An Information Flow Model of Fault Detection

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

*Department of Computer Science
University of Massachusetts
Amherst, MA 01003

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

Abstract

RELAY is a model of how a fault causes a failure on execution of some test datum. This process begins with introduction of an original state potential failure at a fault location and continues as the potential failure(s) transfers to output. Here we describe the second stage of this process, transfer of an incorrect intermediate state 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 concurrent transfer along multiple information flow chains with transfer sets, which identify possible interaction between potential failures, and with transfer routes, which identify actual interactions. Transfer sets, transfer routes, and control dependence transfer are unique to the RELAY model.

The model demonstrates that the process of potential failure transfer is extremely complex and full analysis of real programs may not be practical. Nonetheless, RELAY provides insight into testing and fault detection and suggests an approach to fault-based testing and analysis that may be warranted for critical systems software.

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.

ACM-ISSTA'93-6/93/Cambridge, MA, USA
© 1993 ACM 0-89791-608-5/93/0006/0182...$1.50

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 test 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 RELAY model builds upon the creation/propagation model developed by Morell [Mor84]. For a fault to cause a failure, execution on some test datum causes intermediate incorrect values to be computed that eventually result in a failure. In the RELAY model, an intermediate incorrect value is termed a "potential failure", since the value may or may not cause a failure. In previous work [RT88, RT93], we describe how a potential failure "originates", or is introduced, in the smallest evaluable subexpression containing the fault and then must "transfer" through all subsequent operations in the statement that depend directly or indirectly on the faulty value until the entire statement evaluates to an incorrect value. Transfer through operators within a statement is called "computational transfer". Potential failures occur when a subexpression of a statement or an entire statement evaluates to an incorrect value. In this paper, we are interested in tracking "state potential failures", which are potential failures

that cause the final value of a statement to be incorrect, thereby effecting the "state" of the program. A variable whose value is incorrect is called a "potential failure variable". When a statement that contains a fault evaluates to an incorrect value, an "original state potential failure" is introduced. If the value of the variable is output, it is called a "failure variable".

In our current research, we have focused on how an original state potential failure introduced at some faulty statement transfers along information flow to an output statement and thus results in a failure. This process, called 'information flow transfer", is composed of the components of "data dependence transfer" and/or "control dependence transfer". Information flow transfer can occur simultaneously along more than one chain of information flow to the same output statement. This phenomenon of multiple chain transfer is captured with the concepts of "transfer sets" and "transfer routes". Transfer sets, transfer routes, and control dependence transfer are unique to the RELAY model. The information flow transfer aspects of the RELAY model are the subject of this paper.

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 sometimes called "weak mutation testing" [How82, Bud83]. Weak mutation testing, 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 sometimes called "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 statement, a "necessity" condition for introducing an error, and a "sufficiency" condition for then producing a failure. Offutt's reachability and necessity conditions are similar to Morell's creation condition, and his sufficiency condition is similar to the propagation condition. Neither of these models fully captures how an incorrect state once introduced 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, RT93]. RELAY extends Morell's model and Offutt's definitions by identifying and explicitly describing the ways an incorrect state may transfer during execution and how this transfer occurs along multiple chains of information flow.

Throughout this paper, we illustrate the components of the model with examples. To simplify and clearly present the ideas, we use what are obviously trivial modules and faults. As is clear from this exploration of information flow transfer on just simple modules, the process of potential failure transfer is extremely complex and full analysis of real programs may not generally be practical. Nonetheless, RELAY provides insight into the difficulty of guaranteeing fault detection. This insight is imperative to understanding what fault detection capabilities are lost by methods that do not consider fully the complexity of information flow transfer. Furthermore, although computationally expensive, RELAY suggests an approach to fault-based testing and analysis that may be warranted for critical systems software. Such an application is outlined here. We have also found that RELAY provides a basis for evaluating the transfer capabilities of fault-based testing approaches as well as highlighting areas requiring empirical study. This latter application is beyond the scope of this paper and is discussed elsewhere [TRC92, Tho91a].

This paper is organized as follows. Section 2 defines some related terminology and outlines the underlying assumptions of the model along with some additional simplifying assumptions for this presentation. Section 3 discusses the components of information flow transfer and how these components fit together in the model. Section 4 discusses one application of the model, testing and analysis of critical software systems. Section 5 summarizes the major contributions of RELAY and concludes with research directions suggested by this work.

2 Terminology and Assumptions

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*, G_M, 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 p may be executed on some input x [2]. A *state* 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 the state 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 [FOW87, PC90] for a more complete discussion of program dependence.

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 assigns a value to V. A *use of* V is associated with each node n in G_M that represents a statement that accesses the value of V. 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 is assigned by the statement represented by n, and the set *used(n)*, which is the set of all variables whose value is 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 for V at node m *reaches* a node n if and only if there is a path $(m) \cdot p \cdot (n)$ [3] such that for all nodes l in p, $V \notin defined(l)$ [4].

A node n is *(directly) data dependent* on a node m if and only if there is a definition for V at node m that reaches a use for V at node 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 [5]. Node 9 is data dependent on node 2 because A is defined at node 2, used at node 9, and there is a path such that the definition of A at node 2 reaches the use of A at 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.

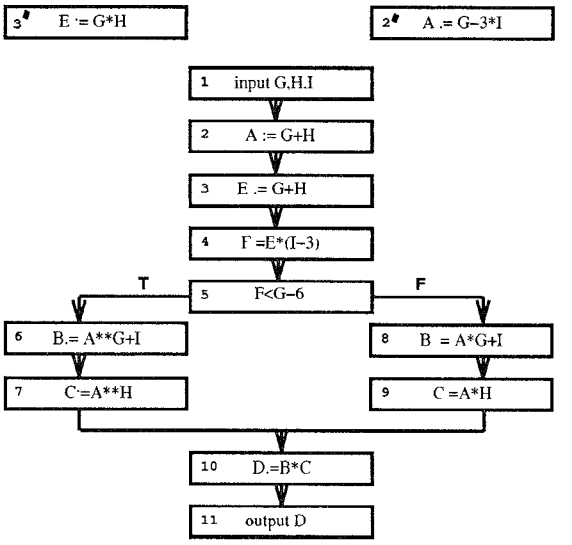

Figure 1: Example Module

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 $(*, d_1, n_1), (u_2, d_2, n_2), ..., (u_{|X|}, d_{|X|}, n_{|X|})$, 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 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 and would be distinguished by a subscript. 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 Assumptions

Our model relies on the "competent programmer" hypothesis [ABD+79, BDLS78], which says that the mod-

[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.

[3] $q \cdot p$ represents the concatenation of path q with path p.

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

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

ule 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 disallows changes that alter the information flow from the faulty node and thus the transfer requirements. This 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. The model allows evaluation and possible relaxation of this assumption. Such an evaluation is shown in [TRC92, Tho91a].

3 Information Flow Transfer

The two components of information flow transfer are "data dependence transfer" and "control dependence transfer". Informally, data dependence transfer occurs when the use of a potential failure variable at some node results in an incorrect value being computed at that node. Control dependence transfer occurs when the incorrect selection of a path results in an incorrect value being assigned to a variable computed along the path.

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. Thus, 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 from that assigned 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 dependence 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 more completely as follows.

		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	6 (8)	6 (9)	36	36
	correct	1	2	3	3	2	0	F	6 (8)	6 (9)	36	36
2	faulty	1	2	4	3	3	3	F	7 (8)	6 (9)	42	42
	correct	1	2	4	3	2	2	F	7 (8)	6 (9)	42	42
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	4 (6)	9 (7)	36	36
	correct	1	2	1	3	2	-4	F	4 (8)	6 (9)	24	24
5	faulty	4	0	2	4	4	-4	T	258 (6)	1 (7)	258	258
	correct	4	0	2	4	0	0	F	18 (8)	0 (9)	0	0

Table 1: Test Data Set for Fault at Node 3

Data dependence transfer occurs from a node m that defines a potential failure variable V to a node n that uses V when the value of V defined at node m reaches node n, and 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 occurs only along chain ii. On the other hand, for test datum 5, transfer occurs along both chains i and ii.

3.1 Transfer Sets and Transfer Routes

To fully model the process of how a fault becomes a failure, RELAY must model transfer of potential failures along multiple information chains. It is important to determine all chains along which transfer may occur and to distinguish between potential transfer and actual transfer. To illustrate this, 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*. See [RT88, Tho91a] for a discussion of computational transfer conditions and their construction. 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.

#	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	$(*,E,3),(E,F,4),(F,BP,5),(BP,C,9),(C,D,10),(D,out,11)$

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

Thus, when there is a single chain, the conjunction of the computational transfer conditions is both necessary and sufficient to transfer along the entire 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 $A := G - 3 * I$ [6]. 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 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 chain is the conjunction of these two conditions along with a path condition to execute the chain:

$$G \neq 0 \ \ (at\ node\ 8) \ \ and \ \ C \neq 0 \ \ (at\ node\ 10)$$
$$and \ \ F < G - 6 = False \ \ (at\ node\ 5).$$

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

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

Both test data in Table 4 introduce an original state potential failure for this fault. Test datum 1 satisfies the "transfer" conditions derived above for both chains but fails to reveal a failure. Thus, the condition is not sufficient 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 necessary to transfer.

[6] Node 3 now is assumed correct for this example.

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 context. This is because at nodes where more than one potential failure variable are 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 on which chains transfer actually occurs.

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". More completely, we define these concepts below.

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.

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:

#	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	$(*, A, 2), (A, C, 7), (C, D, 10), (D, out, 11)$	

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

		test data			evaluated variables							
	module	G	H	I	A	E	F	$F < G - 6$	B	C	D	*output*
1	faulty	1	2	2	3	3	-3	F	5(8)	6(9)	30	30
	correct	1	2	2	-5	3	-3	F	-3(8)	-10(9)	30	30
2	faulty	1	-1	0	0	0	0	F	0(8)	0(9)	0	0
	correct	1	-1	0	1	0	0	F	1(8)	-1(9)	-1	-1

Table 4: Test Data Set for Fault at Node 2

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, satisfies 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 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, shown in Table 3. 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. 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 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.

To represent transfer routes more concisely, we will use a shortened notation and write 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. The transfer routes written in this notation are shown in column three of Table 5.

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. A transfer route identifies the nodes where actual interactions 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

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

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

Table 5: Transfer Routes for Transfer Set TS_X of Example

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 transfers or both transfer, or it is non-transferring, in which case, neither transfers. The different possibilities of transfer within the node are explicitly captured by the 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].

3.2 Summary of the Relay Model

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 one or more 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

4.1 Failure Condition

The RELAY model describes the information required to originate and transfer a potential failure from fault to failure. This information for a particular fault may be captured in a *failure condition*, which guarantees a fault originates a potential failure that transfers to produce incorrect output. To develop such conditions for use in testing and analysis, we hypothesize the existence of a fault and then derive the failure condition for the fault.

The formula for a failure condition is summarized in Figure 2. The failure condition is the conjunction of the original state potential failure condition, the derivation of which is presented in [RT88, Tho91a], and the condition to guarantee transfer, the construction of which is demonstrated here. For a particular fault, the meaning of the failure 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;

- If known to be unsatisfiable, this failure condition means that the module being tested is equivalent to the hypothetically correct module, and the hypothetical fault is not a fault.

Note that if the condition cannot be solved because of the complexity of the condition, then we can draw no conclusion about the existence or absence of the hypothetical fault.

4.2 An Example

Here we show the transfer set condition for TS_X in our example with node 2 as hypothetically faulty. 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 values. We simplify our presentation below by skipping this step.

When constructing a failure condition, 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. Fault independence and dependence of the conditions is discussed more fully in [TRC92, Tho91a].

The transfer set condition for TS_X is derived in the following 4 steps.

Step 1: Construct computational transfer conditions (ctc) [7]

$ctc(A, B, 8) = (G \neq 0)$ at node 8
$ctc(A, C, 9) = (H \neq 0)$ at node 9
$ctc(B, D, 10) = (C \neq 0)$ at node 10
$ctc(C, D, 10) = (B \neq 0)$ at node 10
$ctc(B + C, D, 10) = (B * C) \neq (B' * C')$ at node 10
$ctc(D, out, 11) = (true)$ at node 11

[7] B' and C' represent the values of B and C, respectively, in the hypothetically correct module.

failure condition(fault f) $\equiv$

$$\textit{original state potential failure condition}(f) \wedge \Big(\bigvee_{TS_I(f)} \textit{transfer set condition}(TS_I)\Big)$$

transfer set condition(TS_I) $\equiv$

$$\textit{path condition}(TS_I) \wedge \Big(\bigvee_{tr_k \subset Nodes(TS_I)} \textit{transfer route condition}(tr_k)\Big)$$

transfer route condition(tr_k) $\equiv$

$$\Big(\bigwedge_{n_i \in transferring\ nodes\ (tr_k)} \textit{computational transfer condition}(n_i)\Big) \wedge \Big(\bigwedge_{n_j \in non\text{-}transferring\ nodes\ (tr_k)} \neg\textit{computational transfer condition}(n_j)\Big)$$

Figure 2: Formula for Failure Condition

Step 2: Construct transfer route condition (trc)[8]

$$\begin{aligned} trc(tr_1) &= ctc(A, B, 8) \wedge ctc(A, C, 9) \wedge \\ &\quad ctc(B + C, D, 10) \wedge ctc(D, out, 11) \\ &= (G \neq 0) \wedge (H \neq 0) \wedge ((B * C) \neq (B' * C')) \\ trc(tr_2) &= ctc(A, B, 8) \wedge \neg ctc(A, C, 9) \wedge ctc(B, D, 10) \\ &\quad \wedge ctc(D, out, 11) \\ &= (G \neq 0) \wedge (H = 0) \wedge (C \neq 0) \\ trc(tr_3) &= \neg ctc(A, B, 8) \wedge ctc(A, C, 9) \wedge ctc(C, D, 10) \\ &\quad \wedge ctc(D, out, 11) \\ &= (G = 0) \wedge (H \neq 0) \wedge (B \neq 0) \end{aligned}$$

Step 3: Construct transfer set path condition(pc) [9]

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

Step 4: Construct transfer set condition (tsc)

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

We may similarly derive the transfer set condition

[8] dropping the condition *true* and the node references

[9] The transfer set path condition guarantees execution of all information flow chains in the set.

for TS_Y. As summarized in Figure 2, the disjunction of $tsc(TS_X)$ and $tsc(TS_Y)$ when conjoined with the original state potential failure condition (which includes the path condition to reach the fault) for some hypothetical fault at node 2 yields a failure condition.

4.3 Applying Failure Conditions

Using RELAY as a testing and analysis method is computationally expensive and thus applying it to a large set of faults at all locations 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. 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]. Testing approaches must determine and test for the potentially catastrophic faults associated with these components. For example, in an x-ray machine, the component controlling the level of radiation requires particular scrutiny, and faults that could lead to lethal doses, if possible, should be identified and analyzed.

Given an identified section of critical code, we then construct transfer sets and transfer routes for this code by program dependence analysis. Because construction requires analysis of only the dependence relations between the statements, it is more feasible to analyze transfer across modules, and we may potentially reuse the analysis for several hypothetical faults. To support testing and analysis of critical systems, we may use the failure conditions 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.

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 fact that 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 [Nta81, Nta82, Nta84, LK83, CPRZ86, RW85], program slices [Wei84], and program dependence analysis [PC90].

The RELAY model provides an interesting basis for future work in software analysis and testing. Additional research is needed in how the comprehensive transfer information provided by RELAY may be used in guiding testing. Although we have hypothesized its applicability to safety critical systems, in depth studies are needed. Another area of future research is in empirical studies. The rigor of the RELAY model has been beneficial in demonstrating weaknesses in previous studies on fault based testing approaches [TRC92, Tho91a]. Given the insight of RELAY, we hope to investigate whether certain constructs in the code are more prone to coincidental correctness by looking at transfer through different information flow constructs. 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 firm mutation testing [WH88], may prove sufficient to achieve high fault detection.

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 System 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.

[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.

[DLS79] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Program mutation: A new approach to program testing. In *InfoTech State of the Art Report: Software Testing, Vol. 2*, pages 107–128, 1979.

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

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(5):319–349, July 1987.

[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.

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

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

[Nta81] Simeon C. Ntafos. On testing with required elements. In *Proceedings of COMPSAC '81*, pages 132–139, November 1981.

[Nta82] Simeon C. Ntafos. On required element testing. Technical Report 123, Computer Science Program, University of Texas at Dallas, November 1982.

[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.

[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.

[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.

[RT93] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the relay model of faults and failures. *to appear IEEE Transactions on Software Engineering*, May 1993.

[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.

[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.

[TRC92] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. Information flow transfer in the RELAY model. Technical Report 92-62, Computer Science, University of Massachusetts, Amherst, August 1992.

[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.

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