A Metric Approach to Measuring Fault Coverage of Software Testing in Respect to the FSM Model[*]

Mingyu Yao, Alexandre Petrenko[**] and Gregor von Bochmann

Département d'informatique et de recherche opérationnelle
Université de Montréal, CP. 6128, Succ. “A”, Montréal (Québec), Canada H3C 3J7
Emails: {yao, petrenko, bochmann}@iro.umontreal.ca
Tel: 1-514-343-6111 ext. 3541

Abstract

Software testing is always a trade-off between increased confidence in the correctness of the software system under examination and constraints on the amount of time and effort that can be spent in testing the software system. As a result, the fault coverage or adequacy of the test suite used to test the software system becomes a very important issue as it directly reflects the confidence in the correctness of the system under test. Mutation analysis is a well studied approach to the evaluation of fault coverage of a given test suite. However, it often becomes impractical as it may require to generate a huge number of mutants each of which should then be executed against the given test suite. In this paper, we propose a metric approach to the evaluation of fault coverage of software testing. This approach is developed based on the finite state machine (FSM) model which has been used in the testing of certain software systems such as communication protocols and object-oriented programs as well as the testing of sequential digital circuits. The attractiveness of this approach is its low computational complexity. It calculates the fault coverage of a given test suite by directly analyzing the test suite itself. Therefore, it avoids the generation and execution of mutants. This approach has been implemented and a number of experiments has been carried out. Some of the experimental results are summarized in this paper to show the accuracy of the metric approach compared with the mutation analysis technique.

1 Introduction

[*] This research was supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the Government of Canada, and the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols at University of Montreal.

[**] On leave from the Institute of Electronics and Computer Science, Riga, Latvia.

Software testing is a critical phase of the software development life cycle. Software testing consists of a number of execution scenarios of a software implementation against a selected set of test cases called a test suite. A faulty implementation is said to be detected if its execution against a test case distinguishes its behavior (or output) from what is expected.

Software testing was originally proposed to detect faults in an implementation [Myer 79]. From this fault-detection viewpoint, however, an execution scenario in which no fault is detected provides no useful information at all and therefore calls for more execution scenarios. As such, the testing process of a software implementation will never be stopped if no fault can be detected.

Testing was later proposed to ascertain the correctness of a software implementation in respect to its requirement specification [More 90]. The essential idea of this correctness-proving viewpoint is that an execution scenario in which no fault is detected ensures that the implementation is free of certain faults. Therefore, exhaustive testing, which requires all the possible execution scenarios of the implementation to be carried out, is able to prove the correctness of the implementation. Apparently, exhaustive testing is often impractical since it may involve a huge or even infinite number of execution scenarios to be done. In the more practical and so-called fault-based software testing approaches [More 90], a *fault model* is selected which specifies a set of faults of which an implementation should be tested to be free. However, it can be still too expensive to prove by testing that an implementation is free of all the specified faults as a very large number of test cases may be required. In practice, therefore, a test suite which consists of only a relatively small number of test cases will be actually employed to test the implementation. As such, software testing is often a trade-off between increased confidence in the correctness of the software implementation under examination and constraints on the amount of time and effort that can be spent in testing the software implementation. Whenever such a trade-off is made, one desires to have a measurement of testing effectiveness in terms of the percentage of the specified faults that can be detected.

Mutation analysis is a well studied approach to measuring the *fault coverage* (also called *adequacy*) of a given test suite [SaSp 90]. It involves the *mutation* of a program by the introduction of a syntactic change in the program. Each of the *mutant programs* is then executed against the test cases in the given test suite. The test suite is said to provide full or complete fault coverage if it distinguishes all of the incorrect mutant programs from the original program. Moreover, in case that it does not provide full fault coverage, the ratio of the number of distinguished mutant programs to the total number of incorrect mutant programs yields a precise measure of the fault coverage of the given test suite. Although the mutation analysis technique was originally proposed within the framework of white-box testing, its basic principle can be applied to certain black-box testing as well. The drawback of mutation analysis is that the cost may prevent us from making an exhaustive analysis. The reason is two-fold. Firstly, the number of mutant programs can be, in some cases, very large or even infinite. Secondly, each of the mutant programs may have to be executed against a number of different test cases before being distinguished from the original program. In practice, therefore, it is often beneficial to test only a small, statistically random sampling of mutants against the given test cases [SaSp 90, DDB 91, SiLe 89 and MCS 93]. Certainly, the fault coverage obtained in this way is only an approximation of the real precise fault coverage of the given test suite. Apparently, the accuracy of such an approximated fault coverage relies on the number of randomly sampled mutants.

In this paper, we are going to propose a metric approach to the approximation of the fault coverage of a given test suite. The basic idea of this approach is, by analyzing the given test suite, to make an estimation on the number of incorrect mutants that can be detected by the test suite without the need of generating explicitly the mutants to be executed against the given test cases. The approach is to be developed based on the finite state machine (FSM) model. In addition to its traditional applications in the development of sequential digital circuit systems [Koha 78], the FSM model has been extensively used in recent years in the area of conformance testing of communication protocols [PBD 93, SiLe 89, DDB 91, Ural 91, YPB 93a and YPB 93b]. Currently, it has also attracted much attention in relation with the testing of object-oriented software systems [HoSt 93, TuRo 92].

The rest of the paper is organized as follows. In Section 2, the FSM model is first formally introduced and a framework of software testing based on this model is then presented. The metric approach to the evaluation of fault coverage of a test suite is developed in Section 3. The fault coverage evaluation results for a number of test suites will be summarized in Section 4 to show the accuracy of the metric approach. Finally, in Section 5, the conclusion will be given.

2 A Framework of Software Testing Based on the FSM Model

We will first introduce the finite state machine model and then present a software testing framework based on this model.

2.1 The FSM Model

A *finite state machine* (FSM), often simply called a *machine* throughout this paper, is essentially an *initialized Mealy machine* which can be formally defined as follows.

Definition 2.1 (finite state machine)
A finite state machine is a 7-tuple $<S, X, Y, S_1, \delta, \lambda, D>$, where

S is a set of n states $\{S_1, S_2, ..., S_n\}$ with S_1 as the initial state;
X is a finite set of input symbols;
Y is a finite set of output symbols;
D is a specification domain which is a subset of S x X;
δ is a transfer function δ: D --> S;
λ is an output function λ: D --> Y. ■

An FSM is said to be *completely specified* (*defined*), iff D = S x X. Otherwise it is said to be *partially* or *incompletely specified* (*defined*). Since δ and λ are required to be functions, this FSM model is *deterministic*. That is, for each (S_i, x) [D, there should be exactly one state S_j [S and exactly one output symbol y [Y such that $\delta(S_i, x) = S_j$ and $\lambda(S_i, x) = y$. In this case, we say there is a transition leading from state S_i to S_j with input x and output y. Such a transition is usually written as S_i -x/y-> S_j, or as a triplet < S_i; x/y; S_j >. S_i is said to be the *head* or *starting* state of the transition, while S_j is said to be the *tail* or *ending* state of the transition. An FSM can be given in a graph form, with the states and transitions of the FSM represented by the vertices and arcs of the graph, respectively. As an example, Figure 1 gives a FSM which is partially specified since, at state S_3, no transition is specified for input symbol 1.

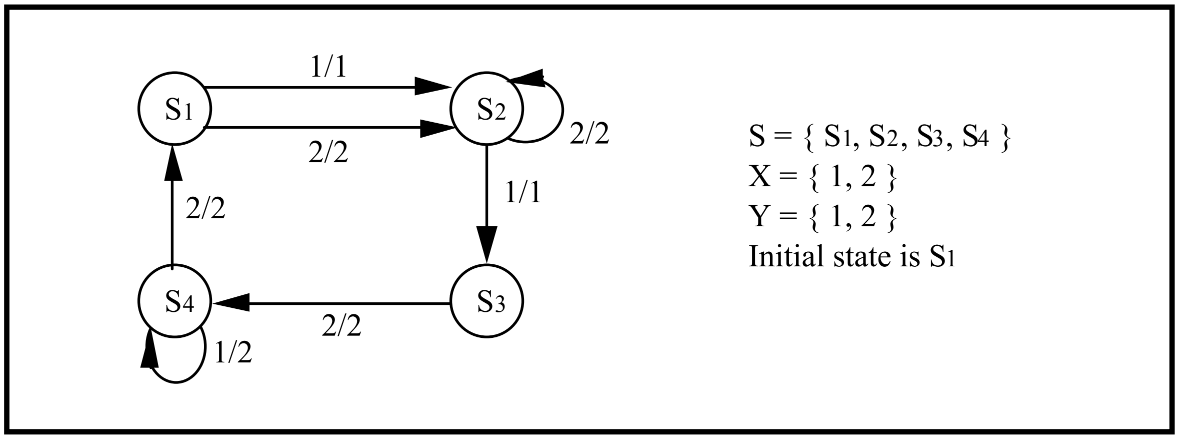


Figure 1: An example FSM

The following notations will be used throughout the paper. For a given symbol set Z, Z^* is used to represent the set of words constructed on Z and "ε" to represent the *empty* word, i.e., the word consisting of no symbols. Also, the dot "$\cdot$" is used to represent the concatenation operation of two words. However, this dot symbol is often omitted when no ambiguity arises. Furthermore, |Z| is used to represent the cardinality of Z.

Definition 2.2 (defined input sequence)

Let $p = x_1x_2\cdots x_k$ [X^*. p is called a defined input sequence for state S_i [S, if there exist k states S_{i1}, S_{i2}, ..., S_{ik} [S and an output sequence $q = y_1y_2 \ldots y_k$ [Y^* such that there is a sequence of transitions

S_i -x_1/y_1-> S_{i1} -x_2/y_2-> S_{i2} --> ... --> S_{ik-1} -x_k/y_k->S_{ik} **(2-1)**

in the finite state machine. ■

We use $\psi(S_i)$ to denote the set of all the defined input sequences for state S_i. A sequence of transitions such as (2-1) can be abbreviated as S_i -p/q-> S_{ik}, which, when we do not care about the output sequence q, can be further simplified as S_i -p-> S_{ik}, with the meaning that the FSM, when in state S_i and given an input sequence p, will enter state S_{ik}. The definitions of the transfer function δ and output function λ can be naturally extended to apply not only to single inputs, but also to sequences of inputs.

Definition 2.3 (extensions of transfer and output functions to input sequences)

Let $p = x_1x_2 \ldots x_k$ [$\psi(S_i)$ and ε be the empty word. Then,

$\delta(S_i, \varepsilon) = S_i$, $\quad \delta(S_i, p) = \delta(\delta(S_i, p'), x_k)$

$\lambda(S_i, \varepsilon) = \varepsilon$, $\quad \lambda(S_i, p) = \lambda(S_i, p').\lambda(\delta(S_i, p'), x_k)$

where $p' = x_1x_2 \ldots x_{k-1}$. ■

Definition 2.4 (compatible states and distinct states)

We say that S_i and S_j are *compatible* states if for : p [$\psi(S_i)$ ($\psi(S_j)$, $\lambda_s(S_i, p) = \lambda_s(S_j, p)$. Otherwise, they are called *distinct* states. ■

According to the above definition, if $\psi(S_i)$ ($\psi(S_j) = \phi$, then S_i is compatible with S_j. If the FSM happens to be completely specified, then the definition of compatible states given above reduces to the definition of *equivalent states* as found in the literature (see for example, [Gill 62, Koha 78]).

Definition 2.5 (reduced machine)

A FSM is said to be *reduced* if and only if no two states are compatible. ■

It is easy to verify that the FSM given in Figure 1 is reduced.

Definition 2.6 (reachable state and strongly connected FSM)

A state S_i is said to be *reachable* (from the initial state S_1) if there exists an input sequence p [$\psi(S_i)$ such that S_1 -p->S_i. A machine is said to be *initially connected* if all the states are reachable. ■

Apparently, all the states of the FSM in Figure 1 can be reached from the initial state and therefore this example FSM is initially connected.

Definition 2.7 (mutant machine)
Let M_1 and M_2 be two given FSMs. M_2 is said to be a mutant machine of M_1 if M_2 is obtained by applying to M_1 each of the following four types of operations, in any order, for a certain number of times (including zero times):
Type 1: change the tail state of a transition;
Type 2: change the output of a transition;
Type 3: add a transition; and
Type 4: add an extra state . ■

The following corollary follows directly from the above definition.

Corollary 2.8
A machine is a mutant machine of itself. ■

2.2 A Testing Framework Based on FSM Model

The FSM model was widely used in traditional hardware testing. In recent years, this model has also received much attention in the testing of certain software systems such as communication protocols [PBD 93] and object-oriented programs [HoSt 93, TuRo 92]. Testing based on the FSM model can be formalized as the problem of *testing a FSM implementation*[Ural 91]: given a FSM representation (specification) of a system (denoted henceforth as M_S) and an implementation of the system (denoted henceforth as M_I), we are required to determine if the implementation machine M_I *conforms to* (i.e., is *correct* with respect to) the specification machine M_S by testing M_I as a black-box. This implies that we should generate from M_S a set of input sequences, called a test suite, and the corresponding set of expected output sequences such that M_I conforms to M_S if and only if, when the input sequences in the test suite are applied to M_I, the observed output sequences from M_I are the same as the corresponding expected output sequences. As already pointed out in the literature [Moor 56, Gill 62, YPB 93a, YPB 93b], this problem is not solvable unless it is dealt within a restricted framework. Therefore, some assumptions should be made about the specification machine M_S and the implementation machine M_I.

Firstly, the restrictions on the specification machine are summarized in the first assumption.

Assumption 1: (reduced and initially connected specification machine)
The given specification machine M_S is reduced and initially connected. ■

Secondly, testing based on the FSM model is essentially a mutation testing. Therefore, for the given specification machine M_S, an implementation machine M_I is actually a mutant machine (of

M_S) obtained from M_S by applying each of the four types of operations listed in Definition 2.7 for a number of times (including zero times). These four types of operations represent the basic types of changes that can be made during the implementation of M_S. However, it should be noted that, in practice, the implementation machine M_I is normally completely defined even though the given specification machine M_S is often only partially specified. Therefore, the following assumption is made throughout this paper.

Assumption 2: (completeness of an implementation machine)
For the given specification machine M_S, an implementation machine M_I is a completely defined mutant machine of M_S. ■

Thirdly, if the number of changes of Type 4 applied to the given specification machine M_S is not limited, the number of mutants of M_S will be infinite and the problem of testing will become intractable. Therefore, in practice, the number of changes of Type 4 is always limited to an upper bound. Throughout this paper, we simply do not allow any change of Type 4 as stated in the next assumption.

Assumption 3: (limited number of states in an implementation machine)
For the given specification machine M_S, any operation of Type 4 is not allowed and therefore the number of states in an implementation machine M_I will not exceed that of M_S. ■

We also note here that additional types of changes, such as changes of inputs, changes of head states and missing states, may be introduced [MiPa 92]. However, these types of changes are not necessary for our discussion as the FSM model is deterministic (Definition 2.1) and implementation machines are assumed to be completely defined (Assumption 2). The following example explains that the same consequences of a missing state can be achieved by changing the tail states of certain transitions (Type 1). Figure 2 (a) shows that, when implementing the FSM given in Figure 1, state S_4 is not implemented (i.e., missing in the implementation) and the transition $< S_3; 2/2; S_4 >$ is changed to $< S_3; 2/2; S_1 >$. However, we can still think that state S_4 is present in the implementation as shown in Figure 2 (b). The reason is that S_4 is no longer reachable and therefore, during the black-box testing, whether S_4 is present or missing in the implementation makes no difference.

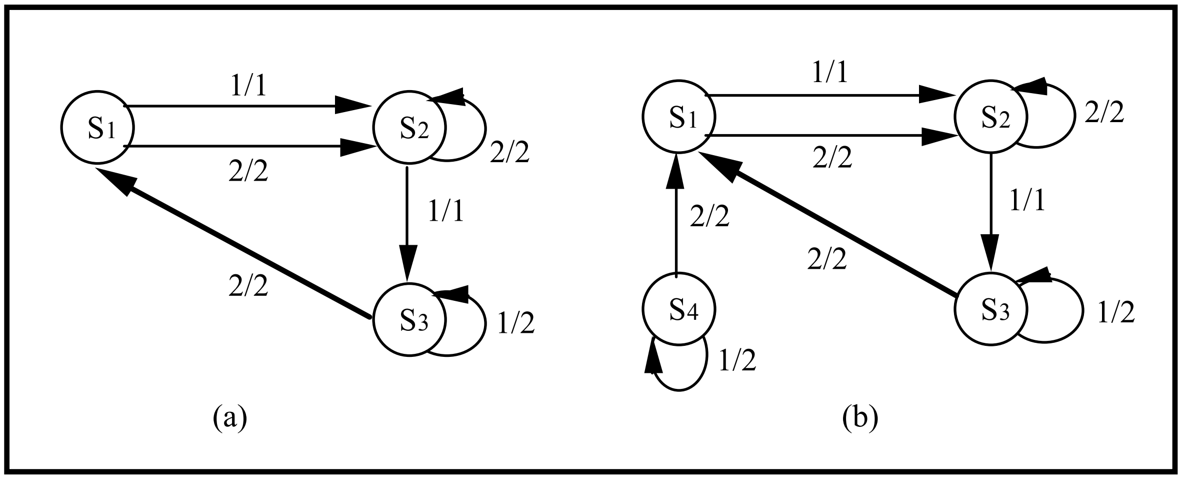


Figure 2: missing state

Therefore, as a matter of fact, all the n states S_1, S_2, ..., S_n of the specification machine M_S are assumed to be present in an implementation machine M_I. However, some of these states may become unreachable in M_I due to the changes of Type 1 introduced during the implementation. Confusion may arise because the same state names S_1, S_2, ..., S_n are used for both M_S and M_I. It is therefore often helpful, although not necessary, to make things clear by renaming the states S_1, S_2, ..., S_n in M_I to I_1, I_2, ..., I_n, respectively. Then without loosing generality, let
$M_S = < \{S_1, S_2, ..., S_n\}, X, Y, S_1, \delta_S, \lambda_S, D_S >$, and
$M_I = < \{I_1, I_2, ..., I_n\}, X, Y, I_1, \delta_I, \lambda_I, D_I >$.

Since M_I is supposed to be completely defined, we know that $D_I = \{I_1, I_2, ..., I_n\}$ x X and therefore $\psi(I_i) = X^*$ and $\psi(S_j)$ { $\psi(I_i)$, for any I_i and S_j.

Now, we need to introduce some important concepts. The first concept required is the so-called *conformance relation* which essentially defines when M_I is a correct implementation of M_S. This concept is defined through the following two definitions.

Definition 2.9 (equivalence of states in respect to a set of input sequences)
Let I_i be a state of M_I and S_j a state of M_S. V is a set of input sequences such that V { $\psi(S_j)$. Then
$I_i -_V S_j$ if $\lambda_I (I_i, p) = \lambda_S(S_j, p)$, for : p [V. ■

Definition 2.10 (conformance relation)
M_I conforms to M_S, written M_I **CONF** M_S, if and only if $I_1 -_{\psi(S1)} S_1$, where I_1 and S_1 are the initial states of M_I and M_S, respectively. ■

The above defined conformance relation corresponds to the notion of weak conformance [SaDa 88, SiLe 89 and MiPa 92]. The relationship between the above defined conformance relation and the types of operations listed in Definition 2.7 is established by the following lemma of which the proof is similar to that of Lemma A.1 given in the appendix of [YPB 93a].

Lemma 2.11

For the specification machine M_S and implementation. Then M_I conforms to M_S if and only if there exists a mapping **f**: $\{S_1, S_2, ..., S_n\}$ -> $\{I_1, I_2, ..., I_n\}$, such that

(1) **f** is one-to-one; and

(2) If S_i - x/y -> S_j is in M_S, then I_k - x/y -> $I_{\neg}$ is in M_I, where I_k = **f**(S_i) and $I_{\neg}$ = **f**(S_j). ■

Since the implementation machine M_I is treated as a black-box, test cases should be generated from the specification machine M_S. The following two definitions formally defines the concepts of test case and test suite.

Definition 2.12 (test case)

A test case is a sequence of inputs which should be of finite length and in $\psi(S_1)$. ■

As is clear from the above definition, a test case always starts from the initial state S_1 of the specification machine M_S. Accordingly, each test case should be applied to the implementation machine M_I when it is in its initial state I_1. Therefore, an important assumption in the testing based on the FSM model is about the availability of the so-called *reliable reset* function and is summarized as our fourth (and final) assumption.

Assumption 4: (availability of reliable reset)

The *reliable reset* is an operation that, when activated, will bring the implementation from any other state back into its initial state. It is assumed to be available in an implementation under test. ■

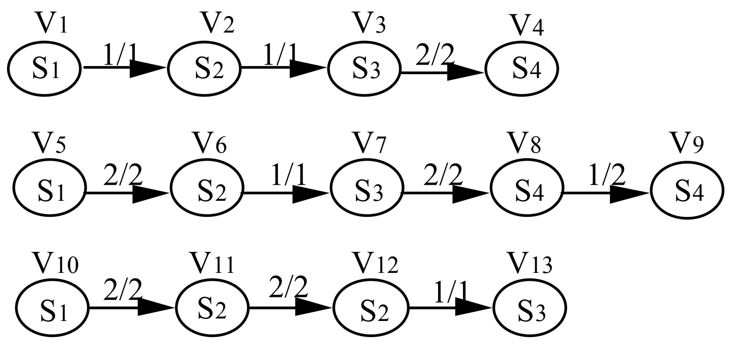
A special input symbol “r” representing the invocation of the reset operation is added to the beginning of each test case.

Definition 2.13 (test suite)

A test suite is a finite set of test cases. ■

TS = { r.1.1.2, r.2.1.2.1, r.2.2.1 }

(a)



(b)

Figure 3: A test suite generated from the example FSM

Figure 3 (a) lists the test cases of a test suite generated from the machine shown in Figure 1. Each of the test cases is prefixed by the reset symbol “r”. Applying the three test cases to the initial state S_1 results in the three sequences of transitions shown in Figure 3 (b) that will be executed.

Definition 2.14 (to pass a test suite)
Let TS be a test suite and p [TS be a test case. We say that a given implementation M_I passes the test case p, written M_I **pass** p, iff λ_I (I_1, p) = λ_S(S_1, p). Further, we say that M_I passes the test suite TS, written M_I **pass** TS, iff M_I **pass** p, for : p [TS. ■

An implementation machine which cannot pass a given test suite is said to *fail* the test suite or to be *detected* by the test suite.

Let **Impl**(M_S) represent the set of all the implementation machines of M_S, i.e., all the completely defined mutant machines with same number of states as M_S. Then we have the following lemma whose validity is obvious (see [Gill 62, SiLe 89]).

Lemma 2.15 (number of implementation machines)
The number of implementation machines, that is the cardinality of **Impl**(M_S), is given by $\left|\mathbf{Impl}\left(M_S\right)\right| = (n|Y|)^{n|X|}$, where n is the number of states of M_S. ■

3 A Metric Approach to the Estimation of Fault Coverage

As with other software testing, the evaluation of fault coverage for a given test suite TS is an important issue in testing based on the FSM model and has been studied in relation with the traditional hardware testing [Koha 78] and, in recent years, in relation with the conformance testing of communication protocols [DaSa 88, DDB 91, SiLe 89 and YPB 94]. Methods that have been proposed are essentially variations of the mutation analysis technique. For instance, instead of using the exhaustive mutation analysis approach, some researchers have introduced a number of classes of mutant machines [DaSa 88, DDB 91, SiLe 89 and MCS 93]. The mutant machines in a class will contain a certain number of faults resulting from the changes of tail states and/or outputs of some transitions. For each class, a limited number of mutant machines are randomly generated which are then executed against the given test suite. Apparently, the fault coverage evaluated in such a way is an estimation of the real fault coverage of the test suite and its accuracy relies on the total number of mutant machines randomly generated and executed. In our recent work [YPB 94], a different procedure has been developed which, without the need of explicitly generating and then executing a certain (and often large) number of mutant machines, can decide if the given test suite provides full fault coverage (i.e., if it can detect all the incorrect implementation machines). When the test suite does not provide full fault coverage, the proposed approach can derive from the test suite, by analyzing it against the specification machine, an incorrect implementation machine which can pass the test suite and therefore allow an additional test case to be generated to distinguish this particular implementation machine from the specification machine. As such, full fault coverage can be achieved by repeatedly applying this procedure. However, this approach does not provide a numeric measure for a test suite which does not have full fault coverage. Consequently, it is impossible to use this approach to compare the fault coverage of two test suites, if none of them provides full fault coverage.

In this section, we are going to present a metric approach to numerically characterize the fault coverage of any test suite. This metric approach avoids the necessity of explicit generation and execution of mutant machines representing possible implementations of the given specification machine M_S. It is developed to have low computational complexity and is therefore quite easy to calculate. First of all, let us introduce the following notations:

$\mathbf{N_1}(M_S)$ - the total number of machines in **Impl**(M_S);
$\mathbf{N_2}(M_S)$ - the number of machines in **Impl**(M_S) which conform to M_S;
$\mathbf{N_3}(M_S)$ - the number of machines in **Impl**(M_S) which do not conform to M_S;
$\mathbf{N_4}(M_S, TS)$ - the number of machines in **Impl**(M_S) which do not conform to M_S but can be detected by the given test suite TS;
$\mathbf{N_5}(M_S, TS)$ - the number of machines in **Impl**(M_S) which do not conform to M_S but can pass the given test suite TS.

Apparently, $\mathbf{N_4}(M_S, TS) = \mathbf{N_3}(M_S) - \mathbf{N_5}(M_S, TS)$. The precise definition of fault coverage of a given test suite in respect to the given specification machine M_S is given as follows.

Definition 3.1 (precise fault coverage)
The precise fault coverage of a test suite TS in respect to M_S, denoted as $\mathbf{FC}_p(M_S, TS)$, is

$$\mathbf{FC}_p(M_S, TS) = \frac{\mathbf{N}_4(M_S, TS)}{\mathbf{N}_3(M_S)} = \frac{\mathbf{N}_3(M_S) - \mathbf{N}_5(M_S, TS)}{\mathbf{N}_3(M_S)} \quad ■$$

The exact value of $\mathbf{N_3}(M_S)$ can be calculated from the given specification machine M_S. As already given in Lemma 2.14,

$$\mathbf{N_1}(M_S) = \left|\mathbf{Impl}(M_S)\right| = (n|Y|)^{n|X|} \qquad \textbf{(3-1)}$$

We can further prove that the following lemma is valid.

Lemma 3.2 (the number of implementation machines which conform to M_S**)**

$$\mathbf{N}_2(M_S) = (n-1)!\,(n|Y|)^{n|X| - |D_S|} \qquad \textbf{(3-2)}$$

where n is the number of states of M_S. ■

Therefore,

$$\mathbf{N_3}(M_S) = \mathbf{N_1}(M_S) - \mathbf{N_2}(M_S) = (n|Y|)^{n|X|} - (n-1)!\,(n|Y|)^{n|X| - |D_S|} \qquad \textbf{(3-3)}$$

Although the exact value of $\mathbf{N_3}(M_S)$ has easily been found, the exact value of $\mathbf{N_4}(M_S, TS)$ or $\mathbf{N_5}(M_S, TS)$ is in general too difficult to find without using the exhaustive mutation analysis technique. However, as we have already mentioned, the exhaustive analysis technique is often not feasible in practice due to the high cost. Therefore, in our approach, we will use an estimated value, denoted as $\mathbf{N_5}(M_S, TS)$, of $\mathbf{N_5}(M_S, TS)$. Substituting $\mathbf{N_5}(M_S, TS)$ for $\mathbf{N_5}(M_S, TS)$ in the calculation of the fault coverage as defined in Definition 3.1 results in the following estimated fault coverage.

Definition 3.3 (estimated fault coverage)
The estimated fault coverage of a test suite TS in respect to M_S, denoted as $\mathbf{FC}_e(M_S, TS)$, is

$$\mathbf{FC}_e(M_S, TS) = \frac{\mathbf{N}_3(M_S) - \mathbf{N}_5(M_S, TS)}{\mathbf{N}_3(M_S)} \quad ■$$

Definition 3.4 (prefix set of a test suite)
The prefix set **AP**(TS) of a test suite TS is the set which consists of all the prefixes of all the test cases in TS, i.e.,
AP(TS) = { p | p is a prefix of some test case in TS }. ■

Definition 3.5 (transition covered by TS**)**

A transition $< S_i; x/y; S_j >$ in M_S is said to be covered by TS, if there are α, αx [**AP**(TS) such that $\delta_S(S_1, \alpha) = S_i$ and $\delta_S(S_1, \alpha x) = S_j$. ■

Definition 3.6 (tail state S_j of a transition distinguished from S_k by TS)
The tail state S_j of a transition $< S_i; x/y; S_j >$ in M_S is said to be distinguished from another state S_k by TS if there are α, αx, $\alpha x\gamma$, β, $\beta\gamma$ [**AP**(TS) such that
$\delta_S(S_1, \alpha) = S_i$, $\delta_S(S_1, \alpha x) = S_j$, $\delta_S(S_1, \beta) = S_k$ and $\lambda_S(\delta_S(S_1, \alpha x), \gamma)$ □ $\lambda_S(\delta_S(S_1, \beta), \gamma)$. ■

We will proceed in two steps to find the estimated value $\mathbf{N_5}(M_S, TS)$. In the first step, we will make an estimation, denoted as $\mathbf{N_6}(M_S, TS)$, on the number of implementation machines of M_S which can pass the given test suite TS. It should be noted that, in general, some of these estimated $\mathbf{N_6}(M_S, TS)$ implementation machines conform to the specification machine M_S, while the others do not. We should therefore in the second step make another estimation, denoted as $\mathbf{N_7}(M_S, TS)$, of how many of those $\mathbf{N_6}(M_S, TS)$ implementation machines conform to the specification machine M_S. Then $\mathbf{N_5}(M_S, TS) = \mathbf{N_6}(M_S, TS) - \mathbf{N_7}(M_S, TS)$ gives us the estimated number of implementation machines which do not conform to M_S and can pass the given test suite.

To find the value of $\mathbf{N_6}(M_S, TS)$, we need to classify the transitions of M_S into two classes: the first class includes the transitions covered by TS, while the second class consists of the transitions not covered by TS. Taking the specification machine given in Figure 1 and the test suite TS shown in Figure 3 (a) as an example, we can easily check that, among the seven specified transitions in Figure 1, the six transitions listed in Table 1 (a) are covered by the test suite TS, while the other transition given in Table 1 (b) is not covered.

For a transition $< S_i; x/y; S_j >$ in M_S, we use **Tail_Dis**($< S_i; x/y; S_j >$, TS) to denote the set of states from which the tail state S_j of transition $< S_i; x/y; S_j >$ is distinguished. Then,
Tail_NDis($< S_i; x/y; S_j >$, TS) = $\{ S_1, S_2, ..., S_n \}$ - **Tail_Dis**($< S_i; x/y; S_j >$, TS)

is the set of states from which the tail state S_j of transition $< S_i; x/y; S_j >$ is not distinguished.

$< S_1; 1/1; S_2 >$ $< S_1; 2/2; S_2 >$ $< S_2; 1/1; S_3 >$

$< S_2; 2/2; S_2 >$ $< S_3; 2/2; S_4 >$ $< S_4; 1/2; S_4 >$

(a) transitions covered by TS

$< S_1; 1/1; S_2 >$

(b) transition not covered by TS

Tail_Dis ($< S_1; 1/1; S_2 >$, TS) = $\{ S_4 \}$
Tail_Dis ($< S_1; 2/2; S_2 >$, TS) = $\{ S_3, S_4 \}$
Tail_Dis ($< S_2; 1/1; S_3 >$, TS) = $\{ S_1, S_2 \}$
Tail_Dis ($< S_2; 2/2; S_2 >$, TS) = $\{ S_4 \}$
Tail_Dis ($< S_3; 2/2; S_4 >$, TS) = $\{ S_1, S_2 \}$
Tail_Dis ($< S_4; 1/2; S_4 >$, TS) = ϕ

(c)

Tail_NDis ($< S_1; 1/1; S_2 >$, TS) = $\{ S_1, S_2, S_3 \}$
Tail_NDis ($< S_1; 2/2; S_2 >$, TS) = $\{ S_1, S_2 \}$
Tail_NDis ($< S_2; 1/1; S_3 >$, TS) = $\{ S_3, S_4 \}$
Tail_NDis ($< S_2; 2/2; S_2 >$, TS) = $\{ S_1, S_2, S_3 \}$
Tail_NDis ($< S_3; 2/2; S_4 >$, TS) = $\{ S_3, S_4 \}$
Tail_NDis ($< S_4; 1/2; S_4 >$, TS) = $\{ S_1, S_2, S_3, S_4 \}$

(d)

Table 1: Intermediate calculation results for the example FSM and TS

For a transition $< S_i; x/y; S_j >$ covered by TS, we can easily calculate **Tail_Dis**($< S_i; x/y; S_j >$, TS) and therefore **Tail_NDis**($< S_i; x/y; S_j >$, TS). As an example, let us consider one of the covered transition $< S_1; 2/2; S_2 >$ given in Table 1 (a). As is clear from Figure 3 (b), this transition is covered twice by the test suite. The tail state S_2 at point V_6 is distinguished from state S_4 at point V_8. The same tail state S_2 at point V_{11} is distinguished from state S_3 at point V_7. Therefore, **Tail_Dis**($< S_1; 2/2; S_2 >$, TS) = $\{ S_3, S_4 \}$ and **Tail_NDis**($< S_1; 2/2; S_2 >$, TS) = $\{ S_1, S_2 \}$. The related results for the other five covered transitions can be found in Table 1 (c) and (d). Since a state in **Tail_NDis**($<S_i; x/y; S_j>$, TS) is not distinguished from the tail state S_j of $<S_i; x/y; S_j>$, changing the tail state S_j of transition $< S_i; x/y; S_j >$ to any state in **Tail_NDis**($< S_i; x/y; S_j >$, TS) will give us an implementation machine which can pass the test suite TS. Therefore, there are |**Tail_NDis**(< S_i; x/y; S_j >, TS)| possible ways to make such a Type 1 change (as defined in Definition 2.7). However, we should note that, as transition < S_i; x/y; S_j > is covered by TS, changing the output symbol "y" to any other output symbol (Type 2 change) will result in an implementation machine which is very likely to be detected by TS. Therefore, to guarantee to generate an implementation machine which can pass TS, we have only one choice of keeping the output "y" of the transition. Consequently, the given transition < S_i; x/y; S_j > covered by TS gives us |**Tail_NDis**(< S_i; x/y; S_j >, TS)| possible ways of generating an implementation machine which can pass the test suite TS.

For a transition < S_i; x/y; S_j > not covered by the given test suite TS, its tail state S_j is not distinguished by TS from any state. Therefore, we have **Tail_NDis**(< S_i; x/y; S_j >, TS) = { S_1, S_2, ..., S_n }. Furthermore, since the transition is not covered by TS, we can change the output symbol "y" to any symbol in Y and still get a mutant machine which can pass TS. Combining the possible ways of changing the tail state (Type 1 change) and the possible ways of changing the output (Type 2 change), we can immediately conclude that, for the transition < S_i; x/y; S_j > which is not covered by TS, there are |**Tail_NDis**(< S_i; x/y; S_j >, TS)| x |Y| = n|Y| possible ways to generate an implementation machine which can pass the test suite. As a result, the set of all the transitions not covered by TS gives us $(n|Y|)^m$ choices to generate an implementation machine which can pass TS (where m is the number of transitions not covered by TS).

As we have assumed in Section 2, an implementation machine should be completely defined. Therefore, for the given specification machine M_S which is in general partially specified, we need to apply the Type 3 operation to add an extra transition for each (S_i, x) [S x X - D_S. Since the tail state of such an extra transition can be any of the n states S_1, S_2, ..., S_n and the output symbol can be any one in Y, we know that there are a total of $(n|Y|)^{n|X| - |D_S|}$ possible ways to generate an implementation machine by adding n|X| - $|D_S|$ extra transitions.

Following from the above discussions, we have

$$\mathbf{N}_6(M_S, TS) = (n|Y|)^{n|X| - |D_S| + m} \qquad \underset{\substack{< S_i;\, x/y;\, S_j > \\ \text{covered}}}{} |\textbf{Tail\_NDis}\,(< S_i; x/y; S_j >, TS)| \qquad \textbf{(3-4)}$$

where m is the number of transitions not covered by TS.

Lemma 3.7
The estimated value $\mathbf{N_6}(M_S, TS)$ given in (3-4) is a lower bound on the number of implementation machines which can pass the test suite TS.

■

Due to the limited space of this paper, the proof of this lemma is omitted here.

Since, in **Impl**(M_S), there are exactly $\mathbf{N_2}(M_S)$ implementation machines which conform to M_S, we can conclude that there are at most $\mathbf{N_7}(M_S, TS) = \mathbf{min}(\mathbf{N_6}(M_S, TS), \mathbf{N_2}(M_S))$ (i.e., the minimum value of the two) implementation machines which conform to M_S and are among the $\mathbf{N_6}(M_S, TS)$ estimated implementation machines. Therefore,

$$\mathbf{N_5}(M_S, TS) = \mathbf{N_6}(M_S, TS) - \mathbf{N_7}(M_S, TS) = \mathbf{N_6}(M_S, TS) - \mathbf{min}(\mathbf{N_6}(M_S, TS), \mathbf{N_2}(M_S)) \qquad \textbf{(3-5)}$$

gives us a lower bound on the value of $\mathbf{N_5}(M_S, TS)$. Consequently, substituting (3-5) for $\mathbf{N_5}(M_S, TS)$ in Definition 3.3 results in the following estimated fault coverage

$$\mathbf{FC}_e(M_S, TS) = 1 - \frac{\mathbf{N}_6(M_S, TS) - \mathbf{min}(\mathbf{N}_6(M_S, TS), \mathbf{N}_2(M_S))}{\mathbf{N}_3(M_S)} \qquad \textbf{(3-6)}$$

which is an upper bound of the precise fault coverage $\mathbf{FC}_p(M_S, TS)$ given in Definition 3.1.

Let us continue our example with the specification machine given in Figure 1 and the test suite shown in Table 1 (a). For this particular example, $|D_S| = 7$, $m = 1$, $n = 4$, $|X| = |Y| = 2$. Therefore, we have $\mathbf{N_1}(M_S) = 16777216$, $\mathbf{N_2}(M_S) = 48$, $\mathbf{N_3}(M_S) = 16777168$, $\mathbf{N_6}(M_S, TS) = 18432$ and finally the estimated fault coverage $\mathbf{FC}_e(M_S, TS) = 99.89042\%$.

Several properties of $\mathbf{FC}_e(M_S, TS)$ given in (3-6) are summarized in the following theorem.

Theorem 3.5
(1) $\mathbf{FC}_p(M_S, TS) \leq \mathbf{FC}_e(M_S, TS) \leq 1$;
(2) $\mathbf{FC}_e(M_S, TS) = 1$ if $\mathbf{FC}_p(M_S, TS) = 1$;
(3) $\mathbf{FC}_e(M_S, TS) = 0$ if $\mathbf{FC}_p(M_S, TS) = 0$; and
(4) $\mathbf{FC}_e(M_S, TS) \leq \mathbf{FC}_e(M_S, TS')$ if $\mathbf{AP}(TS) \{ \mathbf{AP}(TS')$. ■

It is quite straightforward to prove these properties. However, we feel that the meaning of the forth property needs some explanation. We note that **AP**(TS) is the prefix set of TS and that **AP**(TS) { **AP**(TS') implies that TS' has more or longer test cases than TS. The forth property essentially tells us that the estimated fault coverage for TS and TS' coincide with the intuition that TS' provides better fault coverage than TS.

4 Fault Coverage Evaluation Results

The metric fault coverage approach has been implemented under SUN/UNIX. We have done a number of experiments with this approach. To show how accurate the metric approach can estimate the real precise fault coverage of a test suite, we summarize here the experiment results in relation with the specification machine M_S given in Figure 1. The following eight test suites have been generated from that machine.

$TS_1 = \phi$
TS_2 = { r.1 }
TS_3 = { r.1.1.2, r.2.1.2.1 }
TS_4 = { r.1.1.2, r.2.1.2.1, r.2.2.1 }
TS_5 = { r.1.1.2, r.2.1.2.1, r.2.2.1.2 }
TS_6 = { r.1.1.2, r.2.1.2.1.1, r.2.2.1.2 }
TS_7 = { r.1.1.2, r.2.1.2.1, r.2.2.1.2.2 }
TS_8 = { r.1.1.2, r.2.1.2.1.1, r.2.2.1.2.2 }
TS_9 = { r.1.1.2.1.1.1, r.1.1.2.1.2.1, r.1.2.1.2.1, r.1.1.2.2.1.1.2.1, r.1.1.2.2.1.2.1,
r.1.1.2.2.2.1, r.1.2.2.1, r.2.1.2.1.1, r.2.2.1.2.2 }

Applying the metric approach to these test suites yields the estimated fault coverage listed in the second column of Table 2. To assess the accuracy of these estimated fault coverage values, we need to compare them with the real precise fault coverage of these test suite. Fortunately, for the small specification machine given in Figure 1, we have been able to make an exhaustive mutation analysis. We have written a program which generates and executes one by one all the $(4 \times 2)^{(4 \times 2)} = 16777216$ possible implementation machines against each of the above eight test suites. Therefore, we have been able to calculate the real precise fault coverage for these test suites and the results are listed in the third column of Table 2. As expected, the estimated fault coverage are equal to or slightly over the precise fault coverage. The differences between the estimated and precise fault coverage are listed in the forth column of Table 2. Clearly, the estimated fault coverage approaches the precise fault coverage when the latter is either very high (almost 100%) or relatively low. Figure 4 illustrates the fact that our metric approach slightly over estimates the fault coverage of a test suite in some cases. Ideally, we would like to have the estimated fault coverage $\mathbf{FC}_e(M_S, TS)$ to be equal to the corresponding precise fault coverage $\mathbf{FC}_p(M_S, TS)$ as shown in Figure 4 by the bold dashed line. In reality, $\mathbf{FC}_e(M_S, TS)$ is slightly larger than $\mathbf{FC}_p(M_S, TS)$ in certain cases as illustrated by the solid curve.

Actually, we have also applied the metric approach to a number of other more complex examples such as the simplified transport protocol [SaBo 84]. However, due to the complexity of these examples, we have been unable to make exhaustive mutation analysis to obtain their real precise fault coverage. For instance, the simplified transport protocol [SaBo 84] is represented by a machine which has 4 states, 10 inputs and 11 outputs and therefore the total number of implementation machines is 44^{40}. Such a large number of implementations prevented us from making an exhaustive mutation analysis. As such, we are unable to compare the estimated fault

TS_i	$FC_e(M_S, TS_i)$	$FC_p(M_S, TS_i)$	**Deviation**
TS_1	0.00000%	0.00000%	0.00000%
TS_2	50.00014%	50.00014%	0.00000%
TS_3	99.34110%	99.25871%	0.08239%
TS_4	99.89042%	99.66497%	0.22545%
TS_5	99.89042%	99.66898%	0.22144%
TS_6	99.94535%	99.77655%	0.16880%
TS_7	99.94535%	99.88084%	0.06451%
TS_8	99.97282%	99.92032%	0.05250%
TS_9	100.00000%	100.00000%	0.00000%

Table 2: Fault Coverage

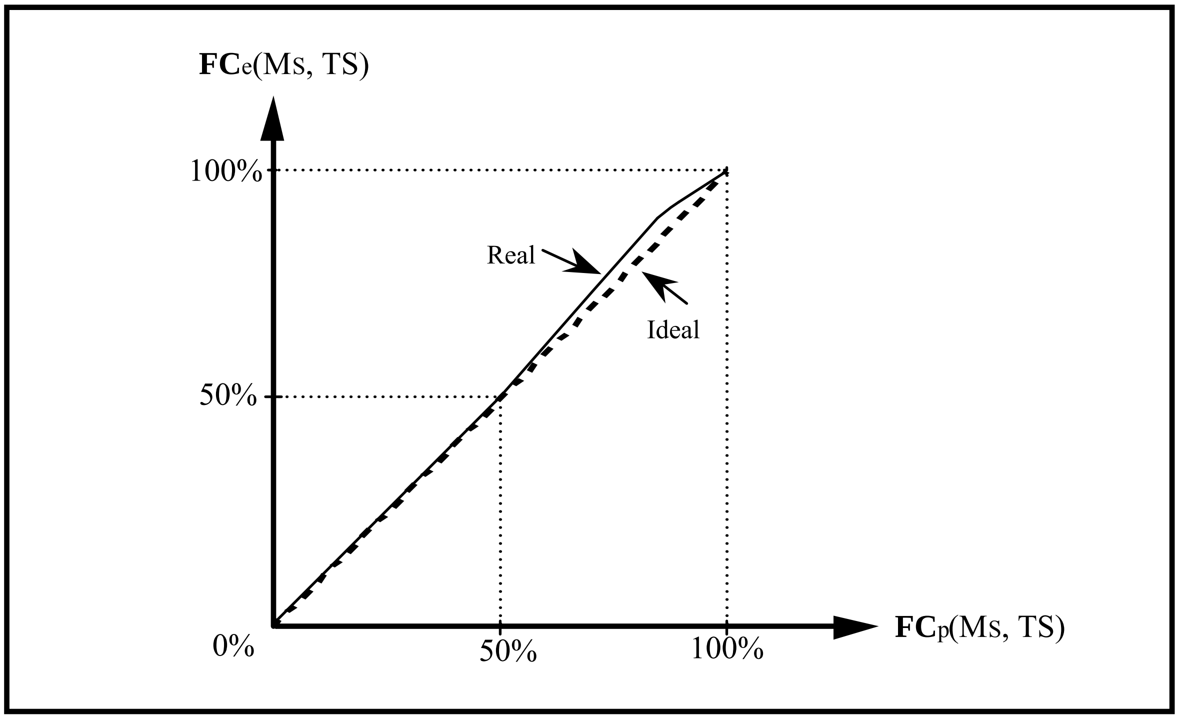


Figure 4: $\mathbf{FC}_e(M_S, TS)$ vs. $\mathbf{FC}_p(M_S, TS)$ **for a given** M_S

coverage for these complex examples with their corresponding precise fault coverage. We are currently applying the metric approach to some realistic protocol machines such as the NBS Class 4 transport protocol [SiLe 89].

5 Conclusions

The FSM model is an important tool in the study of a number of problems, such as conformance testing of communication protocols, object-oriented software testing as well as the development of sequential digital circuits. In this paper, we have presented a metric approach to the evaluation of fault coverage of a test suite in respect to a system specification given in the form of a finite state machine. This approach differs from those methods proposed in [DaSa 88, SiLe 89, DDB 91 and MCS 93] as it avoids the necessity of generating and executing a (large) number of mutant machines. Instead, it evaluates the fault coverage of a given test suite by directly analyzing the test suite against the specification machine. It provides a numeric measure for a test suite no matter whether the test suite has full fault coverage (i.e., 100%) or not. This feature makes the metric approach different from one of our previous work [YPB 94] where a test suite is analyzed only to see if it provides full fault coverage or not. As we have seen in Section 4, applications of this metric approach to a number of example test suites have shown that the estimated fault coverage of a test suite is very close to the real precise fault coverage, especially when the test suite approaches full fault coverage. Furthermore, this approach has very low computational complexity. Actually, it is not difficult to prove that its complexity is $O(L^2)$, where L is the size of a test suite in terms of the total number of inputs in the test suite. Other related work can be found in [MiPa 92, LoSh 92] where they aimed at generating test suites to achieve full fault coverage (or maximal fault coverage as they called) rather than the evaluation of fault coverage of a given test suite.

We also note that the metric approach has been developed under certain assumptions (Assumptions 1-4 as introduced in Section 2) which are the most relaxed ones compared with other work based on the FSM model [SiLe 89, DaSa 88 etc.]. In particular, we have not assumed the specification machine to be completely specified. Therefore, the metric approach can be applied to partially specification machines. We believe that this is very important for its practical applications since the real-world systems, such as protocol machines, are normally partially specified. We are currently using the metric approach to evaluate the fault coverage of a number of test suites for a subset of the NBS Class 4 transport protocol [SiLe 89, MiPa 92].

As for the future work, we will look for a method, guided by the metric approach for fault coverage evaluation, to generate additional test cases to achieve a desired fault coverage if the fault coverage of the original given test suite is too low.

References

[DaSa 88] A. Dahbura and K. Sabnani, “Experience in Estimating Fault Coverage of a Protocol Test”, in Proc. IEEE INFOCOM’88, 1988, pp. 71-79.

[DDB 91] M. Dubuc, R. Dssouli and G.V. Bochmann, “TESTL: A Tool for Incremental Test Suite Design Based on Finite State Model”, 4th International Workshop on Protocol Test Systems, Holland, November 1991.

[Gill 62] A. Gill, “Introduction to the Theory of Finite-State Machines” McGraw-Hill Book Company Inc., 1962, pp. 207.

[HoSt 93] D. Hoffman and P. Strooper, "A Case Study in Class Testing", in Proc. CASCON'93, Toronto, Canada, October 24-28, 1993, pp. 472-482.

[Koha 78] Z. Kohavi, “Switching and Finite Automata Theory”, New York, McGraw-Hill, 1978, pp. 658.

[LoSh 92] F. Lombardi and Y.N. Shen, "Evaluation and Improvement of Fault Coverage of Conformance Testing by UIO Sequences", IEEE Trans. Commun., Vol. COM-40, 8, August, 1992, pp. 1288-1293.

[MCS 93] H. Motteler, A. Chung and D. Sidhu, "Fault Coverage of UIO-based Methods for Protocol Testing", Proc. IWPTS, Pau, France, 28-30 September, 1993, pp. 21-33.

[MiPa 92] R.E. Miller and S. Paul, "Structural Analysis of a Protocol Specification and Generation of a Maximal Fault Coverage Conformance Test Sequence", submitted for publication.

[Moor 56] E.F. Moore, “Gedanken-Experiments on Sequential Machines”, Automata Studies, Princeton University Press, Princeton, New Jersey, 1956.

[More 90] L.J. Morell, “A Theory of Fault-Based Testing”, IEEE Trans. SE-16, 8, August 1990, pp. 844-857.

[Myer 79] G.L. Myers, “The Art of Software Testing”, John Wiley, 1979.

[PBD 93] A. Petrenko, G.v. Bochmann and R. Dssouli, "Conformance Relations and Test Derivation", Proc. IWPTS, Pau, France, 28-30 September, 1993, pp. 157-178.

[Petr 91] A. Petrenko, “Checking Experiments with Protocol Machines”, Proc. of the 4th Int. Workshop on Protocol Test Systems, 1991.

[SaBo 84] B. Sarikaya and G.v. Bochmann, "Synchronization and Specification Issues in Protocol Testing", IEEE Trans. Commun., Vol. COM-32, April 1984, pp. 389-395.

[SaSp 90] M. Sahinoglu and H. Spafford, “Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing”, SERC-TR-79-P, Software Engineering Research Center, Purdue University, September, 1990.

[SiLe 89] D.P. Sidhu and T.K. Leung, “Formal Methods for Protocol Testing: A Detailed Study”, IEEE Trans. SE-15, 4, April 1989, pp. 413-425.

[TuRo 92] C.D. Turner and D.J. Robson, "The Testing of Object-Oriented Programs", Technical Report TR-13/92, University of Durham, 1992.

[Ural 91] H. Ural, “Formal Methods for Test Sequence Generation”, Computer Communications, Vol. 15, No. 5, June 1992, pp. 311-325.

[YPB 93a] M. Yao, A. Petrenko and G.v. Bochmann, "Conformance Testing of Protocol Machines without Reset", Department Publication #861, Département d'informatique et de recherche opérationnelle, Université de Montréal, February 1993, 27 p.

[YPB 93b] M. Yao, A. Petrenko and G.v. Bochmann, "Conformance Testing of Protocol Machines without Reset", Proc. of the 13th IFIP Symposium on Protocol Specification, Testing and Verification, Liege, Belgium, May 25-28, 1993, pp. 241-253.

[YPB 94] M. Yao, A. Petrenko and G.v. Bochmann, “Fault Coverage Analysis in Respect to an FSM Specification”, Accepted by IEEE INFOCOM’94 to be held in Toronto, Canada, June 12-16, 1994.