An Investigation of Fault-Based Testing Using the Relay Model

Margaret C. Thompson
Ph.D. Dissertation
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 91-22
May 1991

An Investigation of Fault-Based Testing Using the Relay Model

A Dissertation Presented

by

Margaret C. Thompson

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

May 1991

Department of Computer and Information Science

An Investigation of Fault-Based Testing Using the Relay Model

A Dissertation Presented

by

MARGARET C. THOMPSON

Approved as to style and content by:

Lori A. Clarke

Lori A. Clarke, Co-Chair

Debra J. Richardson

Debra J. Richardson, Co-Chair

[illegible]

W. Richards Adrion, Member

George Avrunin

George S. Avrunin, Member

[illegible]

W. Richards Adrion, Department Chair
Computer and Information Science

ACKNOWLEDGEMENTS

I am indebted to Professor Debra Richardson, co-chair of my committee, for fast reads, fedex's and fac's, but mostly for being teacher, mentor, and friend. I look forward to our future relationship as colleagues. I am grateful to Professor Lori Clarke, co-chair of my committee, for her support and loyalty for many years. I am also grateful to my committee members Professors Rick Adrion and George Avrunin for reading my dissertation and offering valuable feedback.

I thank my office mates in the Software Development Laboratory for cookie runs, desk stickers, and generally tolerating my current obsession with having the door shut. Remember turning your key counter-clockwise locks the door and clockwise unlocks it.

I thank Renee Kumar for her incredible understanding of University and Graduate School rules and the ways through them, but more importantly for her sense of humor and thoughtfulness which have often helped me keep perspective on the world outside the lowrise.

There are many people outside of the University who have supported, encouraged and generally tolerated me through this process. It has been a privilege to train with the teachers and karate ka at Valley Women's Martial Arts, most especially my training buddies Laurel Turk, Sally Van Wright, and Rochelle Friedman. I thank them for providing me with the space to grow personally and for their love and support.

Crane Willemse has been honest friend who has taught me about acceptance of self and of others. Laurel Turk has for many years been a caring and supportive buddy, and for her willingness to challenge me, I am grateful. Marilyn Johnson has shared her family, life, sense of humor and love, all of which enrich my life immeasurably.

I thank Cheryl Muzio for explaining why there are 31 flavors at Baskin-Robbins and for pointing out the difference between needing room to struggle and needing room to fail and providing me with the former.

This dissertation would not have been possible without the unqualified love from the beans and bears in my life. They made life a little more bearable (beanable?) and a lot more hairy. Thanks to Purnell, Zang Bear, Angus Magilicutty, Zinfandel Bear, Zeta Bear, Lucca Bean, Buster Bean, Lillith, Della Dog and most especially Jazz Boleen. There's nothing like a cold nose after a long day at the zoo.

Abstract

An Investigation of Fault-Based Testing
Using the Relay Model

May 1991

Margaret C. Thompson, B.A., Smith College
M.S., University of Massachusetts
Ph.D. University of Massachusetts
Directed by: Professors Debra J. Richardson and Lori A. Clarke

Fault-based testing techniques attempt to select test data that detect certain types of faults that could exist in a module. This thesis presents a model, called RELAY, of how a fault causes a failure, where a fault is a syntactic mistake in the code and a failure is an observable incorrect behavior. A "potential failure" (intermediate incorrect value) "originates" (is introduced) and "transfers" (moves through the program) to output where a failure occurs. "Computational transfer" involves the transfer of a potential failure within a statement. "Data dependence transfer" involves the transfer of a potential failure from the definition of a variable to a use of that variable. "Control dependence transfer" involves the transfer of a potential failure from the incorrect evaluation of a branching statement to a statement whose execution may be controlled by that branching statement. A potential failure transfers from a faulty statement to output along "information flow chains". RELAY recognizes the possibility that several information flow chains may be transferred along at the same time and models this with "transfer sets", which are sets of chains all of which may be executed at the same time. Identification of transfer sets and recognition of the role of control dependence transfer are unique to RELAY.

This thesis presents three applications for the RELAY model.

1. We use the model to construct "failure conditions", that guarantee detection of a fault.
2. We use the insight provided by RELAY to evaluate fault-based testing criteria. Most criteria do not consider transfer of an originated potential failure to output. For those that do, no guidance for transfer is provided. No approach considers the complexity of transfer through data dependence and control dependence.
3. We use the details provided by RELAY to ask questions about the likelihood of transfer to output for test data that originates a potential failure and about the likelihood of multiple faults masking each other. We evaluate empirical studies and propose further empirical studies in both these areas using patterns of information flow as the basis for such studies.

Table of Contents

List of Tables

LIST OF FIGURES

Chapter 1

Introduction

A goal of testing a program is the detection of defects in the code. In general, exhaustive testing, in which a program is executed for all possible inputs of the program, is not possible. Thus, some subset of the input domain, or test data set, must be selected. A test data selection criterion is a set of rules that guides the tester in how to select a test data set. When a program produces incorrect output for some test case in the test data set, we know the program contains at least one mistake in the code. When a program produces correct output for all test cases in the test data set, we hope we have selected the test data so as to gain some confidence in the reliability of the program.

One group of test data selection criteria, known as fault-based testing, attempt to select test data that would detect a set of common faults that could be present in the code. To do this, these criteria attempt to cause a failure on execution of at least one test case in the test data set. By "fault" we mean a syntactic mistake in the code, and by "failure" we mean an observable incorrect behavior, most commonly incorrect output. This thesis investigates the fault detection capabilities of fault-based testing.

The major contributions of this thesis are:

- Description of a rigorous and complete model, called RELAY, that details how a fault causes a failure. This model assumes, as do fault-based testing criteria in general, that the faulty module closely resembles a correct version of the module and that multiple faults do not mask out each other;

- Application of the model to construct failure conditions that guarantee fault detection for hypothesized faults;

- Analysis of fault detection capabilities of fault-based testing criteria using the failure conditions;
- Examination of empirical studies using insight provided by RELAY.

A great deal of research has been performed in the area of fault-based testing. Much of this research is in the area of mutation testing [DLS79]. In mutation testing, faults are seeded one at a time into the source code to produce a "mutant" program. Test data is then evaluated as to whether original and mutant programs produce different output on at least one test case. The adequacy of a test data set is measured based on whether all non-equivalent mutants are distinguished. In the Portable Mutation Testing Suite [Bud83], several levels of mutation testing are included from weak to strong, depending on the extent of the effect of the fault on the module execution, e.g., mutated expression, mutated statement, entire module. The term "weak mutation testing" was introduced by Howden [How82] and is now part of his larger testing theory known as fault-based functional testing. Woodward and Halewood [WH88] introduce the idea of firm mutation testing as an intermediate step between weak and strong mutation testing.

Another area of fault based testing research has focused on the introduction of an incorrect program state by a fault. Howden's weak mutation testing [How82], Foster's ESTCA [Fos80, Fos83, Fos84, Fos85], Budd's Estimate from the Portable Mutation Testing Suite [Bud81, Bud83], Hamlet's testing with the aid of a compiler [Ham77], and Offutt's Constraint-Based Testing [Off88] all consider ways a statement could be faulty and attempt to select test data that distinguish between the (hypothetically) faulty statement and the (hypothetically) correct statement. Such test data would introduce an incorrect state at the faulty statement during execution if the module contained the hypothesized fault. Zeil's perturbation testing [Zei83, Zei84] takes a similar approach to evaluating a test data set. This approach examines ways in which statements in a program could be perturbed or modified and not introduce an incorrect state by the current set of test data.

Finally, there are several researchers that have considered the need for an incorrect state introduced by a fault to move through execution to effect output. In [Mor88], Morell provides a taxonomy that categorizes fault-based testing criteria according to the extent of the criteria – whether the criteria introduces an incorrect state at the statement containing the fault (local) or causes a failure (global). In his fault-based testing theory [Mor84], Morell defines a model whereby an incorrect state is created at the location of the fault and is propagated to output. In his dynamic symbolic fault-based testing [Mor84, Mor88] based on this theory, Morell determines what faults would be undetected by a test data set by comparing the symbolic computation for a path through a module and the symbolic computation for a path through the same module seeded with a symbolic fault at some location.

Thus, it is clear that fault-based testing is considered an important area of both past and current research. None of this work, however, provides a rigorous and detailed model of how a fault becomes a failure for some test case. The model presented in this thesis details this process and is motivated by an investigation of coincidental correctness, which occurs when a fault does not cause a failure on some test case even though the faulty code is executed. The phenomenon of coincidental correctness is very common. If it were not, then a test data set that executes all statements in a module at least once would be adequate to detect most faults.

The RELAY model extends and refines Morell's theory by more precisely defining the notion of where a potential failure (an incorrect intermediate value) is introduced and by identifying the ways a potential failure may move or "transfer" through a module. In particular, RELAY identifies three types of transfer: computational transfer, data dependence transfer, and control dependence transfer. A potential failure moves from a faulty statement to output along information flow chains, where an information flow chain is a sequence of statements such that each statement in the chain is either control dependent or data dependent on the previous statement in the chain. At each statement in the chain, computational transfer, data dependence transfer or control dependence transfer may be involved. RELAY recognizes the possibility that several information flow chains may be transferred along at the same time and models this with "transfer sets", which are sets of chains all of which may be executed at the same time. Identification of transfer sets and recognition of the role of control dependence transfer are unique to RELAY.

The basic approach of fault based testing is to hypothesize about the existence of a fault in a module and then attempt to select test data that would cause a failure to result if the hypothesized fault were indeed a fault. If we can do this, then correct execution on such test data suggests that the program does not contain that fault. The problem is to determine conditions that would have to be satisfied by test data to *guarantee* fault detection, where "guarantee" means that if the module contains a particular hypothesized fault and we execute the module on test data satisfying such conditions then a failure will result. We also know that if we execute the module on test data that satisfies conditions that guarantee detection and a failure does not result, the hypothesized fault is not a fault. This latter statement is simply the contrapositive of the former statement.

This thesis investigates what conditions would be necessary and sufficient to guarantee detection of a fault and uses those conditions to evaluate the fault detection capabilities of several fault-based testing criteria. Using the RELAY model of how a known fault may result in a failure, we develop "failure conditions" that would guarantee fault detection for a particular hypothesized fault. Failure conditions are constructed that guarantee that a fault introduces an incorrect state at the statement containing the fault and that the incorrect state transfers along some transfer set to output. To develop conditions that guarantee fault detection, we must consider how and where multiple incorrect references may "interact" to mask out the effects of the fault at a statement. To determine this, transfer sets identify at each node the references that reflect incorrect information.

The failure condition may be used to analyze the fault detection capabilities of fault-based testing criteria. Analysis of fault-based testing criteria using the RELAY model points out several weaknesses of these criteria. Most criteria do not consider transfer of an originated potential failure to output. The few that do, do not provide guidance as to how to achieve transfer, and no criteria fully considers the complexity of transfer through data dependence and control dependence. While the RELAY model does not provide the solution to the problems of testing, it does provide insight into the weaknesses of fault-based testing in general.

The RELAY model provides an analytical look at fault-based testing. Such an analysis indicates that the goal of guaranteeing fault detection is, in general, not realistic and suggests the need for empirical study of the problem. This thesis evaluates and analyzes several empirical studies reported in the literature and suggests areas for further study. This evaluation concentrates on two issues in particular. The first investigates how often transfer of an originated state potential failure to output occurs when data is not specifically selected to satisfy transfer conditions. The second investigates how often multiple faults in a module "interact" to mask out each other. With the detailed model of how a fault becomes a failure provided by RELAY, we are able to ask in-depth questions about the nature of transfer in real programs. We recommend further studies using structures of information flow as a basis for empirically investigating transfer.

This thesis is organized as follows.

Chapter 2 describes the terminology and notation. The terminology we use is common in the testing literature and is provided here to clarify our interpretation and use of the terms.

Chapter 3 defines the concepts of origination, transfer, transfer sets and transfer routes and presents the RELAY model of faults and failures that incorporates these components. In addition, this chapter overviews the construction of failure conditions using this model. The construction of the failure condition is divided into two parts. First, we develop the original state potential failure condition, which guarantees introduction of a potential failure at the statement containing the hypothesized fault. Second, we develop the transfer set condition, which guarantees transfer of the potential failure to output along a transfer set.

Chapter 4 presents formal definitions for the original state potential failure condition and demonstrates development of the original state potential failure condition for six classes of faults.

Chapter 5 motivates, describes and develops the transfer set condition.

Chapter 6 presents a survey of related works. The first section of the chapter provides an overview comparing RELAY to other testing research. The second section provides a detailed analysis of the ability of several fault-based testing criteria to introduce an incorrect state at the statement containing the hypothesized fault. The third section discusses how several fault-based testing criteria perform on transferring to output and points out how RELAY differs from these criteria.

Chapter 7 surveys several empirical studies from the literature related to RELAY and suggests additional studies. 34 Chapter 8 summarizes the major contributions of this thesis and discusses future research directions.

Chapter 2

Terminology

In this chapter, we present some general terminology and notation needed to present our work. The terms presented here are from the general literature and are not specific to the RELAY model.

2.1 Program Representation

We consider the analysis 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, $F_M : X_M \to Z_M$. Thus, for any $x \in X_M$, execution of M produces a vector $z = M(x) \in Z_M$. 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 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 pair (N, E), where N is a (finite) set of nodes and $E \subseteq N \times N$ is the set of edges. N contains two 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, and n_{final}, the final node. Associated with n_{start} is the importation of parameter and global variable values from the external environment or calling module. Associated with n_{final} is the exportation of parameter and global values to the external environment or calling module. Each other node in N represents a simple statement or the predicate of a conditional statement in M . For each pair of distinct nodes m and n in N where control may pass directly from the statement represented by m to that represented by n there is an edge (m, n) in E. There is also an edge in E from n_{start} to the entry point of M and an edge from an exit point to n_{final}. 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.

A control flow graph defines the paths within a module. A *subpath* in a control flow graph $G_M = (N, E)$ is a finite, possibly empty, sequence of nodes $p = (n_1, n_2, \ldots, n_{|p|})$ [1] such that for all i, $1 \leq i < |p|$, $(n_i, n_{i+1}) \in E$. A subpath formed by the concatenation of two subpaths p_1 and p_2 is denoted $p_1 \cdot p_2$. An *initial subpath* p is a subpath whose first node is the start node, n_{start}. A *path* P is an initial subpath whose last node is the final node, n_{final} [2] .

A node with more than a single successor node is called a *branching node*. A *loop* in a control flow graph G_M, is a subgraph of G_M corresponding to a looping construct in module M. An entry node of a loop L is a node n in L such that there is an edge (m, n) in G_M where m is not in L. An exit node for a loop L is a node n outside L such that there is an edge (m, n) in G_M, where m is in L. We assume that all loops have single entry and single exit nodes. An *iteration* of a loop L is a subpath within L that begins with the entry node of L, does not return to that node, and ends with a predecessor of either the entry node or the exit node of L.

Each node in a control flow graph may be represented as an abstract syntax tree, where the leaf nodes represent data objects and the internal nodes represent

[1] We denote the length of (the number of elements in) a sequence s by $|s|$

[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).

computational operators. This computation tree describes the statement's hierarchical structure. A subexpression of the statement represented by a node is a subtree of the node's abstract syntax tree. To denote the source code or syntactic subexpression of a node, upper case, e.g., EXP, is used. Lower case, e.g., *exp*, denotes the evaluated expression. If EXP is part of a node that is within a loop, it may be necessary to disambiguate the visit of EXP. The notation exp_i denotes the value of expression EXP during the i^{th} iteration of the loop that includes the node containing EXP[3]. The expression for any n-ary operator *op* may be represented as $op(operand_1, operand_2, ..., operand_n)$. This notation is used when n is unknown. For convenience, when n is known, an expression is written in its in-order form e.g., $operand_1\ op\ operand_2$.

An initial subpath p may be executed on some input x; this execution is denoted $p(x)$. Associated with such execution is a *state* $S_{p(x)}$ that contains the values of all variables after execution of $p(x)$ and the value of the branch predicate for the edge selected on evaluation of the last node in p. This value is stored in a dummy variable called BP. A state is also defined for entry to and exit from a module M. $S_{n_{start}(x)}$ is the state 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; $S_{P(x)}$ is the state 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. Some variables, including BP, may be undefined in any state, including exit from the module. This condition is noted in the state and causes no problem.

[3]For nested loop, this notation would be expanded to indicate the specific iteration of each loop in which the expression is nested.

2.2 Test Data and Testing Oracles

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, $t = [t_1, \ldots, t_m]$. Note that a test datum is distinct from an input vector in that an input vector includes input values for all required inputs for some path in G_M, whereas a test datum may be incomplete or invalid and not execute a path in G_M. For any node n in G_M, the set $DOMAIN(n)$ is the set of test data t for which n may be executed. The *test data domain* D_M for a module M is the domain of inputs from which test data can be selected. Note that D_M is not merely the domain of M since invalid input values are not in X_M. 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 D_M.

To reveal incorrect output by testing, there is usually some test oracle that specifies correct execution of the module [Wey82, How78b]. A test oracle might be a functional representation, formal specification, a 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, $O = \{(x, z)\} \subseteq X_O \times Z_O$, where X_O and Z_O are the domain and range, respectively, of the oracle. Note that an oracle is a relation; thus for any input, an oracle may specify more than one acceptable output. This allows for non-determinism 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 pairs (x, z) for all z.

The standard oracle just defined is in terms of input and output. We may also be interested in a module's behavior on partial execution. This intermediate behavior can be represented with an oracle that includes information about intermediate values, including that for BP, that should be computed by the module — we call this a *state oracle* since it defines the acceptable state(s) for a module's partial executions. A state oracle O_S is a relation $O_S = \{((t, p), S_{p(t)})\}$, that relates a test datum and an initial subpath (t, p) to one or more states $S_{p(t)}$ that are acceptable after execution of p on t. As with the standard input-output oracle, a state oracle is a relation, again to allow for the specification of more than one correct state for a particular test datum and initial subpath pair. A state oracle may derive its intermediate information from some correct module, an axiomatic specification, run-time traces, or monitoring of assertions [How78b].

2.3 Information Flow and Dependence Relationships

RELAY uses information derived from program dependences. Program dependences are syntactic relationships between nodes. Program dependences capture potential flow of information between nodes and include both control flow and data flow information. The definitions presented here are informal. For a more formal and graph theoretic definition and discussion of dependence see [Pod89].

Information may "flow" from the definition of a variable at one node to a use of that definition at another. Let x be a variable in a module M. A *definition of* x is associated with each node n in G_M that represents a statement that can assign a value to x; this definition is denoted $def(x, n)$. A *use of* x is associated with each node n in G_M that represents a statement that can access the value of x; this use is denoted $use(x, n)$.

A *def-use graph* is a control flow graph annotated at each node with information about the definition and use of variables. Associated with each node n is the set *define(n)*, which is the set of all variables to which a value may be assigned by the statement represented by the node, and the set *used(n)*, which is the set of all variables whose value is referenced by the statement represented by the node. In this thesis, to simplify our definitions and discussion we assume at each node there is at most a single variable in *define(n)*.

The RELAY model is concerned not only with the definitions and uses of variables, but also with subpaths from nodes where a definition occurs to nodes where that definition is used. A *definition-clear subpath* with respect to a variable x is a subpath p such that for all nodes n in p, $x \notin define(n)$ [4] . A definition $def(x, n)$ *reaches* a use $use(x, m)$ if and only if there is a subpath $(n) \cdot p \cdot (m)$ such that p is definition-clear with respect to x. A node n_j is *(directly) data dependent* on a node n_i if and only if there is a variable V assigned a value at n_i that is used at n_j, and $def(V, n_i)$ reaches $use(V, n_j)$. Since this is the only data dependence relationship we use, we will refer to it as data dependence, where the direct is implicit.

Information may also flow through the control of execution of one node by another. The *immediate forward dominator* of a (branching) node b is the node where all subpaths leaving b first come together. A node n_j is *(indirectly strongly) control dependent* on n_i if there exists a subpath from n_i to n_j that does not include the immediate forward dominator of n_i. Intuitively, this relationship characterizes the nodes that constitute the "body" of a structured branching node. Since this is the only control dependence relationship we use, we will refer to it as control dependence, where the indirect and strong are implicit.

An *information flow chain* is a sequence of nodes such that each node in the chain is either control dependent or data dependent on the previous node in the chain. More formally, given a control flow graph $G_M = (N, E)$, an information flow chain A in the control flow graph is a sequence of tuples $(u(A)_1, d(A)_1, n(A)_1), \ldots, (u(A)_{|A|}, d(A)_{|A|}, n(A)_{|A|})$, where $|A|$ is the number of tuples in the chain and $\forall i,\ 1 \leq i \leq |A|$, $n(A)_i \in N$, $d(A)_i \in define(n(A)_i) \cup \{\text{'BP'}, \text{'}out\text{'}\}$, and $u(A)_i \in used(n(A)_i) \cup \{\text{'BP'}\}$, such that $\forall k, 1 < k \leq |A|$, $n(A)_k$ is either control dependent or data dependent on $n(A)_{k-1}$.

[4]For languages where a variable may be "undefined" at a node, X must not be undefined at any node n in p.

For branching nodes, where no variable is assigned a value, the symbol 'BP' is used in a tuple in place of the defined variable. For example, at a branching node n that represents the statement *if* $X < 5$, we would have the tuple (X, BP, n). For a tuple that represents a link of control dependence, the symbol BP is used in the tuple in place of the used variable. For example, at a node n that represents the statement $X := A * B$ and that is control dependent on some other node, we would have the tuple (BP, X, n). For output nodes where a value may be communicated to the external environment, the symbol *'out'* is used in a tuple in place of the defined variable. For example, at a node n that represents the statement *output* $X + 6$, we would have the tuple (X, out, n).

An information flow chain is executed by some path or set of paths. Execution of an information flow chain requires execution of the sequence of nodes and for any two consecutive tuples $(V_x, V_y, i), (V_y, V_z, j)$ in the chain where node j is data dependent on node i, execution of a subpath $i \cdot (p) \cdot j$ where p is def-clear with respect to V_y. The path condition of an information flow chain A, *Path-Condition*(A), is the necessary and sufficient condition to execute the chain. *Path-Condition*(A) defines the set of paths in the module that execute chain A, which is called the *(set of) covering subpaths* and is denoted *Paths(A)*. The set of nodes in an information flow chain is $Nodes(A)$. A node i is *syntactically dependent* on some node j if there exists an information flow chain that starts at node i and ends at node j.

With these terms defined, we are now in a position to present our model.

Chapter 3

The Relay Model

This chapter presents the RELAY model of faults and failures. RELAY is a detailed model of how a known fault causes a failure to occur on execution for some test datum. Given a fault in a module, as a minimal requirement to cause a failure, a test datum must execute the fault. Execution of the fault alone, however, is not necessarily sufficient to cause a failure. An incorrect module may produce correct output on some input even when that input executes the fault. The module appears correct, but just by coincidence of the test data selected. As previously noted, this phenomenon is known as coincidental correctness. While the phenomenon of coincidental correctness has been noted by many researchers, there has not been a detailed exploration of how and where it may occur. Such an explanation provides valuable insight into the steps involved in a fault causing a failure. In the first section of this chapter, we present several examples that demonstrate the different ways that coincidental correctness may occur.

The RELAY model of faults and failures is motivated by our investigation of coincidental correctness. Recall from the introduction that a fault is a syntactic defect in some code, a failure is an observable incorrect behavior, most commonly incorrect output, and a potential failure is an intermediate erroneous result (which may potentially lead to failure). For a fault to cause a failure on execution of some test datum, a potential failure must be introduced or "originate" at the faulty expression and move through or "transfer" through computations that occur in the course of execution of the module. Transfer to failure occurs along sets of information flow chains that may be executed together, called "transfer sets". Transfer along transfer sets includes data dependence transfer and control dependence transfer. The second section of this chapter presents this RELAY model of faults and failures.

One application of the RELAY model is the construction of failure conditions that guarantee fault detection for a fault that could exist in the code. Section 3 outlines the process of developing failure conditions. This process is expanded in Chapter 4 and Chapter 5.

It is important to note here two assumptions of the model, both of which are made by all fault-based testing criteria.

The first assumption is that the module being tested is "almost correct". This assumption is similar to the competent programmer hypothesis [ABD+79, BDLS78], which says that the module being tested differs from the correct module by some small set of faults. The faults we consider in this application are restricted to those that do not change the program schema.

A second assumption is that there is either a single fault in the module or that multiple faults do not mask each other. Two faults mask each other if test data that would have detected the faults (cause a failure) if they occurred alone in the module, fails to detect the faults (cause a failure) for the module containing both faults. The implication of this assumption is that we may consider faults one at a time. Multiple fault interaction and its relationship to the concept of coupling [DLS79] are discussed in Chapter 7.

3.1 Coincidental Correctness

Coincidental correctness occurs when the existence of a fault is masked by some computation(s) during the execution of the module on some test datum. This masking out of the fault may happen in the subexpression containing the fault or in different ways at subsequent points during execution.

To demonstrate the different ways coincidental correctness may occur, we consider several test cases for the module shown in Figure 3.1. Suppose that the module in Figure 3.1 contains a fault at node 2 and that node 2 should be

$$A := C * (B + 1).$$

That is, the second reference to 'C' should be a reference to the constant 1. This correct node, labeled $2'$, is shown in the figure next to the faulty node. Table 3.1 lists six test data along with partial execution traces. For each test datum, there are two lines in the table. The first line for a test datum records the variable values on execution of the module with the faulty reference, while the second line records the values for the correct module. The column headed by ref refers to the value of the reference – either c or 1. For each column, the node where the value is assigned is shown in parentheses.

Looking first at test datum 1, we see that $c = 1$. For this test datum, no potential failure is introduced. Coincidental correctness occurs here in the subexpression containing the fault, and no failure can result for this test datum. This is indeed the case when we examine the last column in the table. For both the faulty module (line 1) and the correct module (line 2), execution on test datum 1 yields identical results for output.

Consider now the second test datum. For this test datum, $c \neq 1$, and a potential failure is introduced for this fault on this test datum. The potential failure then transfers through the addition since $(b + c \neq b + 1)$, but is masked out by the multiplication by C, which has the value zero. For this test datum, coincidental correctness happens again within the faulty node, and thus no failure is produced.

For the third test datum, a potential failure is introduced for this fault, and it transfers through all the computations at the faulty node. After execution of node 2, A has an incorrect value. This is seen in the column labeled a, where for the third test datum, evaluation with the faulty reference yields the value 3 and for

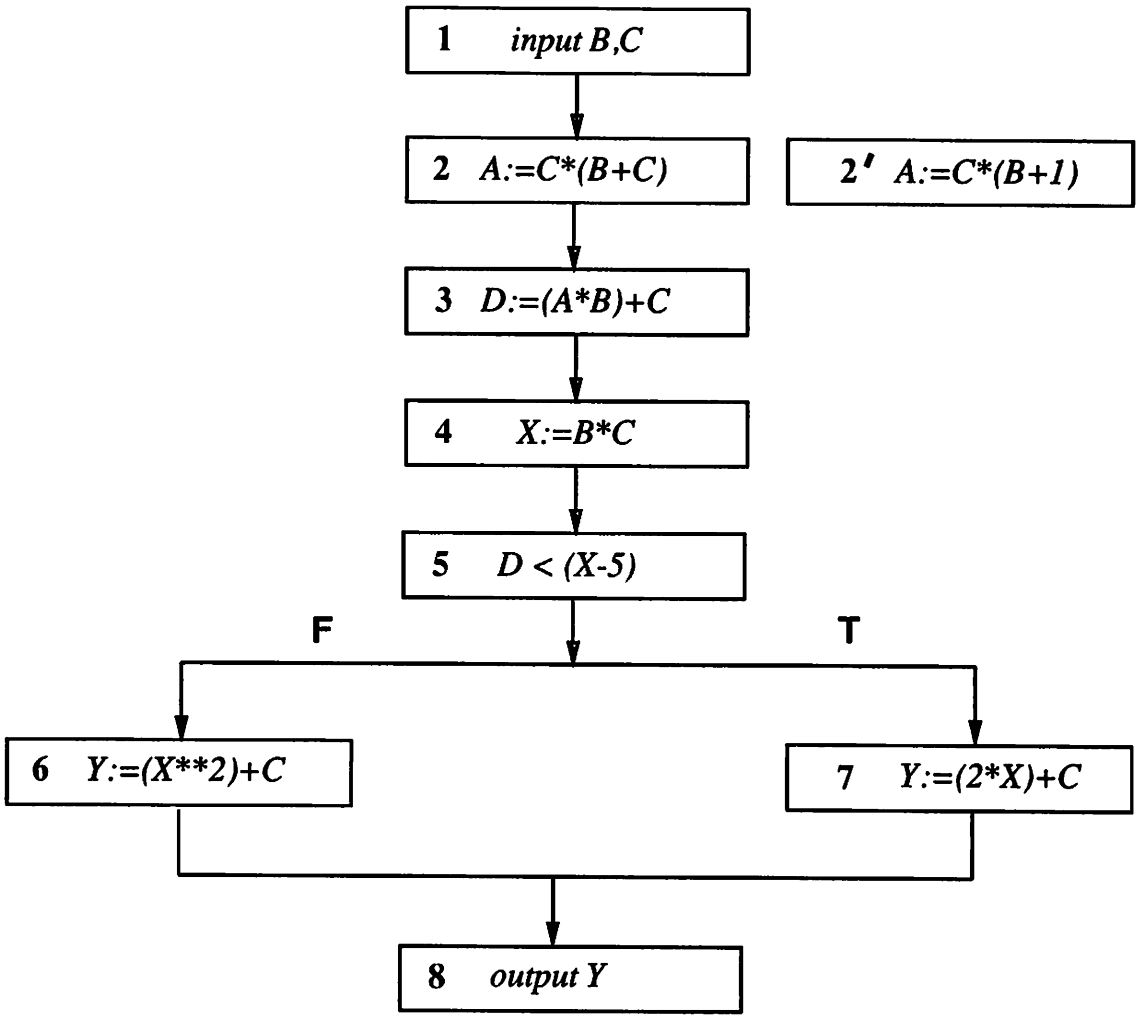

Figure 3.1. Coincidental Correctness Example 1

Table 3.1. Test Data Set For Coincidental Correctness Example 1

	module	t.d.		ref	*a*	*d*	*x*	$d < x - 5$	*y*	*output*
		b	c	c/1	(2)	(3)	(4)	(5)	(6/7)	(8)
1	faulty	1	1	1	2	3	1	F	2	2
	correct	1	1	1	2	3	1	F	2	2
2	faulty	1	0	0	0	0	0	F	0	0
	correct	1	0	1	0	0	0	F	0	0
3	faulty	0	3	3	9	3	0	F	3	3
	correct	0	3	1	3	3	0	F	3	3
4	faulty	2	3	3	15	33	6	F	39	39
	correct	2	3	1	9	21	6	F	39	39
5	faulty	-2	-1	-1	3	-7	2	T	3	3
	correct	-2	-1	1	1	-3	2	F	3	3
6	faulty	1	-3	-3	6	3	-3	F	6	6
	correct	1	-3	1	-6	-9	-3	T	-9	-9

the correct reference the value 1. If we continue to node 3 where A is referenced, we see however, that the incorrect value for A is masked out by multiplication by B. After node 3, D has the same value for both the correct and the incorrect node. A is the only variable that holds an incorrect value at this point and is not referenced at any subsequent nodes. Thus, on this test datum, a potential failure occurs at some intermediate points but is subsequently masked out in the execution of the module, and no failure is revealed for this fault on this test datum.

Execution on the fourth test datum introduces a potential failure and assigns an incorrect value to A. At node 3, where this incorrect value is referenced, the potential failure is not masked out by the computation and an incorrect value is assigned to D. This incorrect value is referenced at node 5; however, the condition $D < X - 5$ evaluates the same for both the incorrect and the correct values of D. As a result, the same branch is selected in the incorrect and the correct module. Since there is no subsequent use of either variable that holds an incorrect value, A or D, no failure can be revealed on this test datum for this fault.

The fifth test datum starts the same as the fourth test datum, with the introduction of a potential failure at node 2 that transfers through computations at node 3. For this test datum, the potential failure also transfers through evaluation of the conditional, which evaluates differently for the correct (F) and the incorrect (T) modules. Looking at the values computed for Y for this test datum along the two branches, however, we see that the same value (3) is computed for Y at node 7 and at node 6. Thus the fault is also masked out for this test datum.

Looking finally at the sixth test datum, a potential failure is introduced, and it transfers through nodes 2 and 3 and through node 5, where an incorrect branch is selected. Furthermore, for this test datum, Y is assigned a value by the incorrectly executing branch (node 6) that is distinct from the value that would have been assigned by the correctly selected branch (node 7); thus, the potential failure transfers at node 6. At node 8, a failure is revealed as reflected by different values output for Y. For this test datum, the potential failure introduced at node 2 is not masked out by any computations in the course of execution, and coincidental correctness does not occur.

As just seen in the example, a potential failure transfers when a computation references a variable that has an incorrect value and the resulting value is incorrect, or a potential failure transfers when an incorrect node is selected, which results in incorrect computations being evaluated, and an incorrect value is assigned to a variable at that node.

In all the sample test data for this example, there is at most a single variable that has an incorrect value referenced at a node or at most one incorrect computation that references no incorrectly valued variables. For this test data, the ways that coincidental correctness occurs are fairly obvious. Coincidental correctness can also occur in more subtle and complicated ways. For example, it is possible for two or more variables that reflect a potential failure to be referenced at the same node and 'interact' in such a way as to mask the potential failure. It is also possible for an incorrect node to be selected, and for that node to reference incorrect values. The incorrectly selected node referencing incorrect variable values may assign the same value(s) as the correctly selected node would with the correct variable values, causing the effects of the fault to be masked out.

These more complicated ways a fault may be masked out on a test datum are illustrated with the example shown in Figure 3.2. Suppose that the module shown in Figure 3.2 is incorrect, and suppose that node 2 should be:

$$X := A * B.$$

Accompanying test data is shown in Table 3.2. This table is slightly different from the previous table. As before, this table includes partial traces for the module with the correct and the incorrect node. In addition, we have included a column for both the computation at node 5 ($x * a$) and a column for the computation at node 6 ($x + b$). The entry in line 1 in the column labeled $x * a$ is in parentheses to indicate this would have been the value computed if this computation at node 5 had been performed. For example, for the first test datum, in the module with the faulty reference, node 6 is selected. The entry in line 2 in the column labeled $x + b$ is in parenthesis to indicated that this would have been the value computed if this computation at node 6 had been performed. For this test datum in the module with the correct reference, node 5 is selected.

For the first test datum in Table 3.2, a potential failure is introduced at node 2, where X is assigned the value 3 in the faulty module and assigned the value 2 in the correct module. The potential failure transfers through the computations at node 3, assigning an incorrect value to D. At node 4, an incorrect branch is selected; in the incorrect module, the true branch is selected, and in the correct module, the false branch is selected. At node 6, the incorrect computation is performed (X+B instead of X*A), and the value computed is distinct from that computed along the other

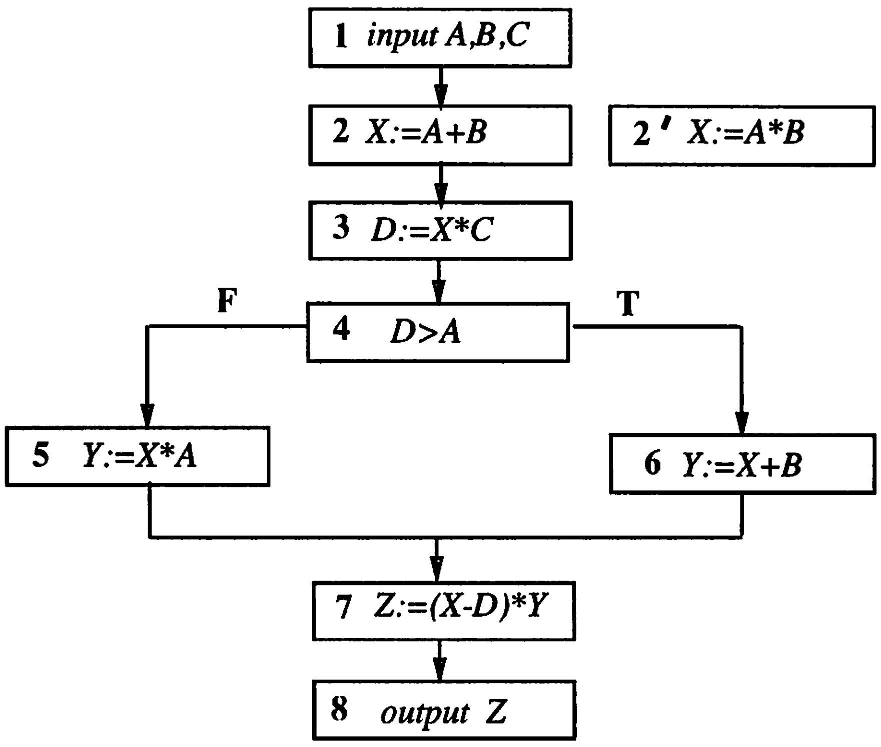

Figure 3.2. Coincidental Correctness Example 2

Table 3.2. Test Data Set for Coincidental Correctness Example 2

	module	t.d.			x	d	$d > a$	$x * a$	$x + b$	y	z	*output*
		a	b	c	(2)	(3)	(4)	(5)	(6)	(5/6)	(7)	(8)
1	faulty	2	1	1	3	3	T	(6)	4	4	4	4
	correct	2	1	1	2	2	F	4	(3)	4	4	4
2	faulty	1	2	-3	3	-9	F	6	(4)	6	162	162
	correct	1	2	-3	2	-6	F	4	(3)	4	48	48

branch — (x+b=4 and x*a=6); however, the incorrect computation references an incorrect value for X. For this test datum, a potential failure is not reflected in the value assigned to Y because the incorrect node (node 6) computes the same value (4) with the incorrect value of X as the correct node (node 5) would compute with the correct value of X. Y then is assigned the same value in both the correct and the incorrect module. At node 7, two variables reflect potential failures – X and D. At this node, coincidental correctness occurs, however, when X and D 'interact' and the same value is assigned to Z in both the correct and incorrect modules. No failure is revealed for this fault on this test datum.

For test datum 2, we see that a potential failure originates in X at node 2 and transfers to D at node 3. The potential failure in D fails to transfer at node 4 as $D > A$ evaluates to false for both the correct and the incorrect module. There are, however, other uses of variables that hold potential failures. At node 5, the potential failure in X transfers to Y, and at node 7 the potential failures in D, X, and Y transfer to Z. For this test datum, coincidental correctness does not occur, and a failure results, even though the potential failure has been masked by some computations.

As seen in the example, coincidental correctness can occur for some test data while not for others. It is also possible that a module contains a discrepancy that produces correct results for all test data that execute the discrepancy. In the first example module discussed, this would be the case if the domain of C were restricted to the values 0 and 1. When this happens, the 'faulty' module and the 'correct' module are equivalent and the discrepancy is not a fault.

This investigation of how a fault may not cause a failure provides the basis of a model of how a fault does cause a failure on execution of some test datum. Such a model of faults and failures is presented in the next section.

3.2 RELAY Model of Faults and Failures

This section presents the RELAY model of how a known fault is manifested as a failure. Recall that a failure occurs when execution of a module on some test datum causes an observable incorrect behavior, most commonly an incorrect output [1]. More precisely, we may define a failure as follows:

> **Definition:** Given a module M with $G_M = (N, E)$ and an oracle $O(X_O, Z_O)$, let $x \in X_M$. A **failure** occurs on execution of $M(X)$ when $(x, M(x)) \notin O(X_O, Z_O)$. The node where a failure occurs is termed a **failure node**.

A failure is caused by one or more faults in a module. A fault may be considered in terms of a transformation applied to some expression in the source code. Associated with a module M containing a fault at some node, there is a hypothetical, correct module M' that is identical to M except at the single node containing the fault.

> **Definition:** Given a module M and a hypothetical correct module M', a **fault** f is a transformation applied to some expression EXP in M such that $f(EXP') = EXP$, where EXP' is the corresponding expression in M', and execution of EXP causes a failure to occur on execution of some test datum.

As previously noted, the faults considered in this thesis are only those transformations that may be applied to or within a single node and that do not affect the program schema of the module (although the model is applicable on a larger scale). Thus, the control flow graph for a faulty module M is identical to the hypothetical correct module M' except in the annotation of the node containing the fault.

[1] We will often refer generally to output, meaning any observable incorrect behavior.

For a fault in a module M to cause a failure on execution of some test datum t, execution of the fault must introduce a potential failure that is not masked out during the course of execution of M to output. When examining the steps involved in the transfer of a potential failure, we will be interested in two types of incorrect intermediate computations – state potential failures and subexpression potential failures.

Sometimes we are interested in an incorrect state after evaluation of a node.

> **Definition:** Given a module M with a state oracle O_S, a **state potential failure** occurs after execution of initial subpath p on test datum t when $((t,p), S_{p(t)}) \notin O_S$.

That is, a state potential failure occurs after execution of p on t when at least one variable in the state (including the dummy variable BP) has an incorrect value.

> **Definition:** A **potential failure variable** is a variable in the state that has an incorrect value after execution of p on t.

While we do not define an oracle that tells us about the correct and incorrect evaluation of subexpressions, we will at times be interested in specifically referring to the incorrect evaluation of a subexpression of a node.

> **Definition:** A **subexpression potential failure** is reflected in an expression EXP in M when $exp \neq exp'$ on some test datum t, where EXP' is the corresponding subexpression in M'. When EXP is part of a node that is within a loop, the visit of EXP is disambiguated by subscripting it with the iteration count of the loop, e.g., $exp_i \neq exp'_i$.

With these terms defined, we may now consider the introduction and transfer of potential failures. Consider first the introduction of a subexpression potential failure by a fault. Some transformations affect pieces of code that considered alone cannot be evaluated. In the RELAY model, we consider introduction of a subexpression potential failure in the smallest subexpression that contains the transformed code. For example, an incorrect arithmetic operator is a transformation that substitutes one arithmetic operator for another. This transformation involves an operator token, which alone cannot be evaluated. In this case, the smallest subexpression containing the transformed piece of code is the expression containing just the operator involved and its operands. Other transformations such as incorrect variable reference, which substitutes a type-compatible variable for another, affect code that can be evaluated just by itself. For these types of transformations, the smallest subexpression containing the transformed code is just the involved code. Introduction of a subexpression potential failure is called origination.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetical, correct module M' with $G_{M'} = (N', E')$, let $n \in N$ contain a fault f such that $n' \in N'$ is the corresponding node in M'. Let $SEXP$ be the smallest subexpression of n containing f such that $f(SEXP') = SEXP$, where $SEXP'$ is the correct subexpression of n'. Let $t \in DOMAIN(n)$. Execution of M on t **originates** a subexpression potential failure in $SEXP$ for f if and only if $sexp \neq sexp'$. A node where a subexpression potential failure originates is call an **originating node**.

Consider the node shown below.

$$A := C * (B + C)$$

One fault that could exist in the node is an incorrect reference. As in one of the previous examples, suppose the second reference to C should be a reference to the constant 1. Then, in this example, the smallest subexpression that contains the fault is just 'C', and a subexpression potential failure originates when this subexpression is executed on a test datum such that $c \neq 1$.

Once a subexpression potential failure originates, it must transfer through all computations at the faulty node to cause a state potential failure, and from there, a state potential failure must transfer throughout execution of the module to some output. The RELAY model identifies three types of transfer: computational transfer, data dependence transfer, and control dependence transfer. Each of these is discussed below and in more detail in later chapters.

Within a node, a subexpression potential failure may be used as part of a larger subexpression of the node. To affect evaluation of the entire node, the subexpression potential failure must transfer through all operators in the node that have the subexpression potential failure nested in an operand. Transfer through computations within a node is called computational transfer.

Definition: Given a module M with $G_M = (N, E)$ and a hypothetical, correct module M', let $n \in N = op(...EXP...)$, $n' = op(...EXP'...)$ be the corresponding node in N' [2], and $exp \neq exp'$. The subexpression potential failure in EXP **computationally transfers** to the parent expression $op(...EXP...)$ if and only if $op(...exp...) \neq op(...exp'...)$.

Consider again the node

$$A := C * (B + C).$$

As before, suppose the reference to 'C' has evaluated incorrectly, originating a subexpression potential failure. The subexpression potential failure computationally transfers through the addition operation and the multiplication operation to affect evaluation of the entire node when $c * (b + c) \neq c * (b + c')$.

When a subexpression potential failure in a node computationally transfers at the node, the entire node evaluates incorrectly. Incorrect evaluation of a node results

[2] Note that only at the hypothetically faulty node where a subexpression potential failure originated are EXP and EXP' syntactically different.

in a state potential failure. If the node evaluating incorrectly represents a statement where a value may be defined for a variable, then at least one variable is assigned an incorrect value in the state and becomes a potential failure variable. If the node evaluating incorrectly is a branching node, then an incorrect branch is selected and the variable BP in the state is assigned an incorrect value.

Definition: The first state potential failure, which occurs when the node containing the fault evaluates incorrectly, is called the **original state potential failure**.

From the original state potential failure, the state potential failure must transfer to subsequent nodes, unless the node containing the fault is a node where a failure may be revealed. There are two types of transfer that involve movement of a state potential failure from node to node. These are data dependence transfer and control dependence transfer.

Recall from Chapter 2 the definition of (direct) data dependence. If n_j is data dependent on n_i, then there is some variable V defined at n_i and used at n_j, and there is a def-clear path with respect to V from n_i to n_j. Data dependence transfer occurs from a node n_i that defines a potential failure variable to a node n_j that uses the potential failure variable when the use of the potential failure variable causes the referencing node n_j to evaluate incorrectly. More formally:

Definition: Data dependence transfer occurs from a definition of V at node n_i to a use of V at node n_j when

1. a subpath $n_i \cdot p \cdot n_j$ is executed, and p is def-clear with respect to V;
2. the subexpression potential failure in V computationally transfers through all computations at n_j.

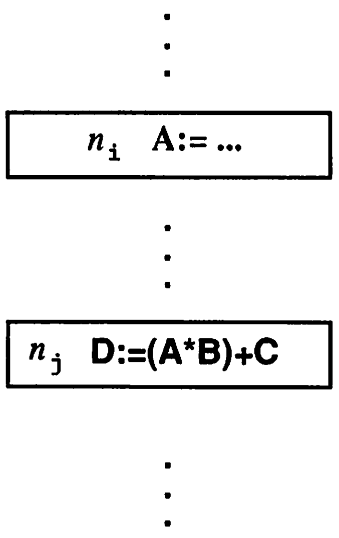

Figure 3.3. Data Dependence Example

Consider the small fragment of code shown in Figure 3.3 and assume there is a path from n_i to n_j that is def-clear with respect to A. Thus, n_j is data dependent on n_i. If A is assigned an incorrect value at n_i, then data dependence transfer occurs from the potential failure variable A at n_i to n_j when a def-clear path with respect to A from n_i to n_j is executed and the subexpression potential failure from the use of A at n_j computationally transfers through the multiplication by B and the addition to C to the assignment of D. When this happens, D becomes a potential failure variable, and the state potential failure transfers through n_j. Note also that A remains a potential failure variable, since it has not been redefined.

It is also possible that while there is not a data dependence relationship between two nodes, one node may affect evaluation of another by controlling when the latter is executed. One such relationship, as previously defined in Chapter 2, is (indirect, strong) control dependence.

Given a branching node n_i, if a state potential failure transfers through the condition, then an incorrect branch has been selected. If there is some node n_j that is control dependent on n_i and defines a value for some variable V, then control dependence transfer occurs when node n_j is incorrectly selected and assigns an incorrect value to V. More completely, we may define control dependence transfer as follows:

Definition: Control dependence transfer occurs from n_i to n_j when

1. n_j is control dependent on n_i;
2. n_i incorrectly selects n_j;
3. n_j computes a value for V that reaches the forward dominator of n_i and that is distinct from the value for V that would have reached the forward dominator if the correct branch had been selected in the correct module.

The incorrect value defined at n_j must reach the immediate forward dominator to insure it is not simply redefined and masked out within the selected branch. The value defined at n_j must compute a value for V that is distinct from that which would have reached the immediate forward dominator in the correct module to insure it can cause potential failures in subsequent computations in the module.

If n_i represents the conditional of an if-then statement, then transfer of the state potential failure through n_i means that the incorrect branch is selected, and transfer through n_j requires that the definition for the variable V at n_j reaches the immediate forward dominator and be distinct from the definition for V that reaches the forward dominator in the correct module if the correct branch had been taken. If n_i represents a control statement for a loop, then transfer of the potential failure through n_i means that an incorrect number of iterations is selected and transfer to n_j requires that the definition for the variable V at n_j for the last iteration be

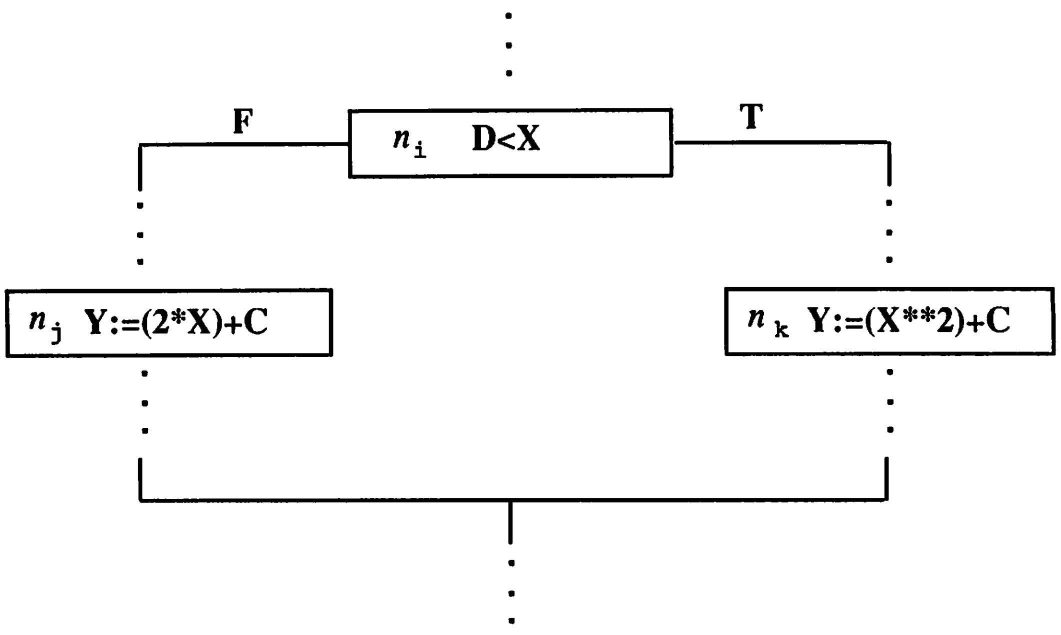

Figure 3.4. Control Dependence Example

distinct from that which would be computed with the correct number of iterations and correct variable values.

Consider the simple code fragment shown in Figure 3.4 and suppose the false branch to incorrectly be selected. (This means that in the hypothetical, correct module without the fault, wherever the fault may be in the module, execution on the same test datum would have caused the true branch to be selected). Control dependence transfer to the assignment of Y at n_j occurs when the computation for Y at n_j is not equal to what would be computed along the else branch at n_k. In this example, this occurs when $((2*x)+c) \neq ((x**2)+c)$, assuming X and C hold correct values at node n_j.

Let us return to the example figure shown in Figure 3.1 and see how the concepts of origination and transfer apply to execution of the module on the test data set in Table 3.1. Consider again the fault at node 2 of an incorrect reference to the second C that should be to the constant 1. The smallest subexpression containing the transformed code that can be evaluated is the affected code itself (e.g., the reference to C versus the reference to 1), and thus we consider origination within the reference to 'C'.

We see that for test datum 1, $c = 1$, and origination does not occur. Because no subexpression potential failure is introduced, we know that no failure can be be revealed for the fault upon execution of the module on this test datum.

For test datum 2, at node 2, $c \neq 1$, and a subexpression potential failure originates; however, it fails to computationally transfer at the node.

For test datum 3, a subexpression potential failure originates and also computationally transfers through the addition and the multiplication operations at the faulty node to affect evaluation of the entire node. At this point, there is an original state potential failure and a potential failure variable A. Node 3 is data dependent on node 2. For test datum 3, however, data dependence transfer does not occur. This is true because $D := (B * A) + C$ evaluates the same for both the faulty module and the correct module on this test datum.

For test datum 4, origination and computational transfer occur at node 2. Data dependence transfer occurs at node 3, where D becomes a potential failure variable. Node 5 is data dependent on node 3; however, data dependence transfer does not occur for this test datum at this node.

For test datum 5, origination and computational transfer occur at node 2. Data dependence transfer occurs at node 3 and at the conditional at node 5. For this test datum, however, control dependence transfer does not occur. This is seen by examination of the column labeled Y in Table 3.1, which shows that for both the faulty and the correct modules, the value 3 is assigned to Y.

Finally for test datum 6, origination and computational transfer occur at node 2, and data dependence transfer occurs through node 3 and then through node 5. At node 7, control dependence transfer occurs, and an incorrect value for Y is assigned. A failure is revealed at node 8.

Now that we have introduced the ideas of origination, computational transfer, data dependence transfer and control dependence transfer, let us see how these components fit together in the process of a fault causing a failure.

For a fault to cause a failure, a subexpression potential failure must originate and computationally transfer at the faulty node to cause an original state potential failure, and from this node, the state potential failure must transfer along some information flow chain(s) to output [3]. Transfer along an information flow chain involves data dependence transfer or control dependence transfer at each link in the chain. From any originating node to some output node, however, it is possible that a state potential failure may transfer along several information flow chains. We are interested in which of these chains may be executed together by the same test data. A transfer set captures these chains and is a set of informations chains such that

1. all chains start with a definition to the same variable at the same originating node;
2. all chains end with output of a value for the same variable at the same failure node;
3. there is a set of paths, each of which executes all the chains in the transfer set;
4. all chains executed by such a set of paths are included in the transfer set.

More formally, we may define a transfer set as follows:

Definition: Given a module M, with $G_M = (N, E)$, a **transfer set** is a set of information flow chains in G_M, $TS = \{A, B, ...\}$, with

A= $(u(A)_1, d(A)_1, n(A)_1), ..., (u(A)_{|A|}, d(A)_{|A|}, n(A)_{|A|})$

[3]Recall from Chapter 2 that an information flow chain is a sequence of nodes such that each node is control dependent or data dependent on the previous node in the chain.

B= $(u(B)_1, d(B)_1, n(B)_1), ..., (u(B)_{|B|}, d(B)_{|B|}, n(B)_{|B|})$

$\vdots$

where $u(A)_i$ is the variable used and $d(A)_i$ is the variable defined at node $n(A)_i$ in the ith tuple of chain A, $|A|$ represents the number of tuples in chain A, and the following properties hold:

1. $\forall$ chains $X, Y \in TS$,

   $(u(X)_1), d(X)_1, n(X)_1) = (u(Y)_1), d(Y)_1, n(Y)_1)$;

2. $\forall$ chains $X, Y \in TS$,

   $(u(X)_{|X|}), d(X)_{|X|}, n(X)_{|X|} = (u(Y)_{|Y|}), d(Y)_{|Y|}, n(Y)_{|Y|})$;

3. $\exists$ a subpath $p = (m_1, ..., m_{|p|})$, such that $m_1 = n(A)_1$ and $m_{|p|} = n(A)_{|A|}$, and $\forall X \in TS \;\; p \in paths(X)$;

4. $\nexists$ a chain $Q \notin TS$ such that

   (a) $p \in Paths(Q)$ and

   (b) $(u(Q)_1, d(Q)_1, n(Q)_1) = (u(A)_1, d(A)_1, n(A)_1)$ and

   (c) $(u(Q)_{|Q|}, d(Q)_{|Q|}, n(Q)_{|Q|}) = (u(A)_{|A|}, d(A)_{|A|}, n(A)_{|A|})$.

Intuitively then, a transfer set is a collection of information flow chains that can be executed by the same path(s). The first node in all chains of a transfer set is called *the transfer set's originating node*, while the last node in all chains of a transfer set is called the *transfer set's failure node*.

For the example shown in Figure 3.2, if we are considering a fault at node 2, there are six information flow chains to output at node 8. These are shown in Table 3.3 [4] . For these chains, we use the symbol $*$ to indicate a reference to a fault

[4]Recall that the symbol BP represents the defined variable in tuples for branching nodes where no variable is defined, e.g., $(D, BP, 4)$, that the symbol BP represents the used variable in tuples that represent a link of control dependence to the previous node, e.g., $(BP, Y, 5)$, and that the symbol *out* represents the defined variable in tuples where a value is output, e.g., $(Z, out, 8)$.

Table 3.3. Information Flow Chains for Coincidental Correctness Example 2

	chain
i	$(*, X, 2)(X, D, 3)(D, BP, 4)(BP, Y, 5)(Y, Z, 7)(Z, out, 8)$
ii	$(*, X, 2)(X, D, 3)(D, BP, 4)(BP, Y, 6)(Y, Z, 7)(Z, out, 8)$
iii	$(*, X, 2)(X, Y, 5)(Y, Z, 7)(Z, out, 8)$
iv	$(*, X, 2)(X, Y, 6)(Y, Z, 7)(Z, out, 8)$
v	$(*, X, 2)(X, D, 3)(D, Z, 7)(Z, out, 8)$
vi	$(*, X, 2)(X, Z, 7)(Z, out, 8)$

in defining the first variable in the chain, e.g., $(*, X, 2)$. For these six information flow chains there are two transfer sets. One transfer set consists of information flow chains (i,iii,v,vi), executed by test data that select the false branch. The other transfer set consists of chains (ii,iv,v,vi), executed by test data that select the true branch. We see from this example that the transfer sets are not necessarily disjoint. This property as well as several others is discussed in Chapter 5.

A transfer set defines the set of chains that may be executed together. It is possible, however, that while a test datum executes all chains in a transfer set, not all chains are transferred along. This happens when transfer fails at some node or nodes in the chains. Thus, while all chains are executed, potential failure information may not be transferring along all chains. A *transfer route* of a transfer set is a subset of the nodes in the transfer set where transfer occurs, such that

1. all nodes for at least one information flow chain in the transfer set are in the transfer route;

2. any node in the transfer route is part of a subchain from the originating node to that node where all nodes in the subchain are in the transfer route.

Define *pred*(n,C) to be the immediate predecessor of node n in chain C, i.e., for $C = (u_1, d_1, n_1)...(u_{i-1}, d_{i-1}, n_{i-1})(u_i, d_i, n_i)...(u_l, d_l, n_l)$, $pred(n_i, C) = n_{i-1}$. More formally, we may define a transfer route as follows:

Definition: Given a transfer set $TS = \{A, B, ...\}$, where $|TS|$ is the number of chains in the transfer set, let $Nodes(TS) = \bigcup_{C \in TS} Nodes(C)$. A **transfer route** of TS is $TR = \{t_1, t_1, ...t_j\} \subseteq Nodes(TS)$ such that the following properties hold:

1. $\exists$ an information flow chain $C \in TS$, such that $Nodes(C) \subseteq TR$;
2. $\forall t_i \in TR \;\; \exists \; C$ such that $pred(t_i, C) \in TR$

A node $n \in TR$ is called a **transferring node**. A node $n \in \{Nodes(TS) - TR\}$ is called a **non-transferring node**.

The set of non-transferring nodes are nodes where transfer does not occur. Transfer may not occur at a node because transfer has failed along all chains up to this node and thus the node references no potential failure variables, or transfer may not occur because while the node references potential failure variables, computational transfer fails at the node.

A transfer route is associated with a particular transfer set, and hence, also implicitly defines a set of non-transferring nodes. In addition, a transfer route associated with a transfer set implicitly defines the set of potential failure variables at each node. When describing a transfer route we will often explicitly specify some or all of this information to assist in our discussion.

There may be several transfer routes for a particular transfer set. For a particular fault at an originating node, execution of a particular test datum, however, executes at most one transfer route of a transfer set. At transferring nodes in the transfer route, data dependence transfer and/or control dependence transfer occurs. At nodes in the transfer route where transfer does not occur, either data dependence transfer and control dependence transfer fails or has failed at previous points such that the node references no potential failure variables.

Consider again the information flow chains shown in Table 3.3. For the transfer set consisting of the information flow chains (i,iii,v,vi),there are six different transfer routes. They are:

1. transfer from X to D at node 3 and transfer from D to BP at node 4 and transfer from (X and BP) to Y at node 5 and transfer from (X and D and Y) to Z at node 7 and transfer from Z to output at node 8;
2. transfer from X to D at node 3 and transfer from D to BP at node 4 and **do not** transfer from (X and BP) to Y at node 5 and transfer from (X and D) to Z at node 7 and transfer from Z to output at node 8;
3. transfer from X to D at node 3 and **do not** transfer from D to BP at node 4 and transfer from X to Y at node 5 and transfer from (X and D and Y) to Z at node 7 and transfer from Z to output at node 8;
4. transfer from X to D at node 3 and **do not** transfer from D to BP at node 4 and **do not** transfer from X to Y at node 5 and transfer from (X and D) to Z at node 7 and transfer from Z to output at node 8;
5. **do not** transfer from X to D at node 3 and transfer from X to Y at node 5 and transfer from (X and Y) to Z at node 7 and transfer from Z to output at node 8
6. **do not** transfer from X to D at node 3 and **do not** transfer from X to Y at node 5 and transfer from X to Z at node 7 and transfer from Z to output at node 8.

If we return again to the test data shown in Table 3.2 for Figure 3.2, we see that test datum 2 executes the third of the transfer routes enumerated above. For this test datum, data dependence transfer occurs at nodes 3,5,7, and 8 and fails at node 4.

Interaction occurs at a node when more than one of the variables referenced at the node is a potential failure variable. While a transfer set defines nodes where potential interaction occur, a transfer route defines nodes where actual interaction occurs. For the third transfer route, interaction occurs at node 7 where X, D, and Y are all potential failure variables. This interaction involves only data dependence links. Interaction may also occur with control dependence links and data dependence links in combination. This is the case for the second transfer route, where interaction occurs at node 5 between BP and X.

In summary, RELAY models the steps involved in a fault causing a failure on execution of some test datum as follows:

1. Introduction of an original state potential failure at the faulty node, which involves:
    (a) origination of a subexpression potential failure in the smallest expression containing the faulty code;
    (b) computational transfer at the faulty node;
2. Transfer of a state potential failure along a transfer route to output, which involves:
    (a) data dependence transfer and/or control dependence transfer at transferring nodes;
    (b) no transfer at non-transferring nodes.

There are several applications for the RELAY model. One such application is construction of the condition that guarantees fault detection for some hypothetical fault. Such an application is outlined in the next section and expanded in subsequent chapters.

3.3 Use of the RELAY Model to Develop Failure Conditions

The RELAY view of faults and failures describes how a particular fault manifests itself as a failure for a particular test datum. This view is dependent on knowledge of the faulty and the correct node. To develop a failure condition that guarantees detection of a fault, we hypothesize about the existence of a fault in some expression and ask the question "what if this expression should be that?". We then apply the RELAY view of how a known fault may cause a failure, to determine how to guarantee a hypothesized fault causes a failure, if indeed it is a fault. This section outlines our approach to deriving such conditions. Further details are provided in Chapters 4 and 5.

To derive the condition that guarantees fault detection, we generate the condition that guarantees origination of a potential failure that transfers until a failure is detected. This model uses the concepts of origination and transfer to define a necessary and sufficient condition to guarantee that a failure is revealed. Sufficient means that if the module is executed on data that satisfies the condition and the node is faulty, then a failure does occur. Necessary, on the other hand, means that if a failure does occur, then the module must have been executed on data that satisfies the condition and the node is faulty.

To begin, we consider an expression in a module being tested and hypothesize the existence of a fault in that expression. Such a hypothesized fault may be viewed, like any fault in a module, as a transformation applied to some subexpression. Associated with a hypothetical fault in a module is an alternate expression that would be correct if indeed the hypothetical fault were a fault. The module M with the hypothetically faulty expression replaced by the alternate forms a hypothetically correct module [5] .

[5]Note that in previous sections, we have referred to a hypothetical, correct module. There the

Definition: Given a module M with $G_M = (N, E)$ and an expression $SEXP$ in M, a **hypothetical fault** is a transformation f such that $f(SEXP') = SEXP$, where $SEXP'$ is the **hypothetically correct alternate**. The module M' is syntactically identical to M except M' contains the expression $SEXP'$, where M contains $SEXP$. M' is called the **hypothetically correct module.**

We develop the condition to detect a hypothetical fault, if indeed the hypothetical fault is a fault, by guaranteeing an original state potential failure would be introduced and would then be transferred along some transfer set to output. Turning first to the introduction of the original state potential failure, we define "origination conditions", "computational transfer conditions", and "original state potential failure conditions" that are necessary and sufficient to guarantee that origination, computational transfer or an original state potential failure, respectively, occur. The definitions provided here are informal; more formal definition are provided in Chapter 4.

The *origination condition* guarantees that if the hypothetically faulty node is executed, then the smallest subexpression containing the hypothetical fault introduces a subexpression potential failure. When the origination condition is infeasible, then the hypothetically faulty expression is equivalent to the alternate one, and no fault exists.

To reveal an original state potential failure, a subexpression potential failure originated at the smallest subexpression containing a hypothetical fault must transfer to effect evaluation of the entire node. A *computational transfer condition* guarantees that a subexpression potential failure in an operand transfers to a parent

existence of the module was hypothetical. Here, we refer to a hypothetically correct module where the correctness of the module is what is hypothetical. As with the hypothetical, correct module, the hypothetically correct module has an identical program schema to the module being tested.

expression. If the computational transfer condition is infeasible, then a subexpression potential failure cannot transfer to affect evaluation of the parent expression — thus, **op**($\ldots EXP \ldots$) is equivalent to **op**($\ldots EXP' \ldots$).

To introduce an original state potential failure, in addition to executing the node, a single test datum must satisfy both the origination condition and the computational transfer condition at the hypothetically faulty node. The *original state potential failure condition* for a hypothetical fault f occurring in $SEXP$ at n is the conjunction of the origination condition for f in $SEXP$ and the computational transfer condition for $SEXP$ in n.

To see how these ideas are applied, let us return to the example module shown in Figure 3.1. Examining node 2, one type of fault that we could hypothesize is an incorrect reference to the second C. We will hypothesize that perhaps the reference should be to the constant 1. The hypothetical fault contains just the expression C. The hypothetically correct alternate is the constant 1. The origination condition distinguishes between the hypothetically faulty subexpression and the hypothetically correct alternate. For the hypothetical fault at this location that we are considering, the origination condition is $c \neq 1$.

The originated subexpression potential failure must then computationally transfer through the addition and the multiplication operators. To transfer through the addition, a test datum must satisfy the condition $b + exp_1 \neq b + exp_1'$, where EXP_1 reflects a subexpression potential failure. This condition simplifies to *true*. To transfer through the multiplication a test datum must satisfy the condition $c * exp_2 \neq c * exp_2'$, where EXP_2 reflects a subexpression potential failure. This condition simplifies to $c \neq 0$. The computational transfer condition for the entire node is the conjunction of these two conditions and is simply $c \neq 0$. The original state potential failure condition is the conjunction of the origination condition and the computational transfer conditions at the node or $(c \neq 1) \wedge (c \neq 0)$.

The original state potential failure condition is defined for a hypothetical fault independent of where the hypothetically faulty node occurs in the module. The test data selected, however, must execute the node within the context of the entire module. Thus, for a hypothetical fault at node n, such test data are restricted to $DOMAIN(n)$. If the conditions are *infeasible* within $DOMAIN(n)$, then no original state potential failure can be introduced and the hypothetical fault is not a fault. In the example we were just considering, inability to select a test datum such that $c \neq 0$ and $c \neq 1$ at node 2 indicates that the reference to C and a reference to 1 are equivalent at this location.

In summary, the meaning of the original state potential failure condition is as follows:

- Execution of the node n on test data that satisfies the condition does not produce an original state potential failure implies that the hypothetical fault is not a fault for all test data that could execute the node (the hypothetically correct alternate then is not correct);

- Execution of the node n on test data that satisfies the condition produces an original state potential failure implies that the hypothetical fault may be a fault and requires additional conditions to transfer the state potential failure to output (as described next) to be satisfied;

- Inability to select test data that executes the hypothetical fault and satisfies the original state potential failure condition indicates that the module with the hypothesized fault and the hypothetically correct module are equivalent (the hypothetical fault is not a fault).

Chapter 4 discusses in more detail the development and application of the original state potential failure condition.

Once an original state potential failure has been introduced, if the node is not an output node, we know that either some variable has been assigned an incorrect value or an incorrect branch has been selected (if indeed the hypothetical fault is a fault). From this point, to guarantee a failure, the state potential failure must transfer along some information flow chain to output [6] . This is unfortunately much more difficult than perhaps it at first appears. This is because, as seen in the previous section, along a single path, a state potential failure may be transferring along several information flow chains. As a result, there may be nodes where multiple potential failures referenced at the node may mask each other.

To reveal a failure, after an original state potential failure has been introduced, a state potential failure must transfer along some transfer set. A *transfer set condition* is the necessary and sufficient condition that guarantees transfer of a state potential failure from the originating node, which is the first node in the transfer set, to the last node in the transfer set, which is a failure node. In order to develop necessary and sufficient conditions to transfer an original state potential failure to output, all potential failure variables at each node must be known. Thus, it is necessary to consider transfer sets when developing the failure condition rather than simply single information flow chains in isolation.

The transfer set condition consists of the *transfer set path condition,* which guarantees execution of all information flow chains in the set, and the disjunction of *transfer route conditions*, which guarantee a failure is revealed at the last node in the transfer set along some transfer route. Recall that a transfer route of a transfer set is a subset of the nodes in the transfer set and defines one way that potential failure information may transfer along the chains in the transfer set. It is necessary to consider combinations of nodes rather than simply considering combinations of

[6]Clearly, if the originating node is an output node, then the original state potential failure condition is also sufficient to cause a failure.

chains because the latter is not sufficient to identify all potential failure variables at each node. A transfer route condition is constructed using computational transfer conditions at the transferring nodes and the complement of computational transfer conditions at the non-transferring nodes.

Consider again the second transfer route for the example module in Figure 3.2, discussed in the previous section:

- transfer from X to D at node 3 and **do not** transfer from D to BP at node 4 and transfer from X to Y at node 5 and transfer from (X and D and Y) to Z at node 7 and transfer from Z to output at node 8.

To construct the transfer route condition, we conjoin the computational transfer condition from X to D at node 3 with the complement of the computational transfer condition from D to BP at node 4 with the computational transfer condition from X to Y at node 5 with the computational transfer condition from X and D and Y to Z at node 7 with the computational transfer condition from Z to output at node 8.

Using the components of the original state potential failure condition and a transfer set condition, we can derive a failure condition that guarantees detection of a fault along a particular transfer set. A *failure condition* for a hypothetical fault occurring in some selected subexpression in a node n along a selected transfer set is the conjunction of $DOMAIN(n)$, which guarantees execution of the node, the original state potential failure condition, which guarantees introduction of an original state potential failure, and the transfer set condition for the transfer set, which guarantees a state potential failure transfers along the transfer set to output. Let us summarize the meaning of this condition.

- Incorrect execution on a test datum that satisfies such a failure condition indicates that the module contains the fault;

- Correct execution on a test datum that satisfies such a failure condition implies that the module does not contain the hypothetical fault for all input;
- Inability to select test data that satisfy such a failure condition implies that another failure condition be derived for a different transfer set, and test data selected to satisfy that failure condition.

Note that we could have defined a failure condition for a transfer route rather than for a transfer set. We have chosen the latter because of the common path condition component of a transfer set condition for all transfer routes, and because transfer sets divide the subpaths in a module that go from an originating node to some failure node into disjoint sets. This property is discussed more in Chapter 5.

When we cannot select test data to satisfy a failure condition for a particular transfer set, we must consider and construct another failure condition for a different transfer set. All failure conditions for a particular hypothesized fault at a node will have $DOMAIN(n)$ and the original state potential failure condition in common. The failure conditions will differ only in the transfer set condition.

The *total failure condition* is the necessary and sufficient condition to reveal a failure for a particular hypothesized fault in a module. It is the disjunction of failure conditions for all transfer sets from an originating node to all output nodes. The meaning of this condition is as follows:

- Ability to satisfy this condition provides the same information as for a failure condition just discussed;
- Inability to satisfy the total failure condition means that we are unable to satisfy the failure condition for any transfer set from the originating node to a failure node. In this case, the module being tested is equivalent to the hypothetically correct module, and the hypothetical fault is not a fault.

In the next chapter, we demonstrate the construction of the original state potential failure condition. Chapter 5 presents the construction of the transfer route condition.

Chapter 4

Original State Potential Failure Condition

A failure condition guarantees fault detection along a selected transfer set and is composed of the original state potential failure condition and the transfer set condition. This chapter expands and formalizes the development of the original state potential failure condition.

The original state potential failure condition for a fault guarantees that execution of the faulty node would introduce an original state potential failure and is composed of the origination condition for the fault and the computational transfer condition required by the node. The origination condition guarantees introduction of a subexpression potential failure in the smallest expression containing the fault. The computational transfer condition at a node guarantees an originated subexpression potential failure effects evaluation of the entire node.

Section 1 presents a formal definition of the origination condition and describes the development of the origination condition for a hypothetical fault. The development of the origination condition is then demonstrated for six fault classes. A fault class is a set of faults that are grouped together according to the type of token to which the fault transformation is applied, e.g., arithmetic operator faults. These six classes have been selected because of their relevance to several other fault-based testing techniques.

Section 2 presents a formal definition of the computational transfer condition. When only a single operand of an expression reflects a subexpression potential failure, "simple" computational transfer conditions apply. At the originating node, only simple computational transfer conditions are required. Simple computational transfer conditions may be developed for individual operators. This section describes the development of the simple computational transfer condition for an operator and demonstrates this development for four operators – assignment, boolean, arithmetic, and relational. These operators are selected because they are the type of operator that may have one of the six fault classes, for which origination condition sets are developed, nested in an operand. Complex computational transfer condition, which apply at nodes where more than one referenced variable is a potential failure variable, are discussed in Chapter 5.

Section 3 demonstrates the application of the fault classes to a module and illustrates how the original state potential failure condition is formed by combining and evaluating the origination conditions for a fault class at a subexpression of a node and the applicable simple computational transfer conditions for the node.

4.1 Origination Conditions

As defined in Chapter 3, the origination condition guarantees that if the hypothetically faulty node is executed, then the smallest subexpression containing the hypothetical fault introduces a subexpression potential failure. More formally, we may define the origination condition as follows:

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically correct module M' with $G_{M'} = (N', E')$, let $n \in N$ contain a hypothetical fault f, and let n' be the corresponding node in N'. Let $SEXP$ be the smallest subexpression of n containing f, and $SEXP'$ be the hypothetically correct alternate expression of n' such that $f(SEXP') = SEXP$, then the **origination condition** is $oc(f, SEXP) = (sexp \neq sexp')$.

To determine the origination condition for a fault, we simply determine when the hypothetical fault and the hypothetically correct alternate are not equal. For example, for a variable reference fault where a reference to some variable V hypothetically should have been a reference to Y, the origination condition is simply $v \neq y$.

In the RELAY model, a fault is viewed as a transformation applied to some expression at a node. In the application presented here, each fault for which the origination condition is developed is an atomic fault, where a (hypothetical) fault f is *atomic* if the node containing the fault, 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. A class of (hypothetical) atomic faults is determined by the set of all semantically-correct replacements for the token. For example, the fault that exchanges an addition operator with a multiplication operator and the fault the exchanges an addition operator with a subtraction operator are examples of atomic faults in the class of incorrect addition operator. A class of faults defines a set of hypothetically correct alternates.

To develop the origination conditions for a class of faults, we first determine the alternate set for the class and then derive the origination condition for each alternate. The *origination condition set* contains the origination condition for each alternate in the set of alternates. Each origination condition must be both sufficient and necessary to guarantee origination of a subexpression potential failure for the corresponding alternate. To satisfy an origination condition set, a test data set must contain for each origination condition at least one test datum that satisfies the condition. In the subsections that follow, we develop the origination conditions for the following fault classes: variable reference fault, variable definition fault, arithmetic operator fault, boolean operator fault, relational operator fault.

Table 4.1. Origination Condition Set for Variable Reference Fault

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

4.1.1 Origination of a Variable Reference Fault

A fault may result when the name of a reference variable is mistakenly replaced by another valid variable name. Any variable reference is hypothetically 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 subexpression potential failure, since an expression could reference V for $\overline{V}$ and not originate a potential failure 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 4.1.

4.1.2 Origination of a Variable Definition Fault

A fault may result when the name of a defined variable is mistakenly replaced by another valid variable name. Given a definition of a variable $V := EXP$ [1], if V

[1]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).

is a faulty variable name, then the correct definition must be in the alternate set $\{[\overline{V} := EXP \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$.

The origination condition for an alternate $\overline{V}$ distinguishes between the assignments $V := EXP$ and $\overline{V} := EXP$. To distinguish these assignments and originate a subexpression potential failure, 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 subexpression potential failure, see Table 4.2, 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 node representing the assignment statement. For cases i,ii, and iii, the values of V, $\overline{V}$, and EXP satisfy the origination condition and evaluation of the expression originates a subexpression potential failure. In case i evaluation of the original expression $V := EXP$ results in $\overline{v} \neq v$, while evaluation of the alternate $\overline{V} = EXP$ results in $\overline{v} = v$ and vice versa for case iii. In cases ii and iii, evaluation of the expression $V := EXP$ results in $v \neq v$, while for $\overline{V} := EXP$, $v = v$ results. Thus, the origination condition is sufficient to originate a subexpression potential failure 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 expressions result in the same values for the variables; hence, no subexpression potential failure originates.

The origination condition set for a variable definition fault at the statement $V := EXP$ is stated in Table 4.3 — $\{[(\overline{v} \neq v)$ 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

Table 4.2. Variable Definition Evaluation

	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 4.3. Origination Condition Set for Variable Definition Fault

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

is satisfied, the origination condition set is satisfied. Thus, $(exp \neq v)$ is sufficient to guarantee origination of a subexpression potential failure 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 set $\{[\overline{v} \neq v \mid \overline{V}$ is a variable other than V that is type-compatible with $V]\}$ must be satisfied to guarantee origination of a subexpression potential failure for a variable definition fault.

4.1.3 Origination of a Boolean Operator Fault

A fault 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 hypothetically 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 $\{(\textbf{not bop}\,(EXP_1))\}$. 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 node containing the expression EXP_1 to guarantee origination of a subexpression potential failure for this particular fault.

Consider now a hypothetically faulty binary boolean operator. Given a boolean expression $(EXP_1\ \textbf{bop}\ EXP_2)$, 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 $(EXP_1\ \textbf{and}\ EXP_2)$ must be distinguished from $(EXP_1\ \textbf{or}\ EXP_2)$; 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 4.4 enumerates all possible cases for this expression, from which it is clear that this condition is both sufficient and necessary to originate a subexpression potential failure. In cases ii and iii, the origination condition is satisfied, and a subexpression potential failure 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 subexpression potential failure originates if and only if the origination condition $[exp_1 \neq exp_2]$ is satisfied.

The origination condition set, summarized in Table 4.5, contains the single condition that $[exp_1 \neq exp_2]$, which is satisfied when exactly one operand is true.

Table 4.4. Boolean Operator Evaluation

	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 4.5. Origination Condition Sets for Boolean Operator Fault

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

4.1.4 Origination of a Relational Operator Fault

A fault 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 $(EXP_1$ **rop** $EXP_2)$, if the relational operator **rop** is faulty, then the correct expression must be in the alternate set $\{(EXP_1\ \overline{\textbf{rop}}\ EXP_2) \mid \overline{\textbf{rop}}$ 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 4.6 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 $(EXP_1\ \overline{\textbf{rop}}\ EXP_2)$. The operator $=$ is one alternative operator; the origination condition that distinguishes $(EXP_1 < EXP_2)$ from $(EXP_1 = EXP_2)$ is

Table 4.6. Relational Operator Evaluation

	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*

$[(exp_1 < exp_2)$ or $(exp_1 = exp_2)]$. As seen from Table 4.6, 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 subexpression potential failure. 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 subexpression potential failure. The origination conditions for the other alternative operators are derived similarly. The origination conditions for relational operator faults are summarized in Table 4.7.

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 subexpression potential failure. When all test data relations are satisfiable, reduc-

Table 4.7. Origination Conditions for Individual Relational Operator Faults

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

Table 4.8. Origination Condition Sets for Relational Operator Fault

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

tion of the origination condition set for any particular relational operator results in two conditions that must be satisfied. These sufficient origination condition sets are summarized in Table 4.8. 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 node 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 be selected to guarantee origination.

4.1.5 Origination of an Arithmetic Operator Fault

A fault 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 $(EXP_1 \ \mathbf{aop}\ EXP_2)$, if the arithmetic operator is faulty, then the correct expression must be in the alternate set $\{(EXP_1\ \overline{\mathbf{aop}}\ EXP_2) \mid \overline{\mathbf{aop}}$ 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.

Table 4.9. Origination Conditions for Individual Arithmetic Operator Faults

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**	$[(exp_1 ** exp_2) \neq (exp_1 \textbf{ div } exp_2)]$

The general form of an origination condition for an alternate is $[(exp_1 \textbf{ aop } exp_2) \neq (exp_1 \overline{\textbf{aop}} exp_2)]$, where **aop** is the hypothetically faulty arithmetic operator. 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 4.9.

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 $\{[(exp_1 + exp_2) \neq (exp_1 - exp_2)], [(exp_1 + exp_2) \neq$

Table 4.10. Origination Condition Set for Arithmetic Operator Fault

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 $(exp_1+exp_2) \neq (exp_1 \textbf{ div } exp_2)]$, $[(exp_1+exp_2) \neq (exp_1**exp_2)]\}$.
$-$	$\{...\}$
$*$	$\{...\}$
$/$	$\{...\}$
$**$	$\{...\}$

$(exp_1*exp_2)]$, $[(exp_1+exp_2) \neq (exp_1/exp_2)$ or $(exp_1+exp_2) \neq (exp_1 \textbf{ div } exp_2)$ [2]], $[(exp_1+exp_2) \neq (exp_1**exp_2)]\}$. The origination condition sets for all arithmetic operator faults of a particular type are derived similarly.

4.2 Simple Computational Transfer Conditions

In order to effect evaluation of a faulty node, an originated subexpression potential failure must transfer through all computations in the node. As stated in Chapter 3, the *computational transfer condition* guarantees that a subexpression potential failure nested in an expression transfers to the parent expression. More formally:

Definition: Given a module M with $G_M = (N, E)$ and a hypothetically correct module M' with $G_{M'} = (N', E')$, let $n \in N$ and n' be the corresponding node in N'. Let **op**$(\ldots EXP \ldots)$ be a subexpression of n and **op**$(\ldots EXP' \ldots)$ be the corresponding subexpression of n' [3].

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

[3] Note again that only at the hypothetically faulty node where a subexpression potential failure originated are EXP and EXP' syntactically different.

Let t $\in$*DOMAIN*(n) and t $\in$*DOMAIN*(n') such that $exp \neq exp'$ for execution $M(t)$, then the **computational transfer condition** for n is **op**$(...exp...) \neq$ **op**$(...exp'...)$.

At an originating node, any operator in the node will have at most a single operand that is a subexpression potential failure. In cases where there is at most a single operand that is a subexpression potential failure, the computational transfer condition for the node is relatively simple to construct and is called a *simple computational transfer condition*.

To construct the simple computational transfer condition at a node, we may consider independently transfer through each operator that contains the subexpression potential failure in an operand. For example, given the expression $A := C*(B+C)$, to guarantee computational transfer from B to the assignment of A, we may develop the condition that guarantees transfer through the addition and then develop the condition that guarantees transfer through the multiplication. This decomposition simplifies the construction of computational transfer conditions for a node and allows the potential application of conditions that guarantee transfer through a particular operator to any node containing the operator. We term the condition that guarantees transfer through a particular operator the simple computational transfer condition for that operator.

To develop the simple computational transfer condition for an operator *op* for an expression $op(..., EXP_i, ...)$, where EXP_i reflects a subexpression potential failure, we determine when $op(..., exp_i, ...) = op(..., exp_i', ...)$. The simple computational transfer condition for the operator is simply the complement of this condition. The simple computational transfer condition for a node is the conjunction of the simple computational transfer condition for each operator that contains the subexpression potential failure in an operand.

Definition: Given a node

$$n = \mathbf{op_t}(\ldots, \mathbf{op_2}(\ldots, \mathbf{op_1}(\ldots, SEXP, \ldots), \ldots), \ldots)$$

the **simple computational transfer condition** for a subexpression potential failure in $SEXP$ is $ctc(SEXP, n) = \bigwedge_{1 \le i \le t} sctc_i$, where $sctc_i$ is the simple computational transfer condition for the operator $\mathbf{op_i}$.

As seen in Chapter 3, interaction may occur at a node when the node references more than one potential failure variable or when a node is incorrectly selected and references potential failure variable(s). At such nodes, we cannot decompose the expression at the node and apply the simple computational transfer condition for each operator in the node to construct the computational transfer condition for the entire node. For these cases, *complex computational transfer conditions* must be developed. These conditions are more dependent on the fault and the fault location than simple computational transfer conditions and are discussed in Chapter 5.

Development of the simple computational transfer conditions for several operators is demonstrated in the subsections that follow. We develop the simple computational transfer conditions for four types of operators – assignment, boolean, relational, and 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.

4.2.1 Simple Computational Transfer Condition for Assignment Operator

The simple computational transfer condition through an assignment operator guarantees that $(v := exp) \neq (v := exp')$. This condition is trivial since for assignment $V := EXP$, any subexpression potential failure produced by evaluation of EXP is reflected in the state after assignment to V. Thus, for this application,

Table 4.11. Simple Computational Transfer Condition for Assignment Operator

operator	expression	transfer condition
$:=$	$V := EXP \neq V := EXP'$	*true*

the simple computational transfer condition through an assignment operator, stated in Table 4.11, is simply (*true*).

4.2.2 Simple Computational Transfer Conditions for Boolean Operators

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 simple computational transfer condition guarantees that **not** (EXP_1) is distinguished from **not** (EXP_1'), where EXP_1 and EXP_1' are distinguished. From Table 4.12, we see that no additional conditions are necessary for transfer of a subexpression potential failure in a unary boolean expression because **not** $(exp_1) \neq$ **not** (exp_1') if and only if $exp_1 \neq exp_1'$.

The simple computational transfer conditions for binary boolean operators guarantee that an expression (EXP_1 **bop** EXP_2) is distinguished from (EXP_1' **bop** EXP_2) and (EXP_2 **bop** EXP_1) is distinguished from (EXP_2 **bop** EXP_1'), when EXP_1 and EXP_1' are distinguished. Since the binary boolean operators are commutative, we need not develop separately the transfer conditions for a subexpression potential failure in the right operand. The binary boolean simple computational transfer conditions depend upon the boolean operator. For the boolean operator **and**, we see from Table 4.12 that (exp_1 **and** exp_2) $\neq$ (exp_1' **and** exp_2) only when $exp_2 = true$. Thus, exp_2 must be *true* to guarantee that a subexpression potential failure in exp_1 transfers through the boolean operator **and**. For the boolean operator **or**, notice

Table 4.12. Boolean Expression Evaluation

exp_1	exp_1'	exp_2	exp_1 **and** exp_2	exp_1' **and** exp_2	exp_1 **or** exp_2	exp_1' **or** exp_2
true	*false*	*true*	*true*	*false*	*true*	*true*
true	*false*	*false*	*false*	*false*	*true*	*false*
false	*true*	*true*	*false*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*false*	*true*

Table 4.13. Simple Computational Transfer Conditions for Boolean Operators

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

that (exp_1 **or** exp_2) $\neq$ (exp_1' **or** exp_2) only when $exp_2 = false$. Hence, exp_2 must be *false* to guarantee transfer of the subexpression potential failure in exp_1 through the boolean operator **or**.

The simple computational transfer conditions for boolean expressions are summarized in Table 4.13.

4.2.3 Simple Computational Transfer Conditions for Relational Operators

The simple computational transfer conditions for a relational operator guarantee that EXP_1 **rop** EXP_2 is distinguished from EXP_1' **rop** EXP_2 and that EXP_2 **rop** EXP_1 is distinguished from EXP_2 **rop** EXP_1', when EXP_1 and EXP_1' are distinguished. We need not actually develop the simple computational transfer conditions for the latter case separately, since for each **rop** the conditions that guarantee transfer of a subexpression potential failure in the right operand are the same as those for transferring a subexpression potential failure in the left operand through the complementary relational operator. For example, the conditions for distinguishing $EXP_2<EXP_1$ from $EXP_2<EXP_1'$ are the same as those for distinguishing $EXP_1 \geq EXP_2$ from $EXP_1' \geq EXP_2$.

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

When the operands in a relational expression are arithmetic expressions, a general representation for the simple computational transfer conditions is easy to write. Selection of test data to satisfy the conditions is difficult, however, because the simple computational transfer conditions through relational operators require more knowledge of the specific hypothetical fault for which the subexpression potential failure is transferred. The simple computational transfer conditions depend upon the relational operator through which the subexpression potential failure must transfer. Let us consider the simple computational transfer condition for the relational operator $<$. We must determine when $(exp_1<exp_2)$ is not equal to $(exp_1'<exp_2)$. This is the case when only one of exp_1 or exp_1' is less than exp_2, which may be written as $(exp_1<exp_2$ and $exp_1' \geq exp_2)$ or $(exp_1 \geq exp_2$ and $exp_1'<exp_2)$. The simple computational transfer conditions for the other relational operators are derived similarly; they are summarized in Table 4.14.

These transfer conditions, although necessary and sufficient to guarantee transfer of a subexpression potential failure 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 exp_1', we can determine more specific conditions that are sufficient to guarantee that the

Table 4.14. Simple Computational Transfer Conditions for Relational Operators

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

subexpression potential failure is transferred through a relational expression for that test datum. For example, consider transfer of a subexpression potential failure 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 exp_1' that $exp_1<exp_1'$, if exp_1 is only slightly less than exp_2, then exp_1' should be greater than or equal to exp_2. Then, for a test datum such that $exp_1<exp_1'$, the additional condition that $(exp_2 - exp_1 = \epsilon)$, where ϵ is the smallest possible positive difference between exp_1 and exp_2, will transfer a subexpression potential failure 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 exp_1' and exp_2. This condition $(exp_2 - exp_1 = \epsilon)$ is sufficient but not necessary to guarantee transfer of a subexpression potential failure in EXP_1 through a $<$ operator under the assumption that $exp_1<exp_1'$. A similar sufficient condition can be derived assuming that $exp_1>exp_1'$.

Sufficient simple computational transfer conditions through each relational operator are reported under each of these assumptions in Table 4.15. In Table 4.15, ϵ is the smallest magnitude positive difference between exp_2 and exp_1 and $-\epsilon$ is the

Table 4.15. Sufficient Simple Computational Transfer Conditions for Relational Operators

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

smallest magnitude negative difference; note that $+\epsilon$ and $-\epsilon$ may not be of the same magnitude.

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 4.15 when applied to any relational operator. If these "$\epsilon-$ conditions" are to be used to transfer a potential failure through a relational expression, then three test data must be selected as follows:

1. $(exp_1 \neq exp_1')$ and $(exp_2 - exp_1 = \epsilon)$,
2. $(exp_1 \neq exp_1')$ and $(exp_2 - exp_1 = -\epsilon)$,
3. $(exp_1 \neq exp_1')$ 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 exp_1', simply because they require satisfaction of the sufficient condition for both relations $(<, >)$. These conditions are only sufficient to guarantee transfer of a subexpression potential failure through a relational operator, however, under the assumption that the relation between exp_1 and 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 4.15 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 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 simple computational transfer conditions, which are introduced in Table 4.14, are both necessary and sufficient to guarantee transfer of a subexpression potential failure through relational operators.

4.2.4 Simple Computational Transfer Conditions for Arithmetic Operators

The simple computational transfer conditions for arithmetic operators guarantee that EXP_1 **aop** EXP_2 is distinguished from EXP_1' **aop** EXP_2 or that EXP_2 **aop** EXP_1 is distinguished from EXP_2 **aop** EXP_1', when EXP_1 and 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 $exp_1\,\mathbf{aop}\,exp_2 = exp_1'\,\mathbf{aop}\,exp_2$. The transfer conditions derived here assume that both operands are of the same type, and that there is no round off error. Simple computational transfer conditions through the following arithmetic operators are considered: $+$, $-$, $*$, $/$ (real operands), and $**$.

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

For the arithmetic operators $*$ and $/$, $(exp_1 * exp_2 = exp_1' * exp_2)$ and $(exp_1 / exp_2 = exp_1' / exp_2)$ and $(exp_2 / exp_1 = exp_2 / exp_1')$ 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 EXP_1' are the base raised to the power EXP_2, we must examine when $exp_1 ** exp_2 = exp_1' ** exp_2$. This is true only when $(exp_2 = 0)$ or $(exp_1 = -exp_1'$ and exp_2 is even). Thus the simple computational transfer conditions for an exponential expression when the subexpression potential failure is contained in the base operand are $(exp_2 \neq 0)$ and $(exp_1 \neq -exp_1'$ or $exp_2 \mathbf{\ mod\ } 2 \neq 0)$. To determine the simple computational transfer conditions when the subexpression potential failure is within the exponent, we must examine the conditions where $exp_2 ** exp_1 = exp_2 ** exp_1'$. This is true when $(exp_2 = 0)$ or $(exp_2 = 1)$ or $(exp_2 = -1$ and $(exp_1, exp_1'$ are both even or both odd)). Thus, the simple computational transfer conditions are $(exp_2 \neq 0)$ and $(exp_2 \neq 1)$ and $(exp_2 \neq -1$ or $exp_1 \mathbf{\ mod\ } 2 \neq exp_1' \mathbf{\ mod\ } 2)$.

The transfer conditions for arithmetic operators are summarized in Table 4.16.

4.3 Original State Potential Failure Condition

The original state potential failure condition is the necessary and sufficient condition that guarantees an original state potential failure occurs at the hypothetically faulty node. More formally, the original state potential failure condition may be defined as follows:

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 be

Table 4.16. Simple Computational Transfer Conditions for Arithmetic Operators

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

a potential fault such that $f(n') = n$. Let $SEXP$ be the subexpression containing the hypothetical fault, where the node $n = EXP = \mathbf{op_t}(\ldots, \mathbf{op_2}(\ldots, \mathbf{op_1}(\ldots, SEXP, \ldots), \ldots), \ldots)$. The **original state potential failure condition** guarantees that $\forall t$ that satisfy the condition, $exp \neq exp'$. The original state potential failure condition for f in $SEXP$ at node n is $ospfc(f, SEXP, n) = oc(f, SEXP) \bigwedge ctc(SEXP, n)$.

In this section, we demonstrate with an example the construction of the original state potential failure condition sets for six classes of hypothetical faults, using the origination condition sets derived in Section 1 and the simple computational transfer conditions derived in Section 2. Consider the module fragment shown in Figure 4.1, and let us apply the six classes of faults to this module. Consider first the class of variable reference faults. Any variable referenced in a module is hypothetically faulty, for example, the reference to X at node 2. Note that the module contains two other variables that are type-compatible with X. For this hypothetical variable reference fault, the origination condition set is

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

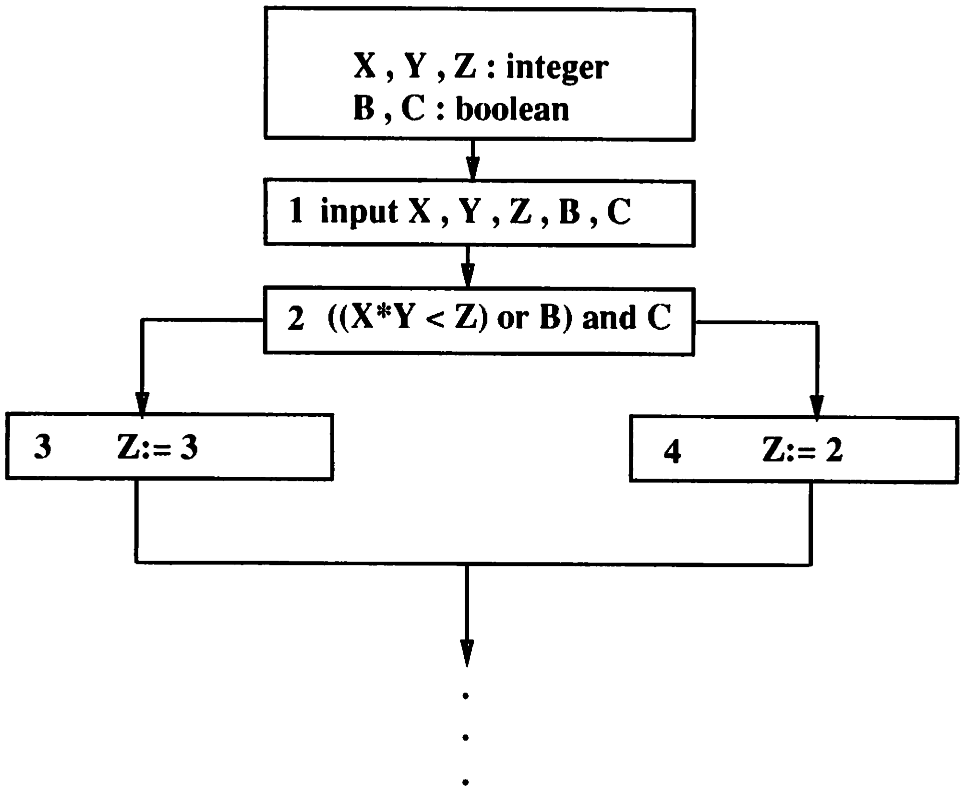

Figure 4.1. Module Fragment for Original State Potential Failure Condition

To introduce an original state potential failure, each origination condition must be satisfied by a test datum that also satisfies the simple computational transfer conditions for each operator that contains X in an operand. For a variable reference fault, in addition to the origination conditions described in Section 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 simple computational transfer conditions for the arithmetic operator $*$, the relational operator $<$, and the boolean operators **and** and **or**. The computational transfer condition for this entire node for this class of faults at this location is thus

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

where $\overline{X}$ represents the alternate variable reference — i.e., $\overline{X} \in \{Y, Z\}$. Combining the origination condition set with the computational transfer condition for the node results in the following original state potential failure 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. The multiplication operator at node 2 is hypothetically faulty. To introduce an original state potential failure for this hypothetical fault, we must satisfy the origination condition set for an arithmetic operator fault for $*$ as well as the simple computational transfer conditions at the node. The origination condition set for the class of arithmetic operator faults for $*$ at node 2 is

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

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 simple computational transfer conditions described in Section 4.2, depending upon the structure of the entire node. The subexpression potential failure originated in the expression $X*Y$ must transfer through the $<$, the **or**, and the **and** operators. The computational transfer condition for this class of hypothetical faults at this location is

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

where $\bar{*}$ is any arithmetic operator other than $*$. The original state potential failure condition set is

$\{[(x*y \neq x+y)\ \mathit{and}\ ((x*y<z \text{ and } x+y \geq z) \text{ or }$
$(x*y \geq z \text{ and } x+y < z))\ \mathit{and}\ (b = \mathit{false})\ \mathit{and}\ (c = \mathit{true})]$,
$[(x*y \neq x-y)\ \mathit{and}((x*y < z \text{ and } x-y \geq z) \text{ or }$
$(x*y \geq z \text{ and } x-y < z))\ \mathit{and}\ (b = \mathit{false})\ \mathit{and}\ (c = \mathit{true})]$,
$[(x*y \neq x \textbf{ div } y)\ \mathit{and}\ ((x*y < z \text{ and } x \textbf{ div } y \geq z) \text{ or }$
$(x*y \geq z \text{ and } x \textbf{ div } y < z))\ \mathit{and}\ (b = \mathit{false})\ \mathit{and}\ (c = \mathit{true})]$,
$[(x*y \neq x{**}y)\ \mathit{and}\ ((x*y < z \text{ and } x{**}y \geq z) \text{ or }$
$(x*y \geq z \text{ and } x{**}y < z))\ \mathit{and}\ (b = \mathit{false})\ \mathit{and}\ (c = \mathit{true})]\ \}$.

Let us now look at the formation of the original state potential failure condition set for a hypothetical relational operator fault. The origination condition set for the class of relational operator faults for $<$ at node 2 is

$\{[x*y = z],\ [x*y > z],\ [x*y \leq z]$,
$[x*y \neq z],\ [\ \mathit{true}\ ]\}$.

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

$$\{[x * y = z],\ [x * y > z]\}.$$

A relational operator fault is contained within a relational expression, which may be part of a boolean expression. Hence, in addition to the origination condition set described in Section 4.1.4, test data may also be required to satisfy simple computational transfer conditions for boolean operators as described in Section 4.2.1. A subexpression potential failure originated in the relational expression in node 2 must transfer through the boolean operators **or** and **and**. The computational transfer condition for this class of hypothetical faults at this location is simply

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

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

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

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 Section 4.1.3, test data may also be required to satisfy simple computational transfer conditions for boolean operators as described in Section 4.2.1. As noted in Section 4.2.3, no additional computational transfer conditions are required through relational operators with boolean operands. A subexpression potential failure that originates at the expression containing **or** at node 2 must only transfer through the boolean operator **and**. The simple computational transfer condition through this operator is

$(c = true)$.

The original state potential failure condition set for the hypothetically faulty boolean operator **or** in this example is

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

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

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

There is no computational transfer condition required since the **and** is the outermost operator at the node. Thus, the original state potential failure condition set for the hypothetically faulty boolean operator **and** in this example is

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

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

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

For the hypothetically faulty variable definition of Z at node 3, the origination condition set is

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

Table 4.17. Sample Test Data For Figure 4.1

	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*

A variable definition fault occurs in the outermost expression of the node. Thus, there is no computational transfer condition that must be satisfied, and any test datum that satisfies the origination condition set is guaranteed to introduce an original state potential failure for a variable definition fault.

We are now in a position to select a test data set that satisfies all the original state potential failure condition sets just developed. A test datum that satisfies an original state potential failure condition at a node n must be selected within $DOMAIN(n)$; further, it must be selected such that the original state potential failure condition is satisfied just before execution of the node. Because the node for which we have developed original state potential failure condition sets is one of the first nodes in the module, selection of test data that satisfies the conditions is relatively easy. There are many possible test data sets that satisfy the original state potential failure conditions developed for this example. Table 4.17 shows just one such set.

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

Next, consider the data selected for node 3, where the only hypothetical fault class, of the chosen classes applied, is the hypothetical variable definition fault. Datum 2 satisfies the single original state potential failure condition in the corresponding condition set, but is not in the *DOMAIN* of the node. Datum 5, however, satisfies the original state potential failure condition and executes the node.

Chapter 5

Transfer Set Condition

As outlined in Chapter 3, a failure condition for some hypothetical fault and transfer set is the conjunction of two components: the original state potential failure condition and the transfer set condition. The transfer set condition guarantees transfer of a state potential failure from an originating node to a failure node along a particular transfer set. This chapter describes the construction of the transfer set condition.

A transfer set is a set of information flow chains from the same originating node to the same failure node all of which may be executed together. A transfer route is a subset of the nodes in the transfer set where transfer actually occurs. To construct necessary and sufficient conditions to transfer from an originating node to a failure node, all potential failure variables at each node must be known. As noted briefly in Chapter 3, this motivates our definitions of transfer sets and transfer routes. In the first section of this chapter, we provide several examples that more fully demonstrate the need for transfer sets and transfer routes. Section 2 introduces some additional terminology associated with transfer sets and transfer routes, states several properties of these structures, and provides proofs for these properties.

The transfer set condition is constructed from two components – the path condition for the transfer set, which guarantees execution of all chains in the transfer set, and a component formed by the disjunction of transfer route conditions for all transfer routes associated with the transfer set. The transfer route condition for a particular transfer route is the conjunction of computational transfer conditions at transferring nodes in the transfer route and the complement of computational transfer conditions at non-transferring nodes. Formal definitions for the transfer set condition and its components are provided in Section 3. A failure condition is the conjunction of the original state potential failure condition for some fault at a node and a transfer set condition for some transfer set from that node. Formal definitions for the failure condition and total failure condition, previously described in Chapter 3, are also provided in Section 3.

Section 4 describes construction of the computational transfer conditions. Section 5 describes construction of the transfer route conditions. Section 6 describes construction of the transfer set conditions. Section 7 describes construction of the failure condition.

All faults at a hypothetically faulty node that do not change the information flow of a module have identical transfer sets and transfer routes. The transfer route conditions, and hence the transfer set conditions, however, may be fault dependent. The fault independence and fault dependence of transfer set conditions is discussed in Section 8.

The conditions developed here are written in terms of a constraint on a variable or variables at a particular node, e.g., $x \neq z$ at node 4. When actually satisfying this condition with test data, the code must be be instrumented at the nodes where variable values are constrained to determine constraint satisfaction or symbolic evaluation must be performed to generate constraints in terms of input variables. Because the focus of this thesis is not on selection of test data, such symbolic evaluation is not performed for our examples.

5.1 Motivation for Transfer Sets and Transfer Routes

We are interested in constructing the condition that guarantees transfer of a state potential failure from some originating node to some failure node. To do this, we must guarantee transfer along some information flow chain. The problem is that constructing the transfer condition along some information flow chain without taking into consideration what other chains are being transferred along does not yield a necessary and sufficient condition that guarantees transfer. This is because the computational transfer condition at any node depends on the potential failure variables at the node. To determine which variables are potential failure variables, we must know all information flow chains from the originating node along which transfer has occurred. A transfer set is the set of chains that may be executed together and defines the set of variables at each node that may be potential failure variables. To determine at a node which of these variables are actually potential failure variables, we must know where transfer has occurred up to that node. The actual potential failure variables at each node are defined by a transfer route, which is a subset of the nodes in the transfer set where transfer does occur.

In this section, we demonstrate with several examples how construction of transfer conditions along information flow chains without considering other chains that are transferred along does not yield sufficient and necessary conditions that guarantee transfer. We do this in two pairs of examples. The first pair demonstrates this for chains that involve only data dependence transfer. The second pair demonstrates this for chains that involve both data dependence and control dependence.

In the first example of each pair of examples, the chains from the originating node to failure node do not interact. We construct the transfer condition for such chains by conjoining the simple computational transfer conditions at each node in the chain. This condition is shown to be sufficient and necessary to guarantee transfer. In the second example for each pair of examples, there are interacting chains. We construct transfer conditions for these chains as in the first example, without considering what other chains may be being transferred along. These conditions are shown to be neither sufficient nor necessary to guarantee transfer.

Before considering the first pair of examples, let us construct in general the necessary and sufficient condition to transfer along a single chain, where there are no chains that interact with this chain along some path(s) in the module. The necessary and sufficient condition to transfer along such a chain, $(*, V_1, n_1)$ (V_1, V_2, n_2) ...$(V_{i-2}, V_{i-1}, n_{i-1})$ (V_{i-1}, V_i, n_i), is the conjunction of the simple computational transfer conditions from V_{j-1} to V_j at each node n_j. The simple computational transfer conditions apply at these nodes since there is at most only a single potential failure variable from which transfer could occur. (If there could be more than a single potential failure variable, then there would be more than a single information flow chain that could be executed together.)

Intuitively, we may argue by induction that this condition is sufficient to transfer an original state potential failure along the single chain:

> **basis:** By definition, the original state potential failure condition alone for fault at node n_1 is sufficient to transfer the originated subexpression potential failure to V_1 at node n_1;
>
> **induction:** Assume the condition formed for $(*, V_1, n_1)...(V_{j-2}, V_{j-1}, n_{j-1})$ is sufficient to transfer from V_1 at n_1 to V_{j-1} at n_{j-1}. By definition, the simple computational transfer condition from V_{j-1} to V_j at node n_j is sufficient to transfer from V_{j-1} to V_j. Thus, the conjunction of the two is sufficient to transfer from the original state potential failure in V_1 at n_1 to V_j at n_j.

Starting at the basis case, we may iteratively construct the sufficient condition to transfer to the next node in the single chain by conjoining the simple computational transfer condition at that node to the condition to transfer up to the previous node.

Intuitively, we may also argue that the condition formed from the conjunction of the simple computational transfer conditions at each node in the chain is necessary to transfer along this single chain, where there are no chains that interact with it.

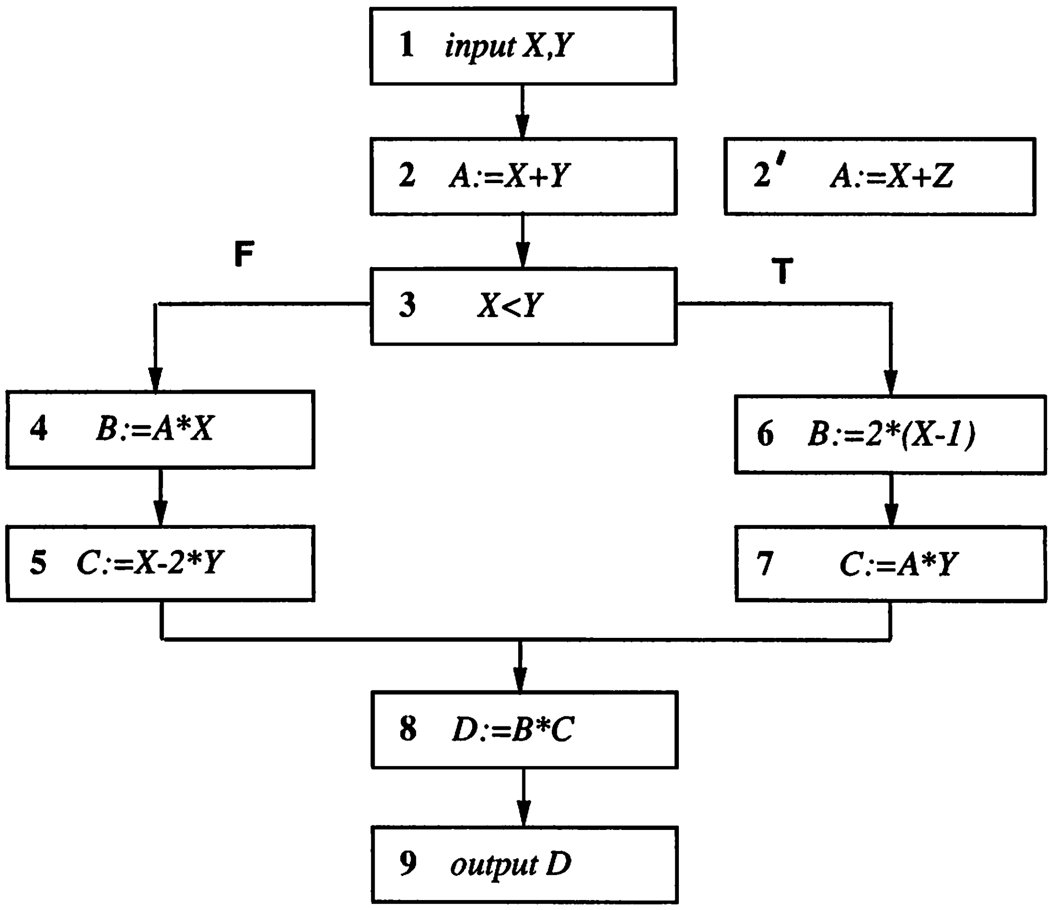

Figure 5.1. Example Module A

When there is no chain interacting with this chain, there is no other chain that could be transferred along from the same originating node to the same failure node. Clearly, if transfer fails at any point in the chain, no failure could be revealed. Assuming that the simple computational transfer condition at each node is the necessary condition to transfer at each node, then the conjunction of the simple computational transfer condition is necessary to transfer along the entire chain.

Let us turn now to two examples, A and B shown in Figure 5.1 and Figure 5.2. Consider first example module A shown in Figure 5.1 and consider a hypothetical fault at node 2. From this node, there are two information flow chains from the originating node to the failure node – node 9. One chain is $(*, A, 2)$ $(A, B, 4)$ $(B, D, 8)$ $(D, out, 9)$. The other chain is $(*, A, 2)$ $(A, C, 7)$ $(C, D, 8)$ $(D, out, 9)$. These chains are executed by separate paths, and thus cannot both be executed by the same test datum. To transfer a state potential failure introduced at node 2 to output, we must transfer along one of the information flow chains. We may construct the transfer condition for each chain as the conjunction of the simple computational transfer conditions at each node. For the first chain along the false branch, in addition to the condition to execute the chain $(x < y = F)$, the condition to transfer any state potential failure from node 2 to node 9 is $x \neq 0$ at node 4 and $c \neq 0$ at node 8. The transfer condition to transfer along the second chain, in addition to the condition to execute the chain $(x < y = T)$, is $y \neq 0$ at node 7 and $b \neq 0$ at node 8.

Now let us examine some test data for this module. Suppose that we are testing for the hypothetical fault at node 2 of an incorrect reference to Y that should be a reference to Z. The hypothetically correct node is shown in the figure alongside the hypothetically faulty node. The original state potential failure condition, which must be conjoined to the transfer condition for one of the information flow chains to guarantee failure along that chain, is $z \neq y$ at node 2. Table 5.1 shows four test data along with partial execution traces; the first line for each test datum shows variable values for the actual module being tested, while the second line shows values that would be computed by the hypothetically correct module. Test data 1 and 2 execute the first information flow chain, which selects the false branch. Both test data satisfy the original state potential failure condition and introduce a state potential failure in A at node 2. Test datum 1, however, fails to satisfy the condition to transfer along this chain and fails to cause a failure at output. This result is expected, since the condition developed is necessary to cause a failure. Test datum 2, on the other hand, does satisfy the condition and would cause a failure. This result is also expected

Table 5.1. Test Data Set For Example Module A

	module	t.d.			ref	a	$x < y$	b	c	d	out
		x	y	z	(y/z)	(2)	(3)	(4/6)	(5/7)	(8)	(9)
1	faulty	2	1	2	1	3	F	6 (at 4)	0 (at 5)	0	0
	correct	2	1	2	2	4	F	8 (at 4)	0 (at 5)	0	0
2	faulty	1	-3	1	-3	-2	F	-2 (at 4)	5 (at 5)	-10	-10
	correct	1	-3	1	1	2	F	2 (at 4)	5 (at 5)	10	10
3	faulty	1	2	1	2	3	T	0 (at 6)	6 (at 7)	0	0
	correct	1	2	1	1	2	T	0 (at 6)	4 (at 7)	0	0
4	faulty	2	3	2	3	5	T	2 (at 6)	15 (at 7)	30	30
	correct	2	3	2	2	4	T	2 (at 6)	12 (at 7)	24	24

because the condition developed is sufficient to transfer to output. Test data 3 and 4 execute the second information flow chain and demonstrate similar behavior.

Turn now to example B shown in Figure 5.2. If we again consider node 2 as a hypothetically faulty node, then there are two information flow chains to the failure node 6. One chain is $(*, A, 2)(A, B, 3)(B, D, 5)(D, out, 6)$. The other chain is $(*, A, 2)(A, C, 4)(C, D, 5)(D, out, 6)$. These chains and the computations performed at each node in the chains are identical to those in the previous example (A) of Figure 5.1 (except for the node numbering). These chains, however, are both executed by the same single path through the module and form a transfer set. Suppose we construct the transfer condition along a chain as before, without considering what other chains might be transferred along. This yields transfer conditions that are identical to those in the previous example. For the first chain, we have the condition $x \neq 0$ at node 3 and $c \neq 0$ at node 5, and for the second chain we have the condition $y \neq 0$ at node 4 and $b \neq 0$ at node 5.

Let us now examine sample test data for this module. Table 5.2 shows test data and partial traces for this module and the hypothetically correct module. Again, we consider the same incorrect reference fault as before. Both test data satisfy

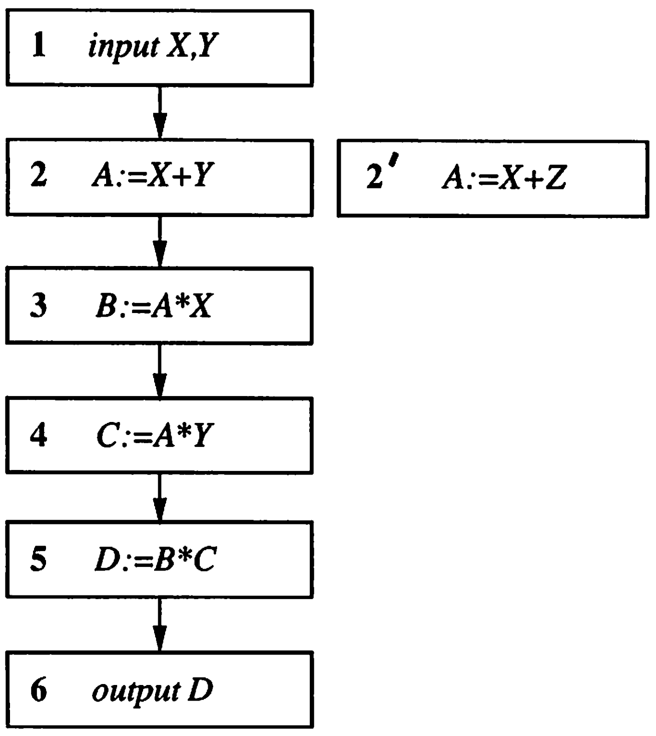

Figure 5.2. Example Module B

Table 5.2. Test Data Set For Example Module B

	module	t.d.			ref	*a*	*b*	*c*	*d*	*out*
		x	y	z	(y/z)	(2)	(3)	(4)	(5)	(6)
1	faulty	1	-3	1	-3	-2	-2	6	-12	-12
	correct	1	-3	1	1	2	2	-6	-12	-12
2	faulty	2	1	2	1	3	6	3	18	18
	correct	2	1	2	2	4	8	4	32	32

the original state potential failure condition for this fault. Test datum 1 satisfies the transfer conditions for both chains, as developed above, but fails to reveal a failure. Thus, for neither chain, is the condition we developed sufficient to guarantee transfer along the chain to output, nor are the conditions for the two chains together sufficient. On the other hand, test datum 3, which does reveal a failure but does not satisfy the transfer condition for either chain, shows that the conditions for the chains are not necessary to transfer a state potential failure from node 2 to node 6.

From this example, we see that we cannot derive the condition that is sufficient and necessary to guarantee transfer of a potential failure along a chain without considering what other chains are being transferred along. To know what chains could be being transferred along, we must know the transfer sets. To know which chains are actually being transferred along, we must know the transfer routes.

In example B, the two information flow chains from node 2 to node 6 may be executed together and thus form a transfer set consisting of the chains $(*, A, 2)$ $(A, B, 3)$ $(B, D, 5)$ $(D, out, 6)$ and $(*, A, 2)$ $(A, C, 3)$ $(C, D, 5)$ $(D, out, 6)$. For this transfer set, there is one node where there may be more than one potential failure variable – node 5. Which of these variables (B or C) are potential failure variables however, depends on which chains transfer has occurred along up to that point. One possibility is that A does not transfer to B at node 3 but does transfer to C at node 4. In this case, at node 5, only C is a potential failure variable. Another possibility is that A does transfer to B at node 3 but does not transfer to C at node 4. Then at node 5, only B is a potential failure variable. A final possibility is that A transfers to B at node 3 and to C at node 4, and at node 5. For this final case, both B and C are potential failure variables. Each of these possibilities represents a transfer route and requires a different condition to guarantee transfer at node 5.

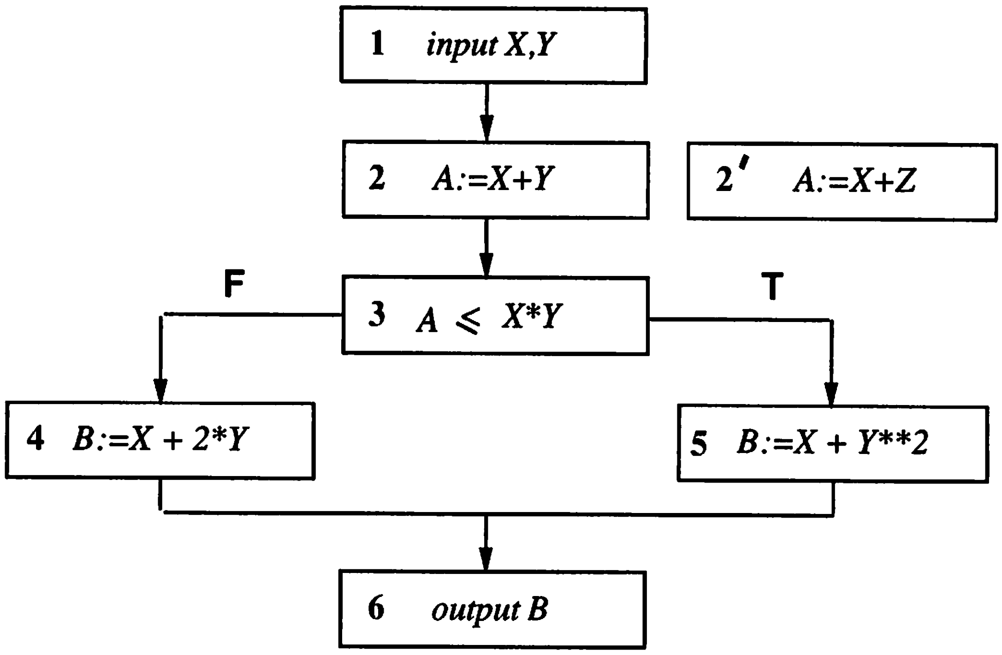

Figure 5.3. Example Module C

In the previous pair of examples, the information flow chains involved only data dependence transfer. Similar problems arise when control dependence transfer is involved as well. This is demonstrated with the next pair of examples, consisting of two modules, C and D, shown in Figures 5.3 and 5.4.

Let us follow the same procedure of developing the transfer conditions for independent chains and comparing them to those for interacting chains for these two examples, C and D. Thus, let us start with the single chains in example C shown in Figure 5.3. If we consider transfer of an originated state potential failure from node 2, there are two information flow chains to output at node 6. They are $(*, A, 2)(A, BP, 3)(BP, B, 4)$ $(B, out, 6)$ and $(*, A, 2)(A, BP, 3)(BP, B, 5)(B, out, 6)$. These chains include control dependence links at nodes 4 and 5 from node 3. These chains are executed by separate paths and cannot both be executed at once. Construction of the transfer condition along each chain is similar to that performed for example module A. To construct the condition for each chain, we conjoin the

Table 5.3. Test Data Set For Example Module C

	module	t.d.		ref	a	$a \leq x * y$	b (node 4)	b (node 5)	*out*
		x	y	(y/z)	(2)	(3)			(6)
1	faulty	2	1	1	3	F	4	(3)	4
	correct	2	1	2	4	F	4	(3)	4
2	faulty	2	2	2	3	F	5	(5)	5
	correct	2	2	1	2	T	(5)	5	5
3	fault	-1	1	1	0	F	1	(0)	1
	correct	-1	1	-1	-2	T	(1)	0	0

transfer condition at each node in the chain. For these chains, at nodes that represent data dependence links, the transfer conditions are the simple computational transfer conditions. To transfer at a node that represents a control dependence link, we must distinguish the computation performed along the branch from what would have been performed along the correct branch.

For the first chain, in addition to the condition to execute the chain ($a \leq x * y = F$), the condition to transfer from node 2 to node 6 is $(a \leq x * y) \neq (a' \leq x * y)$ [1] at node 3 (to transfer along data dependence at node 3) and $(x + 2 * y) \neq (x + y * * 2)$ at node 4 (to transfer along control dependence at node 4). Similar arguments as stated above showing the necessity and sufficiency of this condition may be made.

Table 5.3 shows sample test data along with partial execution traces for this module. In this table, both the computation for B at node 4 and at node 5 are shown. The value in parentheses indicates the value that would have been computed if the other node had been selected.

Suppose we again consider the hypothetical fault at node 2 of an incorrect reference to Y at node 2, which should be a reference to Z. All three test data in the test data set satisfy the original state potential failure condition for this fault.

[1]The notation a' represents the value of a that would have reached node 3 in the correct module.

Test data 1 and 2 select the false branch, executing the first chain. Both test data fail, however, to satisfy the condition constructed to transfer along this chain. Specifically, while test datum 1 distinguishes between the computation at node 4 and that at node 5 (since $4 \neq 3$), it fails to transfer along data dependence at node 3. Test datum 2, on the other hand, transfers through data dependence at node 3, but fails to transfer along control dependence when the computation performed at node 4 evaluates to the same value as at node 5 ($x+2*y = x+y**2 = 5$). No failure results for these test data. This result is expected as the condition we developed to transfer along this chain is necessary to cause a failure.

Test datum 3 also selects the false branch and executes chain 1. This test datum satisfies the condition to transfer along this chain and as seen in the output column would cause a failure if the hypothetical fault is a fault. This result is also expected as the condition developed for chain 1 is sufficient to transfer a state potential failure to output.

Turn now to example module D shown in Figure 5.4. This example includes data dependence from node 2 to nodes 4 and 5 as well as control dependence at nodes 4 and 5 on node 3. From node 2 to node 6, there are four information flow chains:

1. $(*, A, 2)(A, BP, 3)(BP, B, 4)(B, out, 6)$
2. $(*, A, 2)(A, B, 4)(B, out, 6)$
3. $(*, A, 2)(A, BP, 3)(BP, B, 5)(B, out, 6)$
4. $(*, A, 2)(A, B, 5)(B, out, 6)$

Let us consider transfer along the first two chains, which may both be executed by the same path that selects the false branch. Suppose we construct the transfer condition for each of the two chains separately. For chain 1, in addition to the

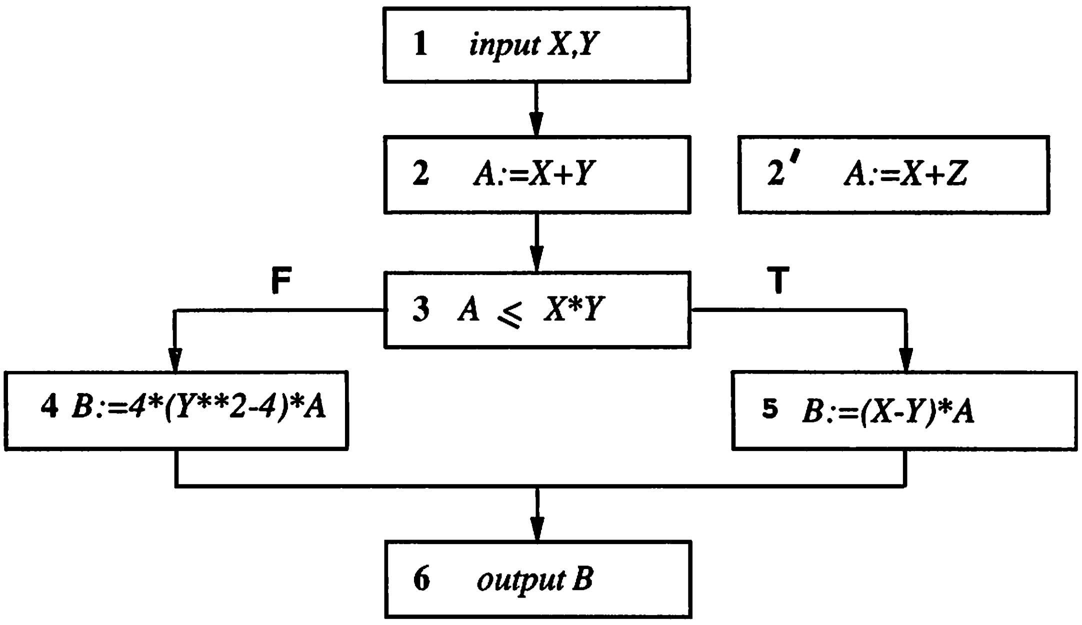

Figure 5.4. Example Module D

condition to execute the chain $(a \leq x * y = F)$, the condition derived to transfer along this chain is $(a \leq x * y) \neq (a' \leq x * y)$ at node 3 (to transfer along data dependence at node 3) and $(x - y) \neq (4 * (y * *2 - 4)$ at node 4 (to transfer along control dependence at node 4). For the second chain, in addition to the condition to execute the chain, the condition derived is $(y \neq \pm 2)$ at node 4 (to transfer along data dependence at node 4).

Table 5.4 shows a test data set for this module. As in previous examples, suppose we are testing for the same hypothetical reference fault at node 2. Both test data satisfy the original state potential failure condition. Test datum 1 satisfies the transfer conditions for both chain 1 and chain 2. This test datum, however, fails to cause a failure for this fault. Thus we see that for neither chain is the condition developed sufficient to guarantee transfer from node 2 to node 6. Test datum 2, on the other hand, fails to satisfy the transfer condition for either chain. Specifically, for chain 2, it fails to satisfy the condition to transfer along control dependence

Table 5.4. Test Data Set For Example Module D

	module	t.d.			ref	a	$a \leq x * y$	b (node 4)	b (node 5)	*out*
		x	y	z	(y/z)	(2)	(3)			(6)
1	faulty	-2	1	-2	1	-1	F	12	(3)	12
	correct	-2	1	-2	-2	-4	T	(48)	12	12
2	faulty	1	-2	-4	-2	-1	F	0	(-3)	0
	correct	1	-2	-4	-4	-3	T	(0)	-9	-9

at node 4 and for chain 1 it fails to satisfy the condition to transfer along data dependence at node 4. Nonetheless, this test datum would still cause a failure for this fault. From this test datum, we see that the conditions developed for both chains are not necessary to guarantee transfer from node 2 to node 6.

The previous examples show that the conditions to transfer along an information flow chain developed without considering other chains that may be transferred along are neither sufficient nor necessary to guarantee transfer to output. In order to construct conditions that are sufficient and necessary, we must know all the potential failure variables from which transfer may occur. This motivates our definition of transfer sets. Transfer sets alone, however, are not sufficient to determine the potential failure variables at any node. This is because, while transfer sets define at each node the variables that could be potential failure variables, whether or not a variable is a potential failure variable at a particular node depends on where transfer has occurred up to that node. Thus for any transfer set, there may be several ways transfer actually occurs. Each of these ways is identified by a transfer route.

These examples identify the issues that are important when considering transfer of an originated potential failure to output and motivate our definition of transfer sets and transfer routes.

5.2 Transfer Sets and Transfer Routes

This section presents some additional terminology and notation associated with transfer sets and transfer routes. In addition, this section presents several interesting properties of transfer sets along with proofs for these properties. The first subsection presents the terminology and notation, while the second subsection presents the properties.

5.2.1 Terminology

Recall from Chapter 3 that a transfer set is a set of information flow chains that all start at the same hypothetically faulty node and end at the same failure location. All chains in a transfer set are executed by some set of subpaths, each of which executes all the chains, and all chains executed by such subpaths are included in the transfer set.

Associated with a transfer set is a condition that is necessary and sufficient to execute all chains in the set. This condition is called the *transfer set path condition.* The transfer set path condition is the conjunction of the path condition for each chain in the set. The path condition for a transfer set defines a set of paths that execute all chains in the set. This set is called the *covering (set of) paths* and is the intersection of the sets of covering paths for each chain in the transfer set.

Definition: Given a transfer set $TS = \{A, B, ...\}$, the **transfer set path condition** is $\textit{path-condition}(TS) = \bigwedge_{C \in TS} \textit{path condition}(C)$.

The **set of covering paths for TS** is $Paths(TS) = \bigcap_{C \in TS} paths(C)$.

Returning to module D in Figure 5.4, for the transfer set from node 2, which consists of the chains $(*, A, 2)(A, BP, 3)(BP, B, 4)(B, out, 6)$ and $(*, A, 2)(A, B, 4)(B, out, 6)$, the path condition is $a \leq x * y$ at node 3. For this transfer set, the set of covering paths is the single path $p = (1, 2, 3, 4, 6)$.

We are interested in nodes where two or more variables used at the node are potential failure variables. At such nodes, we say that interaction occurs. A transfer set defines the set of nodes where interaction potentially occur. Potential interaction occurs at a node if there exists two chains in a transfer set that both define the same variable (including BP) at the node, but use different variables (including BP) at that node. More formally,

Definition: Given a transfer set $TS =$

$$\begin{aligned} A = &\quad (u(A)_1, d(A)_1, n(A)_1), ..., (u(A)_{|A|}, d(A)_{|A|}, n(A)_{|A|}) \\ B = &\quad (u(B)_1, d(B)_1, n(B)_1), ..., (u(B)_{|B|}, d(B)_{|B|}, n(B)_{|B|}) \\ &\qquad \vdots \end{aligned}$$

If there exists i, j, $1 \leq i \leq |A|$ and $1 \leq j \leq |B|$, corresponding to tuples $(u(A)_i, d(A)_i, n(A)_i)$ and $(u(B)_j, d(B)_j, n(B)_j)$, such that $d(A)_i = d(B)_j$ and $n(A)_i = n(B)_j$, but $u(A)_i \neq u(B)_j$, **potential interaction** occurs at a node $n(A)_i = n(B)_j$.

We use the term 'potential' interaction because whether or not interaction actually occurs depends on whether or not transfer occurs up to the point of potential interaction.

As defined in Chapter 3, a transfer route of a transfer set is a subset of nodes in the transfer set at which transfer does occur. Because a transfer route is associated with a particular transfer set from which it has been derived, the non-transferring nodes and the potential failure variables referenced at the node are both implicit in the transfer route. Nonetheless, we will make this information explicit with the following notation. A transfer route may be written as a list of tuples of the form: $(V_1 + V_2 + ... + V_k, T, n)$ for transferring nodes [2] and of the form $\neg(V_1 + V_2 + ... + V_k, T, n)$

[2]Generically, we may also write a tuple $(V_1 + V_2 + ... + V_k, T, n)$ as $(+_{1 \leq j \leq m} V_j, T, n)$.

for non-transferring nodes, where V_i are the potential failure variables that are used at node n and T is the variable to which transfer occurs (or does not occur) at node n. Note that non-transferring nodes that do not reference any potential failure variables because transfer along all chains up to the node has failed are not included in the list of non-transferring tuples. This is because no condition to cause transfer to fail at these nodes needs to be constructed. For any node that could appear more than once in an information flow chain, the visit of the node is disambiguated with a subscript. Thus, the order of the tuples is not significant.

The transfer route for Figure 3.2 previously written as:

- transfer from X to D at node 3 and **do not** transfer from D to BP at node 4 and transfer from X to Y at node 5 and transfer from (X and D and Y) to Z at node 7 and transfer from Z to output at node 8;

is written in our notation as:

$$(X, D, 3); \neg(D, BP, 4); (X, Y, 5); (X + D + Y, Z, 7); (Z, out, 8).$$

Actual *interaction* occurs along a transfer route at a node where more than one of the variables used at the node is a potential failure variable.

Definition: Given a transfer route $TR = \{tuple_i\}$, where that $\forall i,\ 1 \leq i \leq |TR|,\ tuple_i = +_{1\leq j\leq m}V_j, T, n)$ or $tuple_i = \neg(+_{1\leq j\leq m}V_j, T, n)\}$, **interaction** occurs at a node k if there exists $tuple_i = (+_{1\leq j\leq m}\ V_j, T, k)$ or $\neg(+_{1\leq j\leq m}V_j, T, n) \in TR$ such that $m > 1$.

In the transfer route $(X, D, 3); \neg(D, BP, 4); (X, Y, 5); (X + D + Y, Z, 7); (Z, out, 8)$, interaction occurs only at node 7.

Algorithms for identifying transfer sets and transfer routes may be found in an associated technical report [Tho91]. The general approach for identifying transfer sets is to generate all information flow chains from an originating node to a failure node and then determining the sets of chains with consistent path conditions. Identification of transfer routes involves a search on a graphical representation of a transfer set that generates legal combinations of transferring and non-transferring nodes of the transfer set. Identification of these structures is relatively straightforward when the code contains no loops. When loops are added, however, there may be a potentially infinite number of information flow chains and in turn a potentially infinite number of transfer sets. For such code, the identification of these structures is greatly complicated. To identify transfer sets and transfer routes in code involving loops, additional loop analysis must be performed. The approach reported in [Tho91] first generalizes the information flow through the loop, creating generalized acyclic information flow chains. These chains are used to identify generalized transfer sets. Information flow for generalized transfer sets may be specified as needed. To do this, we use an analysis of a loop in terms of single iteration chains. Single iteration chains are information flow chains whose first use may come from outside the loop (or a previous iteration), whose last definition may reach the immediate forward dominator for the loop, and that may be executed in a single iteration through the loop. Single information flow chains may be concatenated together to specify information flow over several iterations. While explicitly specifying the information flow through a loop requires selection of an iteration count through the loop (which corresponds to an unrolling of the loop), the value of the approach is that it allows reuse of the loop analysis and specification of the information flow through the loop only as needed. Explanation of the single iteration chain loop analysis method and example of its application is also found in [Tho91].

5.2.2 Properties of Transfer Sets and Transfer Routes

There are several properties of transfer sets and transfer routes that are interesting to note. They are presented here along with proofs.

Property 1 *There may be several transfer sets from an originating node n to a particular failure node f.*

Proof: Consider the example shown in Figure 3.2 and recall from Chapter 3 that there are two transfer sets from node 2 to node 9. They are:

- $\{(*,X,2)(X,D,3)(D,BP,4)(BP,Y,5)(Y,Z,7)(Z,out,8)$
  $(*,X,2)(X,Y,5)(Y,Z,7)(Z,out,8)$
  $(*,X,2)(X,D,3)(D,Z,7)(Z,out,8)$
  $(*,X,2)(X,Z,7)(Z,out,8)\}$

- $\{(*,X,2)(X,D,3)(D,BP,4)(BP,Y,6)(Y,Z,7)(Z,out,8)$
  $(*,X,2)(X,Y,6)(Y,Z,7)(Z,out,8)$
  $(*,X,2)(X,D,3)(D,Z,7)(Z,out,8)$
  $(*,X,2)(X,Z,7)(Z,out,8)\}$. $\square$

Property 2 *An information flow chain may be part of more than one transfer set.*

Proof: In the example discussed for Property 1, the information flow chains $(*,X,2)(X,D,3)(D,Z,7)(Z,out,8)$ and $(*,X,2)(X,Z,7)(Z,out,8)\}$ occur in both transfer sets. $\square$

Property 3 *The transfer sets from a particular originating node to a particular failure node define disjoint sets of covering paths.*

Proof: This follows from the definition of a transfer set, and a proof by contradiction follows.

Given two transfer sets TS_A and TS_B such that the originating node for TS_A = the originating node for TS_B and the failure node for TS_A =

the failure node for TS_B, and $TS_A \neq TS_B$, suppose that $Paths(TS_A) \cap Paths(TS_B) \neq \emptyset$ – that is the sets of covering paths are not disjoint.

Let $p \in (Paths(TS_A) \cap Paths(TS_B))$. Because $TS(A) \neq TS(B)$, we know there exists a chain $A \in TS_A$ and a chain $B \in TS_B$ such that $A \neq B$ and $(A \notin TS_B \vee B \notin TS_A)$. A is executed by p since $p \in PATHS(TS_A)$, and B is executed by p since $p \in PATHS(TS_B)$. It follows then that $A \in TS_B$ since by definition all chains with the same originating node and same failure node covered by $Paths(TS_B)$ must be in TS_B. It also follows that $B \in TS_A$ for the same reason. But A and B cannot both be in the same transfer set. Therefore $Paths(TS_A) \cap Paths(TS_B) = \emptyset$. $\square$

Property 4 *Any two chains in a transfer set potentially interact at some node.*

Proof: Given a transfer set TS and information flow chains $A, B \in TS$ such that $A \neq B$, we know that A and B start at the same originating node and end at the same failure node. Further, we know since $A \neq B$, that the two chains must diverge (differ) at some point after the originating node. Since they both end at the same node, they must also converge again at some node. At the node where they converge, the chains will define the same variable. Because they are converging, however, they will transfer from different potential failure variables at that definition. (If they transfer from the same variable, then that variable must have been defined along different paths, and thus the chains could not be in the same transfer set.) This point of convergence is by definition a point of potential interaction. $\square$

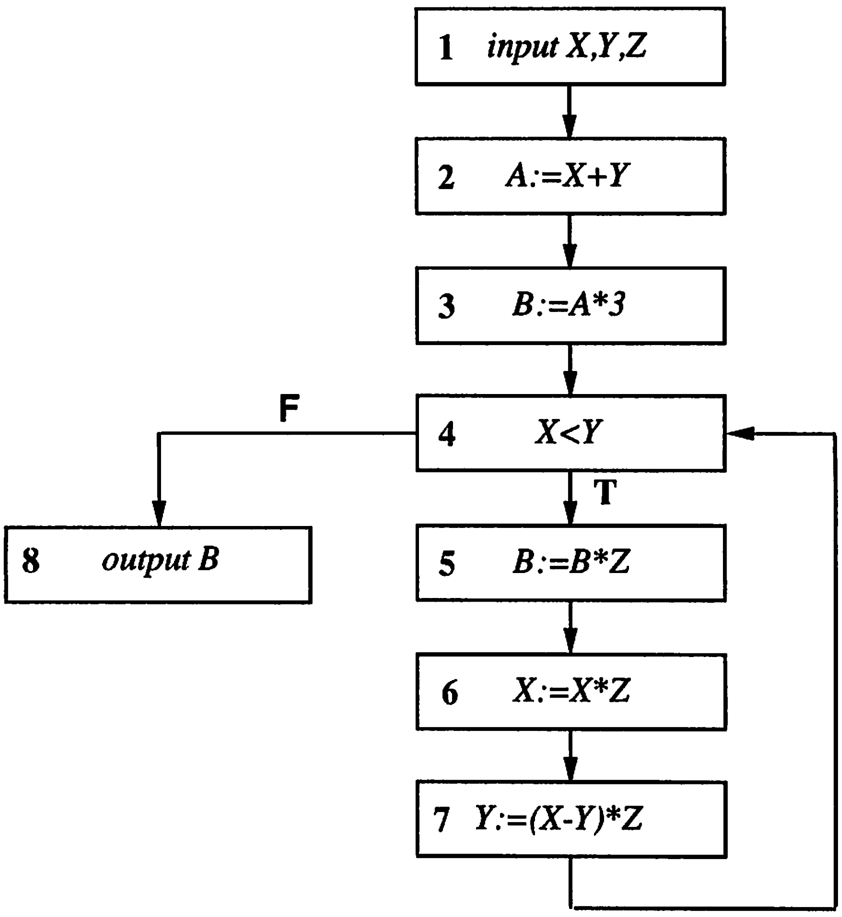

Figure 5.5. Module with Infinite Number of Chains

Property 5 *The number of transfer sets from an originating node to some failure node is potentially infinite.*

Proof: Consider the example shown in Figure 5.5. In this example, there are a potentially infinite number of information flow chains that start with the definition of A at node 2 and end with an output of B at node 8. They are:

$(*, A, 2)(A, B, 3)(B, out, 8)$

$(*, A, 2)(A, B, 5_1)(B, B, 5_i)^*(B, out, 8)$, where the notation 5_i is used to indicate the visit of node 5 in the i^{th} iteration of the loop, $1 < i \leq \infty$, and the kleene star notation is used to indicate that the tuple $(B, B, 5_i)$ may appear from 0 to ∞ times and each time i would be incremented.

For each of these chains, there is a single path through the module that covers the chain and only that chain. Thus, for each information flow chain, there is a transfer set consisting of that single chain. Therefore in this example, there is an infinite number of transfer sets from node 2 to node 8. $\square$

Property 6 *The number of chains in any transfer set is finite.*

Proof: Let $TS = \{A, B ...\}$ be a transfer set. Suppose that $|TS| = \infty$.

We know then that some chains in TS must involve a loop. Without loss of generality, suppose that the originating node and the failure node are both within a loop [3]. There is a finite number of nodes in any loop. Thus, for there to be an infinite number of chains in TS, some of the chains must involve repeated tuples, that is multiple visits to at least one node using the same potential failure variable at that node. The path(s) that executes the chain with the greater number of occurrences of a repeating tuple cannot also execute the chain with the fewer number of occurrences of the repeating tuple. This is because execution of the chain with the greater number of occurrences of the repeating tuple must redefine a variable needed to transfer along the other chain.

This may be seen more easily by considering the following chains shown schematically as:

[3]This assumption is made to simplify our discussion of individual chains in TS. If the originating nodes and failure nodes are outside the loop, then to denote a chain in TS, we must add subchains from the originating node to the loop and subchains from the loop to the failure node.

$$chain1 = (*, d_o, n_o) \cdot W \cdot (u_a, d_a, n_a) \cdot X \cdot (u_f, d_f, n_f)$$
$$chain2 = (*, d_o, n_o) \cdot Y \cdot (u_a, d_a, n_a) \cdot Z \cdot (u_a, d_a, n_a) \cdot X \cdot (u_f, d_f, n_f),$$

where $(*, d_o, n_o)$ represents the first tuple (originating) in all chains in TS, (u_f, d_f, n_f) represents the last tuple (failure) in TS, u_a is the variable used and d_a is the variable defined at node n_a, and X, Y, and Z represent subchains.

Let chain 1 be some chain in TS such that at least one tuple in chain 1 occurs more often in some other chain. Let the repeating tuple be (u_a, d_a, n_a). It is known that some chain 2 exists because it is known that there is some chain with at least one more occurrence of the repeating tuple, and once the second occurrence of (u_a, d_a, n_a) occurs in the chain, the same subchain that went from node n_a to n_f in the first chain (the segment X), may be concatenated after the second occurrence in chain 2.

A path that executes chain 1 must be def-clear with respect to d_a from n_a to n_f. Such a path, however, cannot also execute chain 2. Thus, chain 1 and chain 2 cannot be in the same TS. Therefore, TS must be finite. □

Property 7 *The set of covering paths for any transfer set is potentially infinite.*

Proof: Consider the example shown in Figure 5.6. From node 2 to node 9, there is a single transfer set consisting of the single information flow chain $(*, A, 2)(A, C, 8)(C, out, 9)$. For this chain, there is a potentially infinite set of paths that cover the chain. Using the kleene star notation to indicate repetition of the set of nodes $(5, 6, 7, 4)$, these paths are: $(1, 2, 4, (5, 6, 7, 4)^*, 8, 9)$ □

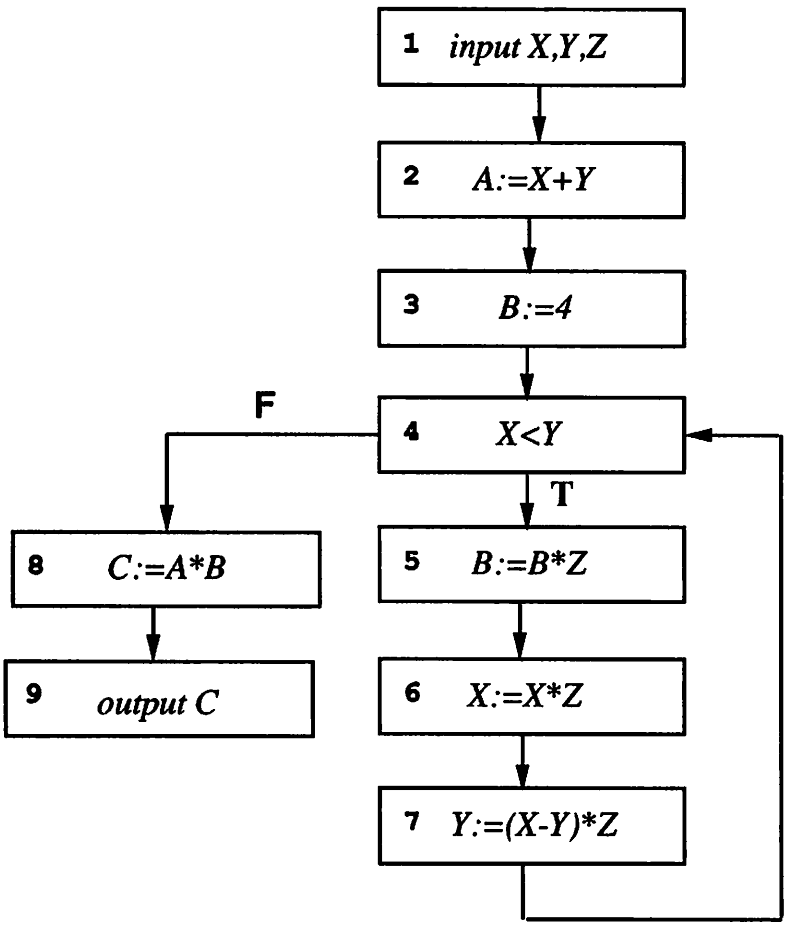

Figure 5.6. Module with Infinite Number of Paths for Transfer Set

5.3 Transfer Route, Transfer Set and Failure Conditions

This section provides formal definitions for the transfer set condition and its components. A transfer set condition is the necessary and sufficient condition to guarantee transfer from the transfer set's originating node to the transfer set's failure node. A transfer set condition is formed from the conjunction of the path condition for the transfer set, which guarantees execution of all information flow chains in the transfer set, with the disjunction of the transfer route conditions for all transfer routes associated with the transfer set. A transfer route condition is the necessary and sufficient condition that guarantees transfer along a particular transfer route. The transfer route condition is formed from the conjunction of the computational transfer conditions at the transferring nodes and the computational non-transfer conditions at the non-transferring nodes.

In Chapter 3, the computational transfer condition is defined as the necessary and sufficient condition to guarantee a subexpression potential failure referenced at a node transfers to effect evaluation of the entire node. As defined there it applies only to nodes where only data dependence is occurring. Here we extend the definition of a computational transfer condition to include transfer at nodes where control dependence transfer is occurring. In addition, we define the computational non-transfer condition as the necessary and sufficient condition to guarantee computational transfer does not occur at a node.

Formal definitions for these components are provided below. Details for the construction of these conditions are provided in the sections that follow.

Definition: Given a tuple $tuple_i = (V_1 + V_2 ... + V_k, T, n)$ from some transfer route, where $n = EXP$, and EXP references the potential failure variables $V_1, V_2 ... V_k$, the **computational transfer condition** at node n is $ctc(tuple_i) = ctc(V_1 + V_2 ... + V_k, T, n) =$

- $exp \neq exp'$, for $BP \notin \{V_1, V_2, ... V_k\}$, where exp is the value for exp computed using $v_1, v_2, ... v_k$ and exp' is the value for exp computed using $v'_1, v'_2, ... v'_k$, where v'_i is the value that reaches n in the hypothetically correct module on execution of the same test datum;
- $t \neq t'$, for $BP \in \{V_1, V_2, ..., V_k\}$, where t is the value of T assigned at node n that reaches the immediate forward dominator of node b and t' is the value that reaches the immediate forward dominator of node b, whose incorrect evaluation caused BP to represent a potential failure variable in the correct module.

The **computational non-transfer condition** for $tuple_i$ is the complement of the computational transfer condition and is denoted

$$\neg ctc(V_1 + V_2 + ... + V_k, T, n).$$

Definition: Let $TR = \{tuple_i\}$ be a transfer route associated with a transfer set TS, such that $\forall\ 1 \leq i \leq |TR|$, $tuple_i = (+_{1 \leq j \leq m} V_j, T, n)$ or $\neg(+_{1 \leq j \leq m} V_j, T, n)\}$. The **transfer route condition** for a TR is $trc(TR) = \bigwedge(ctc(tuple_k)) \ \wedge \ \bigwedge(\neg ctc(tuple_l)), \forall tuple_k = (+_{1 \leq j \leq m} V_j, T, n) \in TR$ and $\forall tuple_l = \neg(+_{1 \leq j \leq m} V_j, T, n) \in TR$.

Definition: Given a transfer set TS and its associated set of transfer routes $TR = \{tr_1, ... tr_k\}$, the **transfer set condition** is the necessary and sufficient condition that guarantees the transfer set is executed and for an original state potential failure introduced at the transfer set's originating node, a failure occurs at the transfer set's failure node. The transfer set condition is $tsc(TS) = Path\text{-}Condition(TS) \ \bigwedge \ (\vee_{1 \leq i \leq k}\ trc(tr_i))$

Given the original state potential failure condition as described in the previous chapter and the transfer set condition, we may construct the failure condition for a particular hypothetical fault and transfer set. The failure condition guarantees failure for a hypothetical fault, if indeed it is a fault, along a particular transfer set. The failure condition for a hypothetical fault at a node is the conjunction of the original state potential failure condition for the hypothetically faulty node and the transfer set condition for some transfer set from the hypothetically faulty node.

> **Definition:** Given a module M with $G_M = (N, E)$ and a hypothetically correct module M'. Let f be a hypothetical fault in some subexpression $SEXP$ of node $n \in N$ with $SEXP'$ the hypothetically correct alternate in the corresponding node $n' \in N'$. Let TS be a transfer set from node n to some failure node m. A **failure condition** for f in n along TS guarantees that $\forall t$ that satisfy the condition, t executes TS and reveals a failure at m. The failure condition is:
>
> $fc(M, f, SEXP, TS) = DOMAIN(n) \bigwedge ospfc(f, SEXP) \bigwedge tsc(TS)$.

This condition is sufficient to guarantee fault detection for a hypothetical fault. If this condition is unsatisfiable, however, the failure condition for another transfer set must be constructed. The disjunction of the failure conditions for all transfer sets from a hypothetically faulty node forms the necessary and sufficient condition to guarantee fault detection and is called the total failure condition.

> **Definition:** Given a module M with $G_M = (N, E)$ and a hypothetically correct module M'. Let f be a hypothetical fault in some subexpression $SEXP$ of node $n \in N$ with $SEXP'$ the hypothetically correct alternate in the corresponding node $n' \in N'$. The **total failure condition** guarantees that $\forall t$ that satisfy the condition, $M(t) \neq M'(t)$. Let $TTS = \{$ transfer sets from n to output nodes$\}$. The total failure condition $tfc(f, SEXP, M) = \bigvee_{TS \in TTS} fc(M, f, SEXP, TS)$.

The next four sections discuss each of these structure more and provides examples for their construction.

5.4 Construction of Computational Transfer Conditions

The transfer route condition for a transfer route is constructed using the computational transfer conditions and computational non-transfer conditions at the nodes in the transfer route. Here we describe the construction of the computational transfer condition for a transferring node in a transfer route. The computational non-transfer condition for a non-transferring node is simply the complement of the computational transfer condition at that node.

At nodes where only a single variable referenced is a potential failure variable, e.g., for tuples in transfer routes of the form (V, T, n), the computational transfer conditions is termed a *simple computational transfer condition.* (As with computational transfer condition in general, the previous definition for simple computational transfer condition is expanded to include reference to the potential failure variable 'BP.) At nodes where more than one potential failure variable is referenced, e.g., for tuples in transfer routes of the form $(V_1 + V_2 + \dots V_k, T, n)$, the computational transfer condition is termed a *complex computational transfer condition.* We separate the computational transfer conditions into these two classes for two reasons. First, the simple computational transfer conditions are relatively easy to construct as compared to the complex computational transfer conditions, and second, the simple computational transfer conditions are in some cases fault independent, while the complex transfer condition in general are not.

In the next two subsections, we describe the construction of the simple and complex computational transfer conditions. For both simple and complex computational transfer conditions, we separately consider the case when the branch predicate variable BP is a potential failure variable and when it is not.

5.4.1 Simple Computational Transfer Condition

Simple computational transfer conditions apply at nodes where a single potential failure variable is reference, and thus no interaction is occurring. There are two places this condition may be required. Simple computational transfer may occur at nodes that reference a single potential failure variable other than BP, and thus have been correctly selected, or at nodes that singly reference BP as a potential failure variable, and thus have been incorrectly selected.

Nodes that have been correctly selected but reference a single potential failure variable are of the form $op_t(..., op_2(...op_1(..., VAR, ...), ...), ...)$, where VAR is the potential failure variable. The simple computational transfer condition for such a node may be constructed by conjoining the simple computational transfer condition for each operator that has VAR nested within an operand. Construction of the computational transfer condition for such nodes is identical to that performed at an originating node and is discussed in Chapter 4. At such nodes, the computational transfer condition guarantees data dependence transfer occurs.

Consider next the development of the simple computational transfer condition for some node n that references a single potential failure variable BP, and thus has been incorrectly selected. At such nodes the computational transfer condition guarantees that control dependence transfer occurs. Such a node n is control dependent on some node b, since in order for n to be incorrectly selected, there had to be some selection or branching point. Either n is a branching node or n assigns a value to some variable V. We discuss each of these possibilities below.

If n is a branching node, then there is no simple node transfer condition constructed. To understand why this is so, consider Figure 5.7. This figure contains the skeleton of a portion of a control flow graph with nodes n and b. Transfer through n means that n evaluates incorrectly, but n has been incorrectly selected, and correct evaluation of n versus incorrect evaluation of n has no meaning. More specifically,

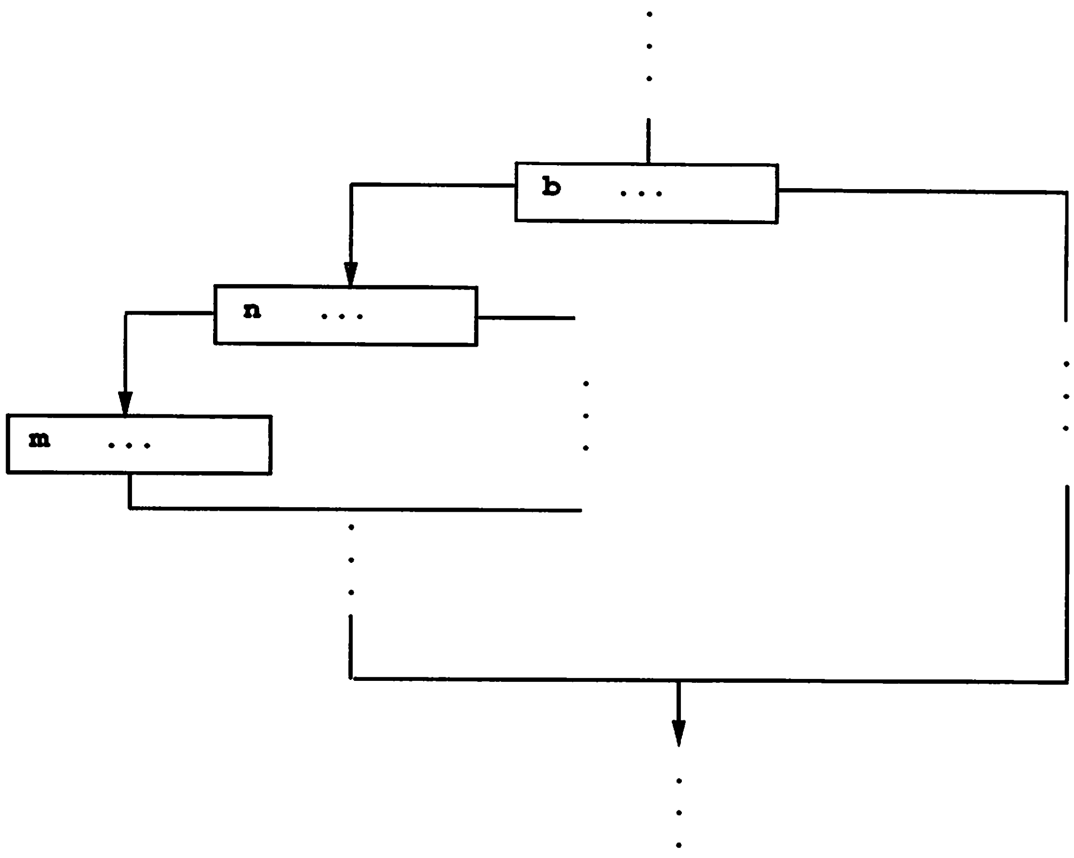

Figure 5.7. Module Fragment with Nested Conditional

any node m selected by n is also control dependent on node b, and thus has already been incorrectly selected by incorrect evaluation of b. Since this node cannot be any more 'incorrectly selected' by n, we do not construct node transfer conditions for nodes like n that have been incorrectly selected but are branching nodes. It should not be forgotten, however, that the evaluation of n may still be constrained by the path condition in the transfer set condition for which the computational transfer conditions are being constructed.

The other possibility is that n assigns a value to some variable V. If the value computed at n is not used at the immediate forward dominator or at a node that is a successor of the immediate forward dominator, then no computational transfer condition at n is constructed. To see why this is true, consider the skeleton of a portion of a control flow graph shown in Figure 5.8. Suppose that node n defines V and that $def(V, n)$ is not used by the immediate forward dominator i of b or at any subsequent point. One possibility is that $def(V, n)$ is not used before the immediate forward dominator. In this case, n has no effect on evaluation of the module. The other possibility is that some node m, between n and i, uses $def(V, n)$ in defining some variable A, that is then used by i or a successor of i. In this case m is also incorrectly selected (since n is), and the computational transfer condition for m will transfer along control dependence assigning a value to A that is distinct from that which would have been assigned along the other branch in the hypothetically correct alternate. While this computational transfer condition developed at m may constrain the value assigned at n, if node n defines a variable that is not used at the immediate forward dominator or at a successor of the immediate forward dominator, a simple computational transfer condition is not specifically constructed at n.

If n is incorrectly selected, references no potential failure variables, and defines a value to a variable V that is used at the immediate forward dominator or at some successor of the immediate forward dominator, then the simple computational transfer conditions is $v \neq v'$, where v' is the value of V that would have reached the forward dominator if n had not been selected. Note that the value of V' computed at n must be computed using values for all variables used at n that would have reached n in the hypothetically correct module. Consider the example shown in Figure 5.3 and suppose node 2 is the originating node and node 4 is incorrectly selected. At node 4, neither X nor Y are potential failure variables, and the simple computational transfer condition is $ctc(BP, B, 4) = (x + 2 * y) \neq (x + y * *2)$.

It is possible that there may be several, in fact a potentially infinite number of, computations for V that could have reached the immediate forward dominator if n

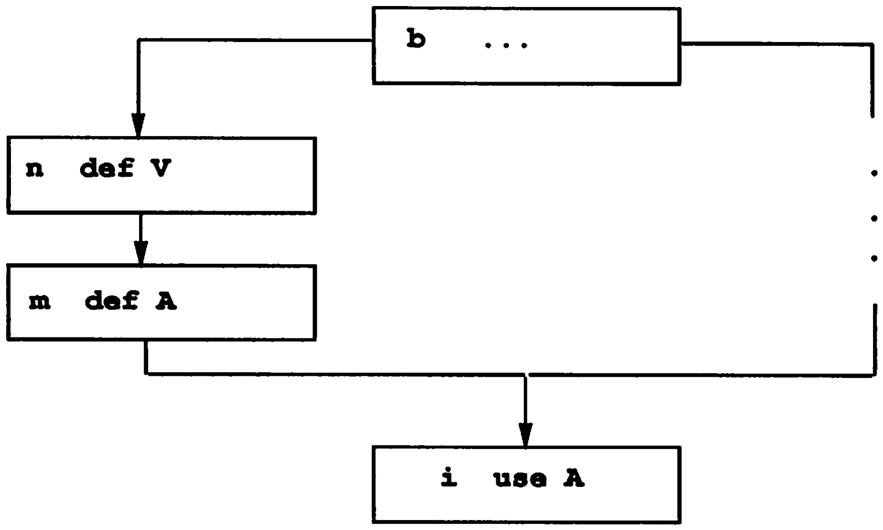

Figure 5.8. Module Fragment with Def-Use Information

had not been selected. For such cases, the simple computational transfer condition would in general be: ($v \neq v'_1$ and $cond_1$) or ($v \neq v'_2$ and $cond_2$) or ... ($v \neq v'_t$ and $cond_t$), where $cond_i$ is the condition that causes the definition V_i to be selected. Consider the simple example shown in Figure 5.9. The simple computational transfer condition at node 4 for the case when node 4 is incorrectly selected and X and Y are not potential failure variables is

$$\begin{aligned} ctc(BP, B, 4) = {} & (x + 2 * y) \neq (x + y * *2) \wedge (x < y = F) \\ \vee {} & (x + 2 * y) \neq (x + y * *4) \wedge (x < y = T). \end{aligned}$$

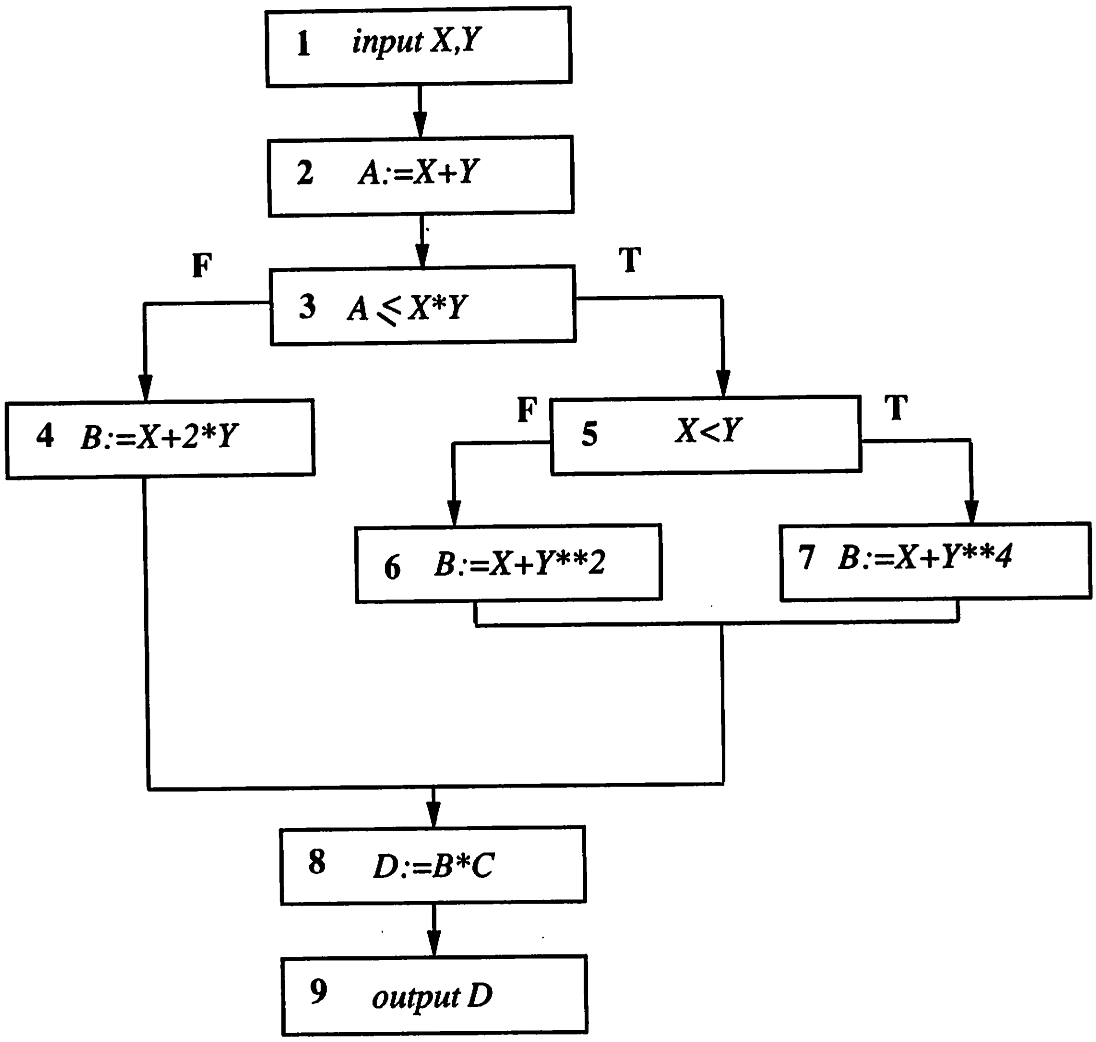

Figure 5.9. Multiple Assignments Along Alternate Branch

5.4.2 Complex Computational Transfer Condition

At nodes where more than one potential failure variable is referenced, interaction occurs, and we must develop complex node transfer conditions. There are two cases to consider: all data dependence interaction (where BP is not a potential failure variable) and data dependence and control dependence interaction (where one of the potential failure variables referenced is BP). The case of multiple control dependence is not possible, since as discussed with simple computational transfer conditions, a node that has been incorrectly selected cannot be 'more incorrectly' selected, and thus multiple control dependence links at a node do not result in interaction.

A complex computational transfer condition for a tuple $(V_1 + V_2 + ... + V_k, T, n)$ guarantees that when $V_1, ... V_k$ are potential failure variables at node n, T also becomes a potential failure variable at node T. Depending on the structure of the computation at node n, transfer to T may occur in several ways. It is possible that potential failure information may be masked out within the computation for some of the potential failure variables at the node while not for others. These different ways are captured within the complex computational transfer condition. For example, consider the expression $A := (B * C) + D$ and suppose that B and D are potential failure variables. One sufficient condition to transfer to A is $c = 0$. This condition essentially masks out the potential failure variable B while D transfers to A. Another possibility is that transfer occurs from both B and D. Both these possibilities are embedded in the complex computational transfer condition. Complex computational transfer conditions for the case when BP is and is not a potential failure variable at a node are described separately below.

When a node references two or more potential failure variables and BP is not a potential failure variable, the node has been correctly selected and data dependence interaction occurs. As seen in the example in Figure 5.2 and its accompanying discussion in Section 5.1, when two or more potential failure variables are referenced, the simple computational transfer condition constructed from the simple computational transfer conditions through the operators at the node are not necessary and sufficient to guarantee transfer at such a node. Such nodes are of the form

$$op_t(\ldots, op_l(\ldots, VAR_1, \ldots, (op_m(\ldots, VAR_2, \ldots op_p(\ldots VAR_i \ldots)\ldots)\ldots)\ldots)\ldots).$$

The computational transfer condition is

$$op_t(\ldots, op_l(\ldots, var_1, \ldots, (op_m(\ldots, var_2, \ldots op_p(\ldots var_i \ldots)\ldots)\ldots)\ldots)\ldots) \neq$$

$$op_t(\ldots, op_l(\ldots, var'_1, \ldots, (op_m(\ldots, var'_2, \ldots op_p(\ldots var'_i \ldots)\ldots)\ldots)\ldots)\ldots),$$

where $VAR_1, VAR_2, \ldots, VAR_i$ are potential failure variables and thus $\forall j, 1 \leq j \leq i, var_j \neq var'_j$.

Consider the example module shown in Figure 5.2. If we consider a hypothetical fault at node 2 and a transfer route where both chains are transferred along, then at node 5 both B and C are potential failure variables, and we must construct a complex computational transfer condition. This condition is $ctc(B+C, D, 5) = b*c \neq b'*c'$, where b' and c' are the values for B and C that reach node 5 in the hypothetically correct module.

It is also possible for a node to reference two or more potential failure variables where one of them is BP. In this case, the node has been incorrectly selected, and control dependence and data dependence interaction occurs. For the same reasons discussed with simple computational transfer conditions when BP is the only potential failure variable referenced, we only develop the complex computational transfer condition for an incorrectly selected node n that defines a variable V that is used at the immediate forward dominator or at a successor of the immediate forward dominator. The complex node transfer condition guarantees that the value assigned to V is different from that which would have reached the immediate forward dominator, using the correct variable values and the correct computation, if n had not been selected. This is essentially how the simple computational transfer condition is derived for nodes that reference only the potential failure variable BP.

Consider the example shown in Figure 5.4. If we are considering a hypothetical fault at node 2 and the transfer route where node 4 is incorrectly selected and A is a potential failure variable, then the computational transfer condition at node 4 is

$$ctc(BP + A, B, 4) = (4 * (y * *2 - 4) * a) \neq ((x - y) * a').$$

As with simple computational transfer conditions, there may be many alternative definitions that could reach the immediate forward dominator. A complex computational transfer condition must consider all such definitions.

5.5 Construction of Transfer Route Conditions

Given the computational transfer conditions for the tuples in a transfer route, we may construct the transfer route condition for that transfer route. This is done simply by conjoining the computational transfer conditions for the transferring nodes in the transfer route and the computational non-transfer conditions for the non-transferring nodes.

Returning to Figure 5.4, let us construct the transfer route conditions for the transfer routes that are associated with the transfer set that consists of the two chains $(*, A, 2)(A, BP, 3)(BP, B, 4)(B, out, 6)$ and $(*, A, 2)(A, B, 4)(B, out, 6)$. There are two transfer routes associated with this transfer set. They are:

$tr_1 : (A, BP, 3); (A + BP, B, 4); (B, out, 6)$

$tr_2 : \neg(A, BP, 3); (A, B, 4); (B, out, 6).$

Construction of the transfer route condition for these two transfer routes requires the following computational transfer conditions: $ctc(A, BP, 3)$ and $ctc(BP+A, B, 4)$

and $ctc(A, B, 4)$ and $ctc(B, out, 6)$. Constructing these conditions as described in the previous section yields the following conditions:

$ctc(A, BP, 3) = (a \leq x * y) \neq (a' \leq x * y)$ at node 3

$ctc(BP + A, B, 4) = (4(y * *2 - 4) * a) \neq ((x - y) * a')$ at node 4

$ctc(A, B, 4) = y \neq \pm 2$ at node 4

$ctc(B, out, 6) = true.$

The transfer route condition for each transfer route is:

$$\begin{aligned} trc(tr_1) &= \ ctc(A, BP, 3) \bigwedge ctc(A + BP, B, 4) \bigwedge ctc(B, out, 6) \\ trc(tr_2) &= \ \neg(ctc(A, BP, 3)) \bigwedge ctc(A, B, 4) \bigwedge ctc(B, out, 6). \end{aligned}$$

More completely, using the actual computational transfer conditions and the computational non-transfer conditions formed by complementing the computational transfer condition, the transfer route conditions are:

$$\begin{aligned} trc(tr_1) = \ & (a \leq x * y) \neq (a' \leq x * y) \text{ at node 3 } \bigwedge \\ & (4(y * *2 - 4) * a) \neq ((x - y) * a') \text{ at node 4 } \bigwedge \\ & true \\ trc(tr_2) = \ & (a \leq x * y) = (a' \leq x * y) \text{ at node 3 } \bigwedge \\ & y \neq \pm 2 \text{ at node 4 } \bigwedge \\ & true. \end{aligned}$$

5.6 Construction of Transfer Set Conditions

The transfer set condition is the conjunction of the path condition for the transfer set with the condition formed from the disjunction of the transfer route conditions for the transfer routes associated with the transfer set. If we continue with the example from above, the path condition for the transfer set $TS = \{(*, A, 2)\ (A, BP, 3)\ (BP, B, 4)\ (B, out, 6);\ (*, A, 2)\ (A, B, 4)\ (B, out, 6)\}$ selects the false branch in the module and is

$$\mathit{Path\ Condition}(TS) = a \leq x * y \text{ at node } 3.$$

Using the transfer route conditions as developed in the previous section (dropping the condition *true*), the transfer set condition for TS is

$$
\begin{aligned}
&tsc(TS) = \\
&\qquad a \leq x * y \text{ at node } 3 \\
&\quad \wedge \\
&\qquad (\ (a \leq x * y) \neq (a' \leq x * y) \text{ at node } 3 \ \wedge \\
&\qquad (4(y * *2 - 4) * a) \neq ((x - y) * a') \text{ at node } 4 \\
&\qquad \vee \\
&\qquad (a \leq x * y) = (a' \leq x * y) \text{ at node } 3 \ \wedge \\
&\qquad y \neq \pm 2 \text{ at node } 4\ ).
\end{aligned}
$$

5.7 Construction of Failure Conditions and Total Failure Conditions

Given the original state potential failure condition as described in the previous chapter and the transfer set condition described in this chapter, we may now construct the failure condition for a particular hypothetical fault and transfer set. The failure condition for a hypothetical fault at a node is the conjunction of the transfer set condition for some transfer set and the original state potential failure condition at the hypothetically faulty node.

Let us continue with the example shown in Figure 5.4 for which we have just constructed a transfer set condition and consider the hypothetical variable reference fault at node 2, where the reference to Y should be a reference to Z. The original state potential failure condition is $y \neq z$ at node 2. Conjoining this condition with the transfer set condition developed in the previous section yields the failure condition:

$fc(f, SEXP, TS) =$

$y \neq z$ at node 2

$\wedge$

$a \leq x * y$ at node 3

$\wedge$

$(\ (a \leq x * y) \neq (a' \leq x * y)$ at node 3 $\wedge$

$(4(y * *2 - 4) * a) \neq ((x - y) * a')$ at node 4

$\vee$

$(a \leq x * y) = (a' \leq x * y)$ at node 3 $\wedge$

$y \neq \pm 2$ at node 4).

This condition is sufficient to guarantee fault detection for this hypothetical fault. If this condition is unsatisfiable, however, the failure condition for another transfer set must be constructed. The disjunction of the failure conditions for all transfer sets from a hypothetically faulty node forms the necessary and sufficient condition to guarantee fault detection is called the total failure condition. In Figure 5.4, there is a second transfer set from node 2 to the single output node 6, which is covered when the *true* branch is selected. The transfer set condition for this transfer set may be constructed as done for the first transfer set. The total failure condition is:

$tfc(f, SEXP, M) =$

$y \neq z$ at node 2

$\wedge$

($a \leq x * y$ at node 3

$\wedge$

($(a \leq x * y) \neq (a' \leq x * y)$ at node 3 $\wedge$

$(4(y * *2 - 4) * a) \neq ((x - y) * a')$ at node 4

$\vee$

$(a \leq x * y) = (a' \leq x * y)$ at node 3 $\wedge$

$y \neq \pm 2$ at node 4).

$\vee$

$a > x * y$ at node 3

$\wedge$

($(a \leq x * y) \neq (a' \leq x * y)$ at node 3 $\wedge$

$(4(y * *2 - 4) * a) \neq ((x - y) * a')$ at node 5

$\vee$

$(a \leq x * y) = (a' \leq x * y)$ at node 3 $\wedge$

$x \neq y$ at node 5))

5.8 Fault Dependence and Independence of Transfer Set Conditions

Many hypothetical faults may apply at a node. When a hypothetical fault does not effect the information flow of a module, the hypothetically faulty module and the hypothetically correct module contain the same transfer sets and transfer routes. In this case, we would like to be able to apply the same transfer set conditions to the original state potential failure condition for several hypothetical faults at a node to construct a failure condition for those faults. This section describes the fault dependence and fault independence of transfer set conditions.

For a transfer set condition to be fault independent, the computational transfer conditions in the transfer route conditions must be fault independent. The fault independence of a computational transfer condition varies depending the type of the computational transfer condition — simple or complex.

Consider first simple computational transfer conditions. For correctly selected nodes, the simple computational transfer condition is fault independent when transfer through none of the operators that contains the single potential failure variable referenced in an operand depends on v and v', where v is the value of the potential failure variable referenced at the node and v' is the value of the variable that would have reached the node in the hypothetically correct module. This is true through boolean, multiplication, addition, division, and assignment operators. It is not true through relational, exponentiation and integer operations (e.g., DIV), although sufficient conditions that do not depend on the value of the subexpression potential failure transferring (and hence the fault) may be available. The simple computational transfer at an incorrectly selected node (i.e., that reference BP) is fault independent when the v' is not fault dependent, where V is the variable defined at the node and v' is the value for V that reaches the immediate forward dominator along the correctly selected branch. The value v' may be fault dependent if its computation uses a potential failure variable or the selection of the computation is fault dependent.

In general, the complex computational transfer condition for a node is fault dependent. This is true whether the node is correctly selected or not. As noted in Section 4, however, at a node that references more than a single potential failure variable, there may be several ways transfer occurs within the node. Some of these ways, for example those that involve masking out of all but one potential failure variable and then transferring the single potential failure variable using the simple computational transfer conditions through individual operators, may be fault independent. Consider the example noted previously, at the node representing the statement $A := (B * C) + D$ if B and D are potential failure variables, a sufficient condition that would transfer the potential failure to A is $c = 0$. This condition causes the subexpression potential failure in B to be masked out while the subexpression potential failure reflected in D computationally transfers to A. This condition is fault independent. (Recall that the simple computational transfer condition through the additional operator is simply *true*.) The ability to apply this approach, however, is completely dependent on the nesting structure of a node's expression. It is important to remember that inability to satisfy such a sufficient condition would not imply equivalence for the hypothetical fault since transfer at the node could still occur for some other sufficient condition. Investigation of sufficient fault independent conditions for complex computational transfer conditions is an area for further research. Such research might investigate the possibility of applying the results of Howden's Algebraic Testing [How78a] as well as other results in algebra theory.

In general, the transfer set condition is fault independent only when the transfer set consists of a single information flow chain. This is because, as shown in Section 2, any two information flow chains in a transfer set potentially interact. The node of this interaction will reference more than one potential failure variable and hence require complex computational transfer conditions.

When the transfer set consists of more than one chain, however, there may be transfer route conditions within the transfer set condition that are fault independent. As a result, there may be a sufficient transfer set condition involving a single transfer route that is fault independent. The transfer route condition is fault independent when the computational transfer (or non-transfer) condition at each node is fault independent. This is only possible for transfer routes where no interactions occur. In such cases, it is then only possible when the simple computational transfer conditions at each node are fault independent.

Consider again the module shown in Figure 5.2. If we are interested in hypothetical faults at node 2, then as noted previously, there is a single transfer set from node 2 to node 6 consisting of the chains:

$$(*, A, 2)(A, B, 3)(B, D, 5)(D, out, 6)$$
$$(*, A, 2)(A, C, 4)(C, D, 5)(D, out, 6).$$

There are three transfer routes associated with this transfer set. They are:

$$tr_1 : \ (A, B, 3); \neg(A, C, 4); (B, D, 5); (D, out, 6)$$
$$tr_2 : \ \neg(A, B, 3); (A, C, 4); (C, D, 5); (D, out, 6)$$
$$tr_3 : \ (A, B, 3); (A, C, 4); (B + C, D, 5); (D, out, 6).$$

The transfer route conditions for the first two transfer routes, tr_1 and tr_2, are fault independent, while the transfer route condition for tr_3 involves a complex computational transfer condition at node 5 and is fault dependent.

We see from this discussion that while in general transfer from an originating node may be fault dependent, in several cases, we may be able to construct a fault independent condition.

5.9 Summary of Application of RELAY to Construct Failure Conditions

One application of the RELAY model of faults and failures is construction of failure conditions that guarantee fault detection for a hypothesized fault. This application is presented in Chapters 4 and 5. While construction of the original state potential failure condition is quite straightforward, construction of the transfer set condition is extremely complex and involves identification of transfer sets and transfer routes and construction of computational transfer conditions. The application is not presented to suggest feasibility of constructing a failure condition. Rather, our investigation shows what would be needed to guarantee fault detection and demonstrates the complexity of the problem. Such a detailed analysis is not provided by any other fault-based testing model or fault-based testing criteria. An analysis of fault-based testing criteria using the insight provided from construction of the failure condition as well as a comparison of the RELAY model with other testing research is provided in the next chapter.

Chapter 6

Analysis and Discussion of Related Works

This chapter describes related fault-based testing approaches and discusses their relationship to the RELAY model and their ability to guarantee fault detection. In general, a review of related fault-based testing criteria shows that most only attempt to originate a potential failure. While several of these criteria succeed for some classes of fault, none in general introduce an original state potential failure. Only Morell's dynamic symbolic fault-based testing provides any information on transfer to output of an originated potential failure. This technique, however, does not incorporate control dependence in transferring a potential failure to output.

Section 1 of this chapter provides an overview of fault-based testing. The first subsection provides a broad categorization of testing approaches to provide a context for fault-based testing, and the second subsection provides a survey of the primary fault-based testing approaches and discusses their relationship to the RELAY model. In Section 2, we analyze three of these approaches for their ability to introduce an original state potential failure. Introduction of an original state potential failure is necessary but not sufficient to guarantee fault detection. In Section 3, we discuss the ability of fault-based testing criteria to reveal a failure, focusing on the works of Offutt and Morell, which are most closely related to RELAY.

6.1 Overview of Fault-Based Testing

6.1.1 Testing Approaches in General

Testing can in general be divided into two categories – black box testing and white box testing. Black box testing views the program or module being tested as a black box and uses only information about the input and output domain and the function computed by the program. Information about the actual implementation is not available to direct the test data selection process. An example of a black box testing technique is probabilistic or random testing, which randomly selects test data based on the run time distribution of the input domain.

White box testing techniques use information about the actual implementation of a module in selecting test data. RELAY is a white box testing approach. We may broadly divide white box testing approaches into three categories – coverage criteria, error (failure) based, and fault-based testing criteria.

Coverage criteria are probably the most well known and include criteria that direct selection of test data that executes some subset of all paths in a program (since in general covering all paths may be infeasible). These path selection criteria include statement testing, which requires each statement in a module be executed by at least one test datum in a set, as well as branch testing, which requires each branch be executed by at least one test datum in a set. In addition, these criteria include data flow path selection criteria, such as those described by Rapps and Weyuker, Laski and Korel, and Ntafos. Rapps and Weyuker [RW85] define several data flow criteria that direct the selection of test data that executes def-clear subpaths between definitions and uses. Their criteria select different subsets of these def-use pairs to be executed. For example, *all-p-uses*, selects a test data set that executes at least one def-clear path for each definition to each branch that uses the definition and each successor of the branching node. Ntafos defines a criteria called ***required k-tuples*** [Nta84] that selects a test data set that executes def-use chains of length k, where a def-use chain is an information flow chain that involves only data dependence links. Laski and Korel [LK83] define *[Ordered] Context Coverage*, which selects a test data set that executes all [ordered] definition contexts of a node, where a [ordered] definition context of a node is a [sequence] set of definitions of variables used at the node that reach the node along the same initial subpath. These coverage criteria are directed towards selection of paths rather than selection of particular test data for those paths. As a result, while these criteria would select test data that would execute part of a chain of data dependence, they do not direct selection of test data that would necessarily transfer at each link in the chain. For a more complete discussion of path selection criteria see [CPRZ86]. For a more complete discussion of data flow path selection criteria and syntactic dependence see [Pod89].

Another group of white box testing techniques are error (or failure) based approaches. These techniques attempt to select test data to detect "errors" in a program. Symbolic evaluation is a program analysis method that provides a functional representation of a program's output in terms of symbolic input. It represents both the subdomains partitioned by the program and the computations associated with each subdomain using symbolic expressions. One example of an error based technique is domain testing [WC80, CHR82], which attempts to detect errors in a path domain by selecting test data on and near the boundaries of the domains. When there is an error in a border of a path domain, then selection of test data on and near a border results in different path domains being selected. A border shift may be caused by a fault in the branching node that defines the border or by a fault in a node that assigns a value upon which the branching node is directly (or indirectly) data dependent. For such a fault, selection of a different path domain at the border defined by the branching node would cause origination of a potential failure for the fault and transfer of that potential failure through to the branching node, which includes computational transfer at the faulty node and data dependence transfer along some chain up to the branching node. Domain testing, however, assumes that the computations performed along all branches are distinct, and thus assumes transfer from the branching node to output rather than guaranteeing it.

Fault-based testing techniques, the focus of this thesis, are the final group of white box testing techniques. Fault-based testing criteria, as well as other testing criteria, are founded on the concepts of reliable and valid introduced by Goodenough and Gerhart [GG75]. A criterion is reliable if every test set selected by the criterion produces the same result, correct or incorrect, on execution of the module. That is, if any test set selected by a reliable criterion detects some fault, then all test sets selected by the criterion would detect the fault, and if any test set selected by the criterion fails to detect some fault, then all test sets selected by the criterion would fail to detect the fault. A criterion is valid if it does not prohibit selection of test data that would detect some fault, and hence some test data set selected by the criterion could detect the fault. The term reliable is redefined by Howden and as defined in [How76] incorporates both the concepts of reliable and valid. A test set is said to be reliable for a program if it causes a failure whenever the program is incorrect. If a test set is reliable for a program and the program executes correctly on the test set, we may infer the program is correct. As shown in [GG75, How76] and by several other researchers, no computable procedure exists to generate a finite test data set that is reliable for a program.

Given this result, Howden takes the approach of investigating the classes of programs and program faults for which reliable test sets can be developed. In [How78a], Howden demonstrates that for several classes of functions, if we assume a program computes some function in one of those particular classes, we can select test data that distinguishes the program from all other programs that compute functions in the class. Such a test data set would be reliable under the assumption of the program being in the class. Hamlet [Ham77] takes a similar approach using the concept of alternative set and requiring selection of test data to differentiate between the program and all alternate programs. Mutation analysis[DLS79] evaluates a test set for its ability to distinguish between a program and alternate programs formed by altering individual expressions in the program. These approaches rely on the "competent programmer hypothesis", which says that a program being tested is either correct or pretty much correct [DLS78, ABD+79]. This approach of distinguishing a program from related alternates is formalized by Budd and Anguin. In [BA80], they investigate the problem that testing cannot show correctness of a program by considering another definition of correctness. This concept of correctness considers correctness of a program within a "neighborhood" of alternate programs. If we assume that a program is correct or if it is incorrect then some program within the neighborhood of the program is correct, then a test data set that distinguishes between the program and all programs in the neighborhood is reliable for the program. The concept of neighborhood testing forms the basis for fault-based testing, where the neighborhood for a program is defined by a set of programs that can be generated by applying a set of hypothetical faults to the program.

6.1.2 Primary Fault-Based Testing Approaches

Formal attempts at fault-based testing fall into two categories: those that evaluate the adequacy of user-selected test data to detect faults and those that guide in the selection of test data to detect faults. In what follows, we first discuss several fault-based test data evaluation techniques and then describe several fault-based test data selection techniques.

The earliest formalized fault-based testing techniques were introduced independently by Hamlet and by DeMillo, Lipton and Sayward. Both techniques *seed* or substitute particular types of faults into the program being tested and evaluate the adequacy of a user-selected set of test data in terms of its detection of the seeded faults. In each of these approaches, the test data set must be augmented iteratively to eliminate the seeded faults. These two evaluation approaches do not provide guidance to how to select test data to distinguish the hypothetical faults.

In addition, both require evaluation of both the original and the mutant program. The philosophy behind these approaches is the belief that the process of finding all seeded faults also eliminates real faults in the source code. This belief is discussed more in Chapter 7.

Hamlet proposed an approach called *testing with the aid of a compiler*[Ham77], whereby faults are seeded into a program by an extended compiler. Rules define faults of expressions in the source code as alternative expressions that are "smaller" than the original expression. The compiler instruments the code to compare the values computed by each alternate and the corresponding original expression. The adequacy of a test data set is then judged based on whether the alternative expression with the seeded fault evaluates differently than the original expression. For the class of seeded faults, this adequacy measure is enough to determine whether test data causes the origination of a potential failure but is not, in general, sufficient to determine if a failure is produced.

DeMillo, Lipton, and Sayward introduced *mutation analysis*[DLS79], which takes a slightly different approach to fault-based testing. The mutation analysis system introduces one fault at a time into the source code to produce a "mutant" program; many different fault types for each location in the program are considered, thus creating lots of mutant programs. The system then executes the original and mutant programs on the user's test data set to determine whether they produce different output results on at least one test datum. If so, the mutant is "killed". The adequacy of the test data set is measured based on whether all non-equivalent mutants are eliminated. As such, mutation analysis determines whether test data causes both origination of a potential failure and transfer to produce a failure for each seeded fault but requires execution of both mutant and original to determine this.

These two approaches, as noted, require explicit construction and execution (or at best partial interpretation) of many alternate programs. Two other evaluation techniques take a more mathematical approach. Rather than evaluate a set of test data for its ability to eliminate faults by execution, the techniques proposed by Zeil and by Morell evaluate a set of test data and determine what faults could exist in the program and remain undetected by execution on the test data.

Zeil's *perturbation testing* [Zei83, Zei84] examines ways in which statements in a program could be perturbed or modified and not be detected by the current set of test data. A functional description of a "perturbation" at a specific statement provides an error term that is evaluated over the current state of execution. Any zero-valued solution to this error term describes a fault that would not be detected by this execution. Perturbation testing, therefore, identifies potential faults of a particular functional class that would not introduce an original state potential failure for user-selected test data. In addition, perturbation testing determines if the output is partially dependent on the error term, thus checking to see that an original state potential failure could transfer to produce a failure; such transfer is not, however, guaranteed since the requirements are not explicit.

Morell describes a model of fault-based testing [Mor84], in which he introduces the ideas of "creating" an initial "error" for a fault and "propagating" it to the output. This model provides the basis for our initial work with RELAY. In his model, Morell defines a creation condition that distinguishes between an expression and an alternate expression formed by applying some transformation to the expression. The creation condition introduces an incorrect state, which is defined at statement boundaries. Thus, the creation condition is similar to RELAY's original state potential failure condition. The propagation condition guarantees that once an incorrect state is introduced, it persists to some point where it can be detected. In this general model, however, how propagation proceeds is not detailed.

One application of his model, described by Morell as *symbolic fault-based testing* [Mor88], provides a technique similar to perturbation testing. This approach replaces an expression within a statement in a program with a "symbolic" fault or alternate. Execution of the program on a user-specified test datum provides a "[specific] propagation equation", which contains the symbolic fault. The original program and the program with a symbolic fault seeded at some expression may also both be "symbolically evaluated" along a particular path — that is, interpreted for symbolic test data rather than actual values. Comparison of the symbolic computation for the original program and the program with the symbolic fault yields a "[general] propagation equation". Both the specific propagation and the general propagation equations may be used to determine hypothetical faults that would not both originate a potential failure and transfer that potential failure to output for user-selected test data, provided the original and the alternate follow the same path. Thus, Morell makes the assumption that the program with the symbolic fault and the program without the symbolic fault execute the same paths. This implies that data dependence transfer at any branching node must fail. As a result, control dependence transfer is not considered. The work of Morell is discussed in more detail in Section 3 of this chapter.

One difficulty of these fault-based test data evaluation approaches is that they do not provide much guidance as to how to select test data that eliminate the faults considered. A number of researchers have developed fault-based techniques that more directly guide the test data selection process. Their works are now outlined. A more complete discussion of the actual fault detection capabilities of specific rules of several of these techniques is provided in Section 2 of this chapter.

Foster introduced the idea of conditions under which a fault manifests itself as an erroneous value [Fos80, Fos83, Fos84, Fos85]. Foster's *error-sensitive test case analysis (Estimate)* consists of conditions that are sufficient to distinguish expressions that may contain a fault from the correct expression for several classes of faults. These conditions are often sufficient to originate a potential failure, but do not guarantee computational transfer at the faulty node to produce an original state potential failure.

Howden clarified these conditions and introduced others in his *weak mutation testing* [How82]. Weak mutation testing has become part of a larger approach called *fault-based functional testing* [How87, How85]. Traditional functional testing requires selection of test data that exercise the input and output domains for all explicit functions of a program over a variety of requirements. Howden extends the concept of a function to include implicit functions defined as units of related computations in a program and applies fault-based and functional testing to these functions. These implicit functions may be expressions, single statements, or groups of statements. It is difficult to evaluate this approach because little guidance is provided in identifying the different levels of functions that should be tested, other than individual expressions in statements. The specific fault-based functional testing rules for individual expressions are often sufficient to originate potential failures for several types of faults at functions at the level of subexpressions, and in some cases, his rules will select test data that produces an original state potential failure in functions at the level of statements. Howden's approach assumes that an oracle exists at that level of function and thus transfer to a larger expression or output is not considered, however, there is no reason to assume such oracles exist. Fault-based functional testing [How85, How87] augments this low-level testing by test data selection rules that are applicable to the synthesis of functions from component functions. This synthesis relies on the correctness of the individual components, which as stated above cannot be assumed because of the very real possibility of no oracle.

Two extensions to mutation analysis that are oriented toward test data selection have been proposed. Both these extensions, it should be remembered, are not intended to serve as test data selection criteria alone, but rather have been developed to assist in the selection of mutant adequate test data. In his mutation testing suite, Budd includes a component called *error-sensitive test monitoring* [Bud83], which consists of conditions that must minimally be satisfied to detect some of the classes of mutants in expressions containing them. These conditions are often sufficient to originate a potential failure but not to transfer a potential failure to output. While Budd recognizes the need for transfer in other levels of his mutation testing suite, these levels provide no guidance in selecting test data.

Offutt has described *constraint-based testing* [Off88] as a part of Godzilla, the test data generation component of the Mothra mutation analysis system. This approach generates the "necessary" conditions, which are similar to origination conditions, for detecting the mutants in the expressions containing them. Offutt recognizes the need for the originated potential failure to transfer to output and satisfy what he calls the "sufficiency conditions", but no guidance is provided for selecting such test data. The system selects test data to satisfy the "necessary" conditions and as with mutation analysis, execution of the program on such test data is compared with execution of the mutant program to determine if the mutant has been killed; otherwise, different test data is tried.

6.2 Analysis for Revealing an Original State Potential Failure

The conditions developed to guarantee introduction of an original state potential failure may be used to evaluate capabilities of a test data selection criterion to introduce an original state potential failure. A test data selection criterion is usually expressed as a set of rules that test data must satisfy. Our analysis approach used here evaluates a criterion in terms of the relationship between its rules and the original state potential failure conditions defined by RELAY for the six fault classes. A rule or combination of rules is judged either to be insufficient to cause an original state potential failure and hence also a failure, to be sufficient to introduce an original state potential failure, or to guarantee introduction of an original state potential failure. This analysis is completely program independent.

In this section, we use the origination and simple computational transfer conditions for the six fault classes developed in Chapter 4 to analyze the fault detection capabilities of three fault-based test data selection criteria — Budd's *Error-Sensitive Test Monitoring* [Bud81, Bud83], Howden's *Weak Mutation Testing* [How82, How85], and Foster's *Error Sensitive Test Case Analysis* [Fos80, Fos83, Fos84, Fos85]. Each of these criteria was selected because its author claims that it is geared toward detection of faults of the six classes previously discussed.

The analysis here examines the abilities of these fault selection criteria to introduce an original state potential failure. The original state potential failure condition is necessary for the detection of a fault but not sufficient. This is because the original state potential failure introduced by satisfaction of these conditions may still be masked out by later computations on the path and thus not transfer to produce a failure. Our analysis shows that none of the criteria guarantees detection of these types of faults.

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

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

The following is not intended to be a complete analysis of the fault detection capabilities of these criteria. Only those faults for which origination and simple computational transfer conditions are developed in Chapter 4 are included in the discussion. A complete analysis must consider a more complete classification of faults.

6.2.1 Budd's *Estimate*

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

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

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

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

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

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

a. $exp \neq v$.

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

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

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

Rule 4 For each edge $(n, m) \in E$, where $BP(n, m)$ is the branch predicate, T contains a test datum t_a such that:

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

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

a. $exp_1 - exp_2 = 0$;

b. $exp_1 - exp_2 > 0$;

c. $exp_1 - exp_2 < 0$;

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

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

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

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

a. $exp_1 > 2$.

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

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

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

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

Estimate's rule 2 is directed toward the detection of variable definition faults. A test datum that satisfies this rule fulfills the origination condition set. The origination condition set, however, contains another condition, $(\overline{v} \neq v)$, that must be satisfied if $(exp \neq v)$ is infeasible. *Estimate* does not satisfy this other condition, and thus a variable definition fault may remain undetected by *Estimate*. Consider the following example:

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

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

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

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

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

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

Estimate's rule 5 captures the general sufficient transfer conditions shown in Chapter 4 in Table 4.14, although *Estimate* does not consider the assumptions noted there. Even if these assumptions were taken into account, one of these sufficient conditions is not by itself sufficient to guarantee transfer through a relational operator. Consider the relational expression in the following code segment:

```
1  read $X, Y$;
2  if $X*Y \geq 10$ then
       ⋮
```

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

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

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

where a subexpression potential failure originates in X at statement 3. No test datum satisfies rule 6 for this node; however, a test datum such that $y \neq 0$ transfers any subexpression potential failure in X. Thus, *Estimate* is sufficient to transfer through most but not all arithmetic operators but does not guarantee transfer.

We are now in a position to determine the ability of *Estimate* to guarantee introduction of an original state potential failure for the six fault classes. Rules are not combined by *Estimate*. Thus, in general, test data selected by *Estimate* that satisfy rules that satisfy origination conditions are not guaranteed to also satisfy rules that satisfy simple computational transfer conditions, and thus transfer of an

Table 6.1. Sample Test Data Selected by *Estimate* for $(A < B)$ **or** Z

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

originated subexpression potential failure is not guaranteed. As an example, consider introduction of an original state potential failure for a hypothetical relational operator fault in the expression $(A<B)$**or**Z (assume for simplicity that A and B are of type integer). The test data shown in Table 6.1 satisfies *Estimate*'s rules 3, 4 and 5 for this expression. Test data i, ii, and iii satisfy rule 5 for the relational expression containing the operator $<$. If this relational operator should have been any other relational operator, this test data would originate a subexpression potential failure; for these test data, however, z =*true*, which will not transfer any subexpression potential failure. Test data iii and iv satisfy rule 3 for the outer boolean expression containing **or**. Data v and vi satisfy rule 4 for the conditional statement. Test data iv and vi are the only data that would transfer any subexpression potential failure originated in the relational expression; these data alone, however, are insufficient to guarantee origination of a subexpression potential failure for the hypothetical relational operator fault. If, for example, the $<$ should be $\leq$, no selected datum both originates and transfers a subexpression potential failure caused by this fault. Thus, *Estimate* does not guarantee introduction of an original state potential failure for this hypothetical relational operator fault.

This may even occur when the same rule satisfies both the origination condition for a hypothetical fault as well as the simple computational transfer conditions that

Table 6.2. Sample Test Data Selected by *Estimate* for (X **and** Y) **or** Z

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

could be required. This is illustrated in the application of rule 3 to the boolean expression (X **and** Y) **or** Z in the following code:

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

The test data shown in Table 6.2 satisfies *Estimate*'s rule 3 for the conditional expression in this example. Test data i and ii satisfy rule 3 for the inner boolean expression containing the operator **and**. Test data iii and iv satisfy rule 3 for the outer boolean expression containing **or**. If the inner operator should have been an **or**, test data i and ii would originate a subexpression potential failure. For these test data, however, z =*true*, which will not transfer any subexpression potential failure. Test data iii and v are the only data that would transfer a subexpression potential failure originated at the inner expression, but for these test data, the values x and y would not originate a subexpression potential failure. Thus, *Estimate* does not guarantee introduction of an original state potential failure for a hypothetical boolean operator fault.

When origination of a potential failure is guaranteed for a class of hypothetical faults, introduction of an original state potential failure is guaranteed by *Estimate* only when the simple computational transfer conditions are trivial. In general, this occurs when the smallest expression containing the hypothetical fault is the outermost expression in the node. The simple computational transfer conditions are always trivial for a variable definition fault. Since *Estimate* is sufficient to originate a subexpression potential failure for this class, it is also sufficient to introduce an original state potential failure. Recall, however, that *Estimate* does not guarantee origination for this class.

6.2.2 Howden's *Weak Mutation Testing*

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

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

a. $v \neq \overline{v}$[1] .

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

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

a. $v \neq exp$.

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

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

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

b. $exp_1 - exp_2 = 0$;

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

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

a. the expression is executed;

b. $exp \neq 0$.

Rule 6 For each arithmetic expression EXP, where k is an upper bound on the exponent in the *exp*, at node n, T contains test data $t_1, t_2, \ldots t_{k+1} \in$ *DOMAIN* (n) such that $\{t_1, t_2, \ldots t_{k+1}\}$ is any cascade set of degree $k + 1$ in *DOMAIN* (n).

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

Rule 1 is obviously directed toward detection of variable reference faults, and a test datum that satisfies this rule does satisfy the origination condition set. This rule, however, is more restrictive than required for this class of faults; it requires a single test datum to distinguish between the hypothetically incorrect variable reference and all other variable references. This rule may not be satisfiable although the origination condition set is feasible. In this case, a non-equivalent alternate may not be distinguished. Thus, *WMT* is sufficient to originate a subexpression potential failure, therefore, but does not guarantee origination for variable reference faults.

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

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

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

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

Next, consider the ability of *WMT* to transfer a subexpression potential failure. Rule 3 selects data that satisfies the simple computational transfer conditions for boolean operators.

WMT's rule 4 is similar to the sufficient simple computational transfer conditions for relational operators shown in Table 4.15. For these transfer conditions to be sufficient, the two assumptions noted there in Chapter 4 must also hold. *WMT* does not consider these assumptions. Hence, even when *WMT*'s relational operator rule is satisfied, a subexpression potential failure may not transfer through a relational operator. Thus *WMT* is insufficient to transfer a subexpression potential failure through a relational operator.

Rule 5 satisfies the simple computational transfer conditions for all arithmetic operators but the exponentiation operator. In cases where the degree of an expression is unknown, rule 6 cannot be used to select a proper cascade set and hence will not guarantee transfer through arithmetic operators. *WMT*, therefore, only partially guarantees transfer through arithmetic operators.

As with *Estimate*, *WMT* origination and transfer are not guaranteed to be satisfied by the same test datum, and hence introduction of an original state potential failure is not guaranteed. As with *Estimate*, this may happen both when the same rule applies for origination as for transfer and when different rules apply. In sum, Howden's *WMT* guarantees introduction of an original state potential failure when origination of a subexpression potential failure is guaranteed for a class of hypothetical faults and the simple computational transfer conditions are trivial. Only for variable definition fault are the simple computational transfer conditions always trivial. *WMT* is sufficient to originate a subexpression potential failure for this class and hence is sufficient to introduce an original state potential failure.

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

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

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

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

Rule 2 For each variable V input at node n and some edge(n, m), T contains test data $t_a, t_b \in DOMAIN\ (m)$ such that the value input for V at node n is:

a. $v_a > 0$;

b. $v_b < 0$.

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

Rule 3 For each logical unit L [2] of each boolean expression $EXP = (\ldots L \ldots)$ at node n, let $EXP^* = (\ldots \neg L \ldots)$, T contains test data $t_a, t_b \in DOMAIN\ (n)$ such that:

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

a. $l = true$ and $exp^* = \neg exp$ [3];

b. $l = false$ and $exp^* = \neg exp$.

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

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

b. $exp_1 - exp_2 = 0$;

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

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

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

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

Foster's *ESTCA* guarantees origination for a boolean operator fault. Rule 3 considers a boolean expression in terms of logical units. A logical unit is a variable or relational expression that is one of the operands or is a subexpression of one of the operands of a boolean expression (EXP_1 **bop** EXP_2). *ESTCA* requires selection of test data such that each such logical unit takes on the value *true* (and the value *false*) and complementing the logical unit complements the entire boolean expression. This rule satisfies the origination condition sets for boolean operator faults. To see this, notice that for any boolean expression EXP_1**bop**EXP_2, three test data are selected,

[3]that is, substituting $\neg$L in EXP complements the value of EXP.

(exp_1, exp_2) = (T,F), (F,T), and (T,T) if **bop** is **and**, or (F,F) if **bop** is **or**. This test data satisfies origination conditions for a boolean operator fault. Thus, *ESTCA* guarantees origination of a subexpression potential failure for the class of boolean operator faults.

Consider now the class of relational operator faults. When satisfiable, *ESTCA*'s rule 4 results in data such that $exp_1 > exp_2$, $exp_1 = exp_2$, $exp_1 < exp_2$. Thus, test data satisfying this rule will originate a subexpression potential failure for hypothetical relational operator faults. This rule, however, is more stringent than required and may be unsatisfiable while the origination condition set is feasible. Thus, *ESTCA* is sufficient to originate a subexpression potential failure for relational operator faults but does not guarantee origination of a subexpression potential failure for relational operator faults.

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

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

Rule 4 is similar to the general sufficient simple computational transfer conditions for relational operators. Like Howden, however, Foster does not consider the assumptions that must hold for these conditions to be sufficient for transfer. Moreover, rather than specifying ϵ to be the smallest satisfiable difference, Foster fixes ϵ at the smallest representable magnitude. As a result, the ability of *ESTCA* to transfer a potential failure through a relational operator is further limited. Thus, *ESTCA* is insufficient to transfer a subexpression potential failure through a relational operator.

Rule 5 attempts to disallow the effect of a variable or subexpression to be masked out by other operations in the statement. While the specifics of how this rule is applied are unclear, one might interpret this as requiring transfer of a subexpression potential failure through arithmetic operators. Under the broadest interpretation, therefore, *ESTCA* guarantees transfer through arithmetic operators.

As with the other criteria, the rules in *ESTCA* that satisfy origination and those that satisfy transfer may not be satisfied by the same test datum. Rule 3, however, does guarantee an original state potential failure is introduced for boolean operator faults. As seen above, this rule satisfies the origination and simple computational transfer conditions for relational operator faults. In addition, when applied to the outermost boolean expression, this rule selects a single datum for each nested binary boolean expression that originates a subexpression potential failure due to a hypothetical fault in the associated boolean operator and transfers that potential failure to the outermost expression. To see this, consider any expression $EXP = EXP_1 \, \mathbf{bop} \, EXP_2$. Some test datum selected for logical units within EXP_1 fulfills the origination condition for hypothetical boolean operator faults in EXP_1. Complementing a test datum selected for a logical unit that is a subexpression of EXP_1 must complement the value *exp*. To force this, if $bop =$ **and** then $exp_2 = true$, or if $bop =$ **or** then $exp_2 = false$. Thus, for any test datum selected for a logical unit that is a subexpression of EXP_1, EXP_2 will take on a value that will transfer any subexpression potential failure originated within EXP_1 to the outer expression EXP. Therefore, *ESTCA*'s boolean operator rule satisfies origination as well as transfer conditions simultaneously and hence guarantees introduction of an original state potential failure for boolean operator faults.

6.2.4 Summary of Analysis

Table 6.3 summarizes the analysis of the three test data selection criteria. The entry insufficient means that the criterion does not include a rule that satisfies the condition. The entry sufficient means that the criterion includes a rule that when satisfied fulfills the condition. The entry partially sufficient means that the criterion includes a rule that is sufficient to distinguish many but not all of the alternates or transfer through many but not all of the operators. The entry guarantees means that the criterion includes a rule that satisfies the conditions when the conditions are feasible, while partially guarantees means the criterion includes a rule that satisfies many but not all of the conditions when feasible.

6.3 Analysis for Revealing a Failure

This chapter discusses the ability of fault-based testing criteria to reveal a failure for a specific fault. The criteria discussed above do not consider transfer of a potential failure and moreover do not guarantee introduction of an original state potential failure and thus cannot guarantee failure. Therefore, we do not consider them further here. Here we discuss Offutt's constraint-based-testing and Morell's symbolic fault-based testing in more detail. These two testing approaches are the most related to RELAY and the only ones that provide any significant discussion of guaranteeing transfer of a subexpression potential failure to output.

Table 6.3. Analysis Summary

	Estimate	*WMT*	*ESTCA*
Origination			
1. Constant Reference Fault	guarantees	guarantees	guarantees
2. Variable Reference Fault	insufficient	sufficient	insufficient
3. Variable Definition Fault	sufficient	sufficient	insufficient
4. Boolean Operator Fault	guarantees	guarantees	guarantees
5. Relational Operator Fault	guarantees	guarantees	sufficient
6. Arithmetic Operator Fault	partially sufficient	partially guarantees	insufficient
Simple Computational Transfer			
1. Assignment Operator	guarantees	guarantees	guarantees
2.Boolean Operator	guarantees	guarantees	guarantees
3.Relational Operator	insufficient	insufficient	insufficient
4.Arithmetic Operator	partially sufficient	partially guarantees	guarantees
Original State Pot. Failure			
1.Constant Reference Fault	insufficient	insufficient	insufficient
2.Variable Reference Fault	insufficient	insufficient	insufficient
3. Variable Definition Fault	sufficient	sufficient	insufficient
4. Boolean Operator Fault	insufficient	insufficient	guarantees
5. Relational Operator Fault	insufficient	insufficient	insufficient
6. Arithmetic Operator Fault	insufficient	insufficient	insufficient

6.3.1 Offutt's *Constraint-Based Testing*

Offutt describes a fault-based testing approach called constraint-based testing [Off88] in which he defines a reachability condition, which is the domain of a hypothetically faulty node as defined in RELAY, a necessity condition, which is similar to Morell's creation condition and RELAY's origination condition, and he defines a sufficiency condition, which is similar to Morell's propagation condition and is the condition to transfer from the originated subexpression potential failure to output. No details are provided of the nature or structure of the sufficiency condition.

The primary result of Offutt's work was support for automated generation of test data sets that satisfy the necessity condition for the Mothra mutation testing system. No assistance, however, is provided for determining sufficiency conditions or for generating test data that satisfies the sufficiency requirement. Offutt argues that the possibility that an intermediate result once introduced will be masked out at a later computation is unlikely. As stated in his dissertation, his argument follows.

> ...
>
> For the necessity condition but not the sufficiency conditions to be met then the mutant does not die and the output of M is identical to that of P. Since the necessity is met, then the state of the mutant program M does diverge from the original program P at the point following the mutated statement. That is, the state of M first diverges from P, then returns to that of P.
>
> This divergence and convergence could happen in one of three ways. First, the test case could be weak — though it had an effect, it did not change the state in a way that would result in a change in the final output. Secondly, the program could be robust enough to recognize the intermediate state as being erroneous and return to a correct state. Thirdly, the intermediate state that the mutant effected could be irrelevant to the final state, in which case the constraint is derived from an equivalent mutant. Although each of these cases is certainly possible, the first seems unlikely – that is, it should occur with relatively low probability. The second case corresponds to fault tolerance in the tested software; the case is interesting enough to warrant further investigation on its own merits. In the absence of fault tolerance however, such robustness is probably rare. The third, of course, provides an opportunity to detect some equivalent mutants, a difficult problem in its own right.

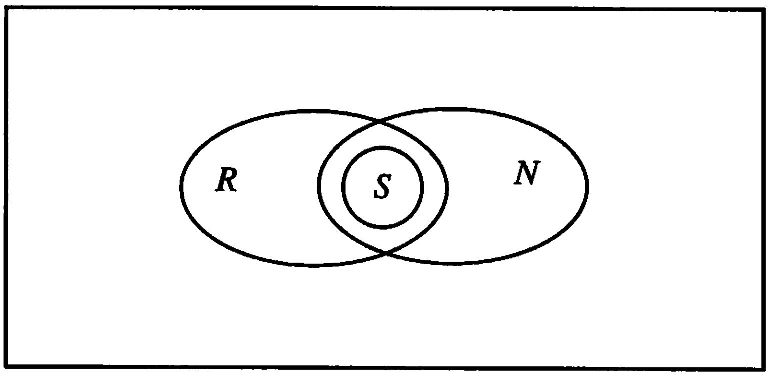

Figure 6.1. Subsets of Input Domain

It is the first case in which we are interested, which Offutt claims is unlikely. To support this claim, Offutt uses a probabilistic argument. His argument is summarized here. Consider Figure 6.1, taken from his dissertation, and three subsets of the input domain, R, N, S, where R is the set of all test data (from the input domain) that reaches the statement containing the mutant, N is the set of all test data that satisfies the necessity conditions, and S is the set of all test data that satisfies the sufficiency and the necessity and reachability conditions. R and N intersect. S is a subset of their intersection. Offutt argues that we are interested in selecting test data from S and asks what is the probability that test data that satisfies reachability and necessity conditions (from the intersection of R and N) does not also satisfy the sufficiency conditions? This probability, which he denotes as p, is the probability that a test datum that is in $R \bigcap N$ is also in S. He notes that p is not constant over all programs but depends on the program and the mutant. Offutt then asks what is the probability of selecting a test datum in S if it is in $R \bigcap N$ at least once in n attempts. He notes that this is the same as not failing to cause a failure, given an incorrect state has been introduced by the mutant, every time in n attempts. The probability of failing on n attempts is $(1-p)^n$. The probability of at least one "success" is $\Gamma = 1-(1-p)^n$. He argues that Γ grows very quickly for positive p (which p must be). This, he continues, says that with even a very low probability of detecting a mutant (selecting a test datum in the set S), if we select a few test cases that satisfy the reachability and the necessity conditions, there is a high probability of also satisfying S. Thus, Offutt argues that multiple selection of test cases that satisfy the reachability and the necessity conditions implies a solution to the sufficiency problem unnecessary.

This same argument, however, can be applied to selection of test data that satisfies any portion of the failure condition, for example, just the reachability condition or perhaps the input domain in general. To see this, consider again the set S which detects a fault. S is a subset of the input domain. There is some probability q that a test datum in S is selected if the test datum is in the input domain. The probability that this test datum fails to detect the fault (is not in S) is $1 - q$. Asking the same question as above, in n attempts, what is the probability that failure occurs each time. This is again $(1 - q)^n$. Thus as above the probability that success occurs at least once in n attempts is $\Gamma = 1 - (1 - q)^n$. As before, Γ here also grows quickly, though perhaps not as quickly, since it is the same equation. Thus, the same argument that Offutt uses to conclude multiple test cases that satisfy reachability and necessity is enough to solve the sufficiency problem may be used to conclude multiple test cases solves the testing problem and a tester need not spend resources determining and satisfying the reachability or necessity conditions or any criterion for that matter. Essentially, the conclusion is that the more test data the higher probability of detecting a fault. We do not disagree, but the use of the probabilistic argument tells us nothing of value about the likeliness or nature of transfer from an originated potential failure to output.

In his dissertation, Offutt also includes some empirical results. These results are discussed more in the next chapter. Briefly, he empirically approximates p, the probability that if a test case satisfies the reachability and necessity conditions it will also satisfy the sufficiency condition, to be between .61 and .94. He uses this result to conclude that only a few multiple test cases per constraint can be used to achieve high mutation score. Because, however, he does not investigate why transfer fails, it is incorrect to suggest that empirical approximation of p provides empirical evidence that selecting multiple test data solves the sufficiency problem. Doing so simply reuses the flawed probabilistic argument above.

Our problems with Offutt's reasoning should not be interpreted as implying that the question of what portions of the failure conditions are often met by random test data is not important. In fact, quite the contrary is true. One thing that the RELAY model and construction of the failure condition show is how difficult it is to guarantee fault detection. Given this, the question is what portions of the failure condition can we construct and satisfy and what is the meaning of satisfying those portions. Such questions are not, however, in any way answered by the probabilistic argument and empirical work cited above.

6.3.2 Morell's *Symbolic Fault-Based Testing*

The RELAY model builds on the theoretical model developed by Morell in [Mor84]. Thus, it is particularly important for us to clearly present the differences between the failure condition based on RELAY and symbolic fault-based testing based on Morell's creation/propagation model.

Morell's model or framework is applied in what he calls symbolic fault-based testing. This approach seeds a symbolic fault and symbolically executes the module either with specific input or on symbolic input providing propagation equations. These equations are used to determine the adequacy of a test data set by determining which symbolic faults would not be distinguished. It is difficult to evaluate the propagation equation approach because in several areas, sufficient details for how a propagation equation is constructed are not provided. Nonetheless, given the complexity of constructing failure conditions and the fault dependence of these conditions, symbolic fault-based testing has several strengths and is a promising area of research. It has, however, several weakness. We discuss those now along with some ideas as to how the information provided by RELAY might be combined with symbolic fault-based testing to strengthen the approach.

This first major weakness is that the importance of control dependence in 'error-propagation', or in RELAY terms information flow transfer, is lost. Morell notes briefly in an example [Mor84, Mor88] that for the propagation conditions to hold, the same paths must be taken in both the module containing the symbolic fault and the original module. This says then that data dependence transfer must fail at branching nodes and thus there can be no subsequent control dependence transfer. Additional analysis must be performed to determine what faults would execute the same paths for the test data being evaluated. How such an analysis would be performed is not provided. In addition, it appears that Morell's approach is unable to evaluate test data where the only chains from a fault to failure include control dependence. Consider the example module shown in Figure 6.2. From node 2 to node 6, there are two transfer sets. One consists of the single chain $(*, A, 2)(A, B, 3)(BP, B, 4)(B, out, 6)$; the other consists of the single chain $(*, A, 2)(A, B, 3)(BP, B, 5)(B, out, 6)$. A symbolic computation for B would not include a symbolic fault at node 3. Here the dependence of B on A is not captured since it only occurs through control dependence. No test data set evaluated using Morell's approach could be adequate to detect a fault at node 2. Extension to Morell's work to capture control dependence would seem feasible by including a symbolic path conditions with the symbolic computation. How this path condition would be used in conjunction with the symbolic computation needs further investigation.

A second weakness of Morell's work is a lack of guidance for selecting test data that would originate and transfer a potential failure to output when a test data

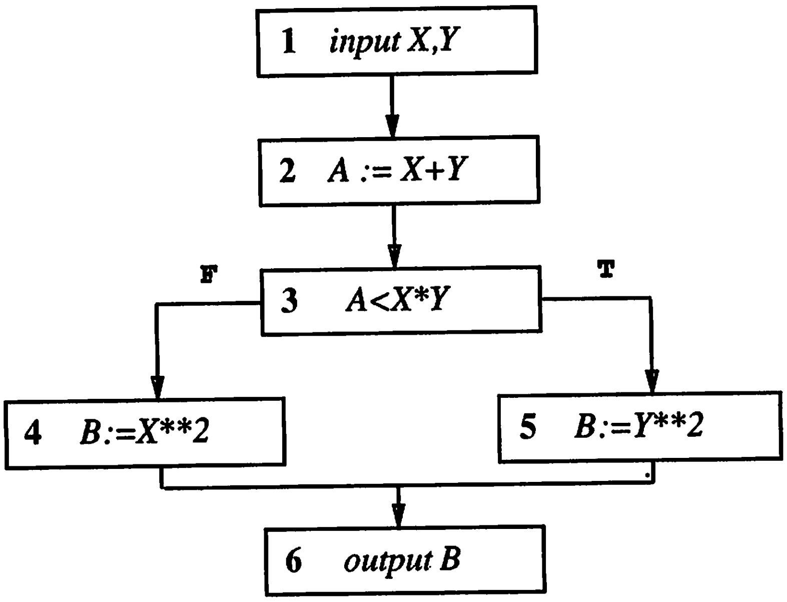

Figure 6.2. Control Dependence Only

set is determined inadequate. If the same path(s) is selected (and thus control dependence transfer cannot occur) and if there is only a single occurrence of the symbolic fault in the symbolic computation, then the simple computational transfer condition could be applied to this expression. Care must be taken, however, that the symbolic computation equation is not simplified as some transfer information could be lost. In cases where a symbolic computation contains more than a single reference to the symbolic fault (and again the same path(s) is selected) it may be possible to apply the technique that derives a sufficient complex computational transfer condition by masking out some of the potential failure variables within a node described in Chapter 5. Hybridization of the transfer ideas in RELAY with the symbolic fault-based testing ideas of Morell is an interesting area for further research.

6.3.3 Summary of Analysis

A review of fault-based testing criteria shows that few consider transfer of an original state potential failure to output. Only a few criteria discuss this need, and only Morell's symbolic fault-based testing provides any guidance for doing this. Symbolic fault-based testing can evaluate a test datum for its ability to reveal a failure for different faults, but only if transfer involves only data dependence.

CHAPTER 7

DISCUSSION OF RELATED EMPIRICAL STUDIES

The analytical work presented in the preceding chapters shows how difficult it is to guarantee fault detection for a single fault in a module. To balance the analytical research, empirical work must be performed. One of the values of the RELAY model is that it provides insight into the process of transfer of a potential failure to output, allowing us to ask interesting and important empirical questions about this process in real programs. This chapter discusses two issues of interest related to the RELAY model. The first issue regards the likelihood of information flow transfer from an original state potential failure to output by data not specifically selected to satisfy a transfer set condition. The second issue examines the problems introduced by multiple faults in a module. For each of these issues, we evaluate related empirical results from the literature and outline further studies.

7.1 Information Flow Transfer

As described by RELAY, to detect a fault, a test datum must execute the faulty expression, originate a potential failure, and transfer the potential failure to output. The *weak mutation hypothesis* [Mar90] states that a test case that executes a faulty expression and originates a potential failure will usually transfer the potential failure to output. Clearly from the examples in the previous chapters, such test data would not always satisfy the transfer conditions. The question is, however, how often does the weak mutation hypothesis hold for real programs. This question has been studied by Offutt and by Marick, whose studies we discuss below. A review of these studies indicates the need for further investigation of the issue of transfer. In particular, we suggest empirically studying how the likelihood of transfer may differ through different structures of information flow.

7.1.1 Related Empirical Studies

Offutt reports the results of several empirical studies in [Off88]. For these studies, he used Godzilla, the test data generation system in the Mothra mutation system [DGK+88] to study a test suite of five Fortran-77 programs. The study of interest examined the "precision", P, of test cases, which Offutt claims to estimate p. Recall from Chapter 6 that p is the probability that a test datum that satisfies the reachability condition (executes the fault) and the necessity condition (originates a potential failure) also satisfies the sufficiency condition (transfers to output). He reports a value for P between .61 and .94 and concludes that, while there is a wide variation of precision, "... if we satisfy the same constraint [the reachability and necessity constraints] with multiple test cases ..., then we can achieve a high mutation score with only a few test cases per constraint."

From a testing point of view, these results would be reassuring. Offutt's results, however, are greatly flawed. This is for several reasons. First, the programs used are quite small – each contains between 12 and 55 lines of code. Further, examination of the programs shows code with a high proportion of conditional statements. As a result, transfer of a potential failure would involve a great deal of control dependence transfer and very little data dependence transfer. It is not clear that these information flow patterns and the size of the programs studied are at all representative. Second, as noted in Offutt's dissertation, Godzilla does not solve for constraints involving internal variables (and other special cases, such as deeply nested constraints). As a result, certain classes of faults were not considered. It is not clear that the results hold in the general case. Thirdly, and most importantly, even if the weak mutation testing hypothesis does hold 60% of the time or more, we cannot make the leap that simply selecting multiple test cases solves the problem without investigating the question of why the weak mutation hypothesis failed those other times. If the hypothesis failed because of particular constructs through which transfer is particularly difficult, then we could be selecting a lot of test data that does not transfer. The question of factors that could be involved in why transfer fails is raised some in the work performed by Marick.

Marick [Mar90] reports on a study he performed on five widely used programs. Four routines were selected from each program containing between 9 and 206 lines. Several simple faults were seeded into each program, and the code was analyzed by hand to determine the circumstances under which the effect of the fault would be observable. Marick defines five categories of faults. First, a fault is in the category *holds* if the weak mutation hypothesis would always hold – that is, it would be detected by any test datum that executes the fault and originates a potential failure. A fault is in the category *almost holds* if the weak mutation hypothesis would usually hold – that is, a fault would be detect in almost all cases by such test data. A fault is in the category *coverage holds* if the fault would not be detected in general by weak mutation testing but would be detected, except in rare cases, by test data that satisfy branch coverage. A fault is in the category *spec likely* if the fault would not be detected by either weak mutation testing or branch coverage but would be detected, except in rare cases, by test data that satisfy ordinary specification based testing techniques such as equivalence class testing with boundary values. Finally, a fault is in the category *does not hold* if neither branch testing, weak mutation testing or specification based testing would detect the fault. Each seeded fault was determined to be in one of these five categories. The results of his analysis are that the weak mutation hypothesis holds 50% of the time, almost holds 20% of the time, coverage holds 16% of the time, and spec likely holds 6% of the time. Weak mutation and branch coverage and specification based testing are not adequate 8% of the time.

Marick further analyzes these results for influencing factors. The factors he considers are: complexity of the code, type of fault, likelihood of the fault, and type of routine in which the fault resides. He found that only one factor of those examined, type of routine in which the fault resides, was significant. In particular, for faults residing in a routine whose only effect is to return a boolean value, he found the percentage of faults detected by weak mutation and branch coverage was less than for all faults. From the point of view of transfer of a potential failure, this is not surprising. What is useful about this information is that it suggests that one area where constructing the transfer condition is particularly important is boolean expressions. For these expressions, as seen in Chapter 4, the transfer conditions are quite straightforward.

It is difficult to evaluate the validity of Marick's results because the actual analysis is performed by hand and the criteria for analysis are not fully described. While we feel that the programs examined are more representative than those used by Offutt, the size of the study is quite modest and must be interpretated as initial results. Nonetheless, the study asks some very interesting questions. In the next subsection, we propose a study that investigates some of these questions.

7.1.2 Proposal for Further Studies

In the studies by Offutt, the question of transfer to output is looked at as one whole component. As seen in RELAY, however, there are several components of transfer from an originated subexpression potential failure to output, including potential failure interaction, control dependence transfer, and data dependence transfer. The nature of transfer involving any of these components may differ in real programs. For example, one possibility is that transfer almost always occurs in real programs when there is no control dependence interaction. Another possibility is that transfer almost always occurs when there is more than one information flow chain being transferred along. Here, we propose investigating the factors that influence transfer in real programs using dependence relations and patterns of information flow as the primary basis for such questions and the types of operators in a computation as a secondary basis.

In this thesis, we identify transfer sets and transfer routes as structures of interest. We propose first investigating the characteristics of these structures in real programs. Next we propose an investigation of the weak mutation hypothesis for different types of information flow structures. Below we outline the questions asked in each of these investigations.

1. Transfer Sets — characteristics of interest:
    - Number of information flow chains,
    - Number of nodes in transfer set,
    - Number of potential interaction points,
    - Degree of interaction at these points (number of potential failure variables referenced at node)
2. Transfer Routes
    - Type of interaction – strictly data dependence, mixed data dependence/control dependence
    - Degree of interaction at interaction points

After determining the characteristics of transfer sets and transfer routes, we propose examining the weak mutation hypothesis in a manner similar to Offutt by seeing how often failure occurs for test data selected only to execute the fault and originate a subexpression potential failure. Such a study, however, would use the details of information flow transfer described in RELAY to direct more in-depth analysis of the question. The following analysis would be performed.

1. Analysis of test data that cause a failure to determine the transfer route executed by a test datum;
2. Analysis of test data that did not cause a failure to determine where and through which type of dependence transfer did and did not succeed;
3. Analysis of these results to determine what characteristics of transfer sets and transfer routes investigated above, if any, influence success or not of a test datum to transfer an originated potential failure to output.

We believe that such study at the detailed level of how coincidental correctness does and does not occur will provide insight types of code where the rigor of constructing and satisfying transfer set conditions is desired or needed.

7.2 Multiple Fault Interaction and the Coupling Effect

As previously noted, testing based on RELAY, as well as most fault-based testing approaches, makes the assumption that there is either a single fault in a module or that multiple faults do not interact in such a way as to mask each other. By this we mean that a test datum that causes a failure for a single fault in a module would also cause a failure when that fault occurs with other faults in the module. Schematically, looking at Figure 7.1, this assumption says that a test datum that distinguishes between the original module (two faults) and alternate module 1 (one fault) would also distinguish between the original module and the correct module (no faults).

To more fully motivate the problem, consider the example shown in Figure 7.2 and suppose that this simple module contains two faults. One fault is at node 2,

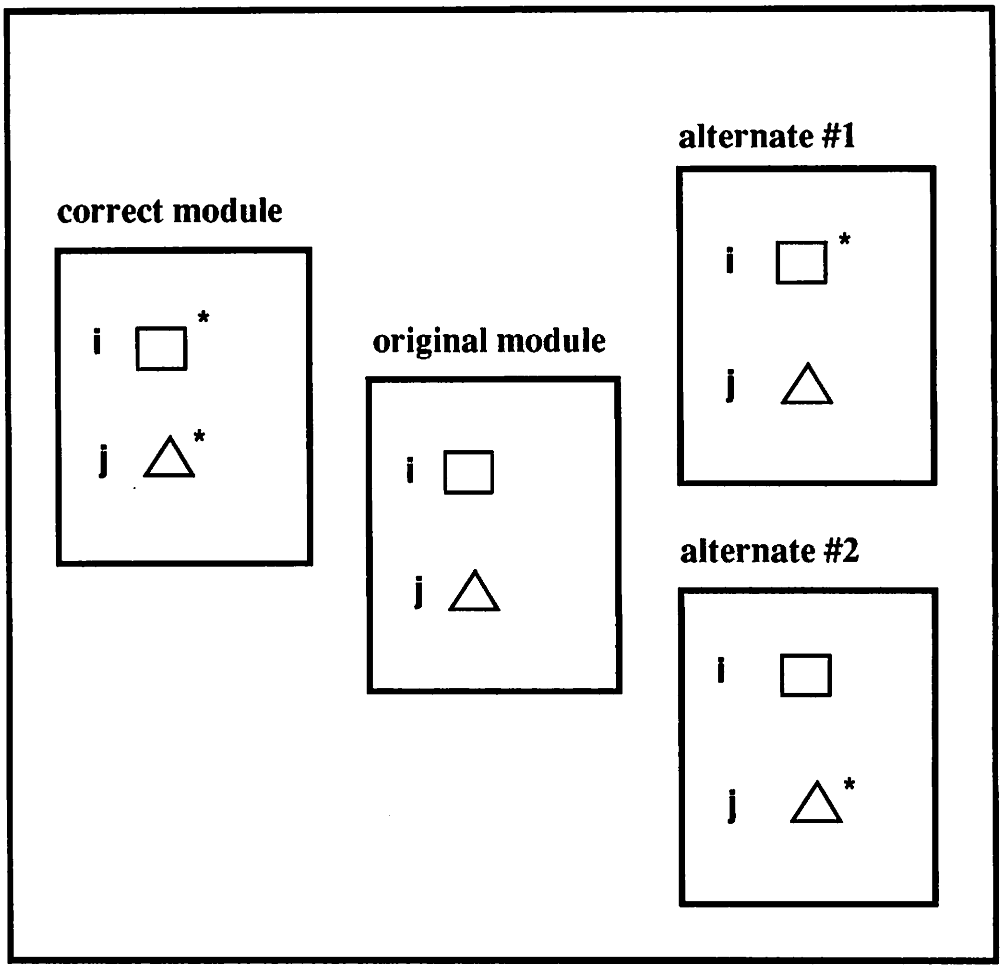

Figure 7.1. Schematic of Fault-Based Testing

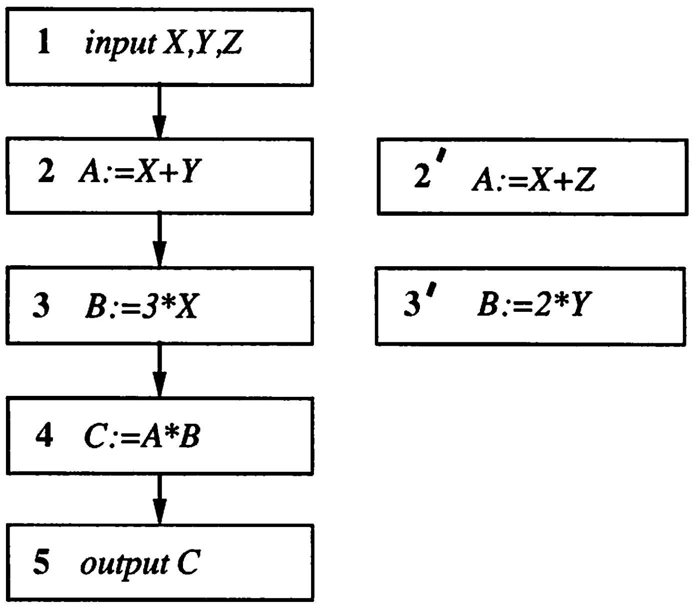

Figure 7.2. Example Module with Two Faults

which should be

$$A := X + Z.$$

A second fault is at node 3, which should be

$$B := 2 * Y.$$

These correct nodes are shown in the figure next to the faulty nodes. Let us test for each of these faults in isolation of the other. To test for the first fault, we derive the failure condition $y \neq z$ at node 2 and $b \neq 0$ at node 4. This condition is satisfied by the test datum $x = 1, y = 3, z = 1$, and as seen in Table 7.1, this test datum distinguishes between this module and the alternate module containing the alternate node 2'. To test for the second fault, we derive the failure condition $3 * x \neq 2 * y$ at node 3 and $a \neq 0$ at node 4. This condition is also satisfied by the test datum $x = 1, y = 3, z = 1$ and as seen in Table 7.2, this test datum distinguishes between this module and the alternate module containing the alternate node 3'. When both

Table 7.1. Test Data Set for Fault at Node 2

module	t.d.			ref	*a*	*b*	*c*	*output*
	x	y	z	(y/z)	(2)	(3)	(4)	(5)
faulty	1	3	1	3	4	3	12	12
alternate #1	1	3	1	1	2	3	6	6

Table 7.2. Test Data Set for Fault at Node 3

module	t.d.			*a*	*b*	*c*	*output*
	x	y	z	(2)	(3)	(4)	(5)
faulty	1	3	1	4	3	12	12
alternate #2	1	3	1	4	6	24	24

faults exist in the module, however, these conditions are not sufficient to guarantee detection of either fault. This is seen in Table 7.3 which shows a partial trace of execution on the test datum $x = 1, y = 3, z = 1$ for the module in Figure 7.2 and the correct module with alternate nodes $2'$ and $3'$. From this table, we see that both modules output 12 for this test datum, and no fault would be detected. Because the module would evaluate correctly for this test datum, we would erroneously conclude that the hypothetical fault at node 2 is not a fault and the hypothetical fault at node 3 is not a fault.

The failure to detect these faults occurs because the potential failure in A, which originated from the first fault at node 2, and the potential failure in B, which originated from the second fault at node 3, interact. Such interaction we call ***multiple fault interaction***. Multiple fault interaction is the equivalent of what Morell defines as 'coupling' [Mor84]. We use the term multiple fault interaction to distinguish it from the older use of the word coupling in mutation analysis [DLS79].

Clearly, the assumption that multiple faults cannot mask each other is not in general valid. In his thesis, Morell [Mor84] states several general theoretical results

Table 7.3. Test Data Set for Faults at Nodes 2 and 3

module	t.d.			*a*	*b*	*c*	*output*
	x	y	z	(2)	(3)	(4)	(5)
faulty	1	3	1	4	3	12	12
correct	1	3	1	2	6	12	12

about multiple fault interaction and shows that it is in general undecidable whether or not multiple fault interactions occur. As most programmers know, it is unlikely that there is a single fault in the module being tested; thus, we must examine this assumption of no multiple fault interaction empirically. In the next subsection, we evaluate empirical studies from the literature that are related to the issue of multiple faults.

7.2.1 Related Empirical Studies

The most related studies involve mutation testing. Recall, that mutation testing is a test data evaluation approach that evaluates a test data set according to its ability to kill single mutant programs. The ability of a test data set to kill single mutant programs is taken to be a measure of the test data sets ability to detect more complex faults. It is believed that test data that would be sensitive to detecting simple faults entered into the code would also detect more complex faults. The justification for this is that some of the mutations would effect code related to any complex fault that could exist in the code and test data that exercises a program sufficiently to distinguish between the original program and the mutant programs would also exercise the program sufficiently to detect the complex fault. In this way the mutated expression and the complex fault are said to be coupled. Let us see what mutation testing is saying in terms of the schematic Figure 7.3. Let the triangle at j be a complex fault that should be a circle. Application of the mutant operators at every possible location in the program will result in at least

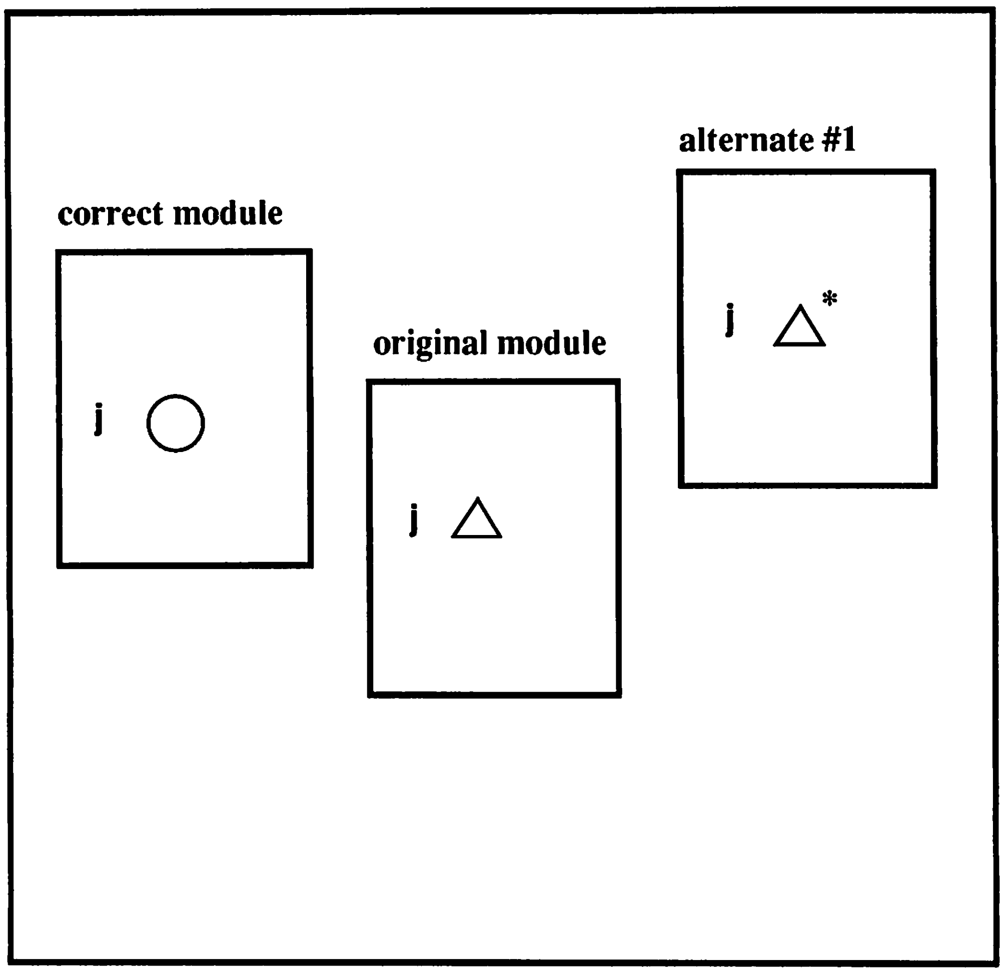

Figure 7.3. Schematic of Mutation Testing

one program that contains a mutated j, indicated by a triangle with a star and labeled in the figure alternate module 1. Mutation analysis says that a test data set that distinguishes between the original module and alternate module 1 would also distinguish between the original module and the correct module not containing the complex fault. Supposed justification for mutation analysis is the hypothesized "coupling effect".

In [DLS79] this effect is stated as "Test data T which causes all the non-equivalent mutants of $M(P)$ to fail is so sensitive that all the non-equivalent mutants of $M^*(P)$ must also fail on T." (M^*P is the set of mutant programs formed from the combinations of single mutants). This, however, is what we mean by multiple fault interaction and in fact does not provide justification for mutation analysis as seen in a comparison of the discussions of Figure 7.1 and of Figure 7.3. The meaning of the coupling effect since its introduction, however, has become less precise and is more generally stated to mean that a test data set that distinguishes programs with simple faults is sensitive enough to distinguish programs with more complex faults [Off89].

Offutt has examined the question of multiple fault interaction in an empirical study of the "coupling effect" [Off89]. There he defines a simple fault as a single mutation applied at a single location and a complex fault as some combination of n simple faults. A complex fault of n simple faults is called *n-order*. To study the question of the coupling effect, Offutt performed first a study of three small Fortran-77 programs. In this study, he generated a set of test data that killed a set of first order mutants for a program. Second order mutants were then constructed, and the test set evaluated for its ability to kill the second order mutants. The test set sufficient to kill first order mutants was sufficient to kill over 99% of the second order mutants for all three programs. Offutt analyzed those second order mutants that were not killed. None of the live second order mutants appeared to be particularly unlikely to be killed. To verify this, the experiment was repeated with different test sets. The high percentage of second order mutants killed was repeated, and for these test sets, different sets of second order mutants remained live. In a second study, Offutt compared the ability of a randomly selected test set to kill first and second order mutants. For all three programs, a higher percentage of second order mutants were killed than the percentage of first order mutants killed. For a two line program and for one of the test suite programs containing only straightline code, some third order results were provided. The percentage of mutants killed increased with the order of the mutant. These results, Offutt concludes, support the belief that the coupling effect does hold.

Let us consider the results of these studies in terms of what they say about multiple fault interaction and what they say about the more general concept of coupling. The first study looks at whether a test set that killed first order mutants would also be sufficient to kill second order mutants. Consider first the concept of coupling. Apart from the question of whether or not the definition of a complex fault is relevant (that question is discussed below with a report of Marick's work on simple and complex faults), the results of this study do not provide evidence for or against the coupling effect. Consider a first order mutant m_1 and a second order mutant (m_1, m_2). Offutt's results indicate that a test set T that kills m_1 also kills (m_1, m_2) 99% of the time. This means that there is a test datum in T that kills m_1, and 99% of the time there is also a test datum in T that kills (m_1, m_2). These results do not speak to the coupling effect, however, because we do not know if the same test datum that kills m_1 also kills (m_1, m_2). Even with such knowledge, these results would not provide evidence for the coupling effect. Consider a single test datum t that kills a particular mutant m_1 and that also kills mutant m_2. The question is whether the test datum is in fact sensitive to m_2 or is m_2 just being killed as a side effect of killing m_1. We cannot answer this because in general, we have no knowledge of whether or not there is any dependence relationship between the location of m_1 and m_2 or if there is any node in the module that is syntactically dependent on both m_1 and m_2. Further, for any specific test datum, we do not know if m_2 was also executed; if it was executed, we do not know whether or not m_2 originated a potential failure, and if it did originate a potential failure, we do not know whether or not that potential failure transferred to a point where it could interact with some potential failure caused by m_1. Such results cannot be used to validate mutation analysis as a test data evaluation method.

Offutt's results do speak to the issue of multiple fault interaction and suggest in Figure 7.1 that a test datum that distinguishes between the original module and alternate module 1 would also distinguish between the original module and the correct module. For the assumption that multiple faults do not mask each other, Offutt's results shows that 99% of the time the assumption holds. Further empirical study is necessary however. This is because 1) the program suite is simply too small and 2) the information flow in the programs is not representative of real programs. Since how multiple faults may interact is through information flow, it is imperative that empirical studies use programs that have representative patterns of information flow.

Let us consider next the results of the second study. These studies actually do speak to the more general justification for mutation analysis, since they suggest that the more mucked up a program, e.g., higher the order of mutant, the less subtle the fault and hence more easily detected. This study, however, cannot be taken to prove or disprove the coupling hypothesis either. This is because 1) again, the study is simply too small and does not examine representative programs, and 2) it is not clear that complex faults may be modeled simply as many simple faults. In [Mar90], Marick specifically examines the question of simple and complex faults. He classified 102 faults collected from USENET bug reports and found that only

23% of all faults were simple, 8% were compound (n-order simple faults), 47% were faults of omission, and 23% were complex of some other sort. Grouping according to the region effected, he found that 43% of the faults were confined to a single expression and 57% effected more than one statement. These results as noted by Marick do not prove or disprove the coupling effect. It does, however, suggest that Offutt's approach to studying the coupling effect by studying n-order mutants may not provide relevant results.

7.2.2 Proposal of Further Studies

We propose an investigation of multiple faults in a vein similar to our investigation of transfer for single faults. Here, we outline the steps of this study.

1. Information flow chains between two nodes are examined, determining the potential for faults at two different locations to interact;
2. Using a smaller set of second order mutants while using a larger test suite, a study similar to Offutt's is performed;
3. For all second order mutants, m_1, m_2, executions of test data are analyzed to determine is both m_1 and m_2 are executed, which mutants originate potential failures, and if potential failures originating from different mutants interact at any point in the program;
4. This analysis is correlated with whether or not the second order mutants were killed.

Results showing that m_2 does not interfere with detection of m_1 even when both m_1 and m_2 are executed would provide evidence that the multiple fault non-interaction assumption made by RELAY, and by fault-based testing in general, holds. Results showing that m_2 does mask detection of m_1 when both are executed would suggest the need for selecting test data that exercise chains of information flow that do not involve both faults. Since we do not know in general where one fault is much less where two faults are, this would suggest the need to select several test data that involve a variety of chains of information flow. Such a requirement would incorporate ideas similar to the data flow coverage technique developed by Laski and Korel [LK83] and described briefly in Chapter 6 that directs selection of test data that selects paths that executes different contexts for a node.

Chapter 8

Conclusion

This chapter provides a summary of the work presented in this thesis, points out its major contributions, and describes future research directions.

8.1 Major Contributions

This thesis presents an investigation of fault-based testing based on a new model of faults and failures called RELAY. This model rigorously defines the steps of how a fault in a module causes a failure and includes the components of origination, computational transfer, data dependence transfer, and control dependence transfer. In addition, RELAY provides a framework of transfer sets and transfer routes in which these components fit. Origination is the introduction of a potential failure, or intermediate incorrect state, in the smallest subexpression containing a fault. Transfer is the movement of a potential failure through execution of the module on some test datum. Three types of transfer are identified. Computational transfer involves the transfer of a potential failure within a statement. Data dependence transfer involves the transfer of a potential failure from the definition of a variable to a use of that variable. Control dependence transfer involves the transfer of a potential failure from the incorrect evaluation of a branching statement to a statement whose execution may be controlled by that branching statement.

Transfer of a potential failure from an originating node to a failure node occurs along an information flow chain. As captured in the model, transfer of a potential failure may occur along several information flow chains at the same time. At nodes where more than one chain is being transferred along, interactions occur. These interactions may involve multiple links of data dependence as well as links of control dependence. RELAY models this potential for interaction with the concept of transfer sets, which are sets of information flow chains from the same node and to the same node that all may be executed by the same path(s). A transfer set defines the set of chains that could be executed together. Execution of a chain, however, does not imply that transfer occurs along the chains. A transfer route is a subset of the nodes in a transfer set where transfer does occur. Thus, a transfer route defines what chains or portions of chains of the transfer set transfer occurs along and as such defines places of actual interaction. Transfer sets and transfer routes form the framework in which the components of transfer fit.

The major contributions of this model are as follows:

- While previous models recognize the two steps of introducing an incorrect state and transferring an incorrect state to output, RELAY is the only model to provide detailed description of the latter.

- While some related research recognizes the movement of a potential failure along data flow, none provide an in-depth investigation of the role of control dependence transfer and the interaction between control dependence and data dependence transfer. Such an in-depth investigation is provided in this thesis.

- While other models consider the transfer of a potential failure along a particular path, none consider how more than one path might execute the same sequence of information flow from one node to another or how that information flow might involve several chains. Our model defines transfer sets and transfer routes which define such equivalence classes, and as such pulls together research in data flow path selection, program slices, and program dependence relationships.

We use the model to determine the condition that must be satisfied to guarantee detection of a fault. Such a condition is termed a failure condition and is composed of two parts, the original state potential failure condition and the transfer set condition.

The original state potential failure condition is composed of the origination condition together with the computational transfer condition at the faulty node. The origination condition guarantees introduction of an incorrect state in the smallest expression containing a fault. The computational transfer condition at the faulty node guarantees transfer of the originated subexpression potential failure to effect evaluation of the entire node. Thus, the original state potential failure guarantees introduction of an incorrect state at the node containing the fault.

The transfer set condition guarantees transfer of an original state potential failure along a particular set of information flow chains to output. The transfer set condition is formed from the conjunction of the path condition for the transfer set, which guarantees execution of all chains in the transfer set, with the disjunction of transfer route conditions. A transfer route condition is the sufficient and necessary condition that guarantees transfer along a particular transfer route. Transfer routes are needed because to derive necessary and sufficient conditions to transfer a potential failure from an originating node to a failure node, all points of interaction must be known. Transfer route conditions are constructed using the computational transfer conditions at nodes where transfer does occur in the transfer route and the complement of the computational transfer conditions at nodes where transfer does not occur. Two types of computational transfer conditions are defined. Simple computational transfer conditions apply at nodes where no interaction occurs – that is, at nodes where only a single potential failure variable is referenced. Complex computational transfer conditions apply at nodes where interaction does occur – that is, at nodes where more than one potential failure variable is referenced.

Simple computational transfer conditions are often fault independent, while complex computational transfer conditions are not.

The major contributions of the work on failure condition presented here are:

- Identification of places in the failure condition that may be fault independent and thus may be applicable to many faults.
- Recognition of how interaction between information flow chains complicates the process of constructing transfer conditions to guarantee transfer of an original state potential failure to output and of how this interaction affects the fault dependence of the conditions.
- Insight into how fault independent conditions could be formed by guaranteeing a failure is masked at some points while not at others, limiting the degree of interaction.

Since previous theoretical results have shown that in general selection of data that guarantees detection of a fault is not possible, it was not our intention to solve the equivalence problem with such conditions. Rather, in investigating what conditions would be necessary to guarantee failure, we hope to gain insight into the process of guaranteeing fault detection. This insight allows us to more fully evaluate other fault-based testing approaches. Such an analytical evaluation is presented in this thesis and demonstrates that existing fault-based testing criteria only attempt to originate a potential failure. Several succeed for some classes of faults. Only the work of Morell provides any information on transfer to output of an original state potential failure. His work seeds a symbolic fault and generates two symbolic computations for a particular path in a module, one with the symbolic fault and one without. The major limitation of his work is that it requires the same path be selected by both the module and the symbolically faulty module. Thus, his model does not investigate the role of control dependence in transferring a potential failure to output. Given the complexity of the transfer set condition, however, his approach is promising, and extensions to his symbolic fault-based testing approach by incorporating aspects of the RELAY model should be pursued.

The analytical evaluation of fault-based testing done with the RELAY model must be balanced by empirical studies. We survey several empirical studies already performed and suggest two areas for future investigation. The first area involves the question of whether or not transfer to output is likely to happen for test data selected to execute a fault and only originate a potential failure for that fault. A review of empirical studies in the literature indicates that this is likely, although the results are not well founded and the actual probability seems variable. With the RELAY model in mind, we suggest further study of the process of transferring a potential failure to output, examining in particular, how the probability of transfer through different patterns of information flow may vary. A second area of further study involves the assumption that multiple faults do not interact to mask each other. A review of empirical studies in the literature suggests that this assumption holds, but again these results are not well founded. More empirical work on a larger more comprehensive test suite is needed.

8.2 Future Directions

Future directions for this work concentrate primarily in the area of practicality and involve issues raised by the question of feasibility and desirability of building a testing system based on the RELAY model. Several of these issues investigate the testing and weakening of the model's current assumptions. Some of these questions require further analytical work while others require empirical study.

The RELAY model assumes that multiple faults do not interact. In addition to the empirical studies of this assumption proposed in Chapter 7, we feel that the work on fault independent sufficient conditions might have application in the area of multiple fault interaction. This is because interaction of potential failures from the same fault is essentially the same as interaction of potential failures from two or more faults. The difference is that the source of a potential failure may not be known in the latter case. While considering all combinations of possible faults at all locations is not possible, there may be some sufficient conditions that might be satisfied to guarantee transfer. Progress in such areas would increase the feasibility of a testing system that uses the transfer set condition or some portion(s) of the transfer set condition.

RELAY assumes that the module being tested is nearly correct and restricts the classes of faults considered to simple faults occurring within a node. The general framework, however, we believe is applicable to larger, more complex faults. Work on extending the RELAY model to more complex types of faults is needed. Marick's empirical study of the types of faults that occur in real programs, discussed in Chapter 7, indicates that most fault-based testing is not addressing the most common faults. Further analysis in defining fault classes other than simple atomic faults is required. Origination conditions must then be developed for these faults. Unless the fault alters the information flow of the program, the RELAY model of transferring the originated potential failure to output should still be applicable. In addition, building upon a program schema (control flow graph) of larger granularity allows consideration of faults that affect multiple statements and still do not affect the program schema.

Transfer sets define equivalence classes of information flow. Investigation into the nature of these equivalence classes is needed. One question is how the conditions for one transfer set are related to the conditions for another. More specifically if the transfer set condition for one transfer set is unsatisfiable, can this tell us anything about the satisfiability of another transfer set condition. This question is of interest for transfer routes as well.

An interesting area of work is in the development of fault independent transfer conditions. In some areas of this thesis, we discuss the construction of sufficient fault independent conditions. A question that should be examined is what are other operators and other types of expressions for which sufficient fault independent conditions could be developed.

Another area of future research is in empirical studies. Several studies are suggested in Chapter 7. The details of the RELAY model provide us with a perspective from which to analyze the process of transfer of a potential failure to output that previously was not available. A primary direction for future empirical study investigates whether certain constructs in the code are more prone to coincidental correctness by looking at transfer through different information flow constructs and at multiple fault interaction. Such studies should provide insight into areas of code that need to be tested differently and how such testing might be achieved.

These investigations into the nature of dependences and transfer in real programs may be used to evaluate whether or not a testing system based on RELAY is desirable or feasible. While a system based on the entire model is not reasonable, integration of RELAY and Morell's symbolic approach might be fruitful.

In summary, this thesis presents the RELAY model of faults and failures, upon which failure conditions that guarantee fault detection can be constructed, fault-detection capabilities of of other testing criteria can be analyzed, and empirical studies can be founded. This thesis proposes future research addressing the practicality and extensibility of fault-based testing based on the RELAY model.

References

[ABD$^+$79] A.T. Acree, T. A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Mutation analysis. Technical Report TR GIT-ICS-79/08, Georgia Institute of Technology, September 1979.

[BA80] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. Technical Report TR 80-19, University of Arizona, 1980.

[BDLS78] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. The design of a prototype mutation system for program testing. In *Proceedings NCC*, 1978.

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

[Bud83] Timothy A. Budd. The portable mutation testing suite. Technical Report TR 83-8, University of Arizona, March 1983.

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

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

[DGK$^+$88] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. An extended overview of the mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 4(11), April 1978.

[DLS79] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Program mutation: A new approach to program testing. Technical Report v. 2, Infotech International, 1979.

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

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

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

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

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

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

[How78a] William E. Howden. Algebraic program testing. *Acta Informatica*, 10, October 1978.

[How78b] William E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4), July 1978.

[How82] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(2):371–379, July 1982.

[How85] William E. Howden. The theory and practice of functional testing. *IEEE Software*, 2(5):6–17, September 1985.

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

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

[Mar90] Brian Marick. Two experiments in software testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, 1990.

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

[Mor88] Larry J. Morell. Theoretical insights into fault-based testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

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

[Off88] A. Jefferson Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, August 1988.

[Off89] A. Jefferson Offutt. The coupling effect: Fact or fiction. In *Proceedings of the Third Workshop on Software Testing, Verification, and Analysis*, December 1989.

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

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

[Tho91] Margaret C. Thompson. Single iteration chain loop analysis and identification of transfer sets and transfer routes for the RELAY model. Technical Report 91-22, Computer and Information Science, University of Massachusetts, Amherst, May 1991.

[WC80] Lee J. White and Edward I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3), May 1980.

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

[WH88] M.R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

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

[Zei84] Steven J. Zeil. Perturbations testing for computation errors. In *Proceedings of the Seventh International Conference on Software Engineering*, March 1984.

[Zei86] Steven J. Zeil. Domain testing and linear fault detection. Technical Report 38, Computer and Information Science, University of Massachusetts, Amherst, August 1986.