Information Flow Transfer in the RELAY Model

Margaret C. Thompson*
Debra J. Richardson†
Lori A. Clarke*

COINS Technical Report 92–62
August 1992

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

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

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grants # MDA972-91-J-1009 and #MDA972-91-J-1012. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

1 Introduction

Software testing is concerned with the selection of test data to produce incorrect output or "failures". When a program produces a failure for some test execution, one knows the program contains at least one mistake or "fault". When a program produces correct output for all executions, one hopes to have selected the test data so as to gain some confidence in the correctness of the program. Unfortunately, it is possible for faulty code to be executed but not cause a failure. This phenomenon, known as *coincidental correctness*, is very common. If it were not, a test data set that covers all statements in a program would be adequate to detect most faults. To understand how to select test data to avoid coincidental correctness, we must understand how a fault may or may not cause a failure on execution of some test datum. The authors have been developing a model of faults and failures, called RELAY, that describes this process. The model provides insight into the difficulty of guaranteeing fault detection, suggests an approach to fault-based testing that, although computationally expensive, may be warranted for critical software, and serves as a basis for evaluating fault-based testing approaches as well as highlighting areas requiring empirical study.

RELAY is related to fault-based test data selection methods, which attempt to select test data that would expose a set of faults if they existed in the code. Some fault-based testing methods focus on introducing an initial incorrect "state" at a faulty expression [Fos80, How82, Bud83, Zei83]. This is generally known as "weak mutation testing" [How82, Bud83]. Weak mutation, however, does not guarantee a failure will result. Subsequent execution may mask the effect of incorrect values and produce correct final results. Several researchers have considered what must happen to cause a failure after an initial incorrect state has been introduced, thereby satisfying what is known as "strong mutation testing" [DLS79, Bud83]. Morell describes a model of fault based testing that introduces the ideas of "creating" an initial "error" for a fault and "propagating" it to the output [Mor84]. Offutt describes a method, called constraint-based testing [Off88], which defines three conditions: a "reachability" condition to force execution of the hypothetically faulty node, a "necessity" condition for introducing an error, and a "sufficiency" condition for then producing a failure. Offutt's necessity condition and sufficiency condition are similar to Morell's creation condition and propagation condition, respectively. Neither of these models fully captures how once an incorrect state is introduced, it remains incorrect until a failure is revealed as an output value.

The RELAY model augments both the weak and strong mutation testing approaches and related models. RELAY formalizes the weak mutation testing method by describing the conditions required to "guarantee" that a fault in some subexpression of a statement produces an incorrect state for the whole statement; this work is described in [RT88, RT86]. RELAY extends Morell's model by identifying and describing the ways an incorrect state may "transfer" through execution. RELAY identifies three types of transfer: "computational transfer", "data dependence transfer", and "control dependence transfer". An incorrect state 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" and "transfer routes". Transfer sets, transfer routes, and control dependence transfer are unique to the RELAY model. The transfer aspects of the RELAY model are the subject of this paper.

This paper is organized as follows. Section 2 provides some general terminology along with an overview of the RELAY model. Section 3 discusses the components of information flow transfer and how these components fit together in the model. Section 4 presents two applications of the model: testing of critical software systems and evaluation of fault-based testing methods. Section 5 summarizes the major contributions of RELAY and concludes with research directions suggested by this work.

2 Foundations

This section lays the foundation for describing information flow transfer in the RELAY model. In particular, we provide an overview of the entire RELAY model as context. We also define some related terminology and give the underlying assumptions of the model and some additional assumptions that greatly simplify the presentation.

2.1 Terminology

We consider the analysis of a *module*, where a module is a procedure or function with a single entry point. A module, M, is represented by a *control flow graph*, which is a directed graph (N, E), where N is a (finite) set of nodes and $E \subseteq N \times N$ is the set of edges. Each 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. A node with more than a single successor node is called a *branching node*.

A *path* in a control flow graph $G_M = (N, E)$ is a finite, possibly empty, sequence of nodes $p = (n_1, n_2, ..., n_{|p|})$ [1] such that for all i, $1 \leq i < |p|$, $(n_i, n_{i+1}) \in E$. A path formed by the concatenation of two paths p_1 and p_2 is denoted $p_1 \cdot p_2$. A path p may be executed on some input x [2]; this execution is denoted $p(x)$. A *state* $S_{p(x)}$ is a vector of values for all variables after execution of path p on input x. If the last node in p is a branching node, then $S_{p(x)}$ includes a dummy variable BP that holds the value for the branch predicate associated with this node.

RELAY uses information derived from program dependences, which 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. See [PC90] for a more complete discussion of dependences.

Let V be a variable in a module M. A *definition of* V is associated with each node n in G_M that represents a statement that can assign a value to V; this definition is denoted $def(V, n)$. A *use of* V is associated with each node n in G_M that represents a statement that can access the value of V; this use is denoted $use(V, n)$. With each node n in a control flow graph, we associate the set *defined(n)*, which is the set of all variables to which a value

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

[2] The input x includes the values of all variables at the start of execution of path p and any data input during execution of the path.

may be assigned by the statement represented by n, and the set *used(n)*, which is the set of all variables whose value may be referenced by the statement represented by n. To simplify our discussion, we assume at each node there is at most a single variable in *defined(n)*. A *definition-clear path* with respect to a variable V is a path p such that for all nodes n in p, $V \notin defined(n)$ [3]. A definition $def(V, m)$ *reaches* a use $use(V, n)$ if and only if there is a path $(m) \cdot p \cdot (n)$ such that p is definition-clear with respect to V.

A node n is *(directly) data dependent* on a node m if and only if there is a $def(V, m)$ that reaches a $use(V, n)$. Since this is the only data dependence relationship we use, we will refer to it simply as "data dependence". Consider the control flow graph shown in Figure 1 [4]. Node 9 is data dependent on node 2 because A is defined at node 2, used at node 9, and there is a def-clear path with respect to A from node 2 to node 9.

Information may also flow by one node controlling execution of another. The *immediate forward dominator* of a (branching) node b is the node where all paths leaving b first come together. A node n is *(indirectly strongly) control dependent* on m if and only if there exists a path from m to n that does not include the immediate forward dominator of m. Intuitively, this relationship characterizes the nodes that are in the "body" of a structured branching construct. Since this is the only control dependence relationship we use, we refer to it as "control dependence". In Figure 1, nodes 6, 7, 8 and 9 are control dependent on node 5.

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. We represent an information flow chain X in a control flow graph $G_M = (N, E)$ as a sequence of tuples $(*, d_1, n_1), (u_2, d_2, n_2), \ldots, (u_{|X|}, d_{|X|}, n_{|X|})$, where $|X|$ is the number of tuples in the chain and $\forall i,\ 1 \leq i \leq |X|,\ n_i \in N$ and $d_i \in defined(n_i)$, and $\forall k, 1 < k \leq |X|,\ u_k \in used(n_k), u_k = d_{k-1}$, and n_k is either control dependent or data dependent on n_{k-1}. At the first node in a chain, the symbol '*' is used in place of the used variable. For branching

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

[4]Nodes $2'$ and $3'$ are not part of the control flow graph and are discussed in subsequent sections.

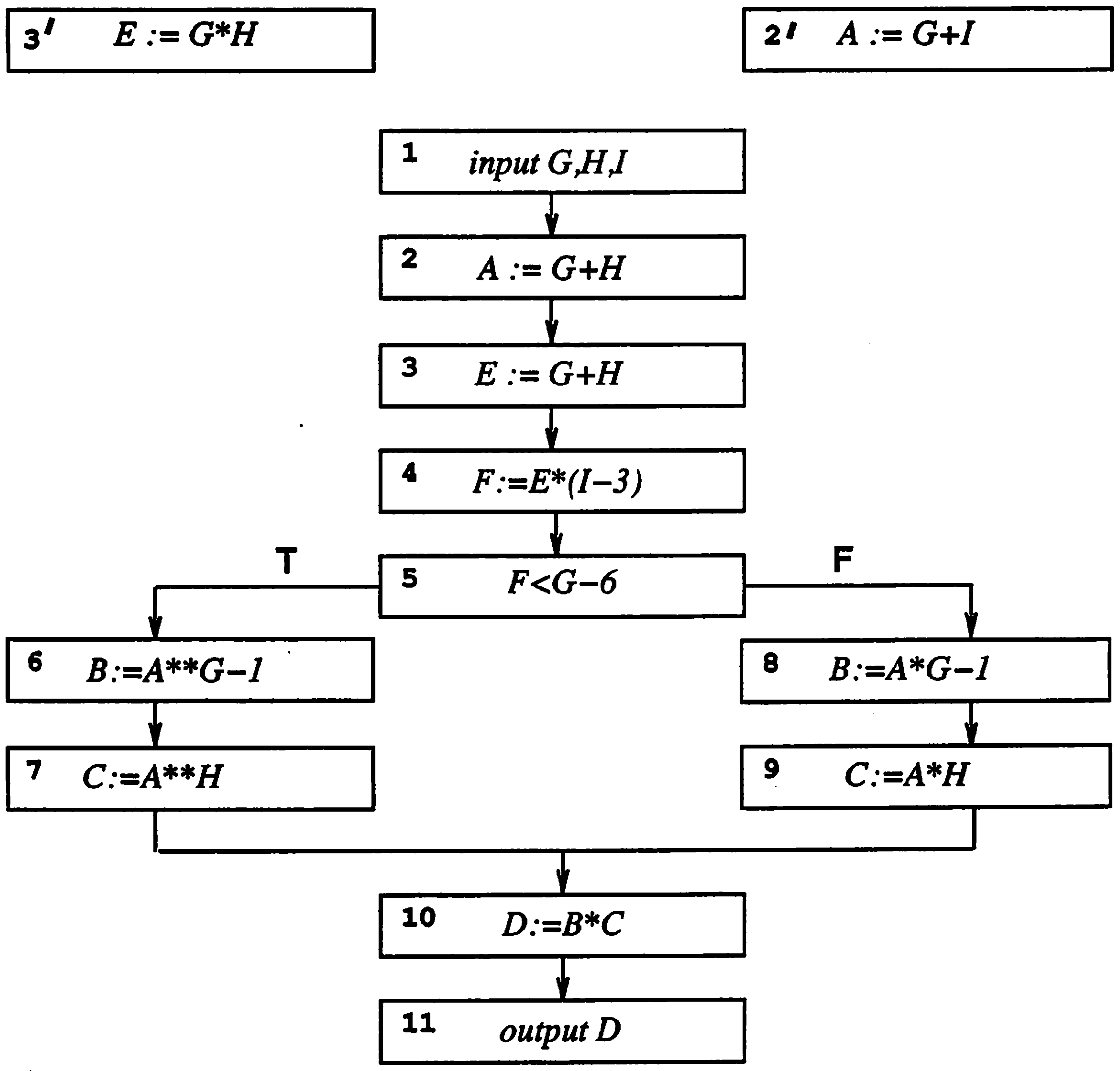

Figure 1: Example Module

nodes, the dummy variable BP, which represents the branch predicate, is used in place of the defined variable. For a tuple that represents control dependence, BP is used in place of the used variable. For nodes where a value is communicated to the external environment, the symbol *'out'* is used in place of the defined variable. Note that due to loops a node could appear more than once in an information flow chain. In Figure 1, one information flow chain from node 3 to node 11 is

$$(*, E, 3), (E, F, 4), (F, BP, 5), (BP, B, 6), (B, D, 10), (D, out, 11).$$

2.2 Model Overview

RELAY is a model of faults and failures. A *fault* is a mistake in the source code and represents a syntactic difference between the given and correct modules. A *failure* is an observable incorrect program behavior. For this paper, we only consider failures that consist of incorrect output. For a fault to cause a failure, execution on some test datum causes intermediate incorrect values that eventually result in observable incorrect output values. An intermediate incorrect value is termed a *potential failure*, since these values may or may not cause a failure. Potential failures occur when a subexpression of a statement or an entire statement evaluates to an incorrect value. In this paper, we will be interested in tracking potential failures that occur at statement boundaries, called *state potential failures*. A *potential failure variable* is a state variable whose value is incorrect after execution of a path.

In RELAY, a potential failure *originates*, or is introduced, in the smallest evaluable subexpression containing the fault. If this smallest subexpression is not the outermost expression in the statement, the potential failure must *transfer* through all subsequent operations in the node that depend directly or indirectly on the faulty value. Transfer through operators within a node is called *computational transfer*. When computational transfer occurs through all operators, the entire statement evaluates to an incorrect value, and an *original state potential failure* is introduced.

For a fault to cause a failure, a potential failure must transfer from a faulty node to an output node along an information flow chain. At each link in the chain, ***data dependence transfer*** and/or ***control dependence*** transfer occurs. This process is called ***information flow transfer***. Computational transfer, similar to that described above for introduction of an original state potential failure, is a component of data dependence transfer and control dependence transfer. Information flow transfer can occur simultaneously along more than one chain to the same output node at the same time. Transfer along multiple information flow chains is captured with the concepts of ***transfer sets*** and ***transfer routes***.

2.3 Model Assumptions

We assume 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. Although the concept of transferring potential failures along information flow can be applied to faults that affect larger portions of code, the faults considered in this presentation are contained within a single node in a control flow graph. Further, the faults we consider in this presentation are restricted to those that do not change the control flow graph and do not change the set *defined(n)* for any node. This last requirement is necessary because changes in *defined(n)* alter the information flow from the faulty node and thus the transfer requirements. This last restriction, however, may be relaxed with additional information flow analysis that takes into account the difference in information flow introduced by the fault.

We also assume there is either a single fault in the module or that multiple faults do not mask each other. Two faults mask each other if a test datum that would have caused a failure for one of the faults occurring alone, fails to cause a failure for the module containing both faults. This assumption allows us to consider faults one at a time. This "non-masking faults" assumption is discussed further in Section 4.2.

Table 1: Illustrative Test Data Set for Fault at Node 3 in Example

		test data			evaluated variables							
	module	G	H	I	A	E	F	$F < G - 6$	B	C	D	*output*
1	faulty	1	2	3	3	3	0	F	2 (8)	6 (9)	12	12
	correct	1	2	3	3	2	0	F	2 (8)	6 (9)	12	12
2	faulty	1	2	4	3	3	3	F	2 (8)	6 (9)	12	12
	correct	1	2	4	3	2	2	F	2 (8)	6 (9)	12	12
3	faulty	1	1	-1	2	2	-8	T	1 (6)	2 (7)	2	2
	correct	1	1	-1	2	1	-4	F	1 (8)	2 (9)	2	2
4	faulty	1	2	1	3	3	-6	T	2 (6)	9 (7)	18	18
	correct	1	2	1	3	2	-4	F	2 (8)	6 (9)	12	12
5	faulty	4	0	2	4	4	-4	T	255 (6)	1 (7)	255	255
	correct	4	0	2	4	0	0	F	11 (8)	0 (9)	0	0

3 Information Flow Transfer

3.1 Introduction of Concepts

To illustrate the components of information flow transfer, we examine several test data for the module in Figure 1 and see how potential failures do and do not transfer. Table 1 lists five test data along with partial execution traces. Suppose that the module contains a fault and node 3 should be $E := G * H$. That is, the addition operator should be multiplication. This correct node is labeled 3′ in the figure. 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 faulty module, while the second line records the values for the correct module. When the modules execute different paths, whereby a variable is defined at different nodes, the node where the value is assigned is shown in parentheses. For all test data in this set, an original state potential failure is introduced, since E has an incorrect value after execution of node 3.

Consider test datum 1. At node 4 where E is referenced, we see that the incorrect value for E is masked out by multiplication, and thus F has the same value in both the correct and the incorrect module. Although node 4 is data dependent on the (incorrect) value held in E, data dependence transfer does not occur. Note that E is the only variable that holds an incorrect value at this point and is not referenced at any subsequent nodes; thus, no failure results for this test datum.

For test datum 2, execution of node 4 assigns an incorrect value to F; thus, data dependence transfer does occur. F is used at node 5, where the branch predicate evaluates to *False* in both the correct and faulty modules. Thus, data dependence transfer fails, and the same branch is selected in both the correct and faulty modules. Since there are no subsequent uses of either E or F, which are the only variables with faulty values, no failure results.

For test datum 3, data dependence transfer succeeds from E to F at node 4 and from F to BP at node 5; thus, an incorrect branch is selected. Since nodes 6, 7, 8, and 9 are control dependent on node 5, we consider whether a potential failure transfers through this incorrect branch selection. After an incorrect branch is selected, transfer occurs when a value is assigned to some variable that is distinct from that which would have been assigned along the correctly selected branch. In this case, one or both of B or C would need to be assigned an incorrect value. For this test datum, however, both B and C are assigned the same value on the incorrectly selected branch as on the correctly selected branch, which masks out the effect of selecting the incorrect branch. In this case, control dependence transfer does not occur.

Consider test datum 4, for which data dependence transfer occurs at nodes 4 and 5, and control dependence transfer to B fails at node 6. Control dependence transfer to C does occur at node 7, however, since C is assigned a different value at node 7 on the incorrectly selected branch than at node 9 on the correctly selected branch. C is used at node 10, where D is assigned different values in the correct and faulty modules, thus data dependence transfer occurs. An incorrect value is output at node 11, thus revealing a failure for execution of the module on test datum 4.

Test datum 5 is another case where a failure occurs. For this test datum, data depen-

Table 2: Information Flow Chains from Node 3 to Node 11 in Example

#	information flow chains
i	$(*, E, 3), (E, F, 4), (F, BP, 5), (BP, B, 6), (B, D, 10), (D, out, 11)$
ii	$(*, E, 3), (E, F, 4), (F, BP, 5), (BP, C, 7), (C, D, 10), (D, out, 11)$
iii	$(*, E, 3), (E, F, 4), (F, BP, 5), (BP, B, 8), (B, D, 10), (D, out, 11)$
iv	$(*, E, 3), (E, F, 4), (F, BP, 5), (BP, C, 9), (C, D, 10), (D, out, 11)$

dence transfer occurs up to node 5. Control dependence transfer occurs both to B at node 6 and to C at node 7. At node 10, data dependence transfer occurs from both B and C to D, and a failure is revealed at node 11.

These example test data illustrate two types of transfer – data dependence transfer and control dependence transfer. We define these concepts as follows.

> *Data dependence transfer* occurs from a node m that defines a potential failure variable V to a node n that uses V when the use of V results in computing an incorrect value at n.
>
> *Control dependence transfer* occurs from a branching node m to a node n that is control dependent on m when n is incorrectly selected and defines an incorrect value for some variable V that reaches the immediate forward dominator of m.

For a fault to cause a failure, transfer must occur from an original state potential failure along some information flow chain(s) to output. Transfer along an information flow chain involves data dependence transfer and/or control dependence transfer at each link in the chain. In the example, for test datum 4, transfer occurs along the information flow chain

$$(*, E, 3), (E, F, 4), (F, BP, 5), (BP, C, 7), (C, D, 10), (D, out, 11).$$

In general, there may be several information flow chains from a faulty node to a failure node. In the module in Figure 1, there are four information flow chains from node 3 to node 11, which appear in Table 2. Notice that more than one information flow chain may be executed by the same test datum. In this example, any test datum that selects the

True branch at node 5 executes both chains i and ii. Execution of a chain, however, does not imply that transfer occurs along it. In fact, transfer may occur along some, all or none of the executed chains. For test datum 4, both chains i and ii are executed but transfer only occurs along chain *i*. On the other hand, for test datum 5, transfer occurs along both chains *i* and *ii*.

3.2 Transfer Sets and Transfer Routes

3.2.1 Motivation

To fully model the process of how a fault becomes a failure, RELAY must model transfer of potential failures along multiple information chains. To illustrate the importance of determining all chains along which transfer may occur and distinguishing between potential transfer and actual transfer, let us consider the conditions that would guarantee transfer of an originated potential failure.

Consider first the simplest case where there is only a single information flow chain along which transfer could occur. In this case, there is at most one potential failure variable used at each node in the chain, since a node using multiple potential failure variables indicates transfer has occurred along a second information flow chain up to that node. The condition to guarantee transfer along a single chain from some faulty node to a particular failure node is the conjunction of the conditions to transfer the potential failure within each node in the chain along with the condition to execute the chain.

The necessary and sufficient condition to guarantee transfer within a single node is called a *computational transfer condition*. Here, we briefly overview the construction of computational transfer conditions when only a single potential failure variable is referenced at a node, first when the single potential failure variable is other than the dummy variable BP and then when it is BP. See [RT88, Tho91a] for a more complete discussion of computational transfer conditions.

At nodes where the potential failure variable is other than BP, the computational transfer condition is the conjunction of the transfer condition for each operator containing the potential failure variable in an operand. For example, consider the statement

$$X := (U + W) * (V + Z).$$

When V is the only potential failure variable at this statement, the faulty value must transfer through the addition operator (true) and then through the multiplication operator ($U + W \neq 0$). The computational transfer condition at the node is simply the conjunction of these individual conditions.

When the single potential failure variable used at a node is BP, indicating control dependence transfer, the computational transfer condition is $EXP_1 \neq EXP_2$, where EXP_1 is the value of the computation selected and EXP_2 is the value that would have resulted were the correct branch selected. For example, in Figure 1, if node 5 incorrectly selects the branch containing node 6 and G and A are not potential failure variables, the computational transfer condition at node 6 is $A * *G - 1 \neq A * G - 1$, which is

$$(A \neq 2 \text{ or } G \neq 2) \text{ and } (A \neq 0 \text{ or } G \neq 1).$$

Intuitively, we may argue by induction that the condition formed from the conjunction of computational transfer conditions at each node in the chain (along with the path condition to execute the chain) is *sufficient* to transfer an original state potential failure along the entire single chain. When there is not another chain that could be transferred along from the same faulty node to the same failure node, we may also argue that the condition formed from the conjunction of the computational transfer conditions at each node in the chain (along with the path condition to execute the chain) is *necessary* to transfer along this single chain. Thus, the conjunction of the computational transfer conditions is both necessary and sufficient to transfer along the entire chain, when there is a single chain.

When more than one chain exists along the same path, the conditions to guarantee transfer for information flow chains are more complicated. Consider again the module shown in Figure 1, and suppose that node 2 is faulty and should be $A := G + I$. This correct node is labeled $2'$ in the figure. From node 2 to node 11, there are four information flow chains, which appear in Table 3. If we construct the condition to transfer along a

Table 3: Information Flow Chains from Node 2 to Node 11 in Example

#	information flow chains	transfer set
i	$(*, A, 2), (A, B, 8), (B, D, 10), (D, out, 11)$	TS_X
ii	$(*, A, 2), (A, C, 9), (C, D, 10), (D, out, 11)$	
iii	$(*, A, 2), (A, B, 6), (B, D, 10), (D, out, 11)$	TS_Y
iv	$(*, A, 2), (A, C, 7), (C, D, 10), (D, out, 11)$	

chain as described above – that is, as if it were the only chain and thus without taking into consideration other chains that might be transferred along concurrently – we find that this approach is inadequate. Consider the condition to transfer along chain i. To transfer at node 8 from A to B, we must guarantee that $G \neq 0$. To then transfer from B to D at node 10, we must guarantee that $C \neq 0$. The condition that should "guarantee" transfer along this single chain is the conjunction of these two conditions along with a path condition to execute the chain:

$$G \neq 0 \ (\textit{at node 8}) \text{ and } C \neq 0 \ (\textit{at node 10}) \text{ and } F \geq G - 6 = (\textit{at node 5}).$$

Similarly, we could construct the conditions that should guarantee transfer along the single chain ii:

$$H \neq 0 \ (\textit{at node 9}) \text{ and } B \neq 0 \ (\textit{at node 10}) \text{ and } F < G - 6 = \textit{False} \ (\textit{at node 5}).$$

Both test data in Table 4 satisfy the original state potential failure condition for this fault $(H \neq I)$. Test datum 1 satisfies the "transfer" conditions derived above for both chains but fails to reveal a failure. Thus, the condition is not <u>sufficient</u> to transfer for either chain, nor are the conditions for the two chains together sufficient. On the other hand, test datum 2 does not satisfy the transfer condition for either chain but does reveal a failure; thus, the conditions for the chains are also not <u>necessary</u> to transfer.

From this example, we see the simple approach described above, which works when there is only one chain along which transfer could occur, is inadequate in the more general

Table 4: Illustrative Test Data Set for Fault at Node 2 in Example

		test data			evaluated variables							
	module	G	H	I	A	E	F	$F < G - 6$	B	C	D	*output*
1	faulty	1	-3	1	-2	-2	4	F	-3(8)	6 (9)	-18	-18
	correct	1	-3	1	2	-2	4	F	1 (8)	-6 (9)	-6	-6
2	faulty	1	-1	2	0	0	0	F	0(8)	0(9)	0	0
	correct	1	-1	2	3	0	0	F	2 (8)	-3 (9)	-6	-6

context. This is because at nodes where more than one potential failure variable is used, the computational transfer condition must take into account all potential failure variables. Otherwise, when combined, two or more potential failure variables may "interact" and mask out all the potential failures referenced at the node. Thus, to determine the necessary and sufficient conditions to transfer a potential failure and to fully model transfer behavior, we must take into consideration all chains transferred along as well as know along which chains transfer actually occurs.

3.2.2 Definitions

A *transfer set* is a collection of information flow chains with the following properties:

1. all chains start at the same faulty node;
2. all chains end at the same failure node and with output of the same designated variable;
3. there is a set of paths such that each path executes all the chains in the transfer set;
4. all chains executed by such a set of paths are included in the transfer set.

Consider again the example module shown in Figure 1. As noted in the previous subsection, if we consider a fault at node 2, there are four information flow chains to the output statement at node 11. For these four information flow chains, there are two transfer sets.

One transfer set TS_X consists of information flow chains (i,ii), executed by test data that select the false branch. The other transfer set TS_Y consists of chains (iii,iv), executed by test data that select the true branch.

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 nodes in a chain. Thus, we not only need to know which chains are actually transferred along, but more precisely, at which tuples in the chains transfer occurs. This is defined by "transfer routes".

Given a transfer set TS, let $Nodes(TS)$ be the set of nodes that are in tuples in the information flow chains in TS. Nodes that could appear more than once in an information flow chain are disambiguated by a subscript indicating the visit to the node on a path covering the chain.

> A *transfer route tr* of a transfer set TS is a subset of $Nodes(TS)$ such that:
>
> 1. all nodes for at least one information flow chain in TS are in tr;
> 2. for every node in tr, there exists a subchain from the faulty node to that node such that all nodes in the subchain are also in tr.

Given a transfer route tr, a node $n \in tr$ is called a *transferring node*. A node $n \in \{Nodes(TS) - tr\}$ is called a *non-transferring node*.

There may be several transfer routes for a particular transfer set. 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 non-transferring nodes in the transfer route, transfer fails or has failed at previous points such that the node references no potential failure variables.

Consider again the information flow chains listed in Table 3 for the example module with a fault at node 2. For transfer set TS_X consisting of the information flow chains (i,ii), there are three different transfer routes, which are provided in Table 5. Because a transfer route is associated with a transfer set, the actual potential failure variable at each node and sequence of transfers is implicit in the set of nodes. This information has been made

Table 5: Transfer Routes for Transfer Set TS_X of Example

route	explicit transfers
$tr_1 = \{8, 9, 10, 11\}$	transfer from A to B at node 8, transfer from A to C at node 9, transfer from B and/or C to D at node 10, transfer from D to output at node 11
$tr_2 = \{8, 10, 11\}$	transfer from A to B at node 8, *do not* transfer from A to C at node 9, transfer from B to D at node 10, transfer from D to output at node 11
$tr_3 = \{9, 10, 11\}$	*do not* transfer from A to B at node 8, transfer from A to C at node 9, transfer from C to D at node 10, transfer from D to output at node 11

explicit in the second column of Table 5 to assist in the discussion. The second test datum in Table 4 executes the first of the transfer routes enumerated above.

At nodes where two or more variables used at the node are potential failure variables, we say that "interaction" occurs. A transfer set identifies the nodes where interactions potentially occur. For tr_1 in Table 5, interaction occurs at node 10 where B and C are both potential failure variables. No interactions occur for transfer routes tr_2 and tr_3. In this example, interaction involves only data dependence, although interaction may also occur with control dependence and data dependence in combination.

It is important for us to note here a subtlety of transfer routes. Transfer routes are defined at the granularity of whole nodes rather than at the level of computations within a node. To elucidate this point, consider a statement containing a computation such as

$$A := (B * C) + (D * E),$$

and suppose that both B and D are potential failure variables at a node representing this statement. This node is either transferring, in which case either B or D or both transfers, or it is non-transferring, in which case, neither transfers. The different possibilities of transfer within the node are explicitly captured by the specific computational transfer conditions, not by the transfer route. This convention simplifies the transfer routes, decreases the number of routes for each transfer set, and allows transfer routes to be determined from an information flow graph.

It is beyond the scope of this paper to present the algorithms for identifying transfer sets and transfer routes; they may be found in [Tho91b]. The general approach for identifying transfer sets is to generate all information flow chains from a faulty node to a failure node and then determine the sets of chains with consistent path conditions. Identification of transfer routes involves a search on a graphical representation of a transfer set to generate legal combinations of transferring and non-transferring nodes. 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. To identify transfer sets and transfer routes in code involving loops, additional loop analysis must be performed. One approach to this loop analysis is also described in [Tho91b].

In summary, RELAY models how a fault causes a failure on execution of some test datum as follows:

1. Introduction of an original state potential failure at the faulty node;
2. Transfer of a state potential failure along potentially many information flow chains:
    (a) a transfer set is the set of information flow chains that may be transferred along concurrently;
    (b) a transfer route is a subset of nodes in a transfer set at which actual transfer occurs;
    (c) data dependence transfer and/or control dependence transfer occurs at transferring nodes;
    (d) transfer does not occur at non-transferring nodes;
    (e) interaction occurs at nodes where multiple potential failure variables are used.

4 Applying Information Flow Transfer

Information flow transfer as captured in the RELAY model may be used in several ways. Here we discuss two areas of application. First, RELAY may be used to guide testing; although computationally expensive, this may be useful for critical systems where the cost of failure is exceptionally high. Second, the rigor provided by RELAY can be used to evaluate fault-based testing methods and their underlying assumptions. The RELAY model provides insight to examine and guide empirical studies as well as an analytical perspective.

4.1 Testing based on the RELAY Model

The RELAY model captures the information required to originate and transfer a potential failure from fault to failure. Based on the model, we can derive a *failure condition*, which guarantees a fault originates a potential failure that transfers to produce incorrect output. To use such conditions in testing, we hypothesize the existence of a fault, derive the failure condition for the fault and test the module on test data satisfying these conditions. This process, under the assumptions discussed in Section 2, guarantees detection of the hypothetical fault if indeed it is a fault.

When constructing such conditions, we must be concerned with several problems. In addition to the general difficulty of selecting test data to satisfy a particular set of conditions, the conditions themselves may be fault dependent and thus require reconstruction for each fault being considered. Fortunately, in some cases fault independent conditions may exist while at other times sufficient (but not necessary) conditions can be constructed.

Using RELAY as a testing method is computationally expensive and thus applying to a large set of faults in a program is impractical. For critical software systems, however, where some failures may be exceptionally costly or intolerable, we can selectively apply a RELAY-based testing approach by focusing the method on system components that may affect critical aspects of the system behavior such as critical variables, statements, and modules. Although an expensive approach, we believe for critical systems, RELAY is a promising method that can provide insight into such systems and allow us to reason about and test them more effectively.

In this subsection, we demonstrate construction of failure conditions, focusing on information flow transfer, discuss fault dependence of the conditions, and outline a testing method based on RELAY.

4.1.1 Failure Condition

The *failure condition* is the necessary and sufficient condition to guarantee a fault causes a failure. The formula for a failure condition is summarized in Figure 2. The failure condition is the conjunction of the original state potential failure condition, which is presented in [RT88, Tho91a], and the condition to guarantee transfer, which we describe here. As demonstrated, it is necessary to consider transfer sets and transfer routes when developing the failure conditions rather than simply single information flow chains in isolation, because all potential failure variables at each node must be taken into consideration. The failure condition includes the disjunction of the transfer set conditions for all transfer sets from the hypothetically faulty node. A *transfer set condition* is the necessary and sufficient condition to transfer an original state potential failure to output along the particular transfer set. The transfer set condition is the conjunction of the transfer set *path condition,* which guarantees execution of all information flow chains in the set, and the disjunction of transfer route conditions (for all transfer routes of a transfer set), each of which guarantees a failure is revealed along the transfer route. 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. The computational transfer condition at a transferring node is the necessary and sufficient condition to guarantee transfer occurs from at least one of the potential failure variables at the node to the defined variable at that node.

Here we demonstrate construction of each of these components – computational transfer conditions, transfer route conditions and transfer set conditions. The conditions are written in terms of a constraint on one or more variables at a particular node – e.g., $G \neq I$ (at node 4). To actually find test data to satisfy this condition, symbolic evaluation must be performed to generate constraints in terms of input variables. We simplify our presentation below by skipping this step.

$$
\begin{aligned}
&\textit{failure condition}(\text{fault } f) \equiv \\
&\qquad \textit{original state potential failure condition}(f) \\
&\quad \wedge \ \Big(\bigvee_{TS_I(f)} \textit{transfer set condition}(TS_I) \Big) \\[1em]
&\textit{transfer set condition}(TS_I) \equiv \\
&\qquad \textit{path condition}(TS_I) \\
&\quad \wedge \ \Big(\bigvee_{tr_k \subset Nodes(TS_I)} \textit{transfer route condition}(tr_k) \Big) \\[1em]
&\textit{transfer route condition}(tr_k) \equiv \\
&\qquad \Big(\bigwedge_{n_i \in \textit{transferring nodes } (tr_k)} \textit{computational transfer condition}(n_i) \Big) \\
&\quad \wedge \ \Big(\bigwedge_{n_j \in \textit{non-transferring nodes } (tr_k)} \neg \textit{computational transfer condition}(n_j) \Big)
\end{aligned}
$$

Figure 2: Formula for Failure Condition

Referring back to the example in Figure 1, consider node 2 as hypothetically faulty. There are four information flow chains from node 2 to output at node 11, which are listed in Table 3. Associated with these information flow chains are two transfer sets, TS_X consisting of chains i and ii and TS_Y consisting of chains iii and iv.

To more concisely represent transfer routes, we will use a shortened notation and write

Table 6: Transfer Routes for Transfer Set TS_X of Example Module

tr_1 :	$(A, B, 8); (A, C, 9); (B + C, D, 10); (D, out, 11)$
tr_2 :	$(A, B, 8); \neg(A, C, 9); (B, D, 10); (D, out, 11)$
tr_3 :	$\neg(A, B, 8); (A, C, 9); (C, D, 10); (D, out, 11)$

a transfer route as a list of tuples of the form $(V_1 + V_2 + ... + V_k, W, n)$ for transferring nodes and of the form $\neg(V_1 + V_2 + ... + V_k, W, n)$ for non-transferring nodes, where V_i are the potential failure variables that are used at node n and W is the variable to which transfer occurs (or does not occur) at node n. Not included in the list of non-transferring tuples are non-transferring nodes that do not reference any potential failure variables because transfer has failed along all chains up to that node. For transfer set TS_X, the transfer routes, which are described in Table 5, are written in this notation in Table 6.

For each tuple in a transfer route, we must construct the computational transfer condition, which guarantees transfer to the variable defined at the node from the potential failure variable(s) referenced at the node. For the transfer routes for TS_X, these computational transfer conditions (*ctc*) are:

$$
\begin{aligned}
&ctc(A, B, 8) = (G \neq 0) \text{ at node } 8\\
&ctc(A, C, 9) = (H \neq 0) \text{ at node } 9\\
&ctc(B, D, 10) = (C \neq 0) \text{ at node } 10\\
&ctc(C, D, 10) = (B \neq 0) \text{ at node } 10\\
&ctc(B + C, D, 10) = (B * C) \neq (B' * C') \text{ at node } 10\\
&ctc(D, out, 11) = \text{true at node } 11
\end{aligned}
$$

The transfer route condition is formed by conjoining the computational transfer condition for each transferring node in the transfer route and the complement of the transfer condition for each non-transferring node. For tr_1, the transfer route condition (*trc*) (dropping the condition true and the node references) is:

$$\begin{aligned}trc(tr_1) &= ctc(A,B,8) \wedge ctc(A,C,9) \wedge ctc(B+C,D,10)\\ &\qquad \wedge ctc(D,out,11)\\ &= (G \neq 0) \wedge (H \neq 0) \wedge ((B * C) \neq (B' * C'))\end{aligned}$$

The transfer set condition is formed from the conjunction of the transfer set path condition with the disjunction of the transfer route conditions for all transfer routes associated with the transfer set. The path condition (pc) for TS_X selects the false branch at node 5 in the module and is:

$$\begin{aligned}pc(TS_X) &= \neg(F < G - 6) \text{ (at node 5)}\\ &= (F \geq G - 6) \text{ (at node 5)}\end{aligned}$$

Thus, the transfer set condition (tsc) for TS_X is:

$$\begin{aligned}tsc(TS_X) &= pc(TS_X) \wedge (trc(tr_1) \vee trc(tr_2) \vee trc(tr_3))\\ &= (F \geq G - 6) \wedge\\ &\quad (((G \neq 0) \wedge (H \neq 0) \wedge ((B * C) \neq (B' * C'))\\ &\qquad \vee((G \neq 0) \wedge (H = 0) \wedge (C \neq 0))\\ &\qquad \vee(G = 0 \wedge (H \neq 0) \wedge (B \neq 0)))\end{aligned}$$

We may similarly derive the transfer set condition for TS_Y. The disjunction of $tsc(TS_X)$ and $tsc(TS_Y)$ when conjoined with the original state potential failure condition (including the path condition to reach the fault) for some hypothetical fault at node 2 yields a failure condition, as summarized in Figure 2. The meaning of this condition may be summarized as follows:

- 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 any input;
- Inability to satisfy this failure condition means that the module being tested is equivalent to the hypothetically correct module, and the hypothetical fault is not a fault.

4.1.2 Fault Dependence

In Section 4.1.1, we demonstrate construction of the failure condition for a single hypothetical fault. Most likely, for a single node, we would want to consider several hypothetical faults in turn. In constructing the failure condition for each hypothetical fault at a node, we would like to be able to determine the transfer set condition once and then conjoin that condition to the original state potential failure condition for each fault that we decide to consider. Often, however, the transfer set condition is fault dependent, and we cannot do this.

For a transfer set condition to be fault independent, the computational transfer conditions must be fault independent. In most cases, where only one potential failure variable is referenced at a node, the computational transfer condition at the node is fault independent. This is true for transfer through boolean, addition, subtraction, multiplication, division, assignment operators and combinations of these operators. Computational transfer is fault dependent through relational, exponentiation and integer operators (e.g., DIV), although sufficient conditions that do not depend on the value of the potential failure transferring (and hence the fault) can often be derived. Necessary and sufficient computational transfer conditions have been derived for those operators where fault independence is possible and sufficient computational transfer conditions have been derived for some other operators [RT88, Tho91a].

Nodes where multiple potential failure variables are referenced indicate where information flow chains in a single transfer set potentially interact. At each interaction node, the computational transfer conditions at the node are usually fault dependent. In some cases, depending on the structure of the expression, sufficient fault independent conditions can be derived. Investigation of sufficient fault independent conditions is an area for further research. Such research might investigate the possibility of applying the results of Howden's *algebraic testing* [How78] or other results in algebra theory.

When the transfer set consists of more than one information flow chain, another approach for avoiding fault dependent conditions is to select a transfer route that has a fault independent transfer route condition. A single transfer route condition is sufficient to transfer a potential failure to output and provides a sufficient transfer set condition. It is important to remember that inability to satisfy such a sufficient condition, however, does not imply equivalence for the hypothetical fault since transfer could still occur for some other transfer route. For example, for TS_X, the transfer route conditions for tr_2 and tr_3 are fault independent but are not satisfiable, while the transfer route condition for tr_1, which is fault dependent, is satisfiable. Thus, to guarantee transfer, we would have to select test data for tr_3.

4.1.3 Testing and Analysis with Failure Conditions

The first step in applying RELAY to critical systems is identification of critical components. This information might be based on safety-critical or mission-critical analyses (such as those proposed by the British MOD Standard 0055 & 0056 [BMD91a, BMD91b]) or software safety analysis [LH83].

Given some group of critical portions of the code, we then construct transfer sets and transfer routes for this code by analyzing a program dependence graph, which includes control flow and data flow information. Next, failure conditions are constructed as described above for a set of hypothesized faults. We may use this failure condition along with the transfer route condition in several ways. First, a failure condition may be used to evaluate a previously selected test data set for its fault detection capabilities. Second, because a failure condition identifies and captures the potential effects of a fault, we can use the condition to assist us in reasoning about the system's behavior. We can analyze this failure condition and transfer route information to determine if a hypothetical fault or state potential failure could lead to a critical failure. A failure condition leading to a critical failure is similar to the "failure scenarios" constructed by software fault tree analysis [LH83]. Third, we may use the failure condition to direct selection of additional test data for execution.

Several optimization approaches should be considered in implementation. First, as previously noted, a number of faults may be hypothesized at a single node. Costs are reduced if those parts of the failure conditions that are independent of a potential fault are isolated and done once for all such hypothetical faults.

In addition, because of the cost of constructing the failure conditions, the failure condition should be constructed incrementally since it may become infeasible at any point. Thus, as each condition (path, origination, and computational transfer) is derived, it should be conjoined to the incremental failure condition and checked periodically for feasibility so as not to waste computation time constructing an already infeasible condition. There are several selections to reconsider when infeasibility is determined — e.g., another path segment to the hypothetically faulty node, another path segment to the node in the transfer set, another transfer route, or a different transfer set may be required.

Further, infeasibility of a failure condition for one transfer route may provide guidance as to what other transfer route would be feasible. For instance, if transfer at a particular node leads to an infeasible incremental failure condition and another transfer route is identical to this point but does not transfer at this node, that transfer route is likely to lead to a feasible failure condition. Likewise, differences between transfer sets can guide the selection of an alternative transfer set. This is an area for additional research.

4.2 Evaluation of Fault Based Testing Methods

The RELAY model can also be used to evaluate the fault detection capabilities of fault-based testing methods. Previously, we analyzed several methods in terms of their ability to reveal an original state potential failure [RT86]. Here, we analyze the information flow transfer of two fault-based testing methods that address at least some aspects of transfer. These two methods rely on various underlying assumptions, which we also analyze to understand the capabilities of the methods. Analytical evaluation should be complemented by empirical evaluation. The RELAY model can be used to examine the validity of empirical studies and their underlying assumptions as well as to suggest further studies.

4.2.1 Symbolic Fault-Based Testing

Morell's *symbolic fault-based testing* [Mor88] is a test data measurement method which measures the adequacy of a given test data set, selected by some other means, for detecting faults "seeded" into a module. Symbolic fault-based testing replaces an expression within a statement with a "symbolic" fault, creating an alternate module. Both the original module and the alternate module (with the symbolic fault) are symbolically evaluated along a particular path. Comparison of the symbolic computation for the original module and the alternate module yields a "[general] propagation equation". The propagation equation identifies hypothetical faults that would not introduce an error (originate a potential failure) that propagates (transfers) to output for the given test data set. This equation suggests additional test data that must be selected to detect the faults that would not have been detected by the given test data.

Same Path Assumption. Symbolic fault-based testing requires that the original and alternate modules execute the same path, referred to here as the "same path" assumption. Morell notes that his approach requires additional analysis to determine what faults would execute the same paths for the test data being evaluated but does not discuss how such analysis would be performed. The same path assumption implies that symbolic fault-based testing does not model control dependence transfer. Thus, this method is unable to measure test data adequacy where the chains from a fault to a failure include control dependence. Empirical studies of the nature of information flow and how often this assumption holds are needed.

Despite these weaknesses, Morell's approach warrants further investigation. Extension to Morell's work to capture control dependence would seem feasible by incorporating the symbolic path condition with the symbolic computation. Hybridization of the transfer ideas in RELAY with the symbolic fault-based testing ideas of Morell is an interesting area for further research.

4.2.2 Mutation Analysis

Mutation analysis, first described by DeMillo, Lipton and Sayward [DLS78], evaluates a test data set (for which the module being tested executes correctly) by comparing actual execution of the module being tested with the execution of alternate modules, each containing a seeded fault. The Mothra mutation analysis system [DGK$^+$88] automatically introduces a simple syntactic change, or "mutation", into the source code to produce an alternate module, or "mutant". Many different mutations are considered independently for each location in the module, thus creating hundreds (or even thousands) of mutants for each module. The system then executes each mutant on the given test data set to determine whether it produces different output results than the module being tested on at least one test datum, in which case the mutant is said to be "killed". Mutation analysis measures the adequacy of a test data set according to its ability to kill mutants and assigns a mutation score, which is the ratio of killed to non-equivalent mutants. A tester might then manually create test data to kill specific mutants that remain live. Mutation testing consists of augmenting the test data set, verifying output correctness of the module being tested, and killing mutants until the tester is satisfied with the mutation score.

Recognizing that creating test data is one of the most human-intensive activities in mutation testing, Offutt developed *constraint-based testing* which generates conditions that are "designed specifically to kill the set of mutants" [Off88]. For each mutation, test data is generated to satisfy the "reachability" and "necessity" conditions, which are necessary and sufficient to execute the hypothetical fault (mutated statement) and to originate a potential failure, respectively. No assistance, however, is provided for determining or satisfying the "sufficiency" condition, which is sufficient for transferring the originated potential failure to output. Offutt implemented a constraint-based test data generator within the Mothra system.

Coupling Effect Assumption. Mutation analysis has an underlying assumption, called the "coupling effect", which states that "test data detecting simple faults are sensitive enough to also detect more complex faults. In other words, complex faults are coupled to simple faults" [DLS78, Off88]. Based on this assumption, mutation analysis extrapolates the mutation score for single, simple mutations as a measure of the ability to detect complex faults. Thus, the coupling effect serves as the primary justification for the utility of mutation testing. Although not elaborated in the mutation testing papers, we believe the intuition being used here is that each complex fault contains at least one atomic fault and that test data that detect the atomic faults would be sufficient to detect the complex faults. Figure 3 schematically illustrates the coupling effect and how it would justify mutation analysis. The original module contains a complex fault at j, where the triangle should be a circle. Introducing simple mutations at every possible location in the program will result in at least one program that contains a mutated j, indicated by a subscripted triangle labeled "alternate #1". Mutation testing would select test data that distinguishes between alternate #1 and the original module. The coupling effect says that this test data would also distinguish between the original module with the triangle and the correct module with the circle.

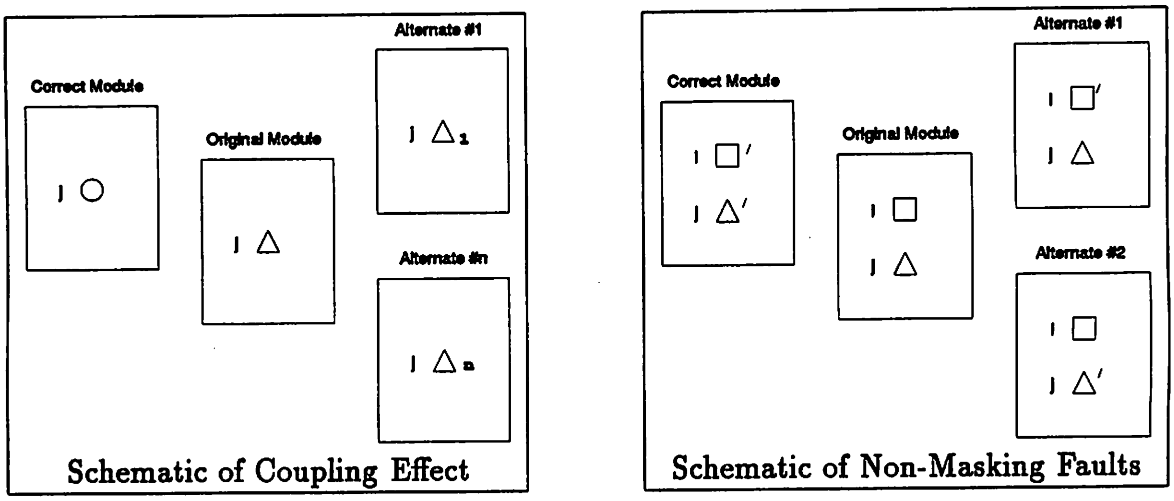

Figure 3: Comparison of Coupling Effect and Non-Masking Faults Assumptions

Using the Mothra system, Offutt performed an empirical study in an attempt to demonstrate the coupling effect and justify mutation analysis [Off89]. For the purposes of this study, he defined a simple fault as a single mutation applied at a single location and a complex fault as some combination of n simple faults, called a *n-order fault*, thus creating a *nth-order mutant*. Offutt studied three small Fortran-77 modules. For each module, he generated a test data set that killed a subset of the first-order mutants of the module. Second-order mutants were then constructed by combining the first-order mutants. The test data set was found to be sufficient to kill over 99% of the second-order mutants for each of the three modules. Offutt analyzed the second-order mutants that were not killed and found none that were particularly unlikely to be killed. To verify this, he repeated the experiment with different test data sets. Again, a high percentage of the second-order mutants were killed, and different second-order mutants remained live.

In evaluating this study, we must first ask whether the initial assumption that a complex fault can be modeled as n simple faults is valid. If one agrees that complex faults can be modeled as combinations of n simple faults, then the results do indeed provide some evidence for the coupling effect.

Certainly on some level, complex faults are composed of a set of simple faults. However, it is probably not true that they are composed of a set of unrelated simple faults. To the contrary, one would suspect that a complex fault is composed of a set of *related* atomic faults. Since it is not clear how this relationship changes the detectability of the complex fault, this study really does not provide evidence for or against the coupling effect. When we ask what it means to detect a complex fault, we must consider the dependences between the atomic faults that comprise the complex fault.

Consider two first-order mutants m_1 and m_2 and a second-order mutant (m_1, m_2). Offutt's results indicate that a test set T that kills m_1 (and m_2) also kills (m_1, m_2) 99% of the time. This means that there is some 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, however, do not indicate that the second-order mutant (m_1, m_2) is in any way coupled to the first-order mutant m_1, because we do not know if the same test datum that kills m_1 also kills (m_1, m_2). Further, for this test datum, we do not know whether m_2 was executed; if it was, we do not know whether it originated a potential failure; and if it did, we do not know whether that potential failure transferred to a point where it could interact with some potential failure caused by m_1. To answer these questions, we need knowledge of the dependences between m_1 and m_2. This knowledge is captured by RELAY's modeling of information flow.

Non-Masking Faults Assumption.

Offutt's results in the experiment described above do speak more directly to the issue of fault interaction and masking. An assumption of RELAY as well as most fault-based testing techniques is that multiple faults in a module do not interact in such a way as to mask each other (or that there is only one fault). Figure 3 illustrates the non-masking faults assumption (and how it differs from the coupling effect). The original module contains two faults at locations i and j. The non-masking faults assumption says that a test datum that would distinguish between the original module and alternate #1 would also distinguish between the original module and the correct module (and likewise for alternate #2). Note that in the non-masking faults assumption, it does not matter whether both faults are executed or if there are dependences between them, only that they do not mask each other.

In Section 3, we demonstrate how potential failures from a single fault could interact to mask out the potential failures and hide the fault. It is not difficult to see how potential failures from multiple faults could interact in the same way and mask one or both faults. Clearly, the assumption that multiple faults cannot mask each other does not always hold. Morell [Mor84] states several theoretical results about fault masking[5] and shows that in general it is undecidable whether or not multiple faults mask each other.

Since it is unlikely that a module being tested has a single fault, we must empirically examine the assumption that multiple faults do not mask each other. Offutt's results from the study described above suggest that multiple faults do not mask each other. In particular, Offutt's results show that 99% of the time, a test data set that kills a single atomic fault in isolation will also kill the fault in the presence of a second atomic fault. These results provide promising although not definitive support of the non-masking assumption. Additional study on a larger, representative suite of modules is needed. To adequately evaluate fault masking, it is imperative that this suite have representative patterns of information

[5]Morell uses the term "coupling".

flow. Further, such studies need to examine the assumption for a single test datum rather than over a test data set. If such studies demonstrate that multiple faults usually do not mask each other, this strengthens the case for fault-based testing. If, on the other hand, the assumption does not hold, then we must consider the effect of additional faults on the detectability of a particular fault. We feel results from an investigation into sufficient fault independent computational transfer conditions for single faults may also be applicable to the problem of multiple faults, perhaps allowing the assumption to be relaxed.

Weak Mutation Hypothesis.

To justify constraint-based testing, Offutt argues for the "weak mutation" hypothesis, which states that "it is unlikely that an intermediate incorrect value is masked out at a later computation" [Off88]. When this assumption holds, a test datum that executes a faulty expression and originates a potential failure transfers the potential failure to output.

To support the weak mutation hypothesis, Offutt argues probabilistically about p, the probability that a test datum satisfying the reachability and necessity conditions also satisfies the sufficiency condition. The probability of not satisfying the sufficiency condition on n attempts is $(1-p)^n$, and the probability of at least one "success" is thus $\Gamma = 1-(1-p)^n$. Offutt argues that Γ grows very quickly, because p is positive. This, he concludes, means that even if the probability of detecting a mutant on a test datum is very low, there is a high probability that if several test data are selected to satisfy the reachability and necessity conditions, then at least one will also satisfy the sufficiency conditions. Thus, Offutt argues that selecting multiple test data that originate a potential failure implicitly addresses transfer and no additional techniques are necessary.

This same argument, however, could be applied to selecting test data that satisfies any portion of the failure condition, for example, just the reachability condition. We could use Offutt's argument to conclude that multiple test data that satisfy the reachability condition have a high probability of detecting the fault. Thus, there would be no reason to do more than statement testing with multiple test data for each statement. It is hard to disagree with the obvious conclusion that the more testing the higher the probability of detecting a fault. Although the question of what portions of failure conditions are probably met by random test data [Woi91] is important, Offutt's probabilistic argument says nothing about this. The RELAY model and the inherent complexity of failure conditions, on the other hand, shows how difficult it is to guarantee fault detection. Clearly from the examples in the previous sections, test data that satisfies the original state potential failure condition would not always satisfy the transfer conditions. The question is, however, how often does the weak mutation hypothesis hold in testing. Both Offutt and Marick have undertaken empirical studies to address this question.

Offutt performed an empirical study that examines the "precision" P of a test data set, which Offutt claims estimates p, the probability, discussed above, that satisfying the reachability and necessity conditions also satisfies the sufficiency condition. He reports P between .61 and .94 and concludes that, although 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" [Off88].

From a testing point of view, these results are reassuring. Because Offutt's results involve small modules, however, additional studies are required. Furthermore, examination of the modules indicates a high proportion of conditional statements. Insight from RELAY tells us that in this case, transfer of a potential failure involves much more control dependence transfer than data dependence transfer. It is not clear that these information flow patterns and the size of the modules studied are representative. Also, Offutt's implementation of constraint-based testing does not handle many types of constraints (such as conditions involving internal variables or nested conditions) and thus excludes many commonly occurring types of faults and forms of transfer. Since transfer conditions are often fault dependent, it is not clear that the results hold more generally for all faults. Finally, and most importantly, even if the weak mutation testing hypothesis does hold 61% or more of the time, we cannot assume that simply selecting multiple test data solves the problem, without investigating the question of why the weak mutation hypothesis frequently fails. If the hypothesis fails because for some constructs transfer is particularly difficult, then we could be selecting a lot of test data that does not transfer.

Marick also evaluated the weak mutation hypothesis and addressed some of the factors involved when transfer fails [Mar91]. Four modules, containing between 9 and 206 lines, were selected from each of five widely used programs. Several simple faults were seeded into each module, and the code was analyzed by hand to determine the circumstances under which the effect of the fault would be observable. Marick found that the weak mutation hypothesis always holds in 50% of the cases. He also found that the weak mutation hypothesis almost holds in 20% of the cases — that is, a fault would be detected in almost all cases by test data that originates a potential failure. In 16% of the cases, the weak mutation hypothesis does not hold, but branch coverage would detect the fault, except in rare cases. In 6% of the cases, "specification-based" testing (such as equivalence class testing with boundary value coverage) would detect the fault where neither weak mutation testing nor branch coverage would. Combining weak mutation testing, branch coverage, or specification-based testing would have been inadequate in only 8% of the cases.

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 module containing the fault. He found the type of module containing the fault to be the only significant factor. In particular, for modules that only return a boolean value, he found the percentage of seeded faults detected by weak mutation testing and branch coverage to be significantly lower than faults in other types of modules. From the point of view of information flow transfer, this is not surprising because these faults are more likely to be masked out due to the collapsing of the value space into binary (boolean) values. This information suggests that one area where constructing the transfer condition is particularly important is boolean expressions. For these expressions, 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, the analysis techniques are not fully described, and the results are somewhat anecdotal. While we feel that the modules examined are more representative than those used by Offutt, the size of the study is quite modest and must be interpreted as initial results. Nonetheless, the study asks some very interesting questions. In particular, it asks how different types of constructs effect the weak mutation hypothesis.

Our future research includes empirical studies that investigate whether certain constructs in the code are more prone to coincidental correctness by looking at transfer through different information flow constructs. The RELAY model provides us with a new perspective from which to analyze the process of transfer of a potential failure to output. Such studies should provide insight into how to handle different types of code. If such code is identified, then some partial information flow transfer conditions through such code, resulting in a mutation approach between strong and weak, known as a firm mutation approach [WH88], may prove sufficient to achieve high fault detection.

5 Major Contributions and Summary

This paper presents the RELAY model of faults and failures, focusing on transfer of an incorrect intermediate state, or potential failure, from a faulty statement to output. Transfer occurs along information flow chains, where each link in the chain involves data dependence transfer and/or control dependence transfer. RELAY models the multiple information flow chains that may be concurrently transferred along with transfer sets, which identify possible interaction between potential failures. RELAY models actual interaction with transfer routes. Transfer sets and transfer routes form the framework that unifies the components of transfer.

While previous work in fault-based testing recognized the two steps of introducing an incorrect state and transferring an incorrect state to output, RELAY fully describes the complexity of these steps. In particular, some previous research recognized data flow transfer, but RELAY provides an in-depth investigation of the role of control dependence transfer and of the interaction between control dependence and data dependence transfer. Moreover, while other research has only considered transfer along a particular path, RELAY considers how transfer may occur concurrently along several intersecting information flow chains. Interactions occur at these intersection points and may mask potential failures. The RELAY model pulls together research in fault-based testing, data flow path selection, program slices [Wei84], and program dependence analysis [PC90].

The RELAY model provides an interesting basis for substantial future work in software analysis and testing. One application of the model is to determine the condition that must be satisfied to guarantee detection of a fault. This failure condition may be used to evaluate previously selected test data, guide selection of additional test data, or analyze fault propagation through a module. It can be selectively applied, and therefore can be used in testing highly critical systems by focusing analysis on critical modules, statements, or variables. Additional research is needed in how the comprehensive transfer information provided by RELAY may be used in guiding testing. In addition, we suggest extending Morell's symbolic fault-based testing, developing sufficient computational transfer conditions for multiple faults, and investigating the weak mutation hypothesis, multiple fault interaction and complex faults.

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.

[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.

[BMD91a] *The Procurement of Safety-Critical Software in Defence Equipment.* British Ministry of Defence, Interim Defence Standard 00-55, Issue 1, April 1991.

[BMD91b] *Hazard Analysis and Safety Classification of the Computer and Programmable Electronic Systerm elements of Defence Equipment.* British Ministry of Defence, Interim Defence Standard 00-56, Issue 1, April 1991.

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

[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.

[How78] William E. Howden. Algebraic program testing. *Acta Informatica*, 10, October 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.

[LH83] Nancy G. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, 9(5):569–579, September 1983.

[Mar91] Brian Marick. The weak mutation hypothesis. In *Proceedings of the Fourth Symposium on Testing, Analysis and Verification*, pages 190–199, October 1991.

[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.

[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.

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

[RT86] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the relay model of error detection. Technical Report 86-65, Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[RT88] Debra J. Richardson and Margaret C. Thompson. The relay model of error detection and its application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, July 1988.

[Tho91a] Margaret C. Thompson. *An Investigation of Fault-Based Testing Using the* RELAY *Model.* PhD thesis, University of Massachusetts at Amherst, May 1991.

[Tho91b] 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.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.

[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.

[Woi91] Denise M. Woit. Realistic expectations of random testing. Technical Report CRL No. 246, Telecommunications Research Institute of Ontario, McMaster University, Hamilton, Ontario, April 1991.

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