exp_1 through the boolean operator **or**. The transfer conditions for boolean operators are summarized in Table 2.

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 2: Transfer Conditions for Boolean Operators

5.2 Construction of Original State Failure Condition Sets

In this section, we illustrate the construction of an original state failure condition set for the relational operator fault class on an example module fragment and show test data that satisfies this condition set. The example module fragment is shown in Figure 4.

Hypothesize that the relational operator at statement 2 is hypothetically faulty. The

```
1   read X, Y, Z, B, C;
2   if (X * Y < Z or B) and C then
        ⋮
```

Figure 4: Module Fragment

origination condition set for $<$ relational operator fault is $\{[x * y = z], [x * y \leq z], [x * y > z], [true], [x * y \neq z]\}$. In fact, the origination conditions $[x * y = z]$ and $[x * y > z]$ are sufficient to satisfy the conditions for all alternates, so a sufficient origination condition set is $\{[x * y = z], [x * y > z]\}$. A potential failure resulting from the $<$ in node 2 must transfer through the boolean operators **or** and **and**. The computational transfer conditions are thus

$(b = false)$ and $(c = true)$.

The origination condition set combines with the computational transfer conditions to form the following original state failure condition set

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

We are now in a position to examine test data set that guarantees that an original state failure is introduced. To do so, data must not only satisfy the original state failure condition but also must execute the node. Hence data that satisfies a failure condition must be a member of the domain of the node. For simplicity, we are considering a node that is not conditionally executed, and hence $dom(2) = D_M$. There are many possible test data sets that satisfy the failure conditions developed for this example. One such set contains the following two datum $(1, 2, 2, false, true)$ and $(1, 3, 2, false, true)$. The first datum satisfies the first failure condition, and the second datum satisfies the second failure condition. If the $<$ operator should have been some other relational operator, then execution for these two test data will reveal an original state failure. If no original state failure is revealed, then the $<$ operator is correct.

6 Analysis of Related Test Data Selection Criteria

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

In this section, we use the origination and transfer conditions for the six fault classes (provided in the appendix) to analyze the fault detection capabilities of three fault-based test data selection criteria — Budd's *Error-Sensitive Test Monitoring* [Bud81, Bud83], Howden's *Weak Mutation Testing* [How78, How85], 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.

As noted, the application of RELAY discussed in this paper is limited to revealing original state failures. Thus, the failure conditions discussed here are necessary for the detection of a fault, but not sufficient, because the original state failure introduced by satisfaction of these conditions may still be masked out by later computations on the path. To guarantee fault detection for a particular class, the failure conditions must be augmented to include information flow transfer. The analysis to follow does not consider whether or not the criteria consider these additional conditions (although in most cases, they do not). As we shall see, however, this limitation of the analysis is of little consequence, since for the most part, the criteria do not guarantee revealing an original state failure. Our analysis shows that none of the criteria guarantees detection of the considered fault classes and points out two weaknesses

that are common to all three criteria. We also discuss how the RELAY model rectifies these common problems.

For each criterion, we first define it in the terminology provided in section 3. Next, we examine the criterion's ability to satisfy the origination conditions for each fault class and also its ability to satisfy transfer conditions through applicable ancestor operators. Then, for each fault class, we discuss the circumstances in which the criterion will guarantee revealing an original state failure, which requires that a single test datum be selected to satisfy both a specific origination condition and the applicable computational transfer conditions for the node. Although a criterion may include rules that satisfy the origination conditions 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 conditions for a fault class, detection is not guaranteed for that class. In the case where only origination is guaranteed, revealing an original state failure is guaranteed only when the 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 $dom(n)$. 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 assume that no test data is selected for an unsatisfiable rule.

In the analysis of each criterion, we analyze all applicable rules for each fault class but do not belabor analysis of those that clearly do not address the class. 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. Some of the conditions are trivially met by any criterion that satisfies statement coverage (e.g., 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 merely mention the satisfaction of such trivial conditions. For the first criterion examined, counter

examples are provided when a rule does not guarantee origination or transfer. Similar counter examples for the subsequent criteria are are not provided but the similarity is noted. Complete detailed analysis is provided in [RT86b].

The following is not intended to be a complete analysis of the fault detection capabilities of these criteria. Only those faults discussed in section 5 are included in the discussion. A complete analysis must consider a more complete fault classification. The analysis presented in this paper, however, provides insight into how our model of fault detection can be used to analyze the strengths and weaknesses of testing criteria.

6.1 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, there exist some node n_a, n_b, n_c such that:

a. $t_a \in dom(n_a)$ and $v = 0$;

b. $t_b \in dom(n_b)$ and $v < 0$;

c. $t_c \in dom(n_c)$ and $v > 0$.

Rule 2 For each each assignment $V := EXP$ at each node n, T contains a test datum $t_a \in dom\ (n)$ 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 $t_a, t_b \in dom\ (n)$ 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. $t_a \in dom(n)$ and $bp(n, n') = true$.

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

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

Rule 7 For each binary arithmetic expression EXP_1 **AOP** C (C **AOP** EXP_1), (where C is a constant), at each node n, T contains a test datum $t_a \in dom(n)$ such that:

a. $exp_1 > 2$.

First, let us consider *Estimate*'s ability to originate potential failures for the six fault classes. Clearly, rule 3 satisfies the origination conditions for boolean operator faults, and rule 5 satisfies the origination conditions for relational operator faults. Thus, *Estimate* guarantees origination of a potential failure 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 code segment[21]:

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

[21] For simplicity, we assume that all variables in this section's examples are either boolean or integer.

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 failure 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 failure caused by a variable definition fault may remain undetected by *Estimate*. Consider the following:

```
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 failure for a variable definition fault, but it does not guarantee origination for this fault class.

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 failure for an 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 failure due to an arithmetic operator fault. For instance, consider the following:

```
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 failure for arithmetic operator faults except for the four noted exceptions, where *Estimate* is insufficient. *Estimate*, however, does not guarantee origination of a potential failure 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 the appendix, 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. Suppose $X*Y$ should be $X+Y$ in the following:

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

Test datum (11,1) would originate a potential failure (since $11 + 1 \neq 11 * 1$) and satisfies rule 5 (since $X * Y$ differs from 10 by a small amount). However, the potential failure 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. Assume a potential failure originates in x at node 3 in the following:

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

No test datum satisfies rule 6 for this node; however, a test datum such that $y \neq 0$ transfers any potential failure 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 revealing an original state failure 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 failure is not guaranteed. This is because *Estimate* does not prescribe any integration of the application of its rules. 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 revealing an original state failure for a relational operator fault in the expression $(A < B)$ **or** Z in the following:

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

The test data shown in Table 3 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 failure; for these test data, however, $z = true$, which will not transfer

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

any potential failure. 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 failure originated in the relational expression; these data alone, however, are insufficient to guarantee origination of a potential failure for a relational operator fault. If, for example, the $<$ should be $\leq$, no selected datum both originates and transfers a potential failure caused by this fault. Thus, *Estimate* does not guarantee revealing an original state failure for this 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:

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

The test data shown in Table 4 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 failure. For these test data, however, $z =$*true*, which will not transfer any potential failure.

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

Table 4: Sample Test Data Selected by *Estimate* for $(X$ **and** $Y)$ **or** Z

Test data iii and v are the only data that would transfer a potential failure originated at the inner expression, but for these test data, the values x and y would not originate a potential failure. Thus, *Estimate* does not guarantee revealing an original state failure for a boolean operator fault.

When origination of a potential failure is guaranteed for a fault class, revealing an original state failure is guaranteed by *Estimate* only when the transfer conditions are trivial. In general, this occurs when the smallest expression containing the 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 failure for this class, it is also sufficient to reveal an original state failure. Recall, however, that *Estimate* does not guarantee origination for this class.

6.2 Howden's *Weak Mutation Testing*

Howden's *Weak Mutation Testing* (*WMT*) [How82, How85, How86] 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 results in one of the fault 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 weak mutation testing.

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

a. $v \neq \overline{v}$ [22] .

Rule 2 For each assignment $V := EXP$ at node n, T contains a test datum $t_a \in dom\ (n)$ 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 $t_1, t_2, \ldots, t_{2^i} \in dom\ (n)$ 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 $t_a, t_b, t_c \in dom\ (n)$ 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 $t_a, t_b \in dom\ (n)$ such that:

a. the expression is executed;

b. $exp \neq 0$.

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 $t_1, t_2, \ldots t_{k+1} \in dom\ (n)$ such that $\{t_1, t_2, \ldots t_{k+1}\}$ is any cascade set of degree $k+1$ in $dom\ (n)$.

[22] 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 failure detection capabilities and we do not include it here.

Howden's *WMT* guarantees origination of a potential failure 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 fault class; it requires a single test datum to distinguish between the faulty 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. Thus, *WMT* is sufficient to originate a potential failure, therefore, but does not guarantee origination for variable reference faults.

WMT'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 failure for a variable definition fault. This rule alone is incomplete, however, since it does not guarantee absence of a fault when it is unsatisfiable. Thus, *WMT* is sufficient but does not guarantee origination for this class.

Rules 5 and 6 are the only rules specifically directed toward exercising arithmetic expressions. For an arithmetic operator fault that exchanges an addition operator for a subtraction operator (and vice versa), rule 5 will guarantee origination of a potential failure. For other arithmetic operator faults, this rule is insufficient. Rule 6 is insufficient to guarantee origination of a potential failure due to an 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:

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

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 *WMT* to transfer a potential failure. Rule 3 selects data that satisfies the boolean transfer condition and guarantees transfer through boolean operators.

WMT'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 the appendix must also hold. *WMT* does not consider these assumptions. Hence, even when *WMT*'s rule 4 is satisfied, a potential failure may not transfer through a relational operator. Thus *WMT* is insufficient to transfer a potential failure through a relational operator.

Rule 5 satisfies the transfer conditions for all arithmetic operators but the exponentiation operator. Rule 6 does not apply because a proper cascade set cannot be selected when the degree of the expression is unknown. *WMT*, therefore, only partially guarantees transfer through arithmetic operators.

As with *Estimate*, *WMT* 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 revealing an original state failure 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. In sum, Howden's *WMT* guarantees revealing an original state failure when origination of a potential failure is guaranteed for a fault class and the transfer conditions are trivial. Only for variable definition fault are the transfer conditions always trivial. *WMT* is sufficient to originate a potential failure for this class and hence is sufficient to reveal an original state failure.

6.3 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, $t_a \in dom(n_{final})$ 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 $t_a, t_b \in dom\ (n')$ 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 [23] of each boolean expression $EXP = (\ldots L \ldots)$ at node n, let $EXP' = (\ldots \neg L \ldots)$, T contains test data $t_a, t_b \in dom\ (n)$ such that:

a. $l = true$ and $exp' = \neg exp$ [24];

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 $t_a, t_b, t_c \in dom\ (n)$ such that:

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 $t_a \in dom\ (n)$ 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 conditions for either a variable reference fault or a 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

[23] A logical unit is either a logical variable, a relational expression or the complement of a logical unit.
[24] that is, substituting ¬L in EXP complements the value of EXP.

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 conditions for a boolean operator fault. Thus, *ESTCA* guarantees origination of a potential failure 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 failure for relational operator faults. This rule, however, is more stringent than required and may be unsatisfiable while the origination condition set is feasible. Thus, *ESTCA* is sufficient to originate a potential failure for relational operator faults but does not guarantee origination of a potential failure 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 failure 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 failure 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 failure through a relational operator is further limited. Thus, *ESTCA* is insufficient to transfer a potential failure 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 failure 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 revealing an original state failure 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 failure due to a fault in the associated boolean operator and transfers that potential failure to the outermost expression. To see this, consider any expression $EXP = EXP_1 \, \mathbf{BOP} \, EXP_2$. Some test datum selected for logical units within EXP_1 fulfills the origination condition for 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 failure 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 revealing an original state failure for boolean operator faults.

6.4 Summary of Analysis

Table 5 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 satisfiable fulfills the condition. The entry partially sufficient means that the criterion includes a rule that is sufficient to distinguish many but not all of the alternates or transfer through many but not all of the operators. The entry guarantees means that the criterion includes a rule that satisfies the conditions when the conditions are feasible, while partially guarantees means the criterion includes a rule that satisfies many but not all of the conditions when feasible.

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

7 Conclusion

In this paper, we have described the RELAY model, which rigorously defines how a fault in a module causes a failure. The model includes origination, computational transfer, and data and control dependence transfer. This paper focuses on using the RELAYmodel to evaluate the fault detection capabilities of testing criteria. This analysis demonstrates how the rules of a test data selection criterion must be carefully designed and tightly integrated to reveal a failure for any fault. Without this precise modeling, 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.

This paper demonstrates four points that distinguish RELAY from other work:

1. RELAY distinguishes between origination of a potential failure in the smallest expression that contains a hypothetical fault and the computational transfer of that potential failure to parent expressions;
2. RELAY provides a detailed model of the transfer of a state failure from the faulty node through information flow until it is externally revealed and further considers both data and control dependence transfer;
3. RELAY provides a mechanism for developing conditions that must be satisfied to guarantee fault detection;
4. RELAY provides a specific framework in which all these components fit.

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

First, RELAY determines origination conditions for the smallest expression containing a fault. It then considers additional computational transfer conditions necessary to reveal a potential failure in parent expressions. Some researchers, such as Foster [Fos80], have presented criteria that are capable of originating a potential failure in the smallest expression, but have not considered the additional conditions necessary to cause a larger expression to evaluate incorrectly. Other researchers, such as Budd [Bud81], 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 rela-

tionship of origination to transfer. RELAY specifically defines such a relationship and details general transfer rules. Other researchers, such as Howden [How86], 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 fault 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. Criteria that consider larger expressions must develop the "failure" condition for that entire expression class. We feel that proving properties about origination conditions of a new fault class is less complicated than proving properties about the revealing conditions for expression classes.

A second major contribution of RELAY is its consideration of information flow transfer. While some criteria that consider hypothetical fault classes in larger expressions may select test data that is capable of producing a state failure, they do not (for the most part) consider what is required for a state failure to transfer to output. Hence, these criteria do not guarantee revealing a failure. Criteria that are directed toward the detection of faults in larger expressions effectively achieve information flow transfer by applying their rules to [partial] 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. The concept of "sufficiency" in Offutt's constraint-based testing [DGK+88, DO91] is similar to transfer, but Offutt does not provide any details on the nature of these conditions. The concept of "propagation" in Morell's symbolic fault-based testing [Mor88, Mor90] is similar to data dependence transfer, but does not consider control dependence transfer. The distinct contributions of RELAY's information flow transfer model are considered further in [Tho91, TRC92], where information flow transfer is fully defined.

Another distinction is that RELAY provides a means of developing conditions that are both necessary and sufficient to reveal a failure. As shown by the analysis, most fault-based testing criteria select test data that are sufficient to originate a potential failure for some fault classes. When these criteria are not satisfiable, however, an undetected fault in the class may remain. Hence, these criteria 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 hypothetical fault in the class is not a fault but rather is an "equivalent discrepancy". Other models of fault-based testing (such as Morell's [Mor90]) do not direct how to construct specific conditions or to select data to guarantee fault detection.

The final significant contribution of RELAY is that it provides a general yet applicable framework that describes how a hypothetical fault originates a potential failure 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 examined criteria is capable of guaranteeing detection of an original state failure for the selected fault classes. The precision of the RELAY model is what enabled this analysis. We plan to do similar analysis of criteria's ability to transfer a potential failure through the model of information flow transfer; such a preliminary analysis appears in [TRC92]. Neither analysis could be accomplished without the formal model of faults and failures.

We continue to evaluate the RELAY model's capabilities by instantiating it for other fault classes. Thus far, we have only considered simple faults in a single node. It is not clear that these are the most common fault types. We believe, however, that our general framework is applicable to larger, more complex faults and are working on extending the application to more complex fault classes. We are also working on applying the model to specifications in an attempt to detect faults introduced early in the software lifecycle [ROT89].

In addition, we are applying this analysis method to other testing criteria. One direction

of future research is to analyze the fault detection capabilities of failure-based (rather than fault-based) testing criteria, such as Cohen's and White's domain testing [WC80, CHR82], and path selection criteria, such as the variety of data flow path selection criteria [RW85, Nta84, LK83, CPRZ86]. We expect that this will provide us with further insight into the relationship of faults and failures in programs and address the strengths and weaknesses of these two very different approaches to testing. As mentioned, we are also investigating the power of the model of information flow transfer in analyzing test criteria.

Finally, the RELAY model enables us to analyze the implications of many assumptions made by testing researchers (such as the competent programmer hypothesis, the coupling effect, and disallowed coincidental correctness, which is assumed by some path-based criteria); some of these assumptions are analyzed in in [TRC92]. Such analysis may allow us to eliminate or tone down some of these assumptions. The analytical perspective provided by RELAY also suggests empirical studies that must be done to balance analytical evaluation and thus consider the impact of these assumptions.

Appendix

A.1 Origination Conditions [25]

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

[25] Origination conditions for the alternates for a particular potential fault class are grouped and reported here as origination condition sets.

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

Table 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[26] Transfer Conditions Through Relational Operators

[26] 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 ϵ–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 ϵ– conditions is infeasible, absence of a fault is not guaranteed by satisfaction of the remaining ϵ-conditions.

References

[BA80] Timothy A. Budd and Dana Angluin. Two Notions of Correctness and Their Relation to Testing. Technical Report TR 80-19, University of Arizona, 1980.

[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), July 1982.

[CPRZ86] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. An Investigation of Data Flow Path Selection Criteria. *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 23–32, July 1986.

[DD77] D.E. Denning and P.J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[DGK+88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, July 1988. IEEE.

[DLS78] Richard DeMillo, R.J. Lipton, and F.G. Sayward. Hints On Test Data Selection: Help For The Practicing Programmer. *Computer*, 4(11), April 1978.

[DLS79] Richard DeMillo, R.J. Lipton, and F.G. Sayward. Program Mutation: A New Approach to Program Testing. Technical Report v. 2, Infotech International, 1979.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[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.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(5):319–349, July 1987.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[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.

[HE78] William E. Howden and Peter Eichhorst. Proving Properties of Programs from Program Traces. In Edward Miller and William E. Howden, editors, *Tutorial: Software Testing and Validation Techniques*, pages 16–19. IEEE, New York, 1978.

[How76] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2(3), September 1976.

[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.

[How86] William E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, October 1986.

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

[HPR88] S. Horwitz, J. Prins, and T. Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 146–157, 1988.

[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.

[LvH85] David C. Luckham and Friedrich W. von Henke. An Overview of ANNA, a Specification Language for Ada. *IEEE Transactions on Software Engineering*, 2(2):9–22, March 1985.

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

[Mor88] Larry J. Morell. Theoretical Insights into Fault-Based Testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 45–62, Banff, July 1988.

[Mor90] Larry J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* Wiley Series in Business Data Processing. John Wiley & Sons, 1979.

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

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[Pod89] H. Andy Podgurski. *The Significance of Program Dependences for Software Testing, Debugging, and Maintenance.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, September 1989.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth Internation Conference on Software Engineering*, Melbourne, Australia, May 1992.

[ROT89] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to Specification-Based Testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 86–96, Key West, Florida, December 1989. ACM SIGSOFT.

[RT86a] Debra J. Richardson and Margaret C. Thompson. A Formal Framework for Test Data Selection Criteria. Technical Report 86-56, Computer and Information Science, University of Massachusetts, Amherst, November 1986.

[RT86b] 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.

[RT86c] Debra J. Richardson and Margaret C. Thompson. RELAY: A New Model of Error Detection. Technical Report 86-64, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[RT88] Debra J. Richardson and Margaret C. Thompson. The RELAY Model of Error Detection and Its Application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[Tho91] Margaret C. Thompson. *An Investigation of Fault-Based Testing using the* RELAY *Model.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, May 1991.

[TRC92] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. Information Flow Transfer in the RELAY Model. Technical Report TR-92-39, Department of Information and Computer Science, University of California, May 1992.

[WC80] Lee J. White and Edward 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.

[Zei84] Steven J. Zeil. Perturbation Testing for Computation Errors. In *Proceedings of the Seventh International Conference on Software Engineering*, March 1984.

[Zei89] Steven J. Zeil. Perturbation Techniques for Detecting Domain Errors. *IEEE Transactions on Software Engineering*, 15(6):737–746, June 1989.