SOFTWARE ERROR ANALYSIS: A REAL CASE STUDY INVOLVING REAL FAULTS AND MUTATIONS

Muriel DARAN
LIS - Technicatome

Pascale THÉVENOD-FOSSE
LAAS - CNRS

7 Avenue du Colonel Roche
31077 Toulouse Cedex - FRANCE

mdaran@laas.fr, thevenod@laas.fr

ABSTRACT

The paper reports on a first experimental comparison of software errors generated by real faults and by 1st-order mutations. The experiments were conducted on a program developed by a student from the industrial specification of a critical software from the civil nuclear field. Emphasis was put on the analysis of errors produced upon activation of 12 real faults by focusing on the mechanisms of error creation, masking, and propagation up to failure occurrence, and on the comparison of these errors with those created by 24 mutations. The results involve a total of 3730 errors recorded from program execution traces: 1458 errors were produced by the real faults, and the 2272 others by the mutations. They are in favor of a suitable consistency between errors generated by mutations and by real faults: 85% of the 2272 errors due to the mutations were also produced by the real faults. Moreover, it was observed that although the studied mutations were simple faults, they can create erroneous behaviors as complex as those identified for the real faults. This lends support to the representativeness of errors due to mutations.

1. INTRODUCTION

Faults can be created at any time in any phase of the software development (during the requirement definition, design phase, coding phase, etc.). They result in **faults** in the program source code such as faulty instruction(s) or data, missing or extra instruction(s), etc. During program execution, when the faulty instruction (or instruction sequence or data) is triggered by an appropriate input pattern, the fault becomes active and may produce one or several errors (incorrect internal states). An **error** may propagate by creating other new error(s); if and when the erroneous data affect the output result(s), i.e. the program supplies wrong output value(s), a **failure** occurs, thus revealing the presence of the fault [Lap 92].

Originally defined by DeMillo [DeM 78], mutation faults have been largely used in previous work either to assess the adequacy of test cases, or to study how a fault creates an error upon activation and the ways this error transfers through execution to an eventual output result. In particular, these investigations have given rise to techniques such as PIE [Voa 92a] or to models such as the RELAY model [Ric 93] applied to evaluate the fault revealing power of test cases or to analyze the adequacy of test data selection criteria in revealing particular faults. Other empirical studies, focused on the effectiveness of mutation testing in detecting errors, have shown that test data generated to reveal simple faults (such as 1st-order mutations) could reveal more complex mutations [Off 89] or even complex known programming faults [DeM 94]. But, none of these studies discussed the controversial issue of the representativeness of mutations with respect to real faults.

This paper is not intended to investigate whether or not mutations constitute a fault model representative of real faults. Indeed, the answer to this question is likely to be negative. But, *does it mean that errors and failures produced by mutations are not similar to those related to real faults?* This is the issue we address here. In [DeM 94], the authors noticed that mutations could cause important variations in the program internal state throughout all the executions carried out during testing. Another purpose of this paper is to explore the nature of these variations and compare them to those produced by known real faults.

The experiments have been conducted on a *critical program from the civil nuclear field* previously involved in our work on software statistical testing [The 93]. The program under study is a version developed by a student from the industrial specification. The experiments involve 12 known real faults and 24 mutations. Thus, 36 faulty versions of the program — each version contains either one real fault or one mutation — were executed on at least one test sequence made up of several input patterns. A noticeable feature of this program is that it is a program with memory. As a

ISSTA '96, San Diego CA USA
© 1996 ACM 0-89791-787-1/95/01..$3.50

result, errors may propagate from one program execution to the following one(s). Errors generated during successive executions of a faulty program were identified by comparison of its execution trace to the correct program execution trace.

Two data bases — one related to the real faults and the other one to the mutations — were constituted in order to record the errors and gather information about their behaviors: how they were created, how the corresponding erroneous data were used in the successive executions. Analyses of these data bases were conducted according to two main objectives: 1) identify the fault behaviors and the error behaviors due to real faults on the one hand, and to mutations on the other hand; 2) identify similarities and discrepancies between errors produced by real faults and errors produced by mutations, and between their respective behaviors. The experimental results show that the errors generated upon activation of the mutations and the behaviors they can exhibit are representative of those observed with the real faults. To our knowledge, this study represents the first detailed comparative analysis between errors generated by real faults and errors generated by mutations on a target program.

The paper is organized as follows. Definitions that support the analyses of fault and error behaviors are given in *Section 2*. *Section 3* describes our experimental framework. *Section 4* reports on the analyses performed on the data bases. Main results are summarized in *Section 5* where we conclude with research directions suggested by this work.

2. Background and Definitions

The study of the manifestation mechanisms of faults, errors and failures involves three steps: 1) fault activation; 2) error propagation; 3) failure occurrence. The second step calls for internal behaviors of the program during execution which can be quite complex: an error may disappear before being detected (i.e. before failure), several errors may compensate each other preventing further propagation, etc. The definition of the internal state of a program given below will allow us to clarify the notion of error as it will be used in the paper, in order to facilitate the analysis of propagation mechanisms.

2.1 Program internal state

The **program internal state** at a point during execution is defined by the values of all variables (global variables, internal variables and output variables) and the value of the program counter which indicates the next instruction to be executed. The execution of an instruction is considered here to be atomic, i.e. the program internal state can only be viewed before and after each instruction. For a program with n variables, the program internal state, denoted PS, is represented by a set of n+1 pairings:

$$PS = \{(var_1, val_1), \ldots, (var_n, val_n), (PC, x)\}$$

where val_i ($i = 1, \ldots, n$) is the current value of variable var_i (including the "undefined" value) and x is the value of the program counter PC, i.e. the address of the next instruction to be executed. At the start of a sequence of program executions on a set of input patterns ($t_1, \ldots, t_p$), i.e. before the execution on t_1, all variables are undefined; then, before the execution on a subsequent input t_j ($j > 1$), PS keeps the value it reached after completion of the execution on t_{j-1}, that is, the program internal state is memorized between two consecutive program executions.

This notion of program internal state differs from the *program state* used by Zeil [Zei 89] which does not include the PC value. It is similar to the *program data state* defined by Voas [Voa 92b], the difference lies in the fact that all variables of the program data state have undefined values before a program execution on an input begins. Indeed, Voas's program data state would not allow us to analyze possible error propagation between successive program executions.

2.2 Error, error trace, error flow

In this paper, the term **error** is used to denote *one* incorrect variable/value pairing (var_i, val_i) or (PC, x) of the program internal state[1]. Hence, an erroneous internal state may contain one or several errors, depending on the number of incorrect pairings. An incorrect (var_i, val_i) pairing corresponds to a *data flow error*, and an incorrect (PC, x) pairing characterizes a *control flow error*. In order to distinguish between the fault activation step and the error propagation step, error(s) produced upon fault activation will be labelled as **immediate error(s)**.

In the experiments we have conducted, each faulty program was executed on a sequence of several input patterns ($t_1, \ldots, t_p$). The term **test case** will be used to denote one input pattern t_i and a sequence of test cases ($t_1, \ldots, t_p$) will be called a **test sequence**. The program execution on one test case corresponds to one *execution cycle*. A program **execution trace** is associated with the completion of a test sequence, thus involving p execution cycles. Given a program and a test sequence, the successive internal states of the program, and then the errors in cases of faulty program, are identified from the analysis of the execution traces[2].

In order to identify and analyze the fault activation and error propagation mechanisms, we will focus on a subset of each program execution trace, called **error trace** [Mur 94],

[1] In our experiments, the correctness is determined by referring to the internal state of the original program, that is the program without any seeded fault (see Section 3.1).

[2] The execution traces were obtained by the command ctrace (available on SunOS 4.1) that allows us to list the text of each executed instruction and the values of all referenced or modified variables.

which represents the sequence of errors identified by comparison between the correct and the incorrect execution traces. An error trace can be quite complex since it may involve several fault activations occurring at different execution cycles, and thus several immediate errors which in turn may propagate. Then, to facilitate the comparison between error traces associated with different faulty programs, the behavior of a particular error (whether immediate or not) will be characterized by its **error flow** which is a subset of the error trace: starting with the target error, the error flow draws the errors created by the propagation of this error up to completion of the test sequence. An error flow which starts with an immediate error will be called an **immediate error flow**.

2.3 Program behaviors

When focusing on a faulty program behavior within a single execution cycle — i.e. independently of the preceding cycles and assuming (implicitly) that the internal state is correct before the execution begins — four issues are possible (see e.g. [Voa 92a]):

1. The fault is not activated during the execution cycle; the program internal state remains correct (including the output results);
2. The fault is activated at least once but no error is created; the program internal state remains correct (including the output results);
3. The fault is activated, an immediate error (or more) is created but output results are correct; this is the problem of *coincidental correctness*. At the end of the cycle, the program internal state is either correct if the errors produced during execution were all corrected before the end of the execution cycle, or erroneous if the errors produced during execution were not used to assign an output variable, or were masked before or when affecting the outputs;
4. The fault is activated, an immediate error (or more) is created and propagates to at least one output variable; the program internal state is erroneous and a failure occurs, thus revealing the presence of the fault.

These program behaviors represent only part of those observed during our experimentation. Indeed, intricate evolutions of the internal state were identified by observing faulty program behaviors on sequences of several execution cycles since, in that case, the internal state after each execution cycle depends on both the test case executed on and the internal state reached at the end of the preceding execution cycle.

Figure 1 shows the possible transitions between the following three states that characterize the program behavior at the execution cycle level:

(i) *Correct:* the program internal state is correct (thus, including the output variables);

(ii) *Incorrect without failure:* the program internal state is incorrect but the errors do not affect the output variables;

(iii) *Failure:* the program internal state is incorrect and at least one error affects an output variable.

The outcoming transitions of state *Correct* — that is, tr_1, tr_2 and tr_3 — correspond to the case where the internal state is correct at the start of the execution cycle: the associated behaviors are those identified above, when focusing on a single cycle. The other transitions relate to execution cycles starting with an incorrect internal state, thus involving other types of behaviors, e.g.:

- A failure may occur without fault activation during the current execution cycle (tr_6, tr_9);
- An error may propagate during successive cycles without being detected (tr_5) or before detection (tr_6);
- An error may disappear before being detected (tr_4);
- Successive failures may be observed (tr_9); we may observe either the same failures during successive cycles due to the same errors present in the internal state, or other failures due to a new fault activation or to the propagation of another error;
- The internal state may be totally (tr_4, tr_7) or partially (e.g. tr_8) corrected, when the current execution cancels error(s) present when it begins; tr_8 is a noticeable case of partial correction which means that in spite of the fact that the observable errors (those affecting the outputs) are cancelled, the internal state remains incorrect;
- A fault can be activated while the internal state is already incorrect, thus creating additional errors; the combination of these errors with the previous ones may facilitate either further propagation (tr_6, tr_9) or error cancellation (tr_4, tr_7, tr_8).

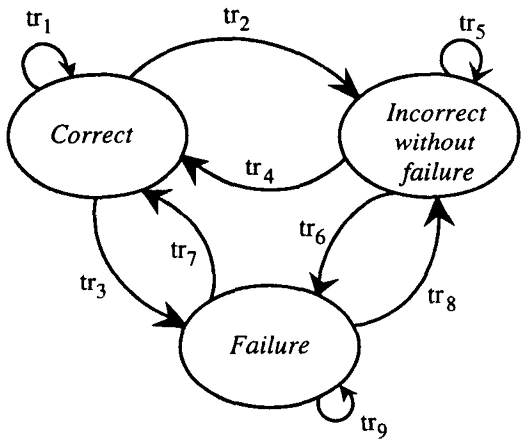

Fig.1: State graph of program behavior

These observations contribute to explain some complex mechanisms of error propagation. To improve the understanding of these mechanisms, it is necessary to analyze internal states produced at different points in an execution cycle (i.e. after execution of each statement). This analysis has been performed in the series of experiments concerning the real faults; our observations will be presented in Section 4.2 and illustrated with examples provided by the case study.

3. EXPERIMENTAL FRAMEWORK

3.1 Original program

The experiments have been conducted on a *critical program from the civil nuclear field* previously involved in our work on software statistical testing [The 93]. It belongs to the part of a nuclear reactor safety shutdown system that periodically scans the position of the reactor's control rods. At each operating cycle, 19 rod positions are processed in the following way: the information (rod positions in Gray code as well as monitoring data) is read through five 32-bit interface cards; then, these data are checked and filtered; after filtering, the measurements of the rod positions are converted into a number of mechanical steps.

The program under study is a version developed by a student from the industrial specification. This version, called STU, is written in the C language and the size of its source code approximates one thousand lines without comments. The program consists of a collection of 15 functions. All the data processed by the program are integers, representing either Booleans or numbers. As the software is critical, the use of pointers was forbidden. Each execution cycle on an input pattern achieves the processing of 19 rod positions. During previous experiments [The 93], 12 residual faults were uncovered in the STU version, thus providing us with a sample of real faults (Section 3.2). In the experiments reported in this paper, the **original program** (assumed to be correct) corresponds to the new version of STU produced by fixing the 12 identified faults.

A noticeable feature of this program is that it is a **program with memory**, due to the check and filtering function: at each execution cycle, the checks performed on the rod positions acquired may depend on the outcome of the checks performed on the rod positions acquired and processed at the preceding execution; the corresponding information being stored into internal variables. As a result, errors infecting these internal variables may propagate from an execution to the following one, before occurrence of a first failure.

3.2 Target faults

The experiments were conducted on *36 faulty programs*: each faulty program differs from the original program in containing either one of the 12 real faults or one mutation.

The **12 real faults** (denoted A, ..., L) are of different types: faults A, G and J are coding faults; faults B to F and fault I result from the lack of understanding of the requirements by the student; faults H, K and L are initialization faults. They correspond to either faulty statements (A, D, I, K, L) or missing statements (B, C, E, H, J) or additional statements (F) or misplaced statements (G), in the corresponding faulty program. **Table 1a** gives a brief description of these faults and the number of instructions involved in the "fix" of each of them.

The **24 mutations** (denoted M1, ..., M24) analyzed in this study were selected from a set of 2419 mutations involved in a mutation analysis previously conducted on our target program with the aim of assessing the efficiency of several statistical test sets [The 95]. These mutations are 1st-order mutations, that is, single-point, syntactically correct changes introduced in the original program. They encompass three types of change: constant value alteration, symbol (i.e. identifier of a variable or array) replacement, and operator replacement by either an operator of the same type or of another type (e.g. an assignment operator can be replaced by a relational operator, or an arithmetic operator by a logical operator).

As the error behaviors due to mutation activation were a priori unknown, we tried to choose a various, though small, sample of mutations. Yet, in order to make feasible the comparison of *immediate* error flows produced by mutations with those produced by real faults, some (but not all) mutations had to be performed on instructions involved in the "fix" of the real faults. Then, the general features of the 24 mutations finally selected are the following ones:

- M1, ..., M9 are constant value alterations; M10, ..., M13 are operator replacements; M14, ..., M24 are symbol replacements;
- 5 mutations (M4, M13, M20, M21, M24) are located on 4 different conditional statements; the 19 others are located on 16 different assignments infecting directly 14 different variables (variables in the left expression of the assignment);
- 16 mutations are located on instructions involved in the "fix" of the real faults: they are listed in **Table 1b**; the 8 others (M8, M9, M13, M20, ..., M24) affect statements selected arbitrarily.

Note that no mutation can affect the statements related to fault F since these statements must be deleted to remove F and the mutations are performed on the fixed program.

(a) Real faults			(b) Mutations	
Description		# statements in the fix	Location with respect to real fault	Type of change
A	Wrong operator in three assignments	3	1st statement → M10 2nd statement → M1 3rd statement → M2	O C C
B	Two missing assignments	2	1st statement → M14 2nd statement → M3	O C
C	Two missing assignments	2	1st statement → M15	S
D	Wrong operator in a conditional statement	1	M4	C
E	One missing assignment	1	M11, M16	O, S
F	Added conditional statement and assignment	3		
G	Incorrect placement of four statements	6	1st statement → M12	O
H	Missing initialization for four variables	4	1st statement → M5 1st statement → M17	C S
I	Wrong expression in an assignment	1	M18	S
J	Missing assignment	1	M19	S
K	Incorrect initial value of a variable	1	M6	C
L	Incorrect initial value of a variable	1	M7	C

Table 1: Characteristics of the 12 real faults and of 16 of the 24 mutations (C: constant value alteration; O: operator replacement; S: symbol replacement)

3.3 Overview of the experiments

In previous work on statistical testing [The 93], 16 sets of test patterns were defined and randomly generated according to different probability distributions over the program input domain. They encompass 4 types of statistical test patterns:

(i) One set of 5300 test patterns generated according to a uniform distribution over the input domain;

(ii) Five sets of 500 test patterns generated according to a structural input distribution, i.e. a distribution derived from the structure of the STU program in order to ensure a balanced coverage of its control flow graph;

(iii) Five sets of 441 test patterns generated according to functional input distributions derived from a hierarchical specification of the program functions based on finite state machines (FSMs) and decision tables (DTs); these distributions ensure a balanced coverage of the FSMs states and DTs rules;

(iv) Five sets of 441 test patterns generated according to functional input distributions derived from another hierarchical specification produced in the STATEMATE environment; these distributions ensure a balanced coverage of the basic states of the statecharts involved in the STATEMATE specification.

Since it would not be feasible to analyze the execution traces associated with such large test sets, we have selected subsets of them in the following way:

1. First, the 16 complete test sets were supplied to each of the 12 programs containing one real fault in order to identify the particular test cases that reveal these faults;
2. Then, for each of these 12 faulty programs, we have selected among the 10 sets of functional test patterns, one test sequence such that: its first test case triggers a program reset, it contains between 2 and 10 test cases, and it reveals the corresponding fault at least once; these small test sequences are denoted T1, ..., T12;
3. The 12 test sequences Ti were supplied to each of the 24 programs containing one mutation, in order to check whether or not at least one of them leads to failure(s); two mutations (namely, M9 and M23) were not revealed;
4. Then, an additional sequence of 7 test cases has been selected which allows the observation of failures due to M9 and of failures due to M23; this test sequence is denoted T13.

To analyze the errors created upon fault activations and their propagation mechanisms, the experiments were conducted on each of the 36 faulty program as follows:

(i) The faulty program and the original program are executed on a test sequence Tj which reveals the fault;

(ii) The differences between the execution traces of the faulty program and the original program are identified;

(iii) Each difference observed corresponds to an error and is recorded in a data base; the set of errors recorded during the execution of Tj represents the error trace.

Two data bases have thus been created: the "real fault data base" which contains the error traces related to the 12 real faults and the "mutation data base" which contains those related to the 24 mutations. They are described below.

3.4 Collected Data Bases

The information collected in an ***error record*** contains:

1. An error number;
2. The error type: data flow error or control flow error;
3. The error identification: incorrect variable or branch predicate;
4. Both correct and incorrect values;
5. The test sequence executed on: T1, ..., or T13;
6. The error location in the execution trace: line number in the source code, test case number and loop index if necessary;
7. Its origin: fault or error(s) that created it;
8. Its consequence: error(s) or failure(s) it creates.

The two last information items, origin and consequence, are used to identify the error traces and the error flows.

During the experiments, we observed similar errors, that is errors with the same error type, identification, correct and incorrect values, and location in the source code. These errors were recorded under the same number. This means that a same error number may appear several times in an error trace, or in an error flow, or in different error traces. Two errors labelled with different numbers are said to be **distinct errors**.

The **real fault data base** contains 1458 error records representing 255 distinct errors. A *reduced* real fault data base containing only the distinct errors has then been generated. The **mutation data base** contains 2272 error records representing 349 distinct errors. A reduced mutation data base has also been generated (349 records). Detailed analyses of the four data bases have been performed, whose main results are described in Section 4.

4. Analyses of Data Bases

First, two types of analyses were performed on the complete real fault data base in order to:

1. Collect information on real fault behaviors from the study of the error traces: immediate errors created upon activation, number of immediate error flows generated, total number of errors produced (Section 4.1);
2. Collect information about error behaviors by examination of the immediate error flows: propagation through execution cycles, interactions between errors, etc. (Section 4.2).

Then, comparative analyses between the two complete data bases on the one hand, and the two reduced data bases on the other hand, were conducted in order to determine the number of common errors (Section 4.3). Finally, the complete mutation data base was analyzed to study similarities and discrepancies between mutation and real fault behaviors (Section 4.4).

4.1 Real fault behaviors

The first twelve test sequences (T1, ..., T12) were used during the experiments on the real faults. **Table 2** gives an overview of some results of error trace analysis. For each real fault, it tabulates:

- The test sequence(s) supplied to the faulty program;
- The number of immediate errors created upon fault activation(s) during the successive execution cycles, together with the number of immediate errors created per activation and their types (incorrect variable, or branch predicate denoted BP);
- The number of immediate error flows identified in the error trace; except for fault A for which the immediate errors affect directly output results, we distinguish between immediate error flows leading to one or several failures (noted FF), and immediate error flows for which no failure is observed (noted FU) leaving the errors not detected after completion of the test sequence;
- The total number of errors recorded in the error trace.

These results call for some general comments. First, due to the presence of loops, a fault may be activated several times during one execution cycle; this explains the fact that the number of immediate errors may be higher than the number of execution cycles (see e.g., faults D and E). Second, immediate errors do not always propagate, either because the incorrect variables are no more used up to completion of the test sequence, or because they have been corrected before being used: hence, the number of immediate error flows may be lower than the number of immediate errors (this is especially true for faults J, K and L which are commented upon below). A total of 88 immediate error flows were

Fault	Test sequence (# test cases)	# immediate errors → immediate errors per fault activation	# immediate error flows (FF or FU)	# errors
A	T1 (6) T2 (9)	3 → 3 incorrect var. 23 → 3 incorrect var.	3 failures 23 failures	3 23
B	T1 (6)	16 → 2 incorrect var.	2 FF & 1 FU	77
C	T3 (6)	4 → 2 incorrect var.	1 FF & 3 FU	107
D	T3 (4)	10 → 1 incorrect BP	4 FF & 6 FU	244
E	T4 (6) T5 (7)	8 → 1 incorrect var. 10 → 1 incorrect var.	1 FF & 7 FU 1 FF & 9 FU	63 87
F	T6 (3)	14 → 1 incorrect var.	1 FF & 12 FU	93
G	T7 (4)	16 → 4 incorrect var.	3 FF & 1 FU	97
H	T7 (4)	4 → 4 incorrect var.	1 FF	91
I	T8 (5)	6 → 1 incorrect var.	1 FF & 5 FU	127
J	T9 (4)	72 → 1 incorrect var.	1 FF & 1 FU	78
K	T10 (5) T11 (5)	18 → 1 incorrect var. 18 → 1 incorrect var.	1 FF & 9 FU 1 FF & 4 FU	83 67
L	T12 (10)	18 → 1 incorrect var.	1 FF & 11 FU	218

Table 2: Analysis of the error traces due to real faults (BP: branch predicate; FF: flow leading to failure(s); FU: flow without failure)

identified in the complete real fault data base, from which only 19 lead to failure occurrence(s) while the 69 others do not (due to error masking or cancellation). As a result, the percentage of errors detected in comparison with the total amount of errors generated is rather weak.

Fault **F** is the only fault resulting in additional statements and it may be interesting to have a closer look at its behavior. Fault F is activated 14 times by the test sequence T6, affecting each time the same variable. Thirteen of the immediate errors propagate and only one error flow affects output results. This error flow leads to failure because the corresponding errors can propagate during two successive execution cycles without being cancelled before affecting the output variables. Most of the immediate error flows (9 FU) concern only one execution cycle and contain less than four errors each; they lead to errors that are not used during the subsequent execution cycles. The three remaining FU propagate during two execution cycles and contain between 10 and 25 errors each; but all of them lead to either cancelled errors or masked errors (e.g. incorrect variables that have the same effect on the output results as the correct ones).

Faults **G and H** were experimented on the same test sequence with the objective of comparing their error traces. Indeed, in previous experiments [The 93], both faults exhibited the same external behavior: they were revealed by the same test cases and they produced the same failures. G and H are not of the same type (see Table 1a) and are not located in the same part of the code; but they affect the same four variables. The comparison of the execution traces shows that:

1. From the four variables infected upon fault activation, only one is concerned by the propagation; thus generating a single immediate error flow per activation;
2. Fault activation modes are different: H is activated once at the first execution cycle, while G is activated at each of the four execution cycles; hence, after completion of the test sequence T7, one immediate error flow (FF) is observed for H, and four immediate error flows (3 FF & 1 FU) for G;
3. The immediate error flow generated by H corresponds to the concatenation of the four immediate error flows generated by G: the corresponding immediate error created by H is equivalent to the one created by G; it is created by H at the first execution cycle and it propagates during the three successive cycles (no. 2, 3, 4), while the corresponding immediate error created by G is generated at each cycle (after each fault activation).

As a result, both faulty programs fail on the same test cases and supplied identical incorrect results.

As regards faults **J, K and L**, a detailed analysis of the error traces and error flows provides us with information to understand why they were seldom revealed. Fault J generates many immediate errors (72 during three successive execution cycles): only two of them propagate, each one creating three errors from which only one leads to failure. K and L create many immediate errors which propagate during five execution cycles before being detected; such long error flows facilitate the occurrence of masking or cancellation mechanisms (this case will be illustrated on Figure 2, in the next section).

4.2 Error behaviors

Investigations on error flows focus on how an error may propagate through executions creating other errors, but also on how an error is masked or cancelled and thus, is not detected. A graphical representation of an error flow is given in **Figure 2**: it represents one of the immediate error flows due to fault K. Each node is associated with an error. For each error, the incorrect variable/value pairing and the number of the execution cycle during which the error was created are indicated. For clarity, this graph is simplified, the location of each error being not mentioned. For example, the notation $(\text{CO\_AQU\_DIF} = 0/1)_1$ means that the variable CO_AQU_DIF was assigned the incorrect value '0' versus the correct value '1' during the first execution cycle. The graph edges denote the cause/effect chain between errors: the error at the arrow's tail is used to create, cancel or mask the error at the arrow's head. Errors in bold represent wrong output results (failure occurrence). Errors in italic affect output variables but are masked or cancelled before the end of the execution cycle (no failure occurrence). Framed errors are those remaining in the program internal state after completion of the test sequence: they could further propagate if an additional test case were executed (without initialization). The other errors (normal style) are cancelled before completion of the test sequence, the incorrect variables being overwritten.

When a variable is not referenced at a given point in either the incorrect execution trace or the correct one, its value is not observed. This case, denoted U (for Unobserved) in Figure 2, occurs when the execution of the original program and the execution of the faulty program follow distinct paths, thus affecting different variables. This notation allows us to identify in the error flow the errors due to the incorrect execution of a branch statement (consequences of a control flow error).

At a given point of the execution, the use of an incorrect variable or incorrect branch predicate may result in the **creation** of new error(s) or/and the **masking** or/and the **cancellation** of other errors. These mechanisms can affect: 1) the same variable, 2) another variable or conditional predicate, 3) several variables (resulting in a division of the error flow in subsequent sub-flows). In the same way, the combined use of several errors (incorrect variables and incorrect branch predicate) may result in the creation, masking or cancellation of errors. The complexity of such behaviors and interactions is increased by the fact that the program internal state and the original (correct) internal state are modified concurrently and a modification in the original internal state may result in the cancellation or the masking of an error in the incorrect internal state.

Error creation and **propagation** have been investigated in other studies [Ric 88, Voa 92a, Voa 92b, Gor 93, Tho 93]. The RELAY model [Ric 88, Tho 93] provides detailed descriptions of propagation mechanisms (computational transfer and information flow transfer) from the creation of immediate errors until the occurrence of a failure, and the authors explored information flow transfer on simple modules. During our experimental study, we observed the same types of transfer, amplified by: 1) the size of the software program, 2) the error ability to propagate between successive execution cycles, 3) the use of AND operators, 4) the parameter passing by value in C.

We have also explored the **masking** and **cancellation mechanisms**. They are often associated in other studies [Voa 92b, Gor 93] but, here, we differentiate between them since they do not have the same effect on the error graph and on the program internal state. Cancellation just stops the extension of an error flow and the cancelled error is either deleted or replaced by a new error in the internal state, whereas masking may stop temporarily the extension of an error flow but the error remains in the internal state and can propagate further.

Cancellation occurs when:

- An incorrect variable is overwritten in the faulty program; either the new value is correct (the error is deleted from the internal state), or it is still incorrect and a new error is created;
- The correct value is modified in the original program and the new value becomes equal to the previously incorrect one;
- In cases of control flow error, when the execution of a branch incorrectly selected is completed, the PC value turns into the correct one;
- etc.

Masking occurs when:

- An incorrect variable is not used in current computations, stopping temporarily the extension of the error flow; this error may be manifested later;
- An incorrect variable (or more) is used in a computation to affect another variable and the resulting value is equal in both programs; this phenomenon, called "blindness" by Zeil [Zei 89] was also observed and explained by Bishop [Bis 89]; it is amplified in our study by the use of OR operators and shift operators in many computations;
- etc.

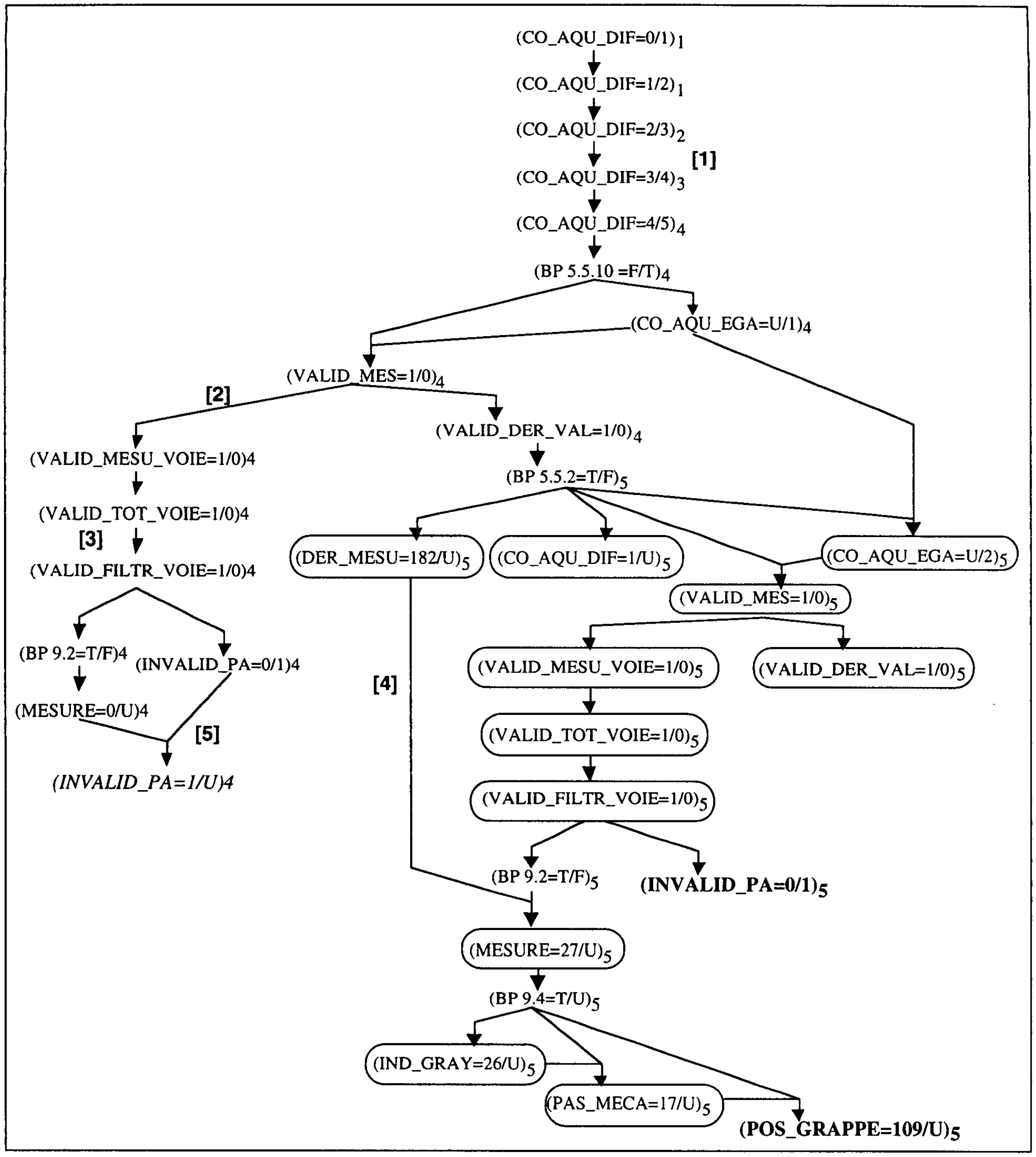

Fig. 2: Example of an error flow graph due to fault K

Some illustrations of the error propagation, masking and cancellation mechanisms are indicated on Figure 2, where different types of interactions and impacts on the error flow graph are observed:

[1] Propagation on the same variable resulting in the simple extension of the error flow; this error evolves through 4 successive execution cycles, both programs following the same path during the first 3 cycles;

[2] Propagation affecting two variables resulting in the division of the error flow into subsequent error flows; the error created on the variable VALID_DER_VAL is masked during the rest of the 4th execution cycle, and then used in the 5th cycle enabling error propagation to an output result (failure occurrence);

[3] Propagation by parameter passing which results in a simple extension of the error flow;

[4] Combined use of two errors resulting in the convergence of the corresponding sub-flows; an incorrect variable is used in an incorrectly selected branch creating a new incorrect variable;

[5] Combined use of two errors resulting in the convergence of the corresponding sub-flows; an incorrect variable is overwritten in the faulty program, by execution of a branch incorrectly selected; then the new value becomes correct, thus cancelling one of the previous error.

4.3 Comparison of errors

The comparison between the complete real fault and mutation data bases shows that:

(i) 1930 errors recorded in the mutation data base are present in the real fault data base (i.e. 85%);

(ii) Among the 342 other records representing "new" errors in the mutation data base, 158 are immediate errors (i.e. 7%) and 184 are propagated errors (8%).

Figure 3 illustrates the result of the comparison between the reduced real fault and mutation data bases, showing the amount of distinct errors shared by the two data bases. These raw results are in favor of a good representativeness of the errors generated by mutations. The 30% miscellaneous distinct errors in the mutation data base represent new errors due to erroneous computations leading to many incorrect values on the same integer variables: a majority of these errors occurred only once and are due to mutations whose behaviors are described in Section 4.4.2. Most of the 13% miscellaneous distinct errors in the real fault data base are either created upon activations of fault K under the test sequence T10, or errors due to fault D. Indeed, the behaviors of these two faults are those which were least reproduced by the selected mutations as explained in the next section.

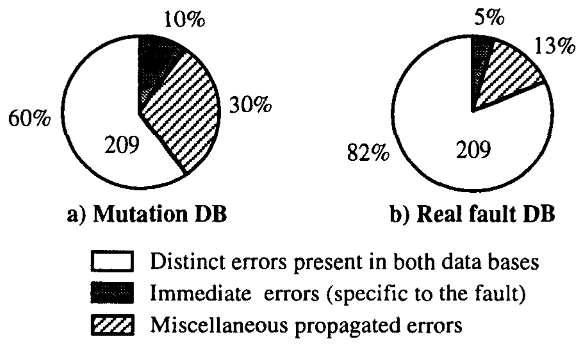

Fig. 3: Analysis of the distinct error sets:
a) in the mutation data base (349 distinct errors)
b) in the real fault data base (255 distinct errors)

4.4 Error flow comparison

For complexity reason and due to the large amount of errors produced, the comparative analysis of error flows was focused on immediate error flows. In this section, the errors in the real fault data base are called *actual error* (for short) and *actual error flows* denote the associated flows.

A total of 41 immediate error flows leading to at least one failure (FF) and 128 immediate error flows resulting in masked or cancelled errors (FU) have been identified in the complete mutation data base. It is worth noting that this proportion of FF and FU obtained from the 24 mutations is similar to the one obtained from the 12 real faults (19 FF and 69 FU, see Section 4.1). This lends support to the assumption that mutations can exhibit error behaviors as complex as those produced by real faults.

A mutation is said to **totally model** a real fault when it generates the same immediate error flows than the real fault, except for the immediate error(s) directly related to the fault syntax and semantic. A mutation **partially models** a real fault when they produce common error flows that result in the same failures or in the same masked or cancelled errors affecting output variables. The comparison of the error flows shows that:

- All the real faults are totally or partially modelled by mutations;
- Only 4 mutations (M4, M9, M23, M24) do not generate any immediate error flow similar to an actual error flow.

These two general results are commented upon in Sections 4.4.1 and 4.4.2, respectively.

4.4.1 Real fault behavior modelling

For purposes of comparison, we consider that an error flow due to a mutation *totally* reproduces an actual error flow when either the two flows are identical (except for the immediate error), or the mutation error flow contains the actual error flow. It *partially* reproduces an actual error flow when it constitutes or contains a sub-flow of the actual error flow; in that case the common sub-flows have to result, after completion of the test sequence, in the same errors affecting the output variables (same failures in case of a FF, and same cancelled or masked errors in case of a FU).

A total of 16 from the 19 actual immediate error flows leading to failure (FF) and 49 from the 69 actual immediate error flows leading to masking (FU) are reproduced — either totally or partially — in the mutation data base. Surprisingly, the 3 actual FF that have not been reproduced upon mutation activation are related to faults D and K that correspond each to a single faulty statement in the source code and thus are similar to mutations of "operator replacement" and "symbol replacement" type, respectively.

Fault	Mutation	Test sequence (# test cases)	Actual immediate error flows (FF or FU) reproduced	Actual immediate error flows not reproduced
A	M10	T2 (8)	same failures	3 from 6 distinct failures
	M1	T5 (7)	same failures	
	M2	T1 (6)	same failures	
B	M14	T1 (6)	2 FF totally	1 FU
	M3	T1 (6)	1 FF partially	
C	M15	T3 (9)	1 FF and 1 FU totally	2 FU
	M20*	T3 (4)	1 FU partially	
D	M4	T3 (2)	no similarity	2 FF and 5 FU
	M20*	T3 (4)	2 FF and 1 FU partially	
E	M11	T4 (6)	1 FF and 7 FU totally	none
	M16	T5 (8)	1 FF and 9 FU totally	
F	M21*	T6 (2)	1 FF and 1 FU partially	11 FU
	M22*	T6 (2)	1 FF and 1 FU partially	
G	M5	T7 (4)	1 FF partially	none
	M12	T6 (2)	1 FF totally	
	M13*	T7 (3)	2 FF partially	
	M17	T7 (4)	3 FF and 1 FU partially	
H	M5	T7 (4)	1 FF partially	none
	M12	T6 (2)	1 FF partially	
	M13*	T7 (3)	1 FF partially	
	M17	T7 (4)	1 FF partially	
I	M18	T8 (4)	1 FF totally and 5 FU partially	none
J	M19	T9 (4)	1 FF and 1 FU totally	none
K	M6	T11 (5)	1 FF and 4 FU totally	1 FF and 1 FU due to T10
	M8*	T3 (8)	3 FU totally	
L	M7	T12 (7)	1 FF and 1 FU totally	none

Table 3: Similarities between real fault and mutation behaviors (* mutations selected arbitrarily)

Table 3 summarizes the similarities observed between the mutation behaviors and the real fault behaviors identified in Table 2. It tabulates:

- For each real fault, the mutations that totally or partially model it;
- For each mutation, the test sequence executed on and the number of actual immediate error flows it reproduces partially or totally; mutations marked with an * are those chosen arbitrarily (not listed in Table 1b), and thus, not located on the instructions involved in the "fix" of the real faults;
- For each fault, the number of immediate error flows not reproduced by the studied mutations.

This table shows that several real faults were totally modelled by mutations under the same test sequence (e.g. faults E, J, K, L). If we have a closer look at the real faults that are located on a single statement, they correspond either to missing statements (faults E and J), or to faulty statements (faults D, I, K and L). For the later type of faults, it is worth noting that the mutations studied don't reproduce totally the same behaviors (e.g. faults D and I), though it would have been possible to select mutations among the 2419 mutations available that generate the same error flows. If we examine how the behaviors generated by real faults that affect several statements (faults B, C, F, G, H) are reproduced by the selected mutations, we can notice that all the actual error flows leading to failures were partially or totally reproduced.

In Section 4.1, it was noticed that **faults G and H** exhibit similar behaviors: this explains that the same four mutations (M5, M12, M13, M17) modelled both faults G and H.

Similarities between **fault K** and mutation M8 are noticeable: in spite of the fact that M8 and K are not located on the same instruction and that the faulty programs were not executed on the same test sequence, M8 reproduced 3 FU related to fault K.

Fault D was incompletely modelled. In particular, there was no similarity with mutation M4 located on the same instruction. The reason is that the activation of D may only produce an incorrect branch predicate whose value is "false" (versus "true" in the original program). M4 affects the same branch predicate, but its activation may only produce the incorrect value "true" (versus "false" in the original program). Hence, D and M4 affect the same location in the code but they always create distinct error flows (no common errors). We will return to the behavior of M4 in the next section.

Fault F corresponds to additional statements in the source code; it was therefore impossible to select a mutation that affects the same location. However, mutations M21 and M22 partially model F. M21 creates an immediate error that is an actual error produced by F; thus one FF generated by M21 corresponds to a sub-flow extracted from the actual FF. M22 creates new immediate errors but it reproduces the same FF as M21 (the corresponding FF are identical from the second error present in the immediate error flows). The actual FU partially reproduced by M21 and M22 are one of the longest described in Section 4.1. Moreover, although fault F behaviors are the least modelled by mutations, all the actual errors generated by F are present in the mutation data base.

To conclude this comparative analysis, it is worth noting that the studied mutations have easily reproduced: 1) the same subtle behaviors as those observed for faults J, K and L; 2) the same behaviors as those identified for omission faults (missing statements).

4.4.2 Mutations M4, M9, M23 and M24

Mutations M9, M23, M24 were selected arbitrarily among the 2419 mutations available. Mutation M4 affects the same location as fault D but it does not produce immediate errors upon the same fault activations. For these four mutations, all the immediate error flows were different from the actual ones. Thus, we only analyzed the representativeness of the errors they produced, independently of the error flows they belong to.

Mutation	**Test sequence** (# test cases)	**# immediate error flows**	**# errors created**	**# new errors**
M4	T3 (2)	14 FF & 14 FU	300	98 (56 distinct)
M9	T13 (7)	1 FF	23	13 (8 distinct)
M23	T13 (7)	1 FF	29	11 (6 distinct)
M24	T9 (4)	2 FF & 27 FU	147	32 (17 distinct)
			Total = 499	Total = 154

Table 4: Analysis of M4, M9, M23 and M24

The results are summarized in **Table 4** which gives, for each mutation:

- The test sequence executed;
- The number of immediate error flows;
- The total amount of the errors present in the error trace;
- The number of "new" errors (i.e. not recorded in the real fault data base) with the number of distinct errors they represent in the reduced mutation data base.

These mutations produced 154 from the 342 new errors contained in the complete mutation data base (see Section 4.3), that is to say 45%. Even if they do not model any real fault, 69% of the errors produced are actual errors: this result is encouraging with respect to the representativeness of the errors produced by mutations.

5. SUMMARY AND CONCLUSIONS

This paper reports on a real case study involving real faults and mutations. The experiments related here were conducted on a student version of a critical program from the civil nuclear field. They focus on the analyses of the errors produced by 12 real faults on the one hand and by 24 mutations on the other hand. The results are in favor of a good representativeness of the errors generated by mutations: 85% of the errors produced by the mutations were also generated by the real faults. This positive outcome is confirmed by the qualitative analysis on error behaviors. It demonstrates how mutations can produce, upon activation, error behaviors as complex as the real faults did. We expect these results to be relevant as the studied mutations were not purposely selected to exhibit the same error behaviors as those identified for real faults. Indeed, we did not know a priori if any similarity could be observed.

One may consider that the faulty programs corresponding to the 12 real faults are syntactically close to the correct program, in spite of the fact that the types of the studied faults are diverse: they are either coding faults, or initialization faults, or the results of a misunderstanding of the requirements by the programmer. Unfortunately, identification of errors by comparison between the execution trace of a faulty program and the execution trace of the correct program would not have been possible if both programs were too syntactically different.

The exploration of error behaviors on a complex program is tedious and may not generally be conceivable: this explains the relative small number of mutations studied. Nevertheless, this exploration is necessary to understand the complex mechanisms of error propagation, error masking and error cancellation through successive execution cycles. Such experiments are currently being conducted on another real case study to see whether or not the results are confirmed.

We also intend to conduct similar experiments on faulty programs containing two or more faults. However, we suspect that the fault and error behaviors that will be observed could be equivalent to those related to the interactions between immediate error flows in our study. That is to say, interactions between faults could be essentially due to the interactions between the immediate error flows they may create. Indeed, we have noticed that a single fault may create several immediate errors and that this fault could be activated several times in a single execution cycle or during successive execution cycles; interactions between immediate error flows have thus been observed. Similar interactions should be identified between error flows produced when several faults infect a program.

Finally, the large data bases of error records now available provide us with valuable information to identify and model fault and error behaviors. We noticed, in our study, that the complexity of an error behavior is not related to the type of fault but rather to the interactions of the errors with the program dependencies [Pod 90]. Further theoretical investigation will concern the use of program dependencies to explain and model error behaviors.

ACKNOWLEDGEMENTS

We wish to thank our colleague Yves CROUZET for having provided us with the mutation analysis environment and the whole set of mutations used in the experiments and for his helpful suggestions about the analyses performed. We would also like to thank the referees for their useful comments, which lead, we hope, to an improved paper.

REFERENCES

[Bis 89] Bishop P.G. and Pullen F.D., "Error Masking: a Source of Failure Dependency in Multi-Version Programs", Proc. *1st IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-1)*, Santa-Barbara, USA, pp. 53-73, 1989.

[DeM 78] DeMillo R.A., Lipton R.J. and Sayward F.G., "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer*, vol. 11, no. 4, pp. 34-41, 1978.

[DeM 94] DeMillo R.A. and Mathur A.P., "On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software", Software Engineering Research Center report, Purdue University, W. Lafayette, USA, 1994.

[Gor 93] Goradia T., "Dynamic Impact Analysis : A Cost-effective Technique to Enforce Error-propagation", Proc. *1st International Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, USA, pp. 171-181, 1993.

[Lap 92] Laprie J.C. (Ed.), *Dependability: Basic Concepts and Terminology*, Springer-Verlag, vol. 5, Dependable Computing and Fault Tolerance Systems, Springer-Verlag, 1992.

[Mur 94] Murrill B.W. and Morell L., "An Experimental Approach to Analyzing Software Semantics Using Error Flow Information", Proc. *2nd International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, USA, p. 200, 1994.

[Off 89] Offutt A.J., "The Coupling Effect: Fact or Fiction?", Proc. *3rd Symposium on Testing, Analysis and Verification (TAV 3)*, Key West, USA, pp. 131-140, 1989.

[Pod 90] Podgurski A. and Clarke L.A., "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance", *IEEE Transactions on Software Engineering* , vol. 16, no. 9, pp. 965-979, 1990.

[Ric 88] Richardson D.J. and Thompson M.C., "The RELAY Model of Error Detection and its Application", Proc. *2nd Workshop on Software Testing, Verification and Analysis*, Banff, Canada, pp. 223-230, 1988.

[Ric 93] Richardson D.J. and Thompson M.C., "An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection", *IEEE Transactions on Software Engineering* , vol. 19, no. 6, pp. 533-553, 1993.

[The 93] Thévenod-Fosse P. and Waeselynck H., "STATEMATE Applied to Statistical Software Testing", Proc. *1st International Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, USA, pp. 99-109, 1993.

[The 95] Thévenod-Fosse P. and Crouzet Y., "On the Adequacy of Functional Test Criteria Based on Software Behaviour Models", Proc. *5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, Urbana-Champaign, USA, pp. 176-187, 1995.

[Tho 93] Thompson M.C., Richardson D.J. and Clarke L.A., "An Information Flow Model of Fault Detection", Proc. *1st International Symposium on Software Testing and Analysis (ISSTA)*, Cambridge, USA, pp. 182-192, 1993.

[Voa 92a] Voas J.M. and Miller K.W., "The Revealing Power of a Test Case", *Journal of Software Testing, Verification and Reliability*, vol. 2, pp. 25-42, 1992.

[Voa 92b] Voas J.M., Morell L.M. and Miller K., "Predicting Where Faults Can Hide from Testing", *IEEE Software*, vol. 8, no. 2, pp. 41-48, 1992.

[Zei 89] Zeil S.J., "Perturbation Techniques for Detecting Domain Errors", *IEEE Transactions on Software Engineering* , vol. 15, no. 6, pp. 737-746, 1989.