A characterization of the general protocol conformance test sequence generation problem for EFSM's

Raymond E. Miller and Junehwa Song
Department of Computer Science
University of Maryland
email : {miller,junesong}@cs.umd.edu

June 9, 1998

Abstract

We discuss the problems which arise in conformance testing when using the EFSM model as well as the problems which arise when extending the concepts used in FSM testing to EFSM. We characterize the problems of stability and executability of transitions, and see what these mean for generating test sequences. Also, we extend the concept of UIO sequences to EFSM, calling them identifying sequences, and discuss the problems which arise in this extension. Finally, we extend the concept of converging transitions to the EFSM model, which can aid in reducing the length of the test sequences.

1 Introduction

Protocol conformance testing has been extensively studied using the *finite state machine* (FSM) model [9, 10, 11, 14, 17, 20, 24]. Research using this approach take advantage of the simplicity of the specification formulation and its structure to derive techniques for obtaining test sequences with various desirable properties. The specification does not involve complex algebraic problems, and the types of faults studied are very limited. One of the very important developments in FSM conformance testing was the introduction of UIO sequences [14], which are used to identify the state reached after testing a transition. Many test sequence generation techniques use UIO sequences in one way or another.

There has also been some efforts to develop conformance test sequences using the *extended finite state machine*(EFSM) model [19, 25, 26]. Some of these approaches are discussed in the context of Estelle, which is a protocol specification language based on the EFSM model [19], and others are discussed more abstractly.

The EFSM model is an extension of a FSM, augmented with variables. These variables, and how they change during transitions, make it necessary to consider the algebraic properties of the modules in testing. The similarity of EFSM's to FSM's, i.e., describing protocols with state transitions and using transition diagrams to specify state transitions, enables one to adapt the test techniques developed for FSM to EFSM, especially to test the flow of control. But this adaptation is more complicated than it appears on the surface. First, it is not simple to separate the transition behavior from the algebraic properties introduced by the variables. Second, transitions interact with each other through actions and properties involving the variables. Moreover, this interaction is not fully observable when internal (or context) variables are involved.

In this paper, we discuss the problems which arise in conformance testing when using the EFSM model as well as the problems which arise when extending the concepts used in FSM testing to EFSM. We characterize the problems of stability and executability of transitions, and see what these mean for generating test sequences. Also, we extend the concept of UIO sequences to EFSM, calling them identifying sequences, and discuss the problems which arise in this extension. Finally, we extend the concept of converging transitions to the EFSM model, which can aid to reduce the length of the test sequences.

One of the basic problems in the conformance testing of a protocol module specified as an EFSM is the selection of an input value which satisfies an enabling predicate for the transition under test. To test satisfiability of a first order predicate is, in general, a semi-decidable problem. This semi-decidability cannot be avoided as long as we try to select input values for testing. We might, however, use some approximation or other simplifying assumption which gives some bound on the difficulty of searching for an input value satisfying the enabling predicate. In this paper, we do not try to overcome that semi-decidability, nor give an approximation method. Rather, we present formulations which characterize those problems mentioned in the previous paragraph in the form of predicates. Thus, the usefulness of our formulation lies in presenting a general characterization of the problem to which simplifying assumptions can be applied, thereby allowing one to study how the complexity of the test generation problem is reduced by particular assumptions.

The rest of this paper is organized as follows. In section 2, we describe our model. The problem of stability is discussed in section 3. Each of the succeeding sections are devoted to the discussion of executability, identifying sequences, and converging transitions. Section 7 concludes this paper.

2 Model

A protocol module is a triple, <S, V, T>.

- S is a tuple $< s_0, S' >$, where s_0 is the initial state, and S' is a set of states.
- V is a triple $<$ I, CV, O $>$, where I is the set of input variables, O the set of output variables, and CV the set of context variables:
  - Input variables are variables controllable by the tester,
  - Output variables are variables observable by the tester,
  - Context variables are variables neither observable nor controllable.

  Some variables can be used as both an input variable and an output variable. From now on, we use I to denote the vector of input variables (not a set), O for that of output variables, and CV for that of context variables.
- T is a set of transitions, where a transition t is a quadruple <Head(t), Tail(t), P(t), A(t)>.
  - Head(t) is the starting state of the transition t
  - Tail(t) is the ending state of the transition t
  - P(t) is the enabling predicate of t.[1]
  - A(t) is the action associated with a transition t. It is composed of a sequence of assignment statements.

We assume that the transitions from each state are deterministically defined. For each pair of transitions t_i and t_j, such that Head(t_i) = Head(t_j), there are no assignments of values to the variables in (I,CV,O) which enable both t_i and t_j.

The behavior of a protocol module depends not only on the state it is in but also on the values of variables. So a global state in an EFSM is a pair, consisting of a state and a tuple of variable values. We call such a tuple of variable values a variable state. A variable state is composed of an instance of (I, CV,O). Since I is selected and set at each step of testing, we are usually interested in tracing only (CV,O).

An action is a sequence of assignment statements, which changes the variable states. Once we represent the state of the variables as three vectors, i.e., (I, CV, O), we can calculate the functional notation for an action by performing its symbolic evaluation[2]. We specify this functional representation of action A(t) by $\widetilde{A(t)}$. [3] Thus, if a transition t is taken, the module changes from a global state $(Head(t), (I, CV, O))$ to a global state $(Tail(t), \widetilde{A(t)}(I, CV, O))$.

[1] P(t) is a boolean expression over variables (I, CV, O). If we need to specify the names of variables involved in the predicate, we specify them as P(t)(I, CV, O)

[2] Details about the symbolic evaluation can be found in [22] and [23]

[3] This function is composed of three parts, for defining new value of I, CV, and O. We can denote it as $\widetilde{A(t)}$(I, CV, O) $\longrightarrow$ (I', CV', O'). Since most times, we are not interested in tracing the I value, we usually write $\widetilde{A(t)}$(I,

3 Stability of transitions and Augmentation of the specification

Consider a transition t from some state s_i to state s_j. Suppose that we can set the input variables in such a way to make the enabling predicate for transition t, *i.e.,* P(t) is true. Then, assuming that no other predicate is true for this setting of input variables(i.e., assuming deterministic behavior), transition t will be taken to reach state s_j and the action for transition t will possibly change the values of some variables. Now, if no transition is enabled from state s_j with the current variable values, then s_j is called stable under this condition. However, it is possible that a transition leaving s_j is enabled, taking the protocol to a second state. This would mean that we could not test that we reached state s_j, nor test that the output variables reached their intended values–since the transition leaving s_j could change these variable values as well. This describes the stability problem.

To simplify the testing procedure, we attempt to apply inputs so after transition t, the system stops at s_j until a new set of inputs is applied. Since there is limited control of the variables, with the output and context variables not under control of the tester, achieving stability may not be possible in some cases, and a sequence of transitions may actually occur rather than just one.

In the discussion below, we formulate the condition under which a transition stops without any further successive transitions. This is called *stopping predicate.* We call a transition which has a satisfiable stopping predicate a *stopping transition.* Further, we formulate the condition under which a sequence of transitions, $t_1, t_2, ..., t_i$, is enabled successively by one input setting. We call such a sequence of transitions a *composite transition.*

Definition 1 *A* stopping predicate *of a transition t is the predicate of (I, CV,O), which, if satisfied, enables transition t only once and does not automatically enable any successive transitions.*

We can formulate the stopping predicate, SP, as follows. [4]

$$SP(t) = P(t) \bigwedge \neg wp(A(t), P), \quad \text{where} \tag{1}$$

$$P = \bigvee_{\forall\, t', Tail(t) = Head(t')} P(t')$$

Definition 2 *A* stopping transition *is a transition whose stopping predicate has at least one assignment of variables, (I, CV, O), that makes it true.*

CV, O) $\longrightarrow$ (CV', O'). Also we use $\widetilde{A_{CV}(t)}$ to denote the part which defines CV' only and $\widetilde{A_O(t)}$ that which defines O'.

[4]The concept of the *weakest precondition* was developed by Dijkstra [21]. Let S be a statement in some programming language and let Q be a predicate. wp(S,Q) is the set of states (described by a predicate) for which S terminates and Q is true on termination. Several axioms were developed related to the weakest precondition. We can use the rule of assignment and the rule of composition for A(t), which is a sequence of assignment statement.

- Assignment : wp(x = e, Q) = Q^x_e
- Composition : wp(S1;S2, Q) = wp(S1, wp(S2,Q))

Q^x_e represents the predicate Q where all free occurrences of x has been replaced by e.

Definition 3 *A* composite transition *of length i, $i > 1$, is a sequence of transitions, $t_1, t_2, ..., t_i$, such that $Tail(t_j) = Head(t_{j+1})$, $1 \leq j < i$, and $t_1, t_2, ..., t_i$ can be enabled successively by setting variable values only once (i.e., only for t_1).*

We now formulate a new set of transitions, T_{new}, from the set of transitions, T, of the original specification. In the new specification of the protocol module, where T is replaced by T_{new}, the stability of transition is guaranteed.

We recursively formulate T_i, before we form T_{new}.

- T_1 = {<Head(t), Tail(t), SP(t), A(t)> | $t \in T$ and SP(t) is satisfiable }[5]
- $T_{i+1} = \{< Head(t'), Tail(t), P(t') \bigwedge wp(A(t'), P(t)), A(t');A(t) >$
  $| \; t \in T_i, t' \in T, Tail(t') = Head(t), P(t') \bigwedge wp(A(t'), P(t))$is satisfiable }[6]
- $T_{new} = \{t \mid t \in T_i, 1 \leq i\}$

The transitions in T_1 are the stopping transitions in T, but the enabling predicates are replaced by stopping predicates. The transitions in T_i, i > 1, are those which were not in T, but formed from the composite transitions of length i. We can consider them as hidden transitions. Note that the enabling predicates are formed in a way that guarantees the stability of the transitions. With T_i, i≥1, we have all transitions, originally specified or hidden, and all are stable. [7]

Let's say we have a transition t, and t is a stopping transition as well as the first transition of a composite transition. Given a test requirement over t, enabling the stopping transition and enabling the composite transition have different meanings in the test requirement. The case of enabling the composite transition is not solely testing transition t. We believe that the statement about the test requirement should reflect this fact.

4 Executability

Some researchers have discussed the test sequence generation problem as producing a sequence of transitions over the transition diagram of the EFSM. A transition diagram shows the global transition behavior of the protocol. In deriving a transition sequence from a transition diagram, we can take advantage of its similarity to the FSM, and adapt some ideas of FSM's test sequence generation. A transition sequence is, however, not only a sequence of input/output pairs as is the case for a test sequence for an FSM.

In [16], a test sequence was derived reflecting the functional behavior of a protocol module. This test sequence consists of a sequence of transitions of the transition diagram. [10] pointed out such a test sequence may not be executable. In [10], def-ob paths and conditional paths are generated from the data flow graph. These are converted to expanded def-ob paths and expanded conditional paths, which are also sequences of transitions of the the transition diagram.

[5] This satisfiability is semi-decidable as we mentioned in the introduction

[6] ; represents the sequencing of statements.

[7] Sometimes, it is possible to have a transition T_∞. This means that we have an infinite enabled loop in the specification. This can be considered as an error in the specification. This problem should be detected in the validation process, rather than in conformance testing

The authors said that their expanded def-ob paths and conditional paths should be generated considering the executability of transitions, however they did not propose how this was to be done.

A transition sequence of a transition diagram may not be executable, because the executability depends not only on the sequence of transitions, which is represented on the transition diagram, but also on the enabling predicate of each transition. Consider a transition sequence $t_1@t_2$, where Tail(t_1) = Head(t_2). Even though they are specified as successive transitions on the transition diagram, it is not guaranteed that t_2 can always be executed after t_1 : it may not be possible to assign the variable values for t_2 such that P(t_2) is satisfied. Furthermore, the executability of t_2 is affected by the input selection for t_1 through the action, A(t_1).

Let's say $D = (I,\ CV,\ O)$ is the space of variable values which satisfies $P(t_1)$. D can be partitioned into $D' = (I', CV', O')$ and $D'' = (I'', CV'', O'')$, where value settings from D' for t_1 makes t_2 be enabled with proper input selection after t_1 is executed and value setting from D" for t_1 prevents t_2 from being enabled no matter what inputs are selected for t_2. Solving the executability problem of the sequence, $t_1@t_2$, reduces to finding the partition *D'* and *D"*.

To guarantee the executability of a transition sequence, it is not sufficient to consider the enabling predicate of each transition in the sequence separately. The actions of the transitions in the initial part of a sequence affect the satisfiability of the transitions in the latter part by changing the values of CV and O, which cannot be directly controlled by the tester in each step of a sequence. Thus, we need to consider the executability of the whole sequence globally by tracing the interactions of transitions. Below, we formulate the enabling condition of a transition sequence, by which we can test the executability of the transition sequence. In the formulation, we use $\widetilde{A(t)}$, the functional representation of A(t), to trace the effect of the action of t to the context variables and output variables. In the discussion below, we use the new transition set, T_{new}, to avoid the complications introduced by the stability problem. By doing this, each transition t_i, $1 \leq i \leq n$, causes a state changes from one stable state to a next stable state.

Definition 4 *The* enabling condition, *EC, of a sequence of transitions* $t_1, t_2, ..., t_n$, *such that* $t_i \in T_{new}$, $1 \leq i \leq n$, *and* $Head(t_{j+1}) = Tail(t_j)$, $1 \leq j < n$, *is defined as follows.*

$$\begin{array}{rll}
EC(t_1, t_2, ..., t_n) & := \; P(t_1)(I_1, CV_0, O_0) & \bigwedge \; \{\widetilde{A(t_1)}(I_1, CV_0, O_0) = (O_1, CV_1)\} \\
& \bigwedge \; P(t_2)(I_2, CV_1, O_1) & \bigwedge \; \{\widetilde{A(t_2)}(I_2, CV_1, O_1) = (O_2, CV_2\} \\
& \bigwedge \; ... & \\
& \bigwedge \; P(t_{n-1})(I_{n-1}, CV_{n-2}, O_{n-2}) & \bigwedge \; \{\widetilde{A(t_{n-1})}(I_{n-1}, CV_{n-2}, O_{n-2}) = (O_{n-1}, CV_{n-1})\} \\
& \bigwedge \; P(t_n)(I_n, CV_{n-1}, O_{n-1}) &
\end{array}$$

Given a sequence of transitions, we can use this enabling condition to test whether the sequence is executable. If EC is satisfiable, then the given transition sequence will be executable. The values of variables which satisfy EC will consist of input values which should be set, and output variables to be observed. To start a transition sequence, the state should be in Head(t_1) with variable state (CV_0, O_0)[8] and the tester should set input values to I_1. Successively setting input values to $I_2, I_3, ...$, the sequence of transition continues with output sequence of $O_1, O_2, ...$

[8]The value of CV can not be observed. The testing process should keep track of the CV value from the specification at each stage of the testing to figure out what the current value of CV is.

We can apply EC to slightly different situations as well. For example, we can use it to connect two executable sequences, i.e., getting *a transfer sequence*[9]. (We assume the availability of an algorithm to find all the paths between two vertices of a graph and enumerate them according to their length.) Given two executable sequences of transitions, seq1 and seq2, we take a path[10] between Tail(t) and Head(t'), where t is the last transition in seq1 and t' is the first transition of seq2, in terms of transitions. Let's call this TS. We, then, form a new transition sequence, seq1@TS@seq2, and test the satisfiability of EC of the new sequence. If it is not satisfiable, we take the next path as TS and repeat the same process until we get an executable connection.

A similar but slightly different situation, which we call a *navigation problem*, may arise as follows. Assume that the current state is s with (CV_0, O_0) and the next transition to test is t. To test t, we need to navigate to Head(t), make the enabling predicate of t satisfiable, and execute t. This can be done using EC as follows.

1. Get the next path $t_1, ..., t_i$ from s to Head(t) from the transition diagram, and let it be TS.
2. If EC(TS@t) is satisfiable with CV_0 and O_0, return $(I_1, ..., I_i, I'/O_1, ..., O_i, O')$.
   If not goto step 1.

If the current state s is the same as Head(t), the paths from s to Head(t) will form loops. In this case, the sequence of $I_1/O_1, ..., I_i/O_i$ may be used just to change the variable state.

5 The Identifying Sequence of a state

In many approaches to generate a conformance test sequence in the FSM model, a UIO sequence is attached to the test segment of a transition under test to see if the transition ended in the correct tail state. [10] introduced the UIO sequences in the conformance test of EFSM's. This is done by transforming the specification into a FSM form and then obtaining the UIO from the FSM.

Getting a UIO sequence in an EFSM is not straightforward, because of the introduction of the variables. In this section, we define the Identifying Sequence, IS, which is the counterpart of a UIO sequence in a FSM, and describe an approach for generating identifying sequences.

Definition 5 *An* Identifying Sequence, *IS, of a* state *s in a* variable state *(CV, O) is a sequence of inputs and outputs, $I_1/O_1, ..., I_n/O_n$, such that*
(1) s generates $O_1, ..., O_n$ if $I_1, ..., I_n$ is applied to s in (CV,O),
(2) but no other state shows the same input output behavior in (CV, O),
(3) no prefix of the sequence displays this unique behavior.

Note that an IS is different from a UIO sequence, in that it depends not only on the state in which a protocol module is, but also on the *variable state*, (CV,O).

We explain a basic algorithm to get an IS. The algorithm is shown in Figure 1. Basically, the problem is a graph search. The goal of the search is the IS of s0 in (CV,O). We search the goal

[9] transfer sequence was introduced in [10]

[10] We may order the paths in some way, for example, shortest path first, and pick the paths in that order

from all the possible input sequences applicable to s0 in (CV,O). We use the following notations in the description.

- i : i is used to index the current length of the partial input sequence. Being at the stage i means that we have generated a partial input sequence of length i-1 and are going to select i_th input.
- I(1..i) : The input sequence selected from the first stage to i_th stage. Also we use I(i) to mean the input selected at stage i.
- S(i) : S(i) means the set of states which we did not successfully distinguish from s0 till the current stage i-1. Initially, S(1) is the set of all states except s0. If we have generated the complete identifying sequence and its length is j, then S(j+1) should be empty.
- state(s, i) : state(s,i) designates the state after executing i steps of transitions by setting the input sequence I(1..i) to state s. That is, state(s,i) is used to trace the state changes.
- context(s, i) : context(s,i) designates the value of context variables after executing i steps of transitions by setting the input sequence I(1..i) to state s.
- output(s, i) : same as context(s, i), but designates the value of output variables.

A state in the search is represented by (1) the partial input sequence generated so far, I(1..i), (2) the set of states which we should still distinguish s0 from S(i). In the beginning, we aim at distinguishing s0 from all other states, (i.e., S(1) includes all the states except s0). At each step of the search, say stage i, we select an input, I(i), and calculate the outputs for s0 and all s's in S(i). We remove the s which shows a different output from that of s0. For calculations, we need to keep track of the changes of state and variable values, both context and output. This is done by state(s, i) , context(s, i) and output(s, i).

One obvious question about the algorithm is the selection criteria for I(i). The wrong selection of I(i) value will not lead to a generation of an identifying sequence, even though there exists one. That is checked by the function BackTrack. BackTrack checks if a loop has been formed in the history of the search process so far without reducing the size of S(i). If so, we backtrack and choose a different value of I(i). It will assure that the algorithm generates an identifying sequence if one exists and reports the non-existence of the identifying sequence, when there is none, assuming the finite domain of variables.

The selection of I(i) will also affect the length of the sequence if there exist more than one. It is difficult to have a selection criteria which will guarantee the shortest sequence without knowing all the correct identifying sequences *a priori*. To get the minimal length identifying sequence, we may use a brute force method. We search if there is an identifying sequence of length 1. If not, we search one of length 2, and so on.

A local optimization criteria or a greedy algorithm approach can be used for the selection of I(i). For example, we can use the following method. For each possible value of I(i), we assign a number to designate the amount of reduction of size of S(i), i.e., |S(i)| - |S(i+1)|, and choose the value with the greatest number. In some cases, we cannot reduce the size of S(i) with whatever

Algorithm **Identifying_Sequence(s0, (CV,O))**

S(1) = all states except s0
context(s0, 0) = CV, output(s0, 0) = O
context(s, 0) = CV, output(s, 0) = O for all s in S(1).

At stage i, do the following.
At each calculation, refer to the current variable states stored in (context(s, i-1), output(s, i-1)).

1. Select a values for I(i) which enables one of the transitions defined from state(s0, i)
   If there are no more such values, go back to stage i-1.
2. Calculate output(s, i) for all s in S(i) and output(s0, i).
3. Form S(i+1) by removing from S(i) all s, such that output(s, i) is not equal to output(s0, i).
4. Calculate context(s, i) for all s in S(i+1) and context(s0, i).
5. If S(i+1) is empty, return I(1,i)
   else if BackTrack(i), backtrack to step 1 and choose a different value of I(i)
   else go to stage i+1.

BackTrack(i)

return true if there exists a j, $1 \leq j < i$, such that

1. S(i+1) = S(j)
2. state(s0, i) = state(s0, j),
   context(s0, i) = context(s0, j), and
   output(s0, i) = output(s0, j)
3. for all s in S(i+1), there is a s' in S(j), such that context(s, i) = context(s', j) and output(s, i) = output(s', j)

Figure 1: Algorithm to generate an *identifying sequence* of a state

value of I(i). This situation may continue for some number of stages. In this case, we may restrict the maximum number of stages where the procedure goes on without reducing the size of S(i). If we confront this maximum number, we backtrack and choose a different input value. Since this just optimizes the local selection of I(i), it does not guarantee the minimal sequence.

So far, we discussed the general idea of getting an Identifying Sequence. We can easily notice that the process of getting an Identifying Sequence is computationally costly, since it involves tracing all the histories of state and variable values as well as having to consider all possible input values.

6 Convergence of transitions

One of the important issues in the research of conformance testing is the reduction of the length of the test sequence. The reduction can be achieved by using overlap in a test sequence. [24] proposed some approaches to utilize overlap among test subsequences. [9] proposed a technique to achieve optimality under certain conditions. The main idea they used is "if there are no converging transitions, then any path of edges followed by a UIO sequence for the tail state of the path tests each transition in the path."

In an EFSM, the same idea can be used to improve the length of test sequence. But the concept of converging transitions should be defined in a different way from that of FSM's, since each transition is associated with an enabling predicate and an action instead of only an input output pair. Different from the case of FSM's, it is not sufficient to consider only the transition diagram to identify converging transitions in EFSM. We formulate a condition under which a transition does not show a converging behavior and define a converging transition with it.

Definition 6 *A* Non-Converging Predicate *of a transition t, NC(t), is a predicate of (I, CV,O) which, if satisfied by (I_0, CV_0, O_0), enables transition t, but there is no transition t', such that Tail(t') = Tail(t), which is enabled with the same variable values, (I_0, CV_0, O_0), and shows the same output.*

We formulate the non-converging predicate of a transition t as follows.

$$NC(t) = \bigwedge_{Tail(t')=Tail(t)} \{P(t') \Longrightarrow (\widetilde{Ao(t)} \neq \widetilde{Ao(t')})\} \bigwedge P(t) \tag{2}$$

Definition 7 *A converging transition is a transition whose non-converging predicate is unsatisfiable.*

Convergence of a transition depends on the variable values, (I, CV, O), as well as the flow of transitions. Given (I, CV, O) which satisfies a non-converging predicate of t, setting I as the input value with (CV, O) as the variable state will uniquely lead to the tail state, a *non-converging behavior*. A converging transition is a transition which cannot show this non-converging behavior with any variable values.

There are different classes of non-converging behaviors. The simplest class will be when the tail state of a transition does not have any other incoming transitions. This case can be identified directly from the transition diagram without evaluating the non-converging predicates (or equivalently, the non-converging predicate is just the enabling predicate). We may call this *structural non-convergence.* The second class is when some transitions share the same tail state, but the sets of variable values which enable each transition have some non-overlapping values. We may call this *algebraic non-convergency.* Taking one of these values will enable only one transition. In this case, we don't have to consider the output behavior of each transition to identify the non-convergency. The non-converging predicate in this case is P(t) $\bigwedge \neg(\bigvee_{Tail(t')=Tail(t)} P(t'))$. In the last and most general class, we should consider both the enabling predicates and output behaviors as specified in the definition above.

If we want to test if each transition ended in the correct tail state in our test procedure, we can use the concept of non-converging predicate to improve the length of test sequence as in [9]. Different classes of non-convergence behaviors will require different costs to identify, as explained above. We may use different classes of non-converging behavior in our test process according to what costs are considered affordable.

7 Conclusion and future research

In this paper, we discussed some problems in test sequence generation for a protocol module specified as an EFSM. We discussed the problems of stability and executability of transitions. We also defined new types of identifying sequences and converging transitions for EFSM's.

The EFSM model has better expressive power than FSM's. But the test sequence generation procedure is much more complicated. This complication primarily comes from two facts. First, the test data selection process should go through the evaluation of the enabling predicate, which is in a first order predicate form. This is a well known semi-decidable problem. Second, transitions interact through variables. This is hard to handle when this interaction occurs through the unobservable and uncontrollable context variables.

Some approximations may give useful and practical results. But these approximations should be justified in the frame of a general characterization of the model. We hope our presentation can be used as the basis for such an approximation effort.

A possible approach to the research may be to classify the specified protocol modules according to the complexity of predicates or interactions between transitions through variables. With this classification, we can start the study of the problem from the simplest class. For example, the simplest class, when classified by the complexity of context variables, is the protocol modules with no context variables, next those with some but only of finite domain, etc.

A different approach will be to transform the EFSM specification to FSM specification in the process of test sequence generation. We can combine the state and variable state, (CV, O), and make it an explicit state of a FSM. We may classify the specified EFSMs in this approach, too. The simplest case is when the domains of both I and (CV, O) are finite. In this case, we can get an FSM easily. If the domain of (CV, O) is finite but the domain of I is infinite, the result will be a machine with a finite number of states but possibly with an infinite number of transitions. However, in this case, we may get the finite partition of input space and form the FSM with these partitions rather than input values. A more general case is when the domain of (CV, O) is infinite, which will result in an infinite number of states. This case will be hard to characterize in a general way. We may be able to find and utilize some specific patterns, like finite partitioning of an infinite number of states or the looping structure if one exists.

References

[1] G. V. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo, "Fault Models in Testing," *Protocol Test Systems* IV, pp.17-31, North Holland, 1992.

[2] S. Budkowski and P. Dembinski, "An introduction to Estelle: A specification language for distributed systems," *Comput. Networks and ISDN Syst.*, vol. 14, no.1, pp.3-23, 1987.

[3] T. Chow, "Testing Software Design Modeled by Finite State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4, pp.178-187, March 1978

[4] L. A. Clarke and D. J. Richardson, "Symbolic evaluation methods for program analysis," in *Program Flow Analysis,* S. S. Muchnick and N. D. Jones, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[5] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, "An extended overview of the Mothra software testing environment," *Proc. 2nd Workshop on Software Testing, Verification, and Analysis* (Banff, AB, Can.), July 1988. Los Alamitos, CA: IEEE Computer Soc., pp.142-151, 1988.

[6] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, vol.17, No. 9, pp.900-910, September 1991.

[7] G. Gonenc, "A method for the design of fault detection experiments," *IEEE Transactions on Computers*, vol. C-19, pp.551-558, June 1970.

[8] Raymond E. Miller and G. M. Lundy, "Testing Protocol Implementations Based on a Formal Specification", *Protocol Test Systems*III, North Holland, pp.289-304, 1991.

[9] Raymond E. Miller and Sanjoy Paul, "On the Generation of Minimal Length Conformance Tests for Communication Protocols", *IEEE/ACM Trans. on Networking*, Vol. 1, No. 1, Feb. 1993.

[10] Raymond E. Miller and Sanjoy Paul, "On Generating Test Sequences for Combined Control and Data Flow for Conformance Testing of Communication Protocols," *Protocol Specification, Testing, and Verification XII*, June 1992, pp 13-27.

[11] Raymond E. Miller and Sanjoy Paul, "Structural Analysis of a Protocol Specifications and Generation of a Maximal Fault Coverage Conformance Test Sequences," *IEEE/ACM Trans. on Networking*, Vol. 2, No. 5, October, 1994, pp 457-470.

[12] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering,* vol. SE-11, No. 4, April 1985.

[13] D. J. Richardson and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification," *IEEE Transactions on Software Engineering,* vol. SE-11, No. 12, pp.1477-1490, December 1985.

[14] K. Sabnani and A. Dahbura,"A Protocol Test Generation Procedure and its Fault Coverage", *Computer Networks and ISDN System 15*, pp.285-297, 1988.

[15] B. Sarikaya and G.V. Bochmann, "Obtaining normal form specifications for protocols," in *Proc. COMNET'85*, Budapest, Hungary, pp.6.113-6.149, 1985.

[16] B. Sarikaya, G. V. Bochmann and E. Cerny, "A Test Design Methodology for Protocol Testing," *IEEE Transactions on Software Engineering,* vol. SE-13, no. 5, pp. 518-539, May 1987.

[17] Hasan Ural and Bo Yang, "A Test Sequence Selection Method for Protocol Testing," *IEEE Transactions on Communications,* vol. 39, No.4, pp.514-523, April 1991.

[18] M. R. Woodward, M. A. Hennell, and D. Hedley, "Experience with path analysis and testing of programs," *IEEE Trans. Software Eng.,* vol. SE-6, pp.278-286, May 1980.

[19] W. Chun and Paul Amer, "Testing case generation for protocols specified in Estelle", *FORTE '90,* Madrid, Nov. 1990.

[20] Samuel T. Chanson and Jinsong Zhu, "A Unified Approach to Protocol Test Sequence Generation," *Proceedings of IEEE INFOCOM '93.*

[21] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM,* vol.18, 1975, pp.453-458.

[22] R. C. Linger, H. D. Mills and B. I. Witt, *Structured Programming : Theory and Practice*, Addison-Wesley, Reading, MA, 1979, Chapter 6.

[23] H. D. Mills, "The New Math of Computer Programming", *Communications of the ACM*, vol 18, No.1, 1975, pp. 43-48.

[24] M. S. Chen, Y. Choi, A. Kershenbaum, "Approaches Utilizing Segment Overlap to Minimize Test Sequences," *Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification*

[25] Chang-Jia Wang and Ming T. Liu, "Generating Test Cases for EFSM with Given Fault Models", *IEEE INFOCOM*, 1993, pp 774-781.

[26] Chang-Jia Wang and Ming T. Liu, "Axiomatic Test Sequence Generation for Extended Finite State Machines", *Proc. 12th International Conference on Distributed Computing Systems*, pp. 252-259, June, 1992.