Fault Coverage Analysis in Respect to an FSM Specification*

Mingyu Yao, Alexandre Petrenko** and Gregor v. Bochmann

Département d'informatique et de recherche opérationnelle
Université de Montréal, CP. 6128, Succ. Centre Ville, Montréal (Québec), Canada H3C 3J7

Abstract

It is shown in this paper that the problem of deciding if a test suite generated from a finite state machine provides complete fault coverage can be converted into the problem of minimizing the test tree representing the test suite. A fault coverage analysis procedure, capable of deciding if a given test suite provides complete fault coverage in respect to a given FSM specification, is then developed. The core of this procedure is a state minimization procedure developed specifically for the class of FSMs whose graphic representations are trees. The fault coverage analysis procedure can cope with partially specified FSM specifications which need not be reduced and faults that increase the number of states up to a chosen upper bound. Two necessary and one sufficient conditions, which in some cases may simplify the fault coverage analysis, are also presented.

1 Introduction

Apart from its traditional applications in the development of sequential digital circuits, the finite state machine (FSM) model has been extensively used in recent years in the conformance testing of communication protocols. Quite a number of methods have been proposed in the literature for generating test suites from finite state machines [Ural91, SiLe89]. It is well known that, however, not all of these methods can generate from a given FSM a test suite which is powerful enough to detect all possible faults in an implementation under test (IUT). Therefore, an important issue related to the test suite generation is to evaluate the *quality*, often called the *fault coverage*, of a test suite generated somehow from a given FSM. Most of the existing work that has been done on this issue are based on Monte-Carlo simulation [DaSa88, SiLe89, DDB91, MCS93] to estimate the fault coverage of a test suite. Our primary purpose in this paper is to develop a systematic approach which can decide if a test suite provides *complete fault coverage* and therefore is capable of detecting all possible faults in an IUT.

To minimize a state machine is to find another machine which has the least number of states but can fulfill all the functions of the original machine. The problem of state minimization was a very active research area in the study of automata theory from the 50's through to the 70's, mostly in relation with the synthesis of sequential circuits. The basic motivation is that an FSM used to model a sequential circuit under development often contains redundant states, i.e., states whose functions can be accomplished by other states. As the number of memory elements required for a physical realization of the FSM, i.e., the sequential circuit, is directly related to the number of states, the minimization of the number of states can in many cases reduce the complexity and cost of the realization. Although it is quite simple to minimize a completely specified machine, it can become in general very complex to minimize a partially specified machine. Therefore most work on state machine minimization has been done for partially specified machines [Gins59, GrLu65, Kell70, Kell71, PaUn59, Unge65].

It will be shown in this paper that the problem of deciding if a test suite provides complete fault coverage can be converted into the equivalent problem of minimizing the state machine which represents the test suite in the form of a tree. Therefore, we will first develop a state minimization procedure for the class of FSMs whose graphic representations are trees. This minimization procedure will combine the advantages of Kella's two state minimization approaches [Kell70, Kell71] and provide some features required for its application to the fault coverage analysis. We will then propose a fault coverage analysis procedure which starts with the conversion of a given test suite into a tree FSM and then calls the state minimization procedure to minimize this tree FSM. If, as a result of the minimization of this tree FSM, an FSM is found which cannot accomplish all the input/output traces specified in the given FSM specification, the test suite is considered to be unable to provide complete fault coverage. On the other hand, if no such FSM is found, the given test suite is said to provide complete fault coverage. The fault coverage analysis procedure can be applied to test suites generated from partially specified specification machines which

---

* 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 Université de Montréal.

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

0743-166X/94 $3.00 © 1994 IEEE

need not be reduced. It can also cope with faults that increase the number of states up to a chosen upper bound.

The rest of the paper is organized as follows. The FSM model is first presented in Section 2. The concept of complete fault coverage of a test suite is defined in Section 3. A state minimization procedure for tree machines is then developed in Section 4. It is then shown in Section 5 that the minimization procedure for tree machines can be used to decide if a test suite provides complete fault coverage. Two necessary and one sufficient conditions are also given there which may in some cases simplify the fault coverage analysis. Finally in Section 6, our approach is compared with related work.

2 The FSM model

A *finite state machine*, often simply called a *machine* throughout this paper, is essentially an initialized *Mealy machine* defined below.

Definition 2.1: A finite state machine is a 7-tuple $<S, X, Y, S_1, \delta, \lambda, D>$, where
S is a state set $\{S_1, S_2, ..., S_n\}$ 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*, iff D = S x X. Otherwise it is said to be *partially* or *incompletely specified*. Since δ and λ are required to be functions, this FSM model is *deterministic*. That is, for each $(S_i, x) \in D$, there should be exactly one state $S_j \in S$ and exactly one output symbol $y \in Y$ such that $\delta(S_i, x) = S_j$ and $\lambda(S_i, x) = y$. In this case, we say there is a transition 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. 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. An example of a graphic representation of an FSM is given in Figure 1.

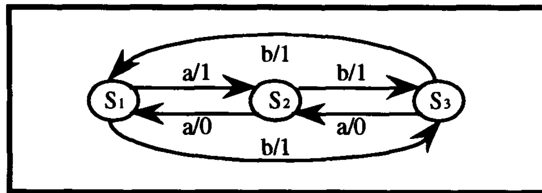

Figure 1: An FSM

For an alphabet Z, Z^* represents the set of words constructed on Z and "ε" represents the empty word, i.e., the word consisting of no symbol. The dot "·" is used to represent the concatenation operation of two words. However, this dot symbol is often omitted when no ambiguity arises.

Definition 2.2: Let $p = x_1x_2 \cdots x_k \in X^*$. p is said to be a defined input sequence for state $S_i \in S$, if there exist k states $S_{i1}, S_{i2}, ..., S_{ik} \in S$ and an output sequence $q = y_1y_2...y_k \in Y^*$ 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 the one in (2-1), can be abbreviated as S_i -p/q-> S_{ik} if we are not interested in the intermediate states. When we do not care about the output sequence, it 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}. Therefore, 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: Let $p = x_1x_2 \cdots x_k \in \psi(S_i)$. Then,
$\delta(S_i, \varepsilon) = S_i$, $\delta(S_i, p) = \delta(\delta(S_i, p'), x_k)$
$\lambda(S_i, \varepsilon) = \varepsilon$, $\lambda(S_i, p) = \lambda(S_i, p').\lambda(\delta(S_i, p'), x_k)$
where $p' = x_1x_2 \cdots x_{k-1}$. ■

Definition 2.4: Two states S_i and S_j of a machine M are said to be *compatible* states, written as $S_i \cong S_j$, if for $\forall\ p \in \psi(S_i) \cap \psi(S_j)$, $\lambda(S_i, p) = \lambda(S_j, p)$. Otherwise, they are said to be *distinct* states and written as $S_i \ncong S_j$. ■

We note that, according to the above definition, if $\psi(S_i) \cap \psi(S_j) = \emptyset$, 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* [Gill62, Koha78].

Definition 2.5: A machine is said to be *reduced* if, for any pair of states S_i and S_j, $S_i \ncong S_j$. ■

Definition 2.6: Let C be a subset of states. C is said to be a *compatible class* if, for any pair of states $S_i, S_j \in C$, $S_i \cong S_j$. ■

Obviously, any subset of a compatible class is also a compatible class.

Definition 2.7: A compatible class C is said to be *maximal* if, for any state $S_i \notin C$, there exists $S_j \in C$ such that $S_i \ncong S_j$. ■

It is easy to see that once we have found all the maximal compatible classes, we have essentially also found all the compatible classes as any compatible class should be a subset of some maximal compatible class. For a subset B of states and an input symbol x, we use **NEXT**(x; B) to denote all the states which can be reached under input x from states in B, i.e.,
NEXT(x; B) = $\{ \delta(S_i, x) \mid S_i \in B$ and $\delta(S_i, x)$ defined $\}$.

Definition 2.8: For a compatible class C and any input x, **NEXT**(x; C) is also a compatible class and is said to be an *implied compatible class* of C under input x. ■

Definition 2.9: Let CS = $\{ C_1, C_2, ..., C_k \}$ be a set of compatible classes.

(1) CS is called a *compatible covering* if $C_1 \cup C_2 \cup ... \cup C_k = \{ S_1, S_2, ..., S_n \}$;
(2) CS is said to be *closed* if for any $C_i \in$ CS and any input x, there exists $C_j \in$ CS such that **NEXT**(x; C_i) $\subseteq C_j$;

(3) CS is said to be a *closed compatible covering* if it is a compatible covering and closed;

(4) CS is said to be a *minimal closed compatible covering* if it is a closed covering containing the least number of compatible classes. ■

Definition 2.10: Let M and M' be two FSMs with the same input symbol set and λ_1 and λ_2 be their output functions respectively. State S_i of M is said to *cover* (or *contain*) state S_j' of M' if and only if $\psi(S_j') \subseteq \psi(S_i)$ and $\lambda_1(S_i, p) = \lambda_2(S_j', p)$, for $\forall p \in \psi(S_j')$. ■

Definition 2.11: Let S_1 and S_1' be the respective initial states of the machines M and M' with the same input symbol set. Then M is said to *cover* M' if and only if S_1 *covers* S_1'. ■

Apparently, if machine M covers machine M', then M' can be replaced by M, since all the functions of M' (in terms of its specified input/output traces) can be accomplished by M. The machine cover relation given in Definition 2.11 is essentially the same as the quasi-equivalence relation in [Gill62] and the **CONF** relation in [YPB93]. When both M and M' are completely specified, the machine cover relation becomes the well-known equivalence relation for initialized machines [Koha78, Gill62, Chow78, etc.].

Definition 2.12: A reduced FSM which covers a non-reduced FSM is said to be a *reduced form* of the non-reduced FSM. A reduced form which has the least number of states is said to be a *minimal form.* ■

We note that, for a completely specified FSM, its reduced form and minimal form are the same and unique. For a partially specified FSM, however, both its reduced and minimal forms can be non unique [Koha78].

3 Complete fault coverage of a test suite

Testing based on the FSM model that has been extensively used in the conformance testing of protocols as well as traditional hardware testing is basically black-box testing. The FSM based black-box testing is essentially "the testing of an FSM implementation" [Ural91, Yann91]: given an FSM specification (denoted as M_S) and an implementation of this FSM (denoted as M_I), one is asked to decide, through the testing of M_I as a black-box, whether M_I is a valid implementation of M_S according to certain predefined criterion often called an implementation relation or a conformance relation [YPB93]. For deterministic FSMs, as discussed throughout this paper, the *machine cover* relation given in Definition 2.11 is the strongest relation that can be tested and is essentially the same as the **CONF** relation defined in [YPB93]. For further discussion, we need to introduce the following concepts.

Definition 3.1: Let S_1 be the initial state of a specification machine M_S. A *test case* (or *test sequence*) is an input sequence of finite length and defined for S_1, i.e., in $\psi(S_1)$. A *test suite* is a finite set of test cases. ■

Each test case starts from the initial state S_1 of the specification machine and will be applied to the initial state of an implementation machine under test. Therefore, a special input symbol "r" called *reset* is used at the beginning of each test case. When the rest symbol "r" is applied, an implementation machine will transfer to its initial state no matter which state it is currently in.

Definition 3.2: Let M_I and M_S be an implementation machine and a specification machine, respectively. Let λ_I and λ_S be their respective output functions, and I_1 and S_1 be their respective initial states. For a test suite TS, M_I is said to pass TS, written M_I pass TS, if $\lambda_I(I_1, t) = \lambda_S(S_1, t)$ for $\forall t \in$ TS. ■

It is well known in the literature [Ural91, Yann91] that testing if a black-box FSM implementation covers (i.e. conforms to) a given specification machine can only be done under certain assumptions. One most notable assumption is that the number of states of an implementation machine should be limited by an upper bound m (which can be larger than n, the number of states of the specification machine). Therefore, we need to introduce the following definition.

Definition 3.3: Let X be the input set of the given specification machine M_S, the set of implementation machines with number of states limited by an upper bound m is denoted by $\mathbf{I}(m, X)$ which consists of all the minimal completely specified machines with no more than m states. ■

As we have seen in the above definition, an FSM representing an implementation (i.e., a machine in $\mathbf{I}(m, X)$) is required to be completely specified. This is because it is treated as a black-box during testing and therefore should give an *observable* response to any input in the input set X [PBD93]. As any non-minimal completely specified machine is equivalent to its minimal form, we have to include only the minimal machines in $\mathbf{I}(m, X)$.

Definition 3.4: Let TS be a test suite. TS is said to be an m-complete test suite in respect to the given specification machine M_S if, for any machine $M_I \in \mathbf{I}(m, X)$, M_I passes TS if and only if M_I covers M_S. ■

The notion of an "m-complete" test suite given above is a more general version of the notion of a "unique" test suite introduced in [VuKo90]. For the specification machine M_S which is completely specified and therefore is in $\mathbf{I}(m, X)$ (when $m \geq n$), a test suite is said to be its unique test suite in respect to the upper bound m if M_S is the only machine in $\mathbf{I}(m, X)$ which can pass the test suite. In the case that the specification machine M_S is partially specified, the notion of an "m-complete" test suite given in Definition 3.4 should be used. An obvious way to verify if a test suite is m-complete is to use the "trial-and-error" method: take a machine from $\mathbf{I}(m, X)$ and check if it can pass the test suite and if so, further check if it covers the given specification machine. Repeat this operation until either a machine is encountered which passes the test suite but does not cover the specification machine or all the machines in $\mathbf{I}(m, X)$ are examined. In the former case we can conclude that the test suite is not m-complete while in the latter case we can say it is m-complete. Apparently, the practical application of this approach is

rather limited due to its high cost. This is the main reason that some researchers [DaSa88, SiLe89, DDB91, MCS93] have used simulation approaches to approximately estimate the fault coverage of a test suite.

4 The minimization of finite state machines

To minimize a non-reduced machine is to find a minimal form of the original machine. The minimization of a completely specified machine is quite easy and can be done in two steps: (1) to find the *minimal equivalence partition* on the set of states of the given machine such that two states are in the same block *if and only if* they are equivalent; and (2) to merge all the states in a block into one state. The machine obtained after these two steps is the reduced and minimal form of the original machine and is equivalent to the original one. However, it becomes much more difficult to minimize a partially specified machine. As previous work has shown [Gins59, GrLu65, PaUn59, Unge65, Koha78, Gill62 etc.], to find a minimal form for a partially specified machine requires to find a minimal closed compatible covering. Unfortunately, there is no simple and precise procedure leading to the selection of a minimal closed compatible covering from all the compatible coverings. Therefore, all the proposed approaches [Gins59, GrLu65, PaUn59, Unge65, Koha78, Gill62 etc.] are inherently based on *trial-and-error*.

Our primary purpose in this section is to develop a state minimization procedure for a special class of machines which we call tree machines. Basically, a tree machine is a finite state machine whose graphic representation is a tree with the initial state of the machine as the root of the tree. It follows from the property of a tree that all the states of a tree machine can be reached from its initial state and for each state S_i (except the initial state) of a tree machine, there is one and only one other state S_j ($j \neq i$) such that there is a transition leading from S_j to S_i. This allows us to simplify the minimization procedure for the tree machines in a way similar to one of Kella's work [Kell71].

Definition 4.1: A *compatible partition* is a compatible covering which consists of pair wisely disjoint compatible classes. A *closed compatible partition* is a compatible partition which is closed. A *minimal closed compatible partition* is a closed compatible partition which consists of a minimum number of compatible classes. ■

The simplification on the minimization procedure in the case of a tree machine is stated in the next lemma.

Lemma 4.2: A reduced form of a given tree machine corresponds to a closed compatible partition of the tree machine. When a closed compatible partition is minimal, its corresponding reduced form is also minimal. ■

Kella proved in [Kell71] a special case of this lemma where the tree machine has only one branch. The proof given there is actually also valid in the general case where the tree machine has several branches. This lemma implies that only minimal closed compatible partitions, rather than minimal closed compatible coverings, need to be found for the construction of the minimal forms of a given tree machine. As is clear from the definitions, a compatible partition is also a compatible covering, but not vice versa. Therefore the set of all compatible partitions is a subset of all compatible coverings and actually in most cases, the former is much smaller than the latter. This implies that the amount of search for a minimal closed compatible partition from the set of all compatible partitions can in most cases be much smaller than the amount of search for a minimal closed compatible covering from the set of all compatible coverings.

Let X_1, Y_1, Z_1, z_1, δ_1, λ_1, D_1 be the input symbol set, the output symbol set, the state set, the initial state, the transfer function, the output function and the specification domain of machine M_1; and similarly X_2, Y_2, Z_2, z_2, δ_2, λ_2, D_2 for machine M_2.

Definition 4.3: M_1 is said to be a *submachine* of M_2 if $X_1 \subseteq X_2$, $Y_1 \subseteq Y_2$, $Z_1 \subseteq Z_2$, $D_1 \subseteq D_2$, $z_1 = z_2$ and δ_1, λ_1 are the respective restrictions of δ_2, λ_2 to D_1. ■

We use $\overline{M}$ to denote a reduced form of M and { $\overline{M}$ } for the set all the reduced forms of M.

Definition 4.4: For machine M_1 and a submachine M_2 of M_1, a reduced form $\overline{M}_1$ of M_1 is said to be based on a reduced form $\overline{M}_2$ of M_2 if $\overline{M}_2$ is a submachine of $\overline{M}_1$. ■

Let $\{V_1, V_2, ..., V_w\}$ be the state set of a given tree machine M. We use M(1), M(2), ..., M(w) to denote its w submachines, where M(i) is obtained from M by deleting the last (w-i) states V_{i+1}, V_{i+2}, ..., V_w and all the transitions leading from/to these states. Then the minimization of M is based on the idea that { $\overline{M}(i)$ } can be generated by adding state V_i to all the reduced forms in { $\overline{M}(i-1)$ }. This is justified by the following lemma.

Lemma 4.5: Each reduced form $\overline{M}(i)$ of M(i) is based on a reduced form $\overline{M}(i-1)$ of M(i-1). ■

This lemma is a special case of a relevant theorem proved in [Kell70]. It has been proved there that this conclusion holds for a more general class of machines which includes the tree machines.

Definition 4.6: A compatible class C of the tree machine M is said to be compatible with a state V_i, written $V_i \cong C$, if $V_i \cong V_j$, for $\forall\ V_j \in C$. Otherwise C is said to be incompatible with state V_i, written $V_i \ncong C$. ■

Definition 4.7: Let E = { C_1, C_2, ..., C_ℓ } be a set of pair wisely disjoint compatible classes, i.e., $C_i \cap C_j = \emptyset$, for $i \neq j$. Then E is said to be *incompatible* with a state V_i, written as $V_i \ncong E$, if $V_i \ncong C_j$, for j = 1, 2, ..., ℓ. ■

We present in the following a procedure which, when given a reduced form $\overline{M}(i-1)$ of M(i-1), will incorporate the next state V_i to generate all the reduced forms of M(i) which are based on this particular $\overline{M}(i-1)$ and have no more than m states, where m is a given integer representing the upper bound on the number of states of any reduced form. Therefore this procedure has three input parameters: the upper bound m, state V_i and the reduced form $\overline{M}(i-1)$. As Lemma 4.2 indicates, the given $\overline{M}(i-1)$ corresponds to a closed compatible partition which, without loosing generality, is denoted as E = $\{C_1, C_2, ..., C_k\}$ on the state set $\{V_1, V_2, ..., V_{i-1}\}$.

Procedure 4.8:

Step1: If $V_i \notin E$, i.e., $V_i \notin C_j$ for $\forall C_j \in E$, go to Step3; otherwise go to Step 2.

Step2: Let $\bar{C}_j = \{V_i\} \cup C_j$, i.e., add V_i to C_j, for all $V_i \leq C_j$. Replace, one at a time, C_j in E by $\bar{C}_j$ to form a compatible covering (actually a compatible partition on the set of states $\{V_1, V_2, ..., V_i\}$. Push all such generated compatible coverings to the stack COVSTACK and then go to Step4.

Step3: If $k < m$, generate all maximal compatible classes of M(i) which include V_i. Add each of these, one at a time, to E to form a compatible covering on $\{V_1, V_2, ..., V_i\}$. Push all such generated compatible coverings to the stack COVSTACK and go to Step4. Otherwise, i.e., if $k = m$, discard the compatible partition E and terminate.

Step4: Pop a compatible covering B from the stack COVSTACK. If COVSTACK is empty, terminate; otherwise go to the next step.

Step5: Check if the compatible covering B is closed.

a) If B is closed, delete in all possible combinations the multiple appearances of states in B to form compatible partitions. Discard all those compatible partitions which are not closed. Record each of the remaining closed compatible partitions as a reduced form M(i). Go to Step4.

b) If B is not closed, i.e., there exists a compatible class $B_j \in$ B and an input symbol x such that **NEXT**(x; B_j) is not included in any compatible classes in B, then go to Step6.

Step6:

a) If B_j only includes states which are not included in any other compatible class in B, go to Step7.

b) If B_j includes some states which are also included in some other compatible classes in B, delete these states from B_j one at a time to form as many new compatible coverings as there are such states. Push all these new compatible coverings to the COVSTACK and go to Step4.

Step7:

a) If the number of compatible classes in B is equal to m, discard B and go to Step 4.

b) If the number of compatible classes in B is less than m, form all the maximal compatible classes of M(i) which include **NEXT**(x; B_j) and missing in the original covering B (after adding back all states deleted in previous steps). Add each of these maximal compatible classes, one at a time, to B to form a number of new compatible coverings. Push these new compatible coverings to COVSTACK and go back to Step4. ■

The explanation of how this procedure works can be found in [YPB94a]. The algorithm of Procedure 4.8 is developed based on Algorithm 2 in [Kell70]. Although these algorithms look quite similar, some improvements of the former on the latter can still be observed. Firstly, as proved in Lemma 4.2, only closed compatible partitions need to be considered in the construction of $\bar{M}$(i)'s. Therefore, in Step 5 of Procedure 4.8, only the closed compatible partitions implied by a closed compatible covering are kept while others are discarded. This will reduce the amount of work required for the construction of $\{\bar{M}(i+1)\}$ when the next state V_{i+1} should be added. Secondly, since an upper bound on the number of states of a reduced form is imposed in Procedure 4.8, a compatible covering can be dropped out from further consideration whenever its number of compatible classes exceeds that upper bound. This feature, not found in Kella's algorithm, is specifically added in Procedure 4.8 for its application in checking the m-completeness of a test suite.

The next procedure, Procedure 4.9, is developed for state minimization of tree machines. Apart from the tree machine M with w states $\{V_1, V_2, ..., V_w\}$ which needs to be minimized, this procedure takes two additional input parameters: an upper bound m and a reference machine M_r which covers the tree machine M. It then calls Procedure 4.8 to incorporate, one at a time, the states V_1, V_2, ..., V_w in search for a reduced form $\bar{M}$ (of M) which has the least number of states and does not cover the reference machine M_r. It stops when either such a reduced form $\bar{M}$ is found or no such reduced form can be found (due to the upper bound m imposed on the number of states of the reduced form).

Procedure 4.9:

Step1: Use some procedure (can be found in many references) to find all pairs of compatible states for the given tree machine M.

Step2: Let $\bar{M}(1) = \{V_1\}$. Add it to the reduced form list RLIST (which is initially empty).

Step3: If the list RLIST is empty, it can be concluded that no reduced form (of the tree machine), which does not cover the reference machine M_r, can be found within the upper bound m on the number of states and therefore terminate.
Otherwise, take a reduced form $\bar{M}$(i) from RLIST which has the least number of states (if more than one is available, take one with the largest i) and then go to the next step.

Step4: If $i = w$, go to Step5.
Otherwise (i.e. $i < w$), call Procedure 4.8, with the upper bound m, state V_{i+1} and the chosen $\bar{M}$(i) as input parameters, to generate all the reduced forms $\bar{M}$(i+1) based on this $\bar{M}$(i). Add all the generated $\bar{M}$(i+1)'s to RLIST and go to Step3.

Step5: Check if $\bar{M}$(i) covers the reference machine M_r. If not, this $\bar{M}$(i) is a reduced form $\bar{M}$ that is being searched for and therefore terminate the procedure. Otherwise, go back to Step3. ■

As priority for further consideration in Step3 is always given to a reduced form which has the least number of states (i.e. its corresponding closed compatible partition has the least number of compatible classes), this

procedure guarantees that the reduced form $\overline{M}$, if can be found, has the least number of states and does not cover the given reference machine M_r. Algorithms can be found in the literature (see, for instance, [Gill62]) to check whether one machine covers the other (Step 5).

5 Checking the completeness of a test suite

We can now use Procedure 4.9 to check if a given test suite is, for an integer m, m-complete in respect to a specification machine M_S. The idea is to first represent the test suite as a tree machine, and then use Procedure 4.9 to minimize this tree machine with m as the upper bound and M_S as the reference machine M_r. The test suite is m-complete if and only if no reduced form can be found which does not cover the specification machine M_S.

The conversion of a test suite into a tree machine can be done in a quite straightforward way. A branch (starting from the root of the tree) is created for each test case in the test suite. The number of edges in a branch is equal to the length of the corresponding test case (without counting the reset symbol "r"). An edge is labeled by a pair of input and output symbols. The concatenation of the labels on the edges of a branch forms an input/output sequence which should be the same as the one obtained when the corresponding test case is applied to the specification machine. Whenever two test cases have a common prefix, their corresponding branches should be merged for that common part so that the tree machine will be deterministic. The procedure for checking the completeness of a test suite in respect to an FSM specification can now be formulated as follows.

Procedure 5.1:

Step1: Convert the given test suite TS into a tree machine M.

Step2: Call Procedure 4.9 to minimize this tree machine M with m as the upper bound and M_S as the reference machine.

Step3: If no reduced machine is found in Step2 which does not cover M_S, the test suite is m-complete; Otherwise, it is not m-complete. ■

The validity of this procedure is justified by the following theorem.

Theorem 5.2: Let TS be a test suite and M be the tree machine representing TS. Then TS is m-complete in respect to the FSM specification M_S if and only if all the reduced forms of M which have not more than m states cover the specification machine M_S. ■

The proof for this theorem is omitted due to limited space. We give an example here to show how this procedure works.

Example 5.3: TS = {r.a.b.b, r.b.a.a} is a test suite derived from the specification machine shown in Figure 1. We are required to check if this test suite is 3-complete. Therefore, we follow Procedure 5.1 to check. The first step of Procedure 5.1 is to convert this test suite into a

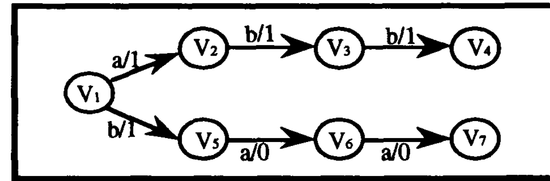

Figure 2: The tree machine

State	Incompatible States
V_1	V_5 V_6
V_2	
V_3	
V_4	
V_5	V_1
V_6	V_1
V_7	

Table 1: The list of incompatible pairs

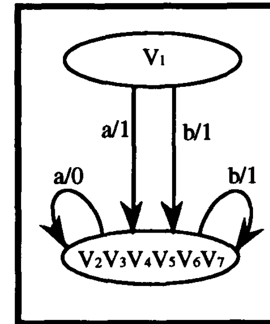

Figure 3: The FSM found by Procedure 4.9

Iterations	RLIST
1	{ V_1 }
2	{ V_1V_2 }
3	{ $V_1V_2V_3$ }
4	{ $V_1V_2V_3V_4$ }
5	{ V_1, $V_2V_3V_4V_5$ } { V_1V_2, $V_3V_4V_5$ } { V_1V_3, $V_2V_4V_5$ } { $V_1V_2V_4$, V_3V_5 }
6	{ V_1, $V_2V_3V_4V_5V_6$ } { V_1V_2, $V_3V_4V_5$ } { V_1V_3, $V_2V_4V_5$ } { $V_1V_2V_4$, V_3V_5 }
7	{ V_1, $V_2V_3V_4V_5V_6V_7$ } { V_1V_2, $V_3V_4V_5$ } { V_1V_3, $V_2V_4V_5$ } { $V_1V_2V_4$, V_3V_5 }

Table 2: Contents of the reduced form list RLIST

tree which is now shown in Figure 2. This tree machine has 7 states V_1, V_2, ..., V_7. The second step of Procedure 5.1 is to call Procedure 4.9 whose first step is to find the compatibility for each pair of states of this tree machine. We have listed in Table 1 the incompatible states for each state of the tree machine. A state not listed as incompatible with another state is therefore compatible with the latter. Table 2 lists the changes of the reduced

form list RLIST when the remaining steps of Procedure 4.9 are executed. At the last iteration, the closed compatible partition $\{V_1, V_2V_3V_4V_5V_6V_7\}$ is found which actually represents the FSM shown in Figure 3. Since this FSM has two states and does not cover the specification machine shown in Figure 1, we can conclude that the test suite TS is not 3-complete. ■

Although the procedure presented above can always be used to check if a given test suite is m-complete, the two necessary conditions and one sufficient condition given below can in some cases be used to give a quicker answer.

Let TS be a test suite for an FSM specification M_S with δ, λ and $\{S_1, S_2, ..., S_n\}$ as its transfer function, output function and state set, respectively. Further, let S_1 be its initial state. To formulate these conditions, we need the following three definitions.

Definition 5.4: A set of input sequences SC is said to be a *state cover set* of the specification machine M_S if, for each state S_i of M_S, there exists *exactly one* input sequence $\alpha \in$ SC such that $S_i = \delta(S_1, \alpha)$. ■

Definition 5.5: A set of input sequences TC is said to be a *transition cover set* of the specification machine M_S if for each transition S_i-x->S_j in the specification machine M_S, there exist two input sequences α, $\alpha.x \in$ TC such that $S_i = \delta(S_1, \alpha)$ and $S_j = \delta(S_1, \alpha x)$. ■

Definition 5.6: Let SC be a state cover set for M_S and TC = $\{\alpha.x \mid \text{for } \alpha \in SC, S_i=\delta(S_1, \alpha) \text{ and } S_i\text{-x->}S_j \text{ in } M_S\}$. Then TC is a transition cover set of M_S and is said to be based on the state cover set SC. ■

Let

AP(TS) = $\{\alpha \mid \text{if } \alpha \text{ is a prefix of some test case in TS}\}$, i.e., AP(TS) consists of all the prefixes of the test cases in TS. Then we can have our first necessary condition.

Necessary Condition 5.7: If, for some $m \geq n$, TS is m-complete in respect to M_S, then AP(TS) should contain a state cover set. ■

This necessary condition essentially says that an m-complete ($m \geq n$) test suite should traverse all the states of the specification machine. Actually, we can have a stronger necessary condition stated below which requires that all the transitions of the specification machine be traversed by a complete test suite.

Necessary Condition 5.8: If, for some $m \geq n$, TS is m-complete in respect to M_S, then AP(TS) should contain a transition cover set. ■

The validity of these two necessary conditions are obvious. Actually we require that Procedure 5.1 be used only after the test suite TS has been checked to satisfy these two necessary conditions. The following condition is a sufficient condition which can be used when the specification machine M_S is reduced. TS will be definitely n-complete, where n is the number of states of the reduced machine M_S, if it satisfies this sufficient condition.

Sufficient Condition 5.9: TS is an n-complete test suite in respect to the reduced specification machine M_S if

(1) AP(TS) contains a state cover set SC and a transition cover set TC based on SC; and

(2) For each pair of sequences $\alpha, \beta \in$ SC such that $\delta(S_1, \alpha) \neq \delta(S_1, \beta)$, there should be two sequences $\alpha\gamma, \beta\gamma \in$ AP(TS) such that $\lambda(\delta(S_1, \alpha), \gamma) \neq \lambda(\delta(S_1, \beta), \gamma)$; and

(3) For $\alpha \in$ (TC - SC) and $\beta \in$ SC such that $\delta(S_1, \alpha) \neq \delta(S_1, \beta)$, there should be two sequences $\alpha\gamma, \beta\gamma \in$ AP(TS) such that $\lambda(\delta(S_1, \alpha), \gamma) \neq \lambda(\delta(S_1, \beta), \gamma)$. ■

The proof for this sufficient condition is omitted here due to limited space. It is interesting to note that any test suite generated by the DS method [Gone70], the UIOv method [Vuon89], the W method [Chow78], the Wp method [Fuji91] or the HSI method [Petr91] satisfies this sufficient condition and therefore is n-complete.

6 Comparison with related work

In this paper, we have introduced the concept of m-completeness of a test suite in respect to an FSM specification. This concept is more general than the concept of uniqueness of a test suite introduced by Vuong and Ko [VuKo90]. The notion of uniqueness of a test suite is applicable only to completely specified machines. The notion of m-completeness of a test suite, however, can be applied to partially specified as well as completely specified machines. It even does not require the specification machines to be reduced. We have also developed a procedure (Procedure 5.1) which is capable of deciding if a given test suite is m-complete in respect to a given FSM specification, where m is an integer which can be larger than the number of states of the specification machine. A tool which implements this procedure is now available. The complexity *upper bound* of the procedure is $O(m^w)$, where w is the number of states of tree machine representing the test suite. However, experiments that have been conducted with the tool has demonstrated that the real complexity in practice is far less.

Procedure 5.1 is based on the state minimization procedure (Procedure 4.9) designed for the so-called tree machines. Procedure 4.9 and the procedure that it calls (Procedure 4.8) combine the advantages of Kella's two approaches [Kell70, Kell71]. As already mentioned in Section 4, they also provide two important features which are not found in previous state minimization procedures [Gins59, GrLu65, Kell70, Kell71, PaUn59, Unge65].

Procedure 5.1 provides a systematic approach for checking the completeness of a test suite and therefore is different from those simulation based approaches [DaSa88, SiLe89, DDB91, MCS93]. It also differs from our recent work [YPB94b] where we proposed a metric approach to estimating the fault coverage of a test suite. Another related work is the CSP method for test suite generation [VuKo90] which could be adjusted to check the m-completeness of a test suite. However, its complexity in practice would be much higher than our method as it would generate all machines, both reduced and non-reduced (within the upper bound on the number of states), which can pass the given test suite. On the other hand, our Procedure 5.1 examines, in the worst case when the given test suite is m-complete, all the reduced

machines within the upper bound on the number of states. Other related work can be found in [LoSh92, MiPa92]. However, their primary purpose is to generate test suites that provide complete fault coverage (or maximal fault coverage as they called) rather than to evaluate the fault coverage of a given test suite.

Apart from its application in checking the completeness of a test suite, Procedure 5.1 can also be used for incremental test suite development. Actually, this function is now available in our tool. If a given test suite (which can be empty) is not m-complete, a machine will be found which does not cover the specification machine. Therefore, an additional test case can be derived which distinguishes this machine from the specification machine. The test suite, which includes the newly generated additional test case, is then checked again for m-completeness. This process is repeated until the test suite has achieved m-complete fault coverage.

Another possible application of the state minimization procedure (Procedure 4.9) is the diagnostics for FSM implementations [GhBo92]. If an FSM implementation for an FSM specification fails to pass a given test suite, a tree machine is constructed with the input sequences (test cases) in the test suite and the corresponding output sequences observed during the testing. This tree machine is then minimized. However, no reference machine is required during the application of Procedure 4.9 in this case and therefore a reduced machine can be definitely found. This reduced machine is actually a minimal form of the tree machine. By comparing this minimal machine with the specification machine, we are able to tell what faults are in the implementation machine.

References

[Chow78] T.S. Chow, "Test Design Modeled by Finite- State Machines", IEEE Trans. SE-4, 3,1978, pp.178-187.

[DaSa88] A.T. Dahbura and K. Sabnani, "An Experience in Estimating Fault Coverage of a Protocol Test", Proc. INFORCOM'88, pp. 71-79.

[DDB91] M. Dubuc, R. Dssouli and G.v. Bochmann, "TESTL: A tool for Incremental Test Suite Design Based on Finite State Model", Proc. IWPTS'91.

[Fuji91] S. Fujiwara, et al., "Test Selection Based on Finite State Models", IEEE Trans. SE-17, 6, June 1991, pp. 591-603.

[Gill62] A. Gill, "Introduction to the Theory of Finite-State Machines", McGraw-Hill Book Company, Inc., 1962, 207 p.

[Gins59] S. Ginsburg, "On the Reduction of Superfluous States in a Sequential Machines", J. ACM, Vol. 6, 1959, pp. 252-282.

[Gone70] G. Gonenc, "A Method for the Design of Fault Detection Experiments", IEEE Trans. Computers, Vol. C-19, No. 6, June 1970, pp. 551-558.

[GhBo92] A. Ghedamsi and G.v. Bochmann, "Test Result Analysis and Diagnostics for Finite State Machines", Proc. of the 12th International Conference on Distributed Systems, Yokohama, Japan, June 9-12, 1992.

[GrLu65] A. Grasselli and F. Luccio, "A Method for Minimizing the Number of Internal States in Incompletely Sequential Networks", IEEE Trans. Electronic Computers, Vol. EC-14, June 1965, pp. 350-359.

[Kell70] J. Kella, "State Minimization of Incompletely Specified Sequential Machines", IEEE Trans. Computers, Vol. C-19, No.4, April 1970, pp. 342-348.

[Kell71] J. Kella, "Sequential Machine Identification", IEEE Trans. Computers (Short Notes), Vol. C-20, No. 3, March 1971, pp. 332-338.

[Koha78] Z. Kohavi, "Switching and Finite Automata Theory", New York, McGraw-Hill, 1978.

[LoSh92] 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.

[MCS93] H. Motteler, A. Chung and D. Sidhu, "Fault Coverage of UIO-based Methods for Protocol Testing", Proc. IWPTS'93.

[MiPa92] 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.

[NaTs81] S. Naito and M. Tsunoyama, "Fault Detection for Sequential Machines by Transition-Tours", Proc. of FTCS, 1981, pp. 238-243.

[PaUn59] M.C. Paull and S.H. Unger, "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions", IRE Trans. Electronic Computers, Vol. EC-8, Sept. 1959, pp. 356-367.

[PBD93] A. Petrenko, G.v. Bochmann and R. Dssouli, "Conformance Relations and Test Derivation", Proc. IWPTS'93.

[Petr91] A.F. Petrenko, "Checking Experiments with Protocol Machines", Proc. IWPTS'91.

[SiLe89] D.P. Sidhu and T.K. Leung, "Formal Methods for Protocol Testing: A Detailed Study", IEEE Trans. Software Engineering, Vol. SE-15, No. 4, April 1989, pp. 413-426.

[Unge65] S.H. Unger, "Flow Table Simplification - Some Useful Aids", IEEE Trans. Electronic Computers, Vol. EC-14, June 1965, pp. 472- 475.

[Ural91] H. Ural, "Formal Methods for Test Sequence Generation", Computer Communications,Vol. 15, No. 5, June 1992, pp. 311-325.

[VuKo90] S.T. Vuong and K.C. Ko, "A Novel Approach to Protocol Test Sequence Generation", Proc. GlobalCOM'90.

[Vuon89] S.T. Vuong, et al., "The UIOv-method for Protocol Test Sequence Generation", Proc. IWPTS'89.

[Yann91] M. Yannakakis, "Testing Finite State Machines", Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, New Orleans, Louisiana, 1991, pp. 476-485.

[YPB93] M. Yao, A. Petrenko and G.v. Bochmann, "Conformance Testing of Protocol Machines without Reset", Proc. of PSTV'93.

[YPB94a] M. Yao, A. Petrenko and G.v. Bochmann, "Fault Coverage Analysis in Respect to an FSM Specification", Publication # 896, Dept. IRO, University of Montreal, Feb. 1994.

[YPB94b] M.Yao, A. Petrenko and G.v. Bochmann, "A Metric Approach to Measuring Fault Coverage of Software Testing in Respect to the FSM Model", submitted for publication.