

RESOLVING INCONSISTENCIES IN EFSM-MODELED SPECIFICATIONS*

M. Ümit Uyar
Ali Y. Duale

Department of Electrical Engineering
The City College of the City University of New York
New York, NY 10031
[umit,duale]@ee-mail.engr.cuny.edu

ABSTRACT

The US Department of Defense requires that all digital ASIC systems used within the department's branches should be specified in VHDL (Very High Speed Integrated Circuit Hardware Description Language). VHDL specifications are typically modeled as Extended Finite-State Machines (EFSMs). Developing efficient algorithms for EFSM models to generate feasible test sequences with acceptable lengths is a challenging task partly because of the inconsistencies among the actions and the conditions. Inconsistency detection algorithms for EFSM models have been developed at the earlier stages of this study. As part of realizable test sequence generation for VHDL specifications, this paper presents algorithms for the removal of inconsistencies in EFSM models. The proposed method allows the direct application of the FSM-based test generation methods by transforming the EFSM models into equivalent FSMs while avoiding the well-known state explosion problem, where possible. The inconsistency detection and removal algorithms are planned to be applied to the communication protocols used within US Army CECOM and NATO.

Keywords: interoperability, military communication protocols, VHDL, conformance testing, EFSM, FSM.

1. INTRODUCTION

Formal specification and modeling techniques contribute towards the proper operation of an implementation before it is integrated with different components, most likely manufactured by different vendors.

Since a deterministic finite state machine (FSM) can represent the control structure of a protocol, FSMs have

been used to model communication protocols to generate test sequences. However, for complex systems such as MIL-STD 188-220 [4] and for VHDL specifications in general, a more powerful mechanism of modeling, called the *Extended Finite-State Machine* (EFSM), is used.

Although EFSMs are powerful enough to model complex systems, generating tests with acceptable lengths for EFSM models is much more difficult than for ordinary FSM models. A major problem is the existence of infeasible paths in an EFSM graph due to conflicts among its actions and conditions. It has been shown that if *inconsistencies* among conditions and actions of an EFSM are removed, the FSM-based test generation methods can be used for the EFSM model [7]. Therefore, developing efficient algorithms that convert *inconsistent* EFSMs to *consistent* EFSMs [7], without causing the state explosion problem, is needed.

An EFSM is called *consistent* if it is free of *condition-to-condition*, *action-to-condition*, and *action-to-action* inconsistencies. A condition-to-condition inconsistency in an EFSM graph occurs when a set of constraints that satisfies a condition fails to be valid for subsequent condition(s) in the same path. A variable that is updated differently in the paths leading to a node of the EFSM graph causes action-to-action and action-to-condition inconsistencies if such a variable is used in actions and conditions of reachable edges from the node.

This paper proposes a method that effectively transforms EFSMs into equivalent FSMs without creating the well-known state explosion problem, where possible. As a result of this transformation, for typical cases, the FSM-based test generation methods can be directly applied to VHDL specifications. The proposed method transforms a general EFSM to a consistent EFSM by using techniques similar to symbolic execution and graph splitting [2, 3, 5]. Communicating EFSMs (CEFSMs) are not considered in this paper.

*Prepared through collaborative participation in the Advanced Telecommunications & Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-000-2.

The rest of this paper is organized as follows: Section 2 introduces the inconsistency removal algorithms and conclusions are given in Section 3.

2. REMOVAL OF INCONSISTENCIES IN EFSM MODELS

Tests generated for an EFSM model may not be realizable in a test laboratory due to infeasible paths caused by the inconsistencies among the conditions and the actions of the EFSM graph. However, if an EFSM is *consistent* (or if its inconsistencies are removed), the FSM-based test generation methods can be directly applied to the EFSM. Detection of the inconsistencies (or lack of them) is presented in [7]. In this section algorithms for the removal of inconsistencies in the EFSM models are presented. The algorithm in Section 2.1, removes the action-to-action and action-to-condition inconsistencies. In Section 2.2, Steps 2.a through 2.e use the graph resulting from Section 2.1 to remove the condition-to-condition inconsistencies. It is assumed that all actions are linear, and all actions and conditions are homogenized (i.e., if a condition or action does not use a variable, its coefficient is 0).

A VHDL specification and its EFSM graph, given in Figures 1 and 2, are used to illustrate the inconsistency removal process. (Note that the signals X , Y , and Z are referred to as variables X , Y , and Z for simplicity.) The conditions and actions for the edges of the graph of Figure 2 are defined in Table 1.

The EFSM graph shown in Figure 2 contains all three types of inconsistencies. For example, conditions for e_3 and e_6 require that $X \leq 0$ and $X > 10$, respectively. Therefore, a path including e_3 and e_6 is infeasible. Furthermore, a negative value of Y may cause an action-to-condition inconsistency between e_4 and e_6 . Action-to-action inconsistencies exist among e_2, e_3, e_4, e_5 and e_6 . In Section 2.3 the inconsistency removal algorithms are applied to the EFSM graph of Figure 2.

2.1 Action-to-action/condition inconsistencies

For a given node s_i in the data flow graph, the cumulative effect of actions up to s_i which modify the variables are represented by a pair of matrices, called the *action update matrix pair*, A_i and \tilde{B}_i . A_i is an $n \times n$ matrix, where n is the number of variables, and \tilde{B}_i is an $n \times 1$ vector. Associated with a given node s_i , there may be multiple action update matrix pairs $A_{i,k}$ and $\tilde{B}_{i,k}$ ($k = 1, \dots, K$, where K is bounded by the number of valid paths up to s_i), each representing a different path leading to node s_i with a different set of variable modifications by the actions. For the initial node of the data

flow graph (i.e., the root), the action update matrices of A_{root} is initialized to the identity matrix and \tilde{B}_{root} elements are initialized to 0.

```

library IEEE;
entity XYZ is
    port (X, Y, inout integer;
          Z: in integer);
end XYZ;
-- architecture of the entity;
-- Note: <= is the VHDL signal assignment operator;
architecture behavior of XYZ is
begin
    process
    begin
        while (Z > 0) loop
            if (X > 0)
                then Y <= Y+10;
                else Y <= Y+11;
            end if;
            if (X > 5)
                then Y <= Y+1;
                else Y <= Y+2;
            end if;
            if ((X > 10 ) and (Y > 5))
                then Y <= Y+3;
                else null;
            end if;
            if (Z = 0 )
                then X <= 0;
                else null;
            end if;
            wait on X;
            wait on Y;
            wait on Z;
        end loop;
    end process;
end behavior;

```

Figure 1: An Example of VHDL Specification.

name	condition	action
e_0	$(Z > 0)$	null
e_1	$(Z \leq 0)$	null
e_2	$(X > 0)$	$Y := Y + 10$
e_3	$(X \leq 0)$	$Y := Y + 11$
e_4	$(X > 5)$	$Y := Y + 1$
e_5	$(X \leq 5)$	$Y := Y + 2$
e_6	$((X > 10) \text{ AND } (Y > 5))$	$Y := Y + 3$
e_7	$(X \leq 10) \text{ OR } (Y \leq 5)$	null
e_8	$(Z = 0)$	$X := 0$
e_9	$(Z \neq 0)$	null
e_{10}	(event on X, Y , and Z)	read X, Y, Z

Table 1: Edge Definitions for the EFSM of Figure 2.

Step 1.a: In breadth-first manner, starting from the initial node, get a node s_i .

Step 1.b: For each action update matrix pair associated with s_i , $A_{i,k}$ and $\tilde{B}_{i,k}$ ($k = 1, \dots, K$), and for each outgoing edge from s_i to s_j , form one or more action

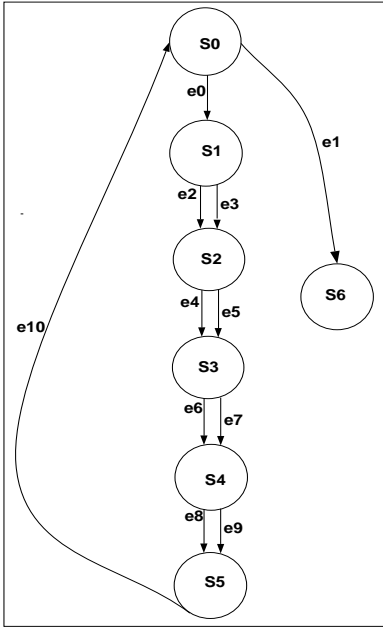


Figure 2: EFSM Model of the VHDL Specification Given in Figure 1.

update matrix pairs of $A_{j,l}$ and $\tilde{B}_{j,l}$ ($l = 1, \dots, L$, where L is the multiplication of the number of action update matrix pairs associated with s_i and the number of outgoing edges from s_i to s_j). Each matrix pair represents the accumulated action modifications above in the graph up to node s_j , including the edge from s_i to s_j .

Step 1.c: Check the conditions and actions of the outgoing edges of s_j . If these conditions and/or actions use any of the variables modified by the previous actions (i.e., by the actions in a path leading into s_i and ending with the chosen outgoing edge from s_i to s_j), then s_j and the subgraph which contains all nodes and edges that are reachable from s_j need to be split into separate parallel subgraphs. The number of subgraphs is determined by the number of different action update matrix pairs that are associated with node s_j . Each split copy of node s_j will be associated with one or more of the corresponding pairs of $A_{j,l}$ and $\tilde{B}_{j,l}$ matrices. Note that a node does not have to be split if the edges incoming to it have *read* actions for the variables that are causing the inconsistencies, which replaces the old value of a variable by an arbitrary new value.

Step 1.d: For each split node of s_j , apply the action update matrix pairs of $A_{j,k}$ and $\tilde{B}_{j,k}$ to the conditions of the edges leaving s_j .

Step 1.e: Repeat the Steps 1.b through 1.d for the next node chosen in the breadth-first manner.

To form an action update matrix pair, as required in Step 1.b, the actions on an edge from s_i to s_j are modified. An action is in the form of $v_i = \tilde{C} * \tilde{V} + d$, where

v_i is the left-hand-side variable of the action, \tilde{C} is an n -element vector of coefficients, \tilde{V} is an n -element vector of variables, and d is a scalar. The current values of the variables, before the actions of this edge are applied, are in the form of $\tilde{V} = A_i * \tilde{V} + \tilde{B}_i$, where A_i and \tilde{B}_i are the update action matrices associated with node s_i . For each one of action update matrix pairs associated with s_i , application of an action update matrix pair into this action variables means the substitution of \tilde{V} in the action: $v_i = \tilde{C}(A_i * \tilde{V} + \tilde{B}_i) + d$ which yields $v_i = \tilde{C} * A_i * \tilde{V} + \tilde{C} * \tilde{B}_i + d$. In the new action update matrix pair, the i th row of $A_{i,j}$ must be replaced by $(\tilde{C} * A_i)$, and i th element of $\tilde{B}_{i,j}$ with $(\tilde{C} * \tilde{B}_i + d)$.

Now consider the application of the action update matrices to the conditions of the edge from s_i to s_j , as needed in Step 1.d. A single condition of an edge is in the general form of: $\tilde{C} * \tilde{V}(op)d$, where op is the operator which can be $=, <, >, ! =, \dots$, etc. This condition will be modified based on the current values of the variables v_0 through v_n , which are represented by the action update matrix pair of $A_{i,j}$ and $\tilde{B}_{i,j}$. In general, the current values of the variables including all the modifications by the edges up to this node is in the form of: $\tilde{V} = A_{i,j} * \tilde{V} + \tilde{B}_{i,j}$. Substituting \tilde{V} values in the condition will result in: $\tilde{C}(A_{i,j} * \tilde{V} + \tilde{B}_{i,j})(op)d$ which will simplify as $\tilde{E} * \tilde{V}(op)f$ where $\tilde{E} = \tilde{C} * A_{i,j}$ is an n -element vector and $f = \tilde{C} * \tilde{B}_{i,j}$ is a scalar.

2.2 Condition-to-condition inconsistencies

For a given node s_i , the conditions that are accumulated in the paths leading up to s_i are represented by *condition matrix triplet*, C_i , \tilde{OP}_i , and \tilde{D}_i . C_i is a $n \times p$ matrix where n is the number of variables and p is the number of conditions that are accumulated up to node s_i . \tilde{OP}_i is a $p \times 1$ vector representing the relations of $=, <, >, ! =, \dots$, etc. The $p \times 1$ vector of \tilde{D}_i contains the scalar values of the accumulated conditions. Similar to the case of action update matrix pairs, there may be multiple condition matrix triplets, $C_{i,k}$, $\tilde{OP}_{i,k}$, and $\tilde{D}_{i,k}$ ($k = 1, \dots, K$ where K is bounded by the number of valid paths up to s_i). Each triplet represents a different set of accumulated conditions from a valid path leading up to node s_i . The condition matrix triplets for the root node are empty.

Step 2.a: In breadth-first manner, starting from the initial node of the graph from Step 1, get a node s_i .

Step 2.b: For each condition matrix triplet of node s_i , $C_{i,k}$, $\tilde{OP}_{i,k}$, and $\tilde{D}_{i,k}$, and for each outgoing edge from s_i to s_j , form a condition matrix triplet $C_{i,l}$, $\tilde{OP}_{i,l}$, and $\tilde{D}_{i,l}$ ($l = 1, \dots, L$ where L is the number of outgoing edges from s_i to s_j). The sizes for condition triplet matrices are $m \times n$ for $C_{i,l}$, $m \times 1$ for $\tilde{OP}_{i,l}$ and $\tilde{D}_{i,l}$ ($m = p + r$ where p is the number of matrix triplets for s_i , and r is

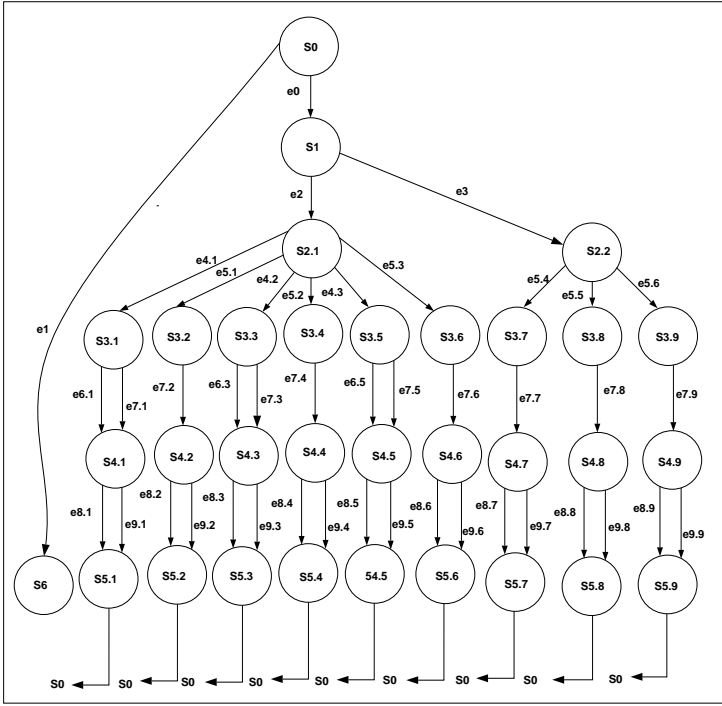


Figure 4: The EFSM Graph After All Inconsistencies Are Removed.

$$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \begin{bmatrix} > \\ > \\ \leq \end{bmatrix} \begin{bmatrix} a \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \begin{bmatrix} > \\ > \\ \leq \end{bmatrix} \begin{bmatrix} b \end{bmatrix}$$

where $a = 10$, $b = -6$ for $s_{3,1}$, $a = 10$, $b = -7$ for $s_{3,2}$, and $a = 10$, $b = -8$ for $s_{3,3}$.

The graph of Figure 3 still contains condition-to-condition inconsistencies (for example, $e_{5,1}$ and $e_{6,1}$). The algorithm of Section 3.2 is now applied to the graph of Figure 3 to remove these inconsistencies. Since the conditions of the edges leaving node s_2 and the outgoing edges of s_1 use X , node s_2 and all the nodes that are reachable from it are split by the number of condition update matrix triplets associated with s_2 , which are:

$$C_{S2,1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \tilde{O}P_{S1,1} = \begin{bmatrix} > \\ > \end{bmatrix} \quad \tilde{D}_{S1,1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$C_{S2,2} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \tilde{O}P_{S2,2} = \begin{bmatrix} > \\ \leq \end{bmatrix} \quad \tilde{D}_{S2,2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The other node split is due to the conditions of edges leaving s_3 which use variable X . For the subgraph starting with $s_{2,1}$, the nodes are split to two parallel ones since there are two different condition update matrix triplets associated with s_3 :

$$C_{S3,1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \tilde{O}P_{S3,1} = \begin{bmatrix} > \\ > \\ > \end{bmatrix} \quad \tilde{D}_{S3,1} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}$$

$$C_{S3,2} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \tilde{O}P_{S3,2} = \begin{bmatrix} > \\ > \\ \leq \end{bmatrix} \quad \tilde{D}_{S3,2} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}$$

However, for the subgraph leaving $s_{2,2}$, only one condition matrix triplet is valid:

$$C_{S3,3} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \tilde{O}P_{S3,1} = \begin{bmatrix} > \\ \leq \\ > \end{bmatrix} \quad \tilde{D}_{S3,1} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}$$

Therefore, the subgraph starting with $s_{2,2}$ is not split for the second time. As discussed before, the splits do not include the initial node s_0 due to the read actions in edge e_{10} . At this point, the infeasible edges are also removed (for example, the edges leaving $s_{2,2}$ with condition $X > 5$, which conflict with the edges leaving s_1 with condition $X \leq 0$ are removed. Figure 4 shows the final graph which is free of all inconsistencies.

3. CONCLUSIONS

This paper presents a method to generate realizable test sequences for EFSM models. Inconsistencies in EFSM models are removed to generate consistent EFSMs. Since a consistent EFSM is effectively an FSM, the FSM-based test generation methods can be directly applied to the consistent EFSM. The inconsistency detection and removal algorithms are planned to be applied to the communication protocols used within US Army CECOM and NATO.

References

- [1] B. Beizer, *Software Testing Techniques*, Thomson Computer Press, Boston, MA., 1990.
- [2] K. T. Cheng and A. S. Krishnakumar, "Automated Generation of Functional Vectors Using the Extended Finite State Machine Model," *ACM Trans. on Design Automation*, Vol 1, No. 1, pp. 57-79, Jan. 1996.
- [3] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. on Software Eng.*, Vol. SE-2, No. 3, pp. 215-221, Sep. 1976.
- [4] M. A. Fecko, M. U. Uyar, P. D. Amer, A. S. Sethi, "Using Semicontrollable Interfaces in Testing Army Communications Protocols: Application to MIL-STD 188-220B," Proc. Advanced Telecommunications/Information Distribution Research Program (ATIRP), College Park, MD., pp. 189-194, Feb. 1999.
- [5] W. E. Howden, "Symbolic Testing and Dissect Evaluation System," *IEEE Trans. on Software Eng.*, Vol. SE-2, No. 4, pp. 266-278, July 1977.
- [6] R. J. Linn and M. U. Uyar, *Conformance Testing Methodologies and Architecture for OSI Protocols*, IEEE Computer Society, Los Alamitos, CA., 1994.
- [7] M. U. Uyar and A. Y. Duale, "Modeling VHDL Specifications as Consistent EFSMs," Proc. of IEEE Military Comm. Conf. (MILCOM), pp. 740-744, Monterey, CA., Oct. 1997.