A New Model for Error Detection

Debra J. Richardson
Margaret C. Thompson

COINS Technical Report 86–64
December 1986

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

This research was supported in part by the National Science Foundation under grants DCR-83-18776 and DCR-84-04217 and by the Rome Air Development Center grant F30602-86-C-0006.

Abstract

This paper defines a formal model for error detection. The model proposes conditions on a set of test data that are necessary and sufficient to guarantee that a fault *originates* an error that is *transferred* through computations and data flow until it is revealed. The model is applied by choosing a fault classification, instantiating these conditions for the classes of faults, and applying them to the program being tested. Such an application guarantees the detection of errors caused by any fault of the chosen classes. The model is applied here to five classes of faults.

1. Introduction

The goal of testing a program is error detection. This is typically done by attempting to select test data for which execution of the program produces erroneous results. If a "good" test data selection criterion is employed and the program executes correctly on the selected test data, then the tester gains assurance that specific types of errors do not exist.

An error is caused by one or more faults in a module. The IEEE Standard [IEEE83] defines <u>test data</u> as "data developed to test a system or system component." It states that an <u>error</u> is "a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically-correct value or condition," while a <u>fault</u> is "an accidental condition that causes a functional unit to fail to perform its required function." Although these definitions provide insight into the accepted meaning of testing concepts, they are imprecise. This paper introduces basic terminology concerning programs and execution and then formally defines a model of errors and error detection within that framework.

A module may be tested at many different levels. The model presented here is directed towards detection of errors in a module known to be close to correct and thus assumes more general functional testing has already been successfully completed.

Our model of error detection builds upon the work of Morrell [Mor84] and is based upon the selection of test data that guarantee the *origination* of an erroneous result that *transfers* to some point at which an error is revealed. This model provides a method by which error detection is applied to a class of faults. The model provides necessary and sufficient conditions that, when satisfied by a test data set, guarantee the detection of an error caused by any fault in the chosen class. If test data is selected to satisfy these conditions, and no errors are detected, the program contains no faults in this class. The model is instantiated once the fault class is chosen, but as a generic model is independent of any particular class of faults.

To demonstrate the effectiveness of the model, we apply it to five classes of faults. For each of these classes, the necessary and sufficient conditions to guarantee detection of any fault in the class are derived and presented here. We believe that the model is widely applicable and are working on further applications.

Our model of error detection also provides a framework within which the capabilities of other testing criteria can be evaluated. An analysis of three test data selection criteria that attempt to detect faults in the five classes to which the model is applied has been performed. This analysis has shown that none of these criteria is completely effective at guaranteeing error detection for these fault classes. It is beyond the scope of this paper to present this comparison; that work is summarized and presented in related papers [RT86b,RT86a].

The next section of this paper introduces a program representation and defines some basic testing concepts using this terminology. The third section defines our new model of error detection within this framework. The fourth section applies this model to five classes of faults. Finally, in section five, we discuss the limitations of the model and some directions for future work.

2. Basic Testing Concepts

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 and consistency in the definition of the criteria. This representation is complete in [RT86b], where a wide variety of testing criteria are also defined. This representation also provides the foundation for the model of error detection that is introduced in the next section.

2.1 Program Representation

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 elements in a domain X_M to elements in a range Z_M, $F_M : X_M \rightarrow Z_M$. Thus, for any $x \in X_M$, execution of M produces a unique $z = M(x) \in Z_M$. An input to a module is a vector x whose elements are values in a designated order for the values of input parameters, imported global variables, and objects of input statements. The elements of an output vector z are values of output parameters, exported global variables, and objects of output statements.

A module implementation can be represented by a directed graph that describes the possible flow of control through the module. A **control flow graph** G_M of a module M is a directed graph, which may be represented by a tuple (N, E), where N is a (finite) set of nodes and $E \subseteq N \times N$ is the set of edges. N contains four special nodes which are added to the graph to facilitate analysis and have no effect on evaluation of the module: n_{start}, the start node; n_{final}, the final node; n_{in}, the input node where parameter and global variable values are imported from the external environment or calling module; and n_{out}, the output node where parameter and global values are exported to the external environment or calling module. Each other node in N represents a simple statement, a group of statements, or the predicate of a conditional statement in M.[1] Each node is represented as an abstract syntax tree, where the leaf nodes represent data objects and the internal nodes represent computational operators. This computation tree describes the statement's hierarchical structure. For each pair of distinct nodes m and n in N, where control may pass directly from the (group of) statement(s) represented by m to that represented by n, there is an edge (m, n) in E. Associated with each edge, (m, n), is a branch predicate, $bp(m, n)$, which is the condition that must hold to allow control to pass directly from node m to node n. 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 edges $p = [(n_1, n_2), ..., (n_{|p|}, n_{|p|+1})]$ such that for all i, $1 \leq i \leq |p|$, $(n_i, n_{i+1}) \in E$. The last node $n_{|p|+1}$ is termed the **open node**, which has been selected by virtue of its inclusion in the last edge but is not visited in the subpath traversal. Given two subpaths $p = [..., (m, n)]$ and $q = [(n, n'), ...]$, there exists a subpath $[..., (m, n), (n, n'), ...]$

[1]This restriction on the form of a control flow graph facilitates the current discussion and is only temporary.

formed by their concatenation and denoted $p{\cdot}q$. An **initial subpath** p is a subpath whose first node is the start node, n_{start}. For any node $n \in N$, the set $INIT(n)$ contains all initial subpaths in G_M whose open node is n. A **path** P [2] is an initial subpath whose open node is the final node, n_{final}. The set of all paths in 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 this paper, we consider only modules with well-formed control flow graphs.

An initial subpath p may be executed on some input x; this execution is denoted $p(x)$. Associated with such execution is a **context** $C_{p(x)}$, which defines the state of the computation. $C_{p(x)}$ contains the values of all variables after execution of $p(x)$. Suppose that some initial subpath $p = ..., (m, n)$ has been executed. Execution is continued by the evaluation of an edge, $(n, n') \in E$. This evaluation includes execution of the node n, selection of the new open node n', and evaluation of the branch predicate $bp(n, n')$, but does not include execution of the open node n'. If n includes an assignment statement, the value(s) of some variable(s) in the context may change. If n includes an input statement, the input vector is extended by the values input to provide x'. If n ends with a conditional statement, evaluation of the condition determines the open node on the extended path — that is, the successor node n' that is selected. This execution provides the updated context for execution of the extended subpath $p' = p \cdot (n, n')$ on input x' — $C_{p\cdot(n,n')(x')}$[3].

A context is also defined for entry to and exit from a module M. $C_{entry(x)}$, where $entry$=[(n_{start}, n_{in}), (n_{in}, m)] and m is the entry node, is the context on entry to the module and contains a defined value for all input parameters and imported global variables and is undefined for all other variables; $C_{P(x)}$ is the context on exit from the module after evaluation of P on x and contains values of all variables at the end of execution of P, including the values for all output parameters and exported global variables.

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

[3]where $x' = x$ if no input occurs.

2.2 Test Data Selection

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 subpath — that is, $t = [t_1,...,t_m]$. The **domain** of an initial subpath p, denoted $dom(p)$, is the set of test data t for which p may be executed. For any node n in G_M, the set $DOMAIN(n)$ is the set of test data t for which n may be executed and is the union of the domains of all initial subpaths in $INIT(n)$; note that $DOMAIN(n_{final}) = X_M$. Thus, $t \in dom(p)$ for some $n \in N$ and $p \in INIT(n)$. Note that a test datum t may be either complete — that is, $\exists P \in PATHS(G_M)$ such that $t \in dom(P)$ — or an incomplete sequence of input values — that is, $\forall P \in PATHS(G_M)$ $t \notin dom(P)$. A test datum t may be incomplete simply because after executing some initial subpath 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 to never terminate. This allows for testing with invalid inputs, which are not in the domain of M but for which M may initiate execution. The **test data domain** D_M for a module M is the domain of inputs from which test data can be selected, $D_M = \{t \mid \exists\, n \text{ in } G_M, p \in INIT(n) : t \in dom(p)\}$. 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, $D_M = DOMAIN(n_{start})$.

A **test data set** T_M for a module M with control flow graph G_M is a finite subset of the test data domain, $T_M \subset D_M$. A **test data selection criterion** S, or simply a **criterion**, is a predicate that assigns a truth value to any pair (G_M, T_M), where G_M is the control flow graph for a module M and T_M is a test data set. A criterion, then, is a set of rules for determining whether a test data set satisfies selection requirements for a particular module.

2.3 Testing Oracles, Errors, and Correctness

To reveal errors by testing, there is usually some test oracle that specifies correct execution of the module [Wey82]. A test oracle might be a functional representation, formal specification, or correct version of the module or simply a tester who knows the module's correct output. In any case, an oracle $O(X_O, Z_O)$ is a relation, $O = \{(x, z)\} \subset X_O \times Z_O$, where X_O and Z_O are the domain and range, respectively, of the oracle. When $(x, z) \in O$, then we write xOz and say that x is O-related to z, meaning that z is an acceptable output for x. If $x \in X_O$ but $(x, z) \notin O$, meaning that z is not acceptable for x, we write $x \not{O} z$. Note that an oracle is a relation; thus, for any input, an oracle may specify more than one acceptable output. This allows for nondeterminism and, in particular, for an oracle to specify a "don't care" case – an input x for which any output is acceptable – by containing the relations (x, z) for all z's.

Although the goal of testing is the detection of errors in a module, it is also important to define our notion of when a module is "correct".

> **Definition:** A module $M : X_M \rightarrow Z_M$ is **equivalent** to an oracle $O(X_O, Z_O)$ if and only if $X_M = X_O$ and $\forall x \in X_M, xOM(x)$.

Thus, a module is equivalent to an oracle if and only if both are defined over the same domain, and for each element of that domain, the module computes one of the values satisfied by the oracle.

Equivalence is a very strong property. It is often the case that a module and an oracle do not have the same domain of definition. Elements of the oracle domain that are not in the module domain may simply be "as yet unimplemented" cases. Alternatively, the oracle may be a specification written in a language that provides type constructs for restricting the range of values for an input, but such restrictions are not allowed in the implementation. In this case, elements of the module domain will not be in the oracle domain. If the implementation explicitly checks for violations of these restrictions, its intent is certainly consistent with the specification. On a similar note, the oracle might be a FORTRAN implementation that is being modernized to an Ada implementation — these two languages have very different mechanisms for specifying types. In each of these situations, although the module is not equivalent to the oracle, it is not necessarily inconsistent with the intended function and hence might not be considered erroneous.

We will work, therefore, with a less strict notion of correctness, called *consistency*, which is concerned only with those inputs for which both the module and the oracle are defined.

Definition: A module $M : X_M \rightarrow Z_M$ is **consistent** with an oracle $O(X_O, Z_O)$ if and only if $\forall x \in (X_M \bigcap X_O) \neq \phi, xOM(x)$.

Consistency holds, therefore, only if the module and the oracle are mutually defined for some elements and the module computes one of the values satisfied by the oracle for each such element, otherwise the module results in an error for some test datum.

Definition: Given a module M with $G_M = (N, E)$ and an oracle $O(X_O, Z_O)$, let $x \in X_M \bigcap X_O$. Execution $M(x)$ reveals an **output error** if $x \not{O} M(x)$.

Both equivalence and consistency are defined above in terms of a "standard" oracle, which judges the correctness of the module's output for valid input data. A tester often has a concept of the "correct" behavior of a module and not just its correct output. Rather than waiting until output is produced to find errors, the tester might check the computation of the module at some intermediate point. This is similar to the approach taken by run-time debuggers and dynamic analysis systems [Bal69,Fai75,RH75,Stu73], where the behavior of variables and the flow of control are monitored in an attempt to locate the source of errors.

This approach to testing can be represented with an oracle that includes information about intermediate values that should be computed by the module — we call this a *context oracle* since it defines the acceptable contexts for a module's execution. A **context oracle** O_C is a relation $O_C = \{((t,p), C_{p(t)})\}$, that relates a test datum and an initial subpath (t,p) to one or more contexts $C_{p(t)}$ that are acceptable after execution of p on t. Thus, the domain of a context oracle O_C is $\{(t,p) \mid t \in D_M, t \in dom(p)\}$. When (t,p) is O_C*-related* to $C_{p(t)}$, we write $(t,p)O_C C_{p(t)}$. A context oracle may derive its intermediate information from some correct module, an axiomatic specification, run-time traces, or monitoring of assertions.

The availability of a context oracle enables a different notion of correctness, which is often more useful in testing. A module is *context-consistent* with a context oracle if and only if for all test data, each context produced by execution on that test data is acceptable.

Definition: A module M is **context-consistent** to a context oracle O_C if and only if $\forall t \in D_M, \forall p : t \in dom(p) \mid (t,p)O_C C_{p(t)}$.

This notion of correctness enables error detection for partial execution on initial data, analysis of whether the module behaves robustly on invalid data, as well as testing of non-terminating modules.

> **Definition:** Given a module M with $G_M = (N, E)$ and a context oracle O_C, let $n \in N$, $p \in INIT(n)$, and $t \in dom(p)$. Execution $M(t)$ reveals a **context error at** n if $(t,p) \not O_C C_{p\cdot(n,n')}(t)$.

Thus, execution of a module that is not context-consistent results in a context error for some test datum.

3. A Model of Fault and Error Detection

In the testing theory introduced by Morrell [MH81,Mor84], an *error* is "created" when an incorrect state is introduced at some location, and it is "propagated" if it persists to the output. We refine this theory by more precisely defining the notion of when an error is introduced and by differentiating between the persistence of an error through computations and its persistence through data flow operations. We define similar concepts, *origination* and *transfer*,[4] as the first erroneous evaluation and the persistence of that erroneous evaluation, respectively.

Depending on one's approach to testing, an error is detected either by revealing an incorrect output or by stopping execution and detecting an incorrect context. An error is caused by some syntactic discrepancy, or *fault*, between the module and some hypothetically-correct module. Testers seldom know what, if anything, is wrong with a module, however, and it is often difficult to select data without any particular error in mind. Another way to test is to select test data aimed at uncovering particular types of faults. The testing scenario might be one in which the tester asks "what if ... ?" — e.g., "what if this expression is wrong and should be like that?" — and attempts to determine the effects of such a transformation. The goal here is either to reveal that the trans-

[4]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.

formation is in fact faulty (and thus the original statement is okay) or to reveal that the original module is faulty. This can be accomplished by selecting test data that distinguishes between the original module and the alternate produced by the transformation. This approach has been taken by several testing methods [Bud81,Bud83,Ham77,Zei83] and has been called "error-based testing" [Mor84,Wey81]. Because of the formal distinction we make between faults and errors, we call it *fault-based testing.*

Our model of error detection formalizes this fault-based testing approach. This approach relies on an assumption that the module being tested bears a strong resemblance to some hypothetically-correct module. Such a module need not actually exist, but we assume that the tester is capable of producing a correct module from the given module and knowledge of the faults and errors detected. With such a concept of the correct module, the tester should be able to identify the location at which a fault first *originates* an erroneous result and monitor the way in which that error *transfers* through the module until it is revealed.

In this section, we present a formal model for error detection that is geared toward the detection of faults by guaranteeing the *origination* and *transfer* of errors. As currently formulated, the model is limited to the detection of errors resulting from a single fault in one node — that is, a fault may involve more than one statement, but must be wholly contained in one node. We first define faults and errors in the framework provided in Section 2. Then, we present the concepts of origination and transfer and define conditions whose satisfaction guarantees the detection of an error for a specific fault.

3.1 A View of Errors and Faults

An error in a module is caused by one or more faults in that module. Both errors and faults can be defined in terms of discrepancies between the tested module and some hypothetically-correct module. Given an oracle O and a context oracle O_C, let M^* be a hypothetically-correct module, which is consistent with O and context-consistent with O_C.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, a **potential fault** f_n at node $n \in N$ is a transformation on some node $n^* \in N^*$ such that $f_n(n^*) = n$.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let EXP be an expression at some node $n \in N$, EXP^* be the corresponding expression at $n^* \in N^*$, and $t \in DOMAIN(n)$. Execution $M(t)$ results in a **potential error** in EXP if and only if the value of EXP differs from the value of EXP^* for $M^*(t)$ — that is, $exp \neq exp^*$. [5]

These terms are qualified by "potential", because an error may not be revealed even though a node containing a fault is executed for some test datum. This anomaly, which is often referred to as "coincidental correctness", may occur because execution of the node does not *originate* a potential error or because a potential error is masked out and does not *transfer* until it is revealed. In either case, the module appears correct, but just by coincidence of the test data selected. It is also possible that despite a discrepancy between the tested module and the hypothetically-correct module, the tested module produces correct output for all input. In this case, the module is not merely coincidentally correct, it is correct (consistent), and thus the transformation is not a fault.

To detect a potential fault, erroneous results must appear for some test datum as an incorrect context (context error) or an incorrect output (output error). To reveal a context error, a potential fault must *originate* a potential error that *transfers* through the node after which an incorrect context results. To reveal an output error, a potential fault must cause a context error that *transfers* from node to node until an incorrect output results.

First, let us consider the original occurence of a potential error within a node. Any potential fault at a node is a transformation at some smallest subexpression of the node. If this subexpression evaluates incorrectly, a potential error *originates*.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let $n \in N, n^* \in N^*$, and f_n be a potential fault in M such that $f_n(n^*) = n$. Let $SEXP$ be the smallest subexpression of n containing f_n and

[5]When an expression EXP is evaluated during execution, the value of the expression is denoted exp (i.e., upper case refers to a syntactic expression, while lower case refers to the value of an expression).

$SEXP^{*}$ be the correct subexpression of n^{*}. Let $t \in DOMAIN(n)$, then f_n **originates** a potential error for execution $M(t)$ if and only if $sexp \neq sexp^{*}$. [6]

A potential error may be masked out by subsequent computations performed within the node or within other nodes on the path. A potential error in some expression *transfers* to a "super"-expression that references the erroneous expression if the evaluation of the "super"-expression remains incorrect.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^{*} with $G_{M^{*}} = (N^{*}, E^{*})$, let $SEXP$ be a subexpression of some node $n \in N$, $SEXP^{*}$ be the correct subexpression of $n^{*} \in N^{*}$, and EXP be an expression in M that references $SEXP$. Let $t \in DOMAIN(n)$ such that $sexp \neq sexp^{*}$ for execution $M(t)$, then the potential error in $SEXP$ **transfers** to EXP for execution $M(t)$ if and only if $exp \neq exp^{*}$.

A potential error may transfer in two different ways. Within a simple statement, the potential error is used in the computation of the statement. To affect evaluation of the entire statement, the potential error must transfer through all ancestor operators in the statement's computation tree.

Definition: Given a module M with $G_M = (N, E)$, and a hypothetically-correct module M^{*} with $G_{M^{*}} = (N^{*}, E^{*})$, let $n \in N, n^{*} \in N^{*}$, and f_n be a potential fault in M such that $f_n(n^{*}) = n$. Let op$(\ldots, EXP, \ldots)$ be a subexpression of n and op$(\ldots, EXP^{*}, \ldots)$ be the correct subexpression of n^{*}. Let $t \in DOMAIN(n)$ such that $exp \neq exp^{*}$ for execution $M(t)$, then the potential error in EXP **computationally transfers** to the parent expression op$(..., EXP, ...)$ for execution $M(t)$ if and only if op$(..., exp, ...) \neq$ op$(..., exp^{*}, ...)$.

A potential error may also transfer from statement to statement, which is termed *data flow transfer*. A potential error transfers through data flow if it reaches another statement — that is, if the potential error is reflected in the value of some variable that is referenced at some other statement — and the smallest subexpression containing that reference results in a potential error.[7]

Now, let us examine how a context error is revealed and is transferred along a path to result in an output error. We say that a potential error at a node transfers to *reveal* a context error if the

[6]Note that the limitation to single faults implies that $SEXP^{*}$ differs from $SEXP$ only by the potential fault.

[7]We do not define data flow transfer formally here as it is not the focus of this paper.

context is unacceptable after execution of the node.

> **Definition:** Given a module M with $G_M = (N, E)$ and a context oracle O_C, let $n \in N$, $p \in INIT(n)$, and $t \in dom(p)$. Let EXP be a subexpression of n such that execution $M(t)$ results in a potential error in EXP. The potential error **reveals a context error** at node n for execution $M(t)$ if and only if $\exists (n, n') \in E$ such that $t \in dom(p \cdot (n, n'))$ and $(t, p) \not{O}_C C_{p \cdot (n,n')(t)}$.

A context error is said to transfer to *reveal* an output error if the context remains incorrect at least until some incorrect output is produced.

> **Definition:** Given a module M with $G_M = (N, E)$, a context oracle O_C, and an oracle $O(X_O, Z_O)$, let $n \in N$, $p \in INIT(n)$, and $x \in dom(p)$ such that $(x, p) \not{O}_C C_{p(x)}$. The context error at node n **reveals** an output error for execution $M(x)$ if and only if the following property holds: $\exists q : x \in dom(p \cdot q)$, $x \not{O} M(x)$ and $\forall q_i : q = q_i \cdot q_j, x \in dom(p \cdot q_i)$ such that $(x, p) \not{O}_C C_{p \cdot q_i(x)}$.

To reveal a context error, evaluation of the potential fault on some test datum must originate a potential error and transfer that potential error through all ancestor operators to the root of the node. To reveal an output error, this incorrect context must then transfer through data flow to other statements on the path, where the potential error that results in each such statement must transfer through computations in that statement, until some statement produces erroneous output. Only if an output error is revealed do we know that the module is incorrect (inconsistent).

> **Definition:** Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let $n \in N$, $n^* \in N^*$, and f_n be potential fault in M such that $f_n(n^*) = n$. f_n **is a fault in** G_M if and only if $\exists x \in X_M \bigcap X_{M^*}$ such that $M(x)$ reveals a context error that transfers to reveal an output error. Otherwise, n is **equivalent to** n^*.

Thus, by definition, a potential fault is a fault only if it produces incorrect output for some test datum.

Figure 1 illustrates the origination of a potential error and its transference throughout a module and how this provides for the discovery of a fault. To detect a fault, execution of a potential fault for some test datum must 1) originate a potential error in the smallest containing subexpression; 2) transfer that potential error through each ancestor operator in the node, thereby revealing a

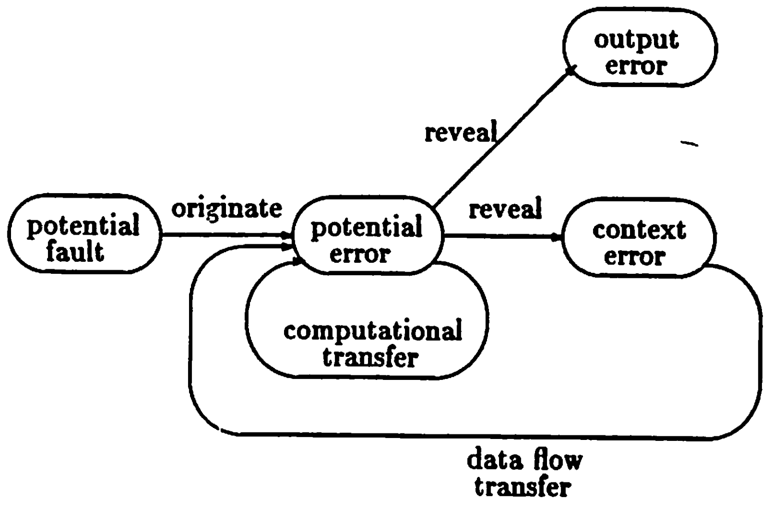

Figure 1: Origination and Transfer of Errors

context error; and 3) transfer that context error through each node on the path to reveal an output error. If there is no test datum for which a potential error originates and "total" transfer occurs, then the potential fault is not a fault, and the node containing the potential fault is equivalent to some hypothetically-correct node.

3.2 Example

The module in Figure 2 is used in this section to illustrate potential faults and the origination and transfer of potential errors produced by those faults.

Suppose that the assignment statement at node 4 should be $Z := (2+Y) * A$. First, let us consider a test datum that results in an output error and demonstrates that the module is indeed incorrect. If node 4 is executed for the test datum $(A = 1, B = -2, X = 2, Y = 4)$, a potential error originates — $(2 * 4) \neq (2 + 4)$. This potential error transfers through the multiplication operator — $(8 * 1) \neq (6 * 1)$ — and the assignment operator — $(Z := 8) \neq (Z := 6)$ — thereby revealing a context error. This context error is not affected by the conditional statement at node 5. Next, the context error transfers through the computation of V at node 6 — $(V := (-2)**8) \neq$

$(V := (-2)**6)$, which is output at node 8. Thus, execution of the module for the test datum $(A = 1, B = -2, X = 2, Y = 4)$ reveals an output error.

Now, consider the ways in which a potential fault might fail to result in an error. First, let us consider a context error that does not transfer to an output error. For the test datum $(A = 1, B = -1, X = 2, Y = 4)$, which follows the same path as that above, a context error is revealed at node 4 and reaches the reference to Z at node 6, but it is masked out by the computation of V — $(V := (-1)**(2*4)) = (V := (-1)**(2+4))$. Thus, the context error does not transfer to the output. Next, let us consider failure to reveal a context error. An incorrect context may not be revealed because no potential error originates for the test datum selected. For instance, if node 4 is executed for $(A = 1, B = -1, X = 2, Y = 2)$, a potential error does not originate — $(2*2) = (2+2)$. Finally, a potential error may not transfer to affect evaluation of the node. Consider node 3 and suppose that the assignment statement should be $Z := (2 + X) * A$. Execution for the test datum $(A = 0, B = -1, X = 4, Y = 2)$ originates a potential error — $(2 * 4) \neq (2 + 4)$ — but it does not transfer through multiplication by A — $(8 * 0) = (6 * 0)$. In fact, this potential fault is not a fault because any potential error that originates at node 3 does not transfer since $A = 0$.

In addition to illustrating the variety of faults and errors, this example demonstrates how coincidental correctness occurs when potential errors are masked out by later computations in a node or along a path. Although this is a contrived example, coincidental correctness is a common phenomenon of testing. If coincidental correctness did not occur, a single arbitrary execution of a node would reveal all faults in that node. Thus, most of the problems associated with test data selection would be eliminated. A goal of test data selection criteria, therefore, is to minimize the occurrence of coincidentally correct results by astutely selecting test data aimed at detecting faults, through the origination and transfer of errors.

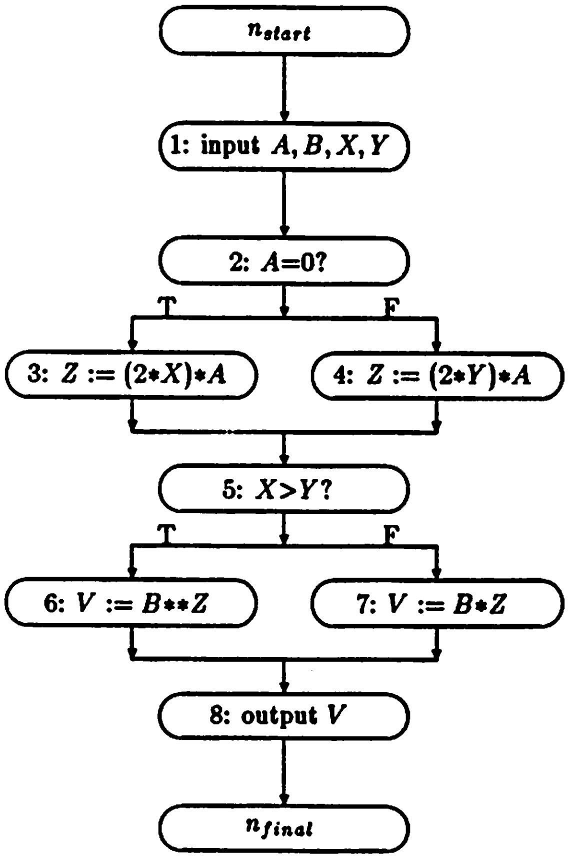

Figure 2: Error Origination and Transference Example

3.3 Error Detection

Here, we provide a model of error detection whereby test data is selected that originates a potential error that transfers until that error is detected. This model uses the concepts of origination and transference to define necessary and sufficient conditions to guarantee that a context error is revealed.

Given an oracle and a module M with $G_M = (N, E)$ that contains a potential fault f_n at node $n \in N$, a test data selection criterion S is said to *guarantee detection* of a fault f_n if for all test data sets T_M where (G_M, T_M) satisfies S, there exists $t \in T_M$ such that f_n originates an error for $M(t)$ that transfers until it is revealed by the oracle. If a context oracle exists, the potential fault must reveal a context error for some test datum.

Definition: Given a context oracle O_C and a module M with $G_M = (N, E)$ that contains a potential fault f_n at node $n \in N$, a test data selection criterion S **guarantees detection** of f_n if and only if $\forall T_M$ such that $S(G_M, T_M), \exists p \in INIT(n), \exists (n, n') \in E, \exists t \in dom(p \cdot (n, n')) \bigcap T_M$ such that $(t,p) O_C C_{p(t)}$ and $(t,p) \not{O}_C C_{p\cdot(n,n')(t)}$.

If error detection is done by a standard (output) oracle, then a context error revealed by f_n must also transfer to the output for some test datum.

Definition: Given an oracle $O(X_O, Z_O)$ and a module M with $G_M = (N, E)$ that contains a potential fault f_n at node $n \in N$, a test data selection criterion S **guarantees detection** of f_n if and only if $\forall T_M$ such that $S(G_M, T_M), \exists p \in INIT(n), \exists q, \exists x \in dom(p \cdot q) \bigcap X_M$ such that $(t,p) \not{O}_C C_{p(t)}, x_{p\cdot q} \not{O} M(x)$, and $\forall q_i$ such that $q = q_i \cdot q_j, (x,p) \not{O}_C C_{p\cdot q_i(x)}$.

Note that guaranteeing detection of an output error is the same as guaranteeing detection of the corresponding fault. Guaranteeing detection of a context error is not sufficient, however, because it is possible that the context error is not transferred to the output.

Let us assume, for the purposes of this paper, that a context oracle exists. We are concerned, then, with the ability of a test data selection criterion to guarantee the detection of a context error. Here, we define *origination*, *transfer*, and *revealing conditions* that are necessary and sufficient to guarantee that a context error is revealed. Sufficient means that if the module is executed on data that satisfies the conditions and the node is faulty, then a context error is revealed. Necessary, on the other hand, means that if a context error is revealed then the module must have been executed on data that satisfies the condition and the node is faulty.

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

The *origination condition* guarantees that the smallest subexpression containing a potential fault originates a potential error.

> **Definition:** Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let $n \in N, n^* \in N^*$ and f_n be a potential fault in M such that $f_n(n^*) = n$. Let $SEXP$ be the smallest subexpression of n containing f_n and $SEXP^*$ be the correct subexpression of n^*, then the **origination condition** for f_n is $oc(f_n) = (sexp \neq sexp^*)$. A test data selection criterion S **guarantees origination** of a potential error for f_n if and only if $\forall T_M$ such that $S(G_M, T_M)$, $\exists t \in T_M$ that satisfies $oc(f_n)$.

The origination condition is the necessary and sufficient condition that must be satisfied by test data to guarantee that the smallest subexpression containing a potential fault originates a potential error. If the origination condition is infeasible, then the potentially faulty expression is equivalent to the correct one, and no fault exists.

The origination condition, as currently defined, is dependent on knowledge of the correct node. Not only do testers not know the correct node, they probably do not even know that a particular node is faulty. Thus, we need a notion of fault detection that is generally applicable, rather than guided by specific knowledge of the existence of faults. This can be achieved by working under the assumption that any node, in fact any subexpression of any node, might be incorrect and considering the potential ways in which that expression might be faulty. These potential faults can be grouped into classes based on some common characteristic of the transformation. For a given class of potential faults in an expression, our goal is to define conditions that guarantee origination of a potential error for any potential fault of that class.

A class of potential faults determines a set of alternative expressions that must contain the correct expression, if the original expression indeed contains a fault of that class.

> **Definition:** Given a module M with $G_M = (N, E)$, let $n \in N$ and F_n be a class of potential faults at n. Let $SEXP$ be the smallest subexpression of n containing any $f_n \in F_n$, then the **alternate set** for F_n, $ALT(F_n)$, is the set of expressions $\{\overline{SEXP} \mid \exists f_n \in F_n \text{ such that } f_n(\overline{SEXP}) = SEXP\}$.

Assuming that we do not know the correct expression, then to guarantee origination of a potential error, the potentially faulty expression must be distinguished from each expression in the alternate set.

Consider the expression $EXP_1 + EXP_2$, for instance. One class of potential faults that may occur in this expression is the class of arithmetic operator faults. If the operator $+$ is faulty, then the correct node must be of the form $EXP_1 \overline{\text{aop}} EXP_2$, where $\overline{\text{aop}}$ is not $+$. Thus, the alternate set for the class of arithmetic operator faults in the expression $EXP_1 + EXP_2$ is $\{EXP_1 \overline{\text{aop}} EXP_2 \mid \overline{\text{aop}}$ is an arithmetic operator other that $+\}$. To guarantee origination of a potential error, test data must be selected that distinguishes $EXP_1 + EXP_2$ from each alternate in the set.

In our model, then, an origination condition is defined for each alternative expression. Each such condition guarantees origination of a potential error if the corresponding alternate were indeed the correct expression. When such an origination condition is infeasible, the alternate is equivalent to the potentially faulty expression — thus, if this alternate was intended, the potential fault would not constitute a fault.

To guarantee detection of a fault in a class, we must guarantee that the potentially faulty expression is distinguished from each nonequivalent expression in the alternate set as any one of these might be the correct expression. Thus, all feasible origination conditions must be satisfied. For a fault class, we define a set of conditions, called the *origination condition set*, that guarantees origination of a potential error in an expression, if the expression contains a fault of this class.

Definition: Given a module M with $G_M = (N, E)$, let $n \in N$ and F_n be a class of potential faults at n. Let $SEXP$ be the smallest subexpression of n containing any $f_n \in F_n$, then the **origination condition set** for F_n is the set of origination conditions $OC(F_n) = \{[(sexp \neq \overline{sexp}) \mid \overline{SEXP} \in ALT(F_n)$ such that $\exists t \in DOMAIN(n)$ that satisfies $(\overline{sexp} \neq sexp)]\}$. A test data selection criterion S **guarantees origination** of a potential error for a fault class F_n if and only if S satisfies $OC(F_n)$ — that is, $\forall T_M$ such that $S(G_M, T_M)$, $\forall oc \in OC(F_n)$, $\exists t \in T_M$ that satisfies oc.

The origination condition set is necessary and sufficient to guarantee that a potential fault of a given class originates a potential error.

To reveal a context error, a potential error originated at the smallest subexpression containing a potential fault must transfer to effect evaluation of the entire node. To do so, it must transfer through all ancestor operators up to and including the operator at the root of the node. The *transfer condition* guarantees that a potential error transfers to a parent expression.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let $n \in N, n^* \in N^*$ and f_n be a potential fault such that $f_n(n^*) = n$. Let op$(\ldots, EXP, \ldots)$ be a subexpression of n and op$(\ldots, EXP^* \ldots)$ be the correct subexpression of n^*. Let $t \in DOMAIN(n)$ such that $exp \neq exp^*$ for execution $M(t)$, then the **transfer condition** for **op** is tc(op) = op$(..., exp, ...) \neq$ op$(..., exp^*, ...)$. A test data selection criterion S **guarantees transfer** of a potential error in EXP through the operator **op** if and only if $\forall T_M$, where $S(G_M, T_M)$, $\exists t \in T_M$ that satisfies tc(op).

The transfer condition is the necessary and sufficient condition that must be satisfied by test data to guarantee that a potential error in a subexpression transfers to the parent expression. If a transfer condition for **op** is infeasible, then no potential error can be transferred through **op** to affect evaluation of the parent expression — thus, op$(\ldots, EXP, \ldots)$ is equivalent to op$(\ldots, EXP^*, \ldots)$.

The transfer conditions that must be satisfied for a potential error are determined by the operators in the node containing the potential error. To guarantee transference of a potential error, the transfer condition must be satisfied for each operator that is an ancestor of the subexpression in which the potential error originates. The *node transfer condition* is the conjunction of all such transfer conditions and must be satisfied to guarantee transfer through the entire node.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let $n \in N, n^* \in N^*$ and f_n be a potential fault such that $f_n(n^*) = n$. Let $SEXP$ be the subexpression in which the potential error originates, where the node n has the form $\text{op}_t(\ldots,\text{op}_2(\ldots,\text{op}_1(\ldots,SEXP,\ldots),\ldots),\ldots)$. The **node transfer condition** for f_n is $TC(f_n) = \bigwedge tc_i, 1 \le i \le t$, where tc_i is the transfer condition for the operator op_i. A test data selection criterion S **guarantees transfer** of a potential error in $SEXP$ through the node n if and only if $\forall T_M$, where $S(G_M, T_M)$, $\exists t \in T_M$ that satisfies $TC(f_n)$.

Since a potential fault must both originate a potential error and transfer it through the node, a criterion must select a single test datum that satisfies both the origination and node transfer conditions to guarantee a fault's detection. The *revealing condition* for a potential fault f_n occurring in node n is the conjunction of the origination condition and the node transfer condition for f_n and n.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically-correct module M^* with $G_{M^*} = (N^*, E^*)$, let $n \in N, n^* \in N^*$ and f_n be a potential fault such that $f_n(n^*) = n$. Let $SEXP$ be the subexpression in which the potential error originates, where the node n has the form $\text{op}_t(\ldots,\text{op}_2(\ldots,\text{op}_1(\ldots,SEXP,\ldots),\ldots),\ldots)$. The **revealing condition** for f_n and n is $rc(f_n) = oc(f_n) \wedge (\bigwedge tc_i, 1 \le i \le t)$, where each tc_i is the transfer condition for the operator op_i. A test data selection criteria S **guarantees revelation** of a context error for f_n if and only if $\forall T_M$ such that $S(G_M, T_M)$, $\exists t \in T_M$ that satisfies $rc(f_n)$.

The revealing condition is the necessary and sufficient condition that guarantees that the potential fault reveals a context error. If the revealing condition is infeasible, then no data exists for which a context error can be revealed, and the potentially faulty node and the correct node are equivalent.

Again, for a fault class, a revealing condition exists for each alternative expression. Each feasible condition in this set must be satisfied by a test data selection criterion to guarantee that a context error is revealed for any fault in a particular fault class. The *revealing condition set* contains a revealing condition for each alternate in the alternate set.

Definition: Given a module M with $G_M = (N, E)$, let $n \in N$ and F_n be a class of potential faults at n. Let $SEXP$ be the smallest subexpression of n containing any $f_n \in F_n$. where the node n has the form $\text{op}_t(\ldots,\text{op}_2(\ldots,\text{op}_1(\ldots,SEXP,\ldots),\ldots),\ldots)$. The **revealing condition set** for F_n is the set of revealing conditions $RC(F_n) =$

$\{oc \wedge (\bigwedge tc_i, 1 \leq i \leq t) \mid oc = (sexp \neq \overline{sexp}), \overline{SEXP} \in ALT(F_n)$ and tc_i is the transfer condition for the operator op_i such that $\exists t \in DOMAIN(n)$ that satisfies $(oc \wedge (\bigwedge tc_i, 1 \leq i \leq t))\}$. A test data selection criterion S **guarantees revelation** of a context error for a fault class F_n if and only if S satisfies $RC(F_n)$ — that is, $\forall T_M$ such that $S(G_M, T_M)$, $\forall rc \in RC(F_n)$, $\exists t \in T_M$ that satisfies rc.

The revealing condition set is necessary and sufficient to guarantee that a potential fault of a particular class reveals a context error.

Our model of error detection is based on the generic revealing condition sets that have been defined. The model is applied by selecting a fault classification and instantiating the origination and transfer conditions for the classes of faults. These instances of the origination and transfer conditions are then evaluated on the nodes in a module's control flow graph to provide the actual revealing condition sets that must be satisfied to guarantee the detection of any fault in the chosen classification. The next section provides an instantiation of the model for five classes of faults and then illustrates this instantiation on several nodes of a module.

As noted, the model is currently limited to the detection of context errors. Thus, the revealing condition set is necessary, but not sufficient for the true detection of a fault. This is because the context error introduced by satisfaction of these conditions may still be masked out by later computations on the path and thus not transfer to produce an output error. To describe the conditions under which a criterion guarantees the detection of a fault of a particular class through an output error, this condition set must be augmented to include data flow transfer conditions. We believe that our model of defining conditions that guarantee the origination and transference of an error will not require reformulation when extended to include transfer from node to node as described by data flow transfer conditions.

The model is also limited to detecting errors that result from a single fault in a node. When detecting context errors, this is not a significant limitation. Error detection can proceed from node to node where each node in the module is considered only after all of its predecessors (with the exception of loops) have been considered. If a context error is detected at a node, the potential fault is corrected before proceeding to any successor node. This ensures that the context is acceptable prior to execution of the node under consideration. This limitation may be more of a weakness when data flow transfer is considered, since we must then be concerned with the interaction between statements. We intend to reconsider the limitations of this restriction when we extend the model to data flow transfer conditions.

We believe that our model of error detection is applicable to a wide variety of fault classes. The model leads to more rigorous results than other criteria that are geared toward detecting these faults. This is shown in another paper [RT86a], where we use the model to evaluate the capabilities of three test data selection criteria in relation to the five fault classes illustrated in Section 4..

4. Application of the Model

In this section, we apply the model developed in Section 3 and develop revealing conditions for context errors for five fault classes. These five classes have been selected because of their relevance to a number of test data selection criteria, in particular those criteria analyzed in [RT86b].

There are two qualifications of this application that should be noted. First, each class for which revealing conditions are developed is a class of atomic faults, where a (potential) fault f_n is *atomic* if the node n differs from the correct node n^* by a single token. Atomic faults may be classified according to what token in a node is faulty and how it differs from the correct node. Likewise, a class of atomic potential faults is determined by the class of all semantically-correct replacements for the token. Second, we represent each simple statement and the predicate of each conditional statement in a module as a single node in the control flow graph. Thus, the transfer conditions developed are computational transfer conditions. In sum, the application presented in this section provides revealing conditions for context error for single statements potentially containing an atomic fault in one of five classes.

To determine the revealing conditions for a class of potential faults, we must instantiate the origination condition set for the class as well as the applicable transfer conditions. The transfer conditions through a particular operator, however, do not depend on the fault and are applicable to many classes of faults. Hence, this section is divided into three subsections. First, for each class of faults, we develop the origination condition set. Then, we derive the transfer conditions for four types of expressions — assignment, boolean, arithmetic, and relational. Finally, we demonstrate how revealing conditions for context errors are formed by combining and evaluating the origination conditions for a fault class at a subexpression of a node and the applicable transfer conditions for an expression.

4.1 Origination Conditions

To develop the origination condition set for a class of faults, we first determine the alternate set, and then derive the origination condition for each alternate. Each origination condition must be both sufficient and necessary to guarantee origination of a potential error for the corresponding alternate. Sufficiency means that if the expression should be the alternate and the module <u>is</u> executed on a test datum satisfying the origination condition, then a potential error <u>does</u> originate. Necessity means that if the expression should be the alternate and the module <u>is not</u> executed on a test datum satisfying the origination conditions, then a potential error <u>does not</u> originate.

To guarantee origination of a potential fault in a particular class, the original expression must be distinguished from each of the nonequivalent alternates. A test datum must be selected that satisfies each feasible origination condition, thereby satisfying the origination condition set. One or more origination conditions in the origination condition set may consist of disjunctive conditions that overlap (thus, satisfying the common disjunct serves to satisfy each such origination condition). When this is the case, we may reduce the origination conditions in the origination condition set forming a *sufficient* origination condition set whose satisfaction implies satisfaction of the origination condition set. This reduction process must ensure that each origination condition in the origination condition set has a nonempty intersection with some origination condition in the sufficient origination condition set.

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 1: Origination Condition Set for Variable Reference Fault

4.1.1 Origination of a Variable Reference Fault

A potential error may result when the name of a referenced variable is mistakenly replaced by another valid variable name. Any variable reference is potentially faulty. Given a reference to a variable V (a potential access to the value of V), if V is a faulty variable name, then the correct reference must be in the alternate set $\{[\overline{V} \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$.

For a variable reference V, the origination condition for a variable $\overline{V}$ in the alternate set is $[\overline{v} \neq v]$. This origination condition is both necessary and sufficient to originate a potential error, since an expression could reference V for $\overline{V}$ and not originate a potential error if and only if $(v = \overline{v})$ for all data in a test data set. If this condition is not feasible, then the original and the alternate variable references are equivalent. The origination condition set contains this origination condition for each $\overline{V}$ in the alternate set and therefore is $\{[\overline{v} \neq v \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$. The origination condition set is summarized in Table 1.

4.1.2 Origination of a Variable Definition Fault

A potential error may result when the name of a defined variable is mistakenly replaced by another valid variable name. Given a definition of a variable $V := EXP$ [8], if V is a faulty variable name, then the correct definition must be in the alternate set $\{[\overline{V} := EXP \mid \overline{V}$ is a variable other

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

than V that is type-compatible with V]}.

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

To demonstrate that this condition is both necessary and sufficient to originate a potential error see table 2, which enumerates all combinations of the values of pertinent variables and expressions for both the original and the alternate before and after evaluation of the statement. For cases i,ii, and iii, the values of V and EXP satisfy the origination condition, and evaluation of the assignment statement originates a potential error. In cases i and iii, evaluation of the original expression $V := EXP$ results in $\overline{V} \neq v$, while evaluation of the alternate $\overline{V} = EXP$ results in $\overline{V} = v$. In cases ii and iii, evaluation of the statement $V := EXP$ results in $V \neq v$, while for $\overline{V} := EXP$ $V = v$ results. Thus, the origination condition is sufficient to originate a potential error for a variable definition fault. To see that the condition $[(exp \neq v)$ or $(\overline{v} \neq v)]$ is necessary, consider case iv for which the origination condition is not satisfied. Evaluation of the original and the alternate statements result in the same values for the variables; hence no potential error originates.

The origination condition set for a variable definition fault at the statement $V := EXP$ is stated in Table 3 — $\{[(\overline{v} \neq v)$ or $(exp \neq v) \mid \overline{V}$ is a variable other than V that is type-compatible with V]}.

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

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

Table 2: Variable Definition

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

Table 3: Origination Condition Set for Variable Definition Fault

compatible with V]} must be satisfied to guarantee origination of a potential error for a variable definition fault.

4.1.3 Origination of a Boolean Operator Fault

A potential error may result when a boolean operator is mistakenly replaced by another boolean operator. The boolean operators we consider are the binary operators **or** and **and** and the unary operator **not**.

Consider first a potentially faulty unary boolean operator. Given a boolean expression **bop** (EXP_1), where the boolean operator is **not** or **null**, if the unary boolean operator **bop** is faulty, then the correct expression is equivalent to the expression (**not bop** (EXP_1)). Hence, the alternate set is $\{(\textbf{not}\,\textbf{bop}(EXP_1))\}$. The origination condition is $[exp_1 \neq \textbf{not}\ exp_1]$, which is satisfied by all values of exp_1. Therefore, we need only guarantee execution of the statement containing the expression EXP_1 to guarantee origination of a potential error for this particular fault.

Consider now a potentially faulty binary boolean operator. Given a boolean expression $(EXP_1$ **bop** $EXP_2)$, where **bop** is **and** or **or**, if the binary boolean operator **bop** is faulty, then the correct expression must be in the alternate set $\{(EXP_1\ \overline{\textbf{bop}}\ EXP_2) \mid \overline{\textbf{bop}}$ is a binary boolean operator other than **bop** $\}$. If **bop** is **and**, then $(EXP_1$ **and** $EXP_2)$ must be distinguished from $(EXP_1$ **or** $EXP_2)$; vice versa, if **bop** is **or**. The origination condition for either binary boolean operator and its alternate is $[(exp_1\ \textbf{and}\ exp_2) \neq (exp_1\ \textbf{or}\ exp_2)]$ or simply $[exp_1 \neq exp_2]$. Table 4 enumerates all possible cases for this expression, from which it is clear that this condition is both sufficient and necessary to originate a potential error. In cases ii and iii, the origination condition is satisfied, and a potential error originates. In cases i and iv the origination condition is not satisfied, and the original expression and the alternate expression evaluate the same. Thus, a potential error originates if and only if the origination condition $[exp_1 \neq exp_2]$ is satisfied.

The origination condition set, summarized in Table 5, contains the single condition that $[exp_1 \neq exp_2]$, which is satisfied when exactly one operand is true.

	exp_1	exp_2	not (EXP_1)	$(EXP_1$ and $EXP_2)$	$(EXP_1$ or $EXP_2)$
i	*true*	*true*	*false*	*true*	*true*
ii	*true*	*false*	*false*	*false*	*true*
iii	*false*	*true*	*true*	*false*	*true*
iv	*false*	*false*	*true*	*false*	*false*

Table 4: Boolean Operator Evaluation

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

Table 5: Origination Conditions for Boolean Operator Fault

4.1.4 Origination of a Relational Operator Fault

A potential error may result when a relational operator is mistakenly replaced with another relational operator. We consider six relational operators: $<, \leq, =, \geq, >, \neq$. Given a relational expression $(EXP_1 \text{ rop } EXP_2)$, if the relational operator rop is faulty, then the correct expression must be in the alternate set $\{(EXP_1\ \overline{\text{rop}}\ EXP_2) \mid \overline{\text{rop}}$ is a relational operator other than rop $\}$. Each origination condition depends on both original and alternative relational operators.

For any relational expression, there are three possible relations for which test data may be selected — $(exp_1 < exp_2)$, $(exp_1 = exp_2)$, $(exp_1 > exp_2)$. Table 6 enumerates the evaluation of any relational expression with data satisfying these three relations, which is useful in developing the origination condition set for the class of relational operator faults for each relational operator. As an example, let us construct the origination condition set for the relational operator $<$. We must determine the origination condition that distinguishes $(EXP_1 < EXP_2)$ from each alternate $(EXP_1 \overline{\text{rop}} EXP_2)$. The operator $=$ is one alternative operator; the origination condition that distinguishes $(EXP_1 < EXP_2)$ from $(EXP_1 = EXP_2)$ is $[(exp_1 < exp_2$ or $(exp_1 < exp_2)]$. As seen

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

Table 6: Relational Operator Evaluation

from table 6, for a test datum satisfying either of these two relations, the original and alternative expressions evaluate differently; this condition is, therefore, sufficient for origination of a potential error. For a test datum satisfying $(exp_1 > exp_2)$, which does not satisfy the origination condition, the expressions evaluate the same; hence, this condition is necessary for origination of a potential error. The origination conditions for the other alternative operators are derived similarly. The origination conditions for relational operator faults are summarized in Table 7.

The origination condition set for the class of relational operator faults for a particular operator is the set of all origination conditions that distinguish that original operator from some other alternate. For a less than (<) fault, for instance, the origination condition set is $\{[exp_1 = exp_2]$, $[(exp_1 < exp_2)$ or $(exp_1 = exp_2)]$, $[exp_1 > exp_2]$, $[(exp_1 < exp_2)$ or $(exp_1 = exp_2)$ or $(exp_1 > exp_2)]$, $[(exp_1 < exp_2)$ or $(exp_1 > exp_2)]\}$. The origination condition sets for other relational operator faults are derived similarly.

For a particular relational operator, a test datum relation may satisfy the origination condition for more than one alternate. As was done with the origination condition set for variable definition faults, the origination condition set may be reduced to form a set of origination conditions that is sufficient for origination of a potential a potential error. When all test data relations are satisfiable, reduction of the origination condition set for any particular relational operator results in two

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 $(exp_1 > exp_2)]$	$[exp_1 \leq exp_2]$
$\leq, =$	$[exp_1 < exp_2]$	$[exp_1 < exp_2]$
$\leq, \neq$	$[(exp_1 = exp_2)$ 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 $(exp_1 = exp_2)]$	$[exp_1 \leq exp_2]$
$\neq, >$	$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)]$	$[exp_1 \leq exp_2]$
$\geq, >$	$[exp_1 = exp_2]$	$[exp_1 = exp_2]$

Table 7: Origination Conditions for Relational Operator Faults

conditions that must be satisfied. These sufficient origination condition sets are summarized in Table 8. It is important to remember that when the sufficient origination condition set is infeasible due to the semantics of the program — that is, at least one of the two conditions cannot be satisfied due to the domain of the statement containing the relational expression — it is possible that an alternate that is not equivalent to the original expression has not yet been distinguished. If the third relation is feasible, data that satisfies it must be selected to ensure that all nonequivalent alternates are distinguished.

Consider for example, the origination condition set for the relational operator <. The sufficient origination condition set for this operator is $\{[exp_1 = exp_2], [exp_1 > exp_2]\}$, since at least one of the relations in the set satisfies the origination condition for each alternate. Suppose, however, that $(exp_1 = exp_2)$ is infeasible; a single datum satisfying the relation $(exp_1 > exp_2)$ is not sufficient to distinguish $(EXP_1 < EXP_2)$ from $(EXP_1 = EXP_2)$; data for which $(exp_1 < exp_2)$ must also

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

Table 8: Origination Condition Set for Relational Operator Fault

be selected to guarantee origination. If $(exp_1 < exp_2)$ is also infeasible, then $(EXP_1 < EXP_2)$ and $(EXP_1 = EXP_2)$ are equivalent.

4.1.5 Origination of an Arithmetic Operator Fault

A potential error may result when an arithmetic operator is mistakenly replaced by another arithmetic operator. We consider six arithmetic operators: $+, -, *, **$ (real and integer operands), / (real division), and **div** (integer division), and we assume both operands to be of the same type. Given an arithmetic expression $(EXP_1 \text{ aop } EXP_2)$, if the arithmetic operator is faulty, then the correct expression must be in the alternate set $\{(EXP_1 \overline{\text{aop}} EXP_2) \mid \overline{\text{aop}}$ is an arithmetic operator other than aop that is type-compatible with EXP_1 and $EXP_2\}$. Each origination condition depends on the original and alternative arithmetic operators.

The general form of an origination condition for an alternate is $[(exp_1 + exp_2) \neq (exp_1 \overline{\text{aop}} exp_2)]$. For some alternates, it is possible to determine a simpler origination condition by determining the conditions under which the alternate and the original expressions evaluate equivalently and complementing. For example, consider the origination condition to distinguish between the operator $+$ and an alternate operator $-$. The original expression $(EXP_1 + EXP_2)$ and the alternate expression

$(EXP_1 - EXP_2)$ are indistinguishable only when $(exp_2 = 0)$. Thus, the origination condition is the complement, $[exp_2 \neq 0]$. Origination conditions are not, however, always this easy to simplify, so we report here only the general origination conditions in Table 9.

operators	origination conditions
+, −	$[(exp_1+exp_2) \neq (exp_1-exp_2)]$
+, *	$[(exp_1+exp_2) \neq (exp_1*exp_2)]$
+, /	$[(exp_1+exp_2) \neq (exp_1/exp_2)]$
+, div	$[(exp_1+exp_2) \neq (exp_1 \textbf{ div } exp_2)]$
+, **	$[(exp_1+exp_2) \neq (exp_1{**}exp_2)]$
−, *	$[(exp_1-exp_2) \neq (exp_1*exp_2)]$
−, /	$[(exp_1-exp_2) \neq (exp_1/exp_2)]$
−, div	$[(exp_1-exp_2) \neq (exp_1 \textbf{ div } exp_2)]$
−, **	$[(exp_1-exp_2) \neq (exp_1{**}exp_2)]$
*, /	$[(exp_1*exp_2) \neq (exp_1/exp_2)]$
*, div	$[(exp_1*exp_2) \neq (exp_1 \textbf{ div } exp_2)]$
*, **	$[(exp_1*exp_2) \neq (exp_1{**}exp_2)]$
/, **	$[(exp_1/exp_2) \neq (exp_1{**}exp_2)]$
**, div	$[(exp_1{**}exp_2) \neq (exp_1 \textbf{ div } exp_2)]$

Table 9: Origination Conditions for Arithmetic Operator Faults

The origination condition set for the class of arithmetic operator faults for a particular operator is the set of all origination conditions that distinguish that original operator from some alternate. The origination condition set for a faulty addition (+) operator, for example, is thus $\{[(exp_1+exp_2) \neq (exp_1-exp_2)], [(exp_1+exp_2) \neq (exp_1*exp_2)], [(exp_1+exp_2) \neq (exp_1/exp_2)$ or $(exp_1+exp_2) \neq (exp_1 \text{ div } exp_2)$ [9] $], [(exp_1+exp_2) \neq (exp_1{**}exp_2)]\}$. The origination condition sets for all arithmetic operator faults of a particular type are derived similarly.

[9]Only one of these conditions is applicable, depending upon the type of the operands.

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

Table 10: Origination Conditions for Arithmetic Operator Fault

4.2 Transfer Conditions

In this section, we develop the computational transfer conditions for four types of operators: assignment, boolean, relational, arithmetic. Transfer conditions are provided for both unary and binary operators of these types. The expression tree for any n-ary operator of these types is the binary translation of the n-ary tree derived using associativity rules.

A computational transfer condition guarantees that a potential error in an operand of an expression is not masked out by the computation of a parent operator. Thus, given an expression (EXP_1 op EXP_2), where a potential error exists in EXP_1, the transfer condition guarantees that (EXP_1 op EXP_2) also produces a potential error. More specifically, given EXP_1 containing a potential fault and $\overline{EXP_1}$ an alternate, the existence of a potential error in EXP_1 implies that $EXP_1 \neq \overline{EXP_1}$, and the transfer condition guarantees that $(EXP_1 \, \mathrm{op} \, EXP_2) \neq (\overline{EXP_1} \, \mathrm{op} \, EXP_2)$. The transfer condition is the complement of the conditions under which a potential error is not transferred.

4.2.1 Transfer through Assignment Operator

When each node contains only a single simple statement, the transfer condition through an assignment operator guarantee that $(V := EXP) \neq (V := \overline{EXP})$. This condition is trivial since for assignment $V := EXP$, any potential error produced by evaluation of EXP is reflected in the context after assignment to V. Thus, for this application, the transfer condition through an assignment operator, stated in table 11, is simply (*true*)

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

Table 11: Transfer Conditions for Assignment Operator

4.2.2 Transfer through Boolean Operator

For transfer through a boolean operator, we must consider both unary as well as binary boolean operators.

Consider first transfer through a unary boolean operator. The unary boolean transfer conditions guarantee that **not** (EXP_1) is distinguished from **not** $(\overline{EXP_1})$, where EXP_1 and $\overline{EXP_1}$ are distinguished. From Table 12, we see that no additional conditions are necessary for transfer of a potential error in a unary boolean expression because **not** $(exp_1) \neq$ **not** $(\overline{exp_1})$ if and only if $exp_1 \neq \overline{exp_1}$.

The binary boolean transfer conditions guarantee that an expression $(EXP_1$ **bop** $EXP_2)$ is distinguished from $(\overline{EXP_1}$ **bop** $EXP_2)$ and $(EXP_2$ **bop** $EXP_1)$ is distinguished from $(EXP_2$ **bop** $\overline{EXP_1})$, when 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 error in the right operand. The binary boolean transfer conditions depend upon the boolean operator. For the boolean operator **and**, we see from Table 12 that $(exp_1$ **and** $exp_2) \neq (\overline{exp_1}$ **and** $exp_2)$

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

Table 12: Boolean Expression Evaluation

only when $exp_2 = true$. Thus, exp_2 must be *true* to guarantee that a potential error in exp_1 transfers through the boolean operator **and**. For the boolean operator **or**, notice that (exp_1 **or** exp_2) $\neq$ ($\overline{exp_1}$ **or** exp_2) only when $exp_2 = false$. Hence, exp_2 must be *false* to guarantee transfer of the potential error in exp_1 through the boolean operator **or**.

The transfer conditions for boolean expressions are summarized in Table 13.

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 13: Transfer Conditions for Boolean Operators

4.2.3 Transfer through Relational Operators

The transfer condition for a relational operator guarantees that EXP_1**rop**EXP_2 is distinguished from $\overline{EXP_1}$ **rop** EXP_2 and that EXP_2 **rop** EXP_1 is distinguished from EXP_2 **rop** $\overline{EXP_1}$, when EXP_1 and $\overline{EXP_1}$ are distinguished. We need not actually develop the transfer conditions for the latter case separately, since for each **rop** the conditions that guarantee transfer of a potential error in the right operand are the same as those for transferring a potential error in the left operand through the complementary relational operator. For example, the conditions for distinguishing

$EXP_2 < EXP_1$ from $EXP_2 < \overline{EXP_1}$ are the same as those for distinguishing $EXP_1 \geq EXP_2$ from $\overline{EXP_1} \geq EXP_2$.

When the operands in a relational expression are boolean expressions, the only semantically-correct relational operators are = and $\neq$. Distinguishing EXP_1 and $\overline{EXP_1}$ implies that $exp_1 = \text{not}(\overline{exp_1})$. If $exp_1 = exp_2$, then $\overline{exp_1} \neq exp_2$, and if $exp_1 \neq exp_2$, then $\overline{exp_1} = exp_2$. Thus, no additional transfer conditions are necessary for a potential error in EXP_1.

When the operands in a relational expression are arithmetic expressions, a general representation for the transfer conditions is easy to write. Selection of test data to satisfy the conditions is difficult, however, because the transfer conditions through relational operators require more knowledge of the specific potential fault for which the potential error is transferred. The transfer conditions depend upon the relational operator through which the potential error must transfer. Let us consider the transfer condition for the relational operator <. We must determine when $(exp_1 < exp_2)$ is not equal to $(\overline{exp_1} < exp_2)$. This is the case when only one of exp_1 or $\overline{exp_1}$ is less than exp_2, which may be written as $(exp_1 < exp_2$ and $\overline{exp_1} \geq exp_2)$ or $(exp_1 \geq exp_2$ and $\overline{exp_1} < exp_2)$. The transfer conditions for the other relational operators are derived similarly; they are summarized in Table 14.

These transfer conditions, although necessary and sufficient to guarantee transfer of a potential error through a relational operator, require specific information about the value of the alternative expressions, and hence are very difficult to apply. These conditions have been investigated more extensively in [Zei86]. With limited knowledge about the relation between exp_1 and $\overline{exp_1}$, we can determine more specific conditions that are sufficient to guarantee that the potential error is transferred through a relational expression for that test datum. For example, consider transfer through the relational operator < in the expression $EXP_1 < EXP_2$. If it is known for a test datum that distinguishes between exp_1 and $\overline{exp_1}$ that $exp_1 < \overline{exp_1}$, if exp_1 is only slightly less than exp_2, then $\overline{exp_1}$ should be greater than or equal to exp_2. Then, for an originating test datum such that $exp_1 < \overline{exp_1}$, the additional condition that $(exp_2 - exp_1 = \epsilon)$, where ϵ is the smallest possible

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 14: Transfer Conditions for Relational Operators

positive difference between exp_1 and exp_2, will transfer a potential error that originates within EXP_1. This assumes that the smallest positive difference between exp_1 and exp_2 is no greater than the smallest positive difference between $\overline{exp_1}$ and exp_2. This condition $(exp_2 - exp_1 = \epsilon)$ is sufficient, but not necessary, to guarantee transfer of a potential error in EXP_1 through a $<$ operator under the assumption that $exp_1 < \overline{exp_1}$. A similar sufficient condition can be derived assuming that $exp_1 > \overline{exp_1}$. Sufficient transfer conditions through each relational operator are reported under each of these assumptions in table 15 [10] .

The conditions $exp_2 - exp_1 = \epsilon$, $exp_2 - exp_1 = -\epsilon$, $exp_2 - exp_1 = 0$, where ϵ is a small positive value, are often cited in the literature. Although not specifically cited as "transfer" conditions nor generally geared towards the concept of transfer, it is interesting to note that these are a generalization of the sufficient conditions in Table 15 when applied to any relational operator. If these "$\epsilon-$ conditions" are to be used to transfer a potential error through a relational expression,

[10]In table 15, 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 not be of the same magnitude.

operator	transfer conditions assuming $exp_1 < \overline{exp_1}$	transfer conditions assuming $exp_1 > \overline{exp_1}$
$<$	$exp_2 - exp_1 = \epsilon$	$exp_2 - exp_1 = -\epsilon$
$\leq$	$exp_2 - exp_1 = \epsilon$	$exp_2 - exp_1 = -\epsilon$
$=$	$exp_2 - exp_1 = 0$	$exp_2 - exp_1 = 0$
$\neq$	$exp_2 - exp_1 = 0$	$exp_2 - exp_1 = 0$
$>$	$exp_2 - exp_1 = -\epsilon$	$exp_2 - exp_1 = \epsilon$
$\geq$	$exp_2 - exp_1 = -\epsilon$	$exp_2 - exp_1 = \epsilon$

Table 15: Sufficient Transfer Conditions through Relational Operators

then three test data must be selected as follows:

1. $(exp_1 \neq \overline{exp_1})$ and $(exp_2 - exp_1 = \epsilon)$,
2. $(exp_1 \neq \overline{exp_1})$ and $(exp_2 - exp_1 = -\epsilon)$,
3. $(exp_1 \neq \overline{exp_1})$ and $(exp_2 - exp_1 = 0)$.

The advantage of combining these conditions is that their application is somewhat independent of the relation between exp_1 and $\overline{exp_1}$, simply because they require satisfaction of the sufficient condition for both relations $(<, >)$. These conditions are only sufficient to guarantee transfer of a potential error through a relational operator, however, under the assumption that the relation between exp_1 and $\overline{exp_1}$ is the same for each of the three test data selected to satisfy all three $\epsilon-$ conditions listed above. This assumption must hold in order to guarantee that one of the sufficient conditions in Table 15 is satisfied. 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. Furthermore, if any of these $\epsilon-$ conditions is infeasible, absence of a fault is not guaranteed by satisfaction of the remaining $\epsilon-$ conditions

The transfer conditions, which are introduced in Table 14, are both necessary and sufficient to guarantee transfer of a potential error through relational operators.

4.2.4 Transfer through Arithmetic Operators

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

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

For the arithmetic operators $*$ and $/$, $(exp_1 * exp_2 = \overline{exp_1} * exp_2)$ and $(exp_1/exp_2 = \overline{exp_1}/exp_2)$ and $(exp_2/exp_1 = exp_2/\overline{exp_1})$ only when $exp_2 = 0$. Thus to guarantee transfer through an outer multiplication or real division expression, exp_2 must not be 0.

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

The transfer conditions for arithmetic operators are summarized in Table 16.

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

Table 16: Transfer Conditions for Arithmetic Expressions

4.3 Revealing Conditions for Context Errors

In this section, we illustrate the formation of revealing conditions from the origination condition sets for the classes of potential faults and from the node transfer conditions for the expressions. Consider the module fragment and that portion of the control flow graph seen in Figure 3.

Any variable referenced in a module is potentially faulty. Suppose the reference to X in node 2 is faulty. Note that the module contains two other variables that are type-compatible with X. For this potential variable reference fault, the origination condition set is

$\{[x \neq y],$
$[x \neq z]\}.$

In general, each origination condition must be satisfied by a test datum that also satisfies transfer conditions through each ancestor operator. For a variable reference fault, in addition to the origination conditions described in 4.1.1, any of the transfer conditions presented in section 4.2 may be applicable. In this example, an "originating test datum" must also satisfy transfer conditions through the arithmetic operator $*$, the relational operator $<$, and the boolean operators and and or. The node transfer condition is thus

$(y \neq 0)$ *and* $((x * y < z$ and $\overline{x} * y \geq z)$ or $(x * y \geq z$ and $\overline{x} * y < z))$ *and* $(b = false)$ *and* $(c = true)$,

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

3      Z := 3
   else
4      Z := 2
   ⋮
```

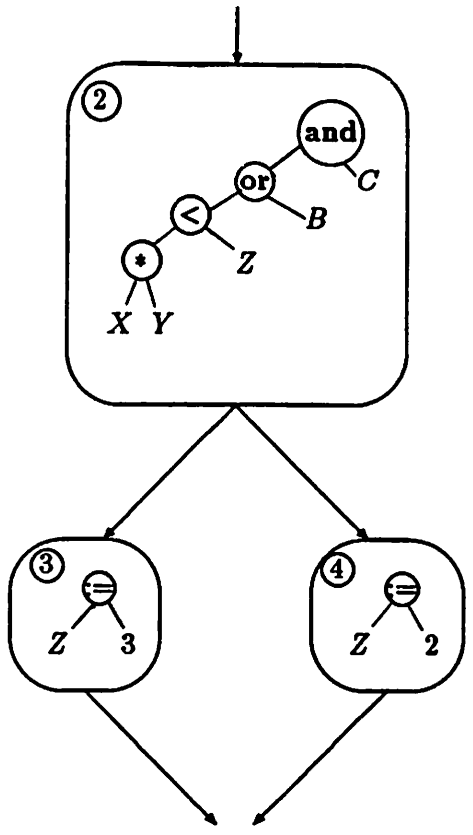

Figure 3: Module Fragment

where $\bar{x}$ represents the alternate variable reference — i.e., $\bar{x} \in \{y, z\}$. Combining the origination condition set with the node transfer condition results in the following revealing condition set

$$\{[(x \neq y) \textit{ and } (y \neq 0) \textit{ and } ((x * y<z \text{ and } y * y \geq z) \text{ or } (x * y \geq z \text{ and } y * y<z)) \textit{ and } (b = \textit{false}) \textit{ and } (c = \textit{true})],$$
$$[((x \neq z) \textit{ and } (y \neq 0) \textit{ and } ((x * y<z \text{ and } z * y \geq z) \text{ or } (x * y \geq z \text{ and } z * y<z)) \textit{ and } (b = \textit{false}) \textit{ and } (c = \textit{true})]\}.$$

Consider now the class of arithmetic operator faults. Suppose the multiplication operator in node 2 is potentially faulty. To reveal a context error for this potential fault, we must satisfy the origination condition set for an arithmetic operator fault for * as well as the node transfer condition. The origination condition set for the class of arithmetic operator faults for * in node 2 is

$$\{[(x*y) \neq (x+y)],$$
$$[(x*y) \neq (x-y)],$$
$$[(x*y) \neq (x \operatorname{div} y)],$$
$$[(x*y) \neq (x**y)])\}$$

In general, an arithmetic operator fault is contained within an arithmetic expression, which may be an operand of an arithmetic expression or a relational expression. The relational expression may then be part of a boolean expression. Hence, in addition to satisfying the origination conditions described in section 4.1.5, test data to detect this fault may also be required to satisfy any of the transfer conditions described in section 4.2, depending upon the structure of the entire node. The potential error originated in the expression $X*Y$ must transfer through the <, the **or**, and the **and** operators. The node transfer condition is

$$((x * y<z \text{ and } x\bar{\mp}y\geq z) \text{ or } (x * y\geq z \text{ and } x\bar{\mp}y<z)) \textit{ and } (b = \textit{false}) \textit{ and } (c = \textit{true}),$$

where $\bar{\mp}$ is any arithmetic operator other than $*$. The revealing condition set for a context error is thus

$$\begin{aligned}
\{&[(x*y \neq x+y) \textit{ and } ((x * y<z \text{ and } x+y \geq z) \text{ or } (x * y \geq z \text{ and } x+y < z)) \textit{ and} \\
&\quad (b = \textit{false}) \textit{ and } (c = \textit{true})], \\
&[(x*y \neq x-y) \textit{ and } ((x * y < z \text{ and } x-y \geq z) \text{ or } (x * y \geq z \text{ and } x-y < z)) \textit{ and} \\
&\quad (b = \textit{false}) \textit{ and } (c = \textit{true})], \\
&[(x*y \neq x \textbf{ div } y) \textit{ and } ((x * y < z \text{ and } x \textbf{ div } y \geq z) \text{ or } (x * y \geq z \text{ and } x \textbf{ div } y < z)) \textit{ and} \\
&\quad (b = \textit{false}) \textit{ and } (c = \textit{true})], \\
&[(x*y \neq x{**}y) \textit{ and } ((x * y < z \text{ and } x{**}y \geq z) \text{ or } (x * y \geq z \text{ and } x{**}y < z)) \textit{ and} \\
&\quad (b = \textit{false}) \textit{ and } (c = \textit{true})]\ \}
\end{aligned}$$

Let us now look at the formation of the revealing condition set for a relational operator fault. The origination condition set for the class of relational operator faults for < in node 2 is

$$\begin{aligned}
\{&[x * y = z], \\
&[x * y > z], \\
&[x * y \leq z], \\
&[x * y \neq z], \\
&[\ \textit{true}\ ]\}.
\end{aligned}$$

Since all three relations are feasible at node 2, a sufficient origination condition set is

$$\begin{aligned}
\{&[x * y = z], \\
&[x * y > z]\}.
\end{aligned}$$

A relational operator fault is contained within a relational expression, which may be part of a boolean expression. Hence, in addition to the origination conditions set described in 4.1.4, test data may also be required to satisfy transfer conditions through boolean operators as described in

4.2.1. A potential error resulting from the < in node 2 must transfer through the boolean operators **or** and **and**. The node transfer condition is simply

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

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

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

Consider now the class of boolean operator faults. The origination condition set for the potentially faulty boolean operator **or** is

$$\{[(x * y < z) \neq b]\}.$$

A boolean operator fault is contained within a boolean expression, which may be contained within a larger boolean expression or within a relational expression. Therefore, in addition to satisfying the origination conditions set described in 4.1.3, test data may also be required to satisfy transfer conditions through boolean operators as described in 4.2.1. However, no additional transfer conditions are required through relational operators. A potential error that originates from the **or** in node 2 must only transfer through the boolean operator **and**. The node transfer condition through this operator is

$(c = \mathit{true})$.

The revealing condition set for the potential faulty boolean operator **or** in this example is thus

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

The boolean operator **and** is also potentially faulty; the origination condition set is

$$\{[((x * y < z) or b) \neq c]\}.$$

There are no transfer conditions since the **and** is the outermost operator in the node. Thus, the revealing condition set for the potential faulty boolean operator **and** in this example is thus

$\{[((x * y < z) orb) \neq c]\}$.

Consider finally the class of variable definition faults. The origination condition set is

$\{[(\overline{v} \neq v)$ or $(exp \neq v) \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$.

For the potentially faulty variable definition of z in node 3, the origination condition set is

$\{[(x \neq z)$ or $(3 \neq z)]$,
$[(y \neq z)$ or $(3 \neq z)]\}$.

A variable definition fault occurs in the outermost expression of the node. Thus, there are no transfer conditions that must be satisfied, and any test datum that satisfies the origination condition set is guaranteed to reveal a context error for a potential variable definition fault.

We are now in a position to select a test data set that satisfies all the revealing conditions sets just developed. A test datum that satisfies a revealing condition must be selected within the domain of the module; further it must be selected such that the revealing conditions are satisfied before execution of the node. Because the node for which we have developed revealing conditions is one of the first nodes of the module, selection of test data that satisfies the conditions is relatively easy. There are many possible test data sets that satisfy the revealing conditions developed for this example. Table 17 shows just one such set.

	Input Values				
datum	x	y	z	b	c
1	1	2	3	*false*	*true*
2	1	2	1	*false*	*true*
3	1	3	2	*false*	*true*
4	1	2	2	*false*	*true*
5	1	2	1	*true*	*true*

Table 17: Sample Test Data Satisfying Revealing Conditions

First, consider the data selected for node 2. Datum 1 satisfies both revealing conditions in the revealing condition set for the potential faulty variable reference of X. For the potentially faulty arithmetic operator *, datum 1 satisfies the first revealing condition, datum 2 satisfies the second and third conditions, while datum 3 satisfies the fourth condition in the revealing condition set. For the potentially faulty relational operator <, datum 4 satisfies the first revealing condition, and datum 5 satisfies the second condition in the sufficient revealing condition set. Datum 1 satisfies the revealing condition set for the potentially faulty boolean operator **or**, while datum 2 satisfies the revealing condition set for the potentially faulty boolean operator **and**.

Next, consider the data selected for node 3, where the only potential fault in the chosen classes is the potentially faulty variable definition. Datum 2 satisfies the single revealing condition in the corresponding revealing condition set, but is not in the *DOMAIN* of the node. Datum 5, however, datum 5 satisfies the revealing condition and executes the node.

In this section, we have demonstrated how revealing condition sets for context errors are formed through evaluation of the origination and transfer conditions instantiated in sections 4.1 and 4.2 for the five fault classes.

5. Conclusion

In this paper, we define a formal model for error detection that is useful in light of the way most people test programs. The model proposes revealing conditions on test data selection that are necessary and sufficient to guarantee that a fault originates an error that transfers through computations and data flow until it is revealed. The paper applies the model to five classes of atomic faults. The model has also been used to analyze three test data selection techniques in terms of their capabilities to detect these five faults. The model has been applied elsewhere to six other fault classes and has provided similar results.

The underlying concepts of the model are generally applicable. Its further utility, however, depends on the granularity at which the model can be practically applied. We are examining the granularity of the origination conditions by looking at how alternate sets can be defined for more general classes of faults, such as non-atomic faults and faults that are not statement specific.

The model of error detection defined herein is directed toward the detection of context errors. We are currently extending the model to include data flow transfer conditions, which can be applied between nodes and within a node that consists of a group of statements. This will provide the necessary and sufficient conditions to guarantee revealing an output error.

REFERENCES

[Bal69] Robert M. Balzer. Exdams — extendable debugging and monitoring system. *1969 Spring Joint Computer Conference, AFIPS*, 34, 1969.

[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.

[Fai75] Richard Fairley. An experimental program-testing facility. *IEEE Transactions on Software Engineering*, SE-1(4), December 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.

[IEE83] *IEEE Standard Glossary of Software Engineering Terminology, Standard 729-1983.* Software Engineering Technical Committee of the IEEE Computer Society, 1983.

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

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

[RH75] C. V. Ramamoorthy and S. F. Ho. Testing large software with automated software evaluation systems. *IEEE Transactions on Software Engineering*, SE-1(1):46–58, March 1975.

[RT86a] Debra J. Richardson and Margaret C. Thompson. *A Comparison of Test Data Selection Criteria*. Technical Report 86-65, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[RT86b] 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.

[Stu73] Leon G. Stucki. Automatic generation of self-metric software. *Record of the 1973 IEEE Symposium on Software Reliability*, 1973.

[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.

[Zei86] Steven J. Zeil. *Domain Testing and Linear Fault Detection*. Technical Report 38, Computer and Information Science, University of Massachusetts, Amherst, August 1986.