INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

FEASIBLE TEST GENERATION BY ELIMINATION OF INCONSISTENCIES IN EFSM MODELS OF COMPUTER AND COMMUNICATION SYSTEMS

By

Ali Y. Duale

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2000

UMI Number: 9986320

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Sept 15, 2000 — Date
M. Ü. UYAR — Chair of Examining Committee

Sept. 18, 2000 — Date
Executive Officer

Professor M. Ümit Uyar

Professor Taarek Saadawi

Professor Myung lee

Professor Michael Conner

Professor Ravindran Kaliappa

Professor Joseph Barba

Dr. Mariusz Fecko

Supervisory Committee

The City University of New York

Abstract

FEASIBLE TEST GENERATION BY ELIMINATION OF INCONSISTENCIES IN EFSM MODELS OF COMPUTER AND COMMUNICATION SYSTEMS

By
Ali Y. Duale

Advisor: Prof. M. Ümit Uyar

Conformance testing aims at the detection of possible discrepancies between an implementation and its specification. Furthermore, conformance testing is essential for achieving a seamless interoperability among the components of a heterogeneous system. Finite-state machines (FSMs) are used to model and generate conformance test sequences for the control structures of communication protocols. On the other hand, the extended finite-state machines (EFSMs) are used when the data portion of a communication protocol is considered.

Although methods to automatically generate conformance test sequences from the FSMs are available, the test generation from the EFSMs is among the most challenging aspects of conformance testing. If the interdependencies among the variables used in the actions and conditions of the EFSMs are not considered during the test generation, the test sequences may be unrealizable in a test laboratory.

This dissertation presents a method that enables the generation of realizable test sequences from a class of EFSMs. *Inconsistencies* among the actions and conditions of EFSM models are defined. Algorithms for the detection and elimination of these inconsistencies are developed. These algorithms eliminate inconsistencies by creating new nodes and edges for the EFSM graph.

However, these nodes and edges are created only when needed, thus avoiding unnecessary growth of the state space. For the cases where the state explosion is unavoidable, the size of the new graph is constantly monitored as the algorithms eliminate the inconsistencies.

Once inconsistencies are eliminated, realizable test sequences can be generated from the resulting *consistent* EFSMs by using the test generation methods available for the FSM models.

The proposed methodology is currently being used to solve the *conflicting timers problem* which arises when a protocol has multiple timers running concurrently. Due to the conflicting timers, a test sequence of a protocol such as the Estelle specification of the MIL-STD-188-220 may be interrupted by unexpected timeouts. Preliminary results show that generating test sequences for the MIL-STD 188-220 after eliminating the timer inconsistencies significantly improves the test coverage by including more transitions into the test sequences without timer interruptions.

Acknowledgments

Completing a Ph.D. thesis is a once in a lifetime opportunity and, at the same time, a challenging task. Without the help and support of many people, it would be unlikely that I could succeed in obtaining the Ph.D. degree. As I am writing the final words of this dissertation, I am delighted to take this opportunity to thank those who assisted me to reach this accomplishment.

I would like to express my sincere gratitude and thanks to my advisor and committee chair, Prof. M. Ümit Uyar. I was fortunate to work with him for the past four and half years in which he provided me valuable knowledge. To the very end, he continued to sacrifice his spare time including many weekends for our discussions. Without his expertise in the field, dedication, and sacrifices, this work could not be done.

I would like to thank the committee members for their inputs to this thesis. I am indebted to Prof. Saadawi for his continuous help throughout my graduate work. Special thanks go to Dr. Fecko for his constructive comments on the thesis and the efforts that he has spent on our collaborative work.

Without the scholarships and the financial support from the MAGNET (CUNY), AMP (CUNY), Design Automation Conference (IEEE/ACM), CASI (CCNY), and ATIPR (ARL) programs, I would not be able to concentrate on this research.

It is a great pleasure for me to mention that this work would not be accomplished without the patience and support of my wife, Kaltun, my three children (Luqman, Zalma, and Ayan), and my parents. For many years, my uncle, Mahmoud, has been both an inspiring and a motivating

source for me.

Last, but not the least, I thank everyone who contributed to this thesis including those who taught me formally and/or informally, the authors whose names appear in the Bibliography, and every fellow student and friend who reviewed portions of this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Communication and computer systems are built by using complex software and hardware components. Undetected errors in these systems may lead to unexpected behavior. In such an environment, systems must be rigorously tested before they are deployed as services.

In communication and computer systems, detecting errors is not only crucial for the proper operation of these systems, but also a challenging issue. Recently, substantial effort has been directed towards the test generation for communication protocols. Due to the complexity of these protocols, manual test sequences generation is often time consuming and inefficient. Methods that automatically generate tests for the communication protocols and computer systems specifications are needed.

When a communication protocol is implemented, several issues related to the behavior of the implementation are considered. The capability of the implementation to integrate with other

component(s) is checked through the *interoperability testing*. The efficiency of a product is determined by employing the *performance testing*. The ability of an implementation to recover from errors is determined by using the *robustness testing*. *Conformance testing*, which is the most formidable among the different types of testing, investigates whether an implementation represents its specification. In other words, in conformance testing discrepancies between a specification and its implementation are detected.

Starting in 1981, the International Federation for Information Processing (IFIP) held a series of workshops and conferences on the topic of protocol specification, testing, and verification. In 1983, the International Organization for Standardization (ISO) initiated the standardization of testing communication protocols. The ISO Open System Interconnection (OSI) Basic Reference Model provides guidelines for such standardization [53].

Since testing procedures should not obligate manufacturers to disclose the internal structures of their products, the components are typically represented as black-boxes [10, 25, 53]. However, manufacturers are required to provide detailed descriptions of the products' external behavior. According to the Reference Model for ISO (ISO7498), the external behavior of an open system represents its standard behavior [10].

The black-box representation hinders the testing process. The challenge for the conformance tester is to find out if an implementation under test (IUT) is equivalent to its specification by analyzing only the IUT's external behavior without the knowledge of the IUT's internal implementation details. Based upon the external behavior of an implementation at its interfaces, a conformance tester reaches a verdict of whether an IUT conforms to its specification.

An ideal way of testing a component would be to conduct *exhaustive testing* where all possible combination of inputs are applied to an IUT. However, due to the limited resources and the time available for testers, exhaustive testing is impossible even for simple protocols. At the other extreme, is to employ *random testing* which only considers a subset of inputs. Obviously, this type of testing will not be efficient or reliable since it may fail to test critical portions of the IUT. Therefore, test sequences that cover all specified behavior of the IUT without performing exhaustive testing are needed.

Ambiguities in natural languages typically cause interpretation problems for manufacturers. The main cause of errors is due to mistakes that occur during the translation of specifications. Different vendors may produce different products from the same specification if the specification is given in an informal language such as English. Problems arising from interpretations can be avoided by using formal models and/or languages in protocol specifications.

For the ease of the test generation methods, a model for the IUT is developed from the specification. FSMs and EFSMs are among the most popular models used to represent communication protocols.

Conformance test generation for specifications modeled as FSMs has been an active area of research. There are techniques that automatically generate test sequences for protocols specified as deterministic FSMs. However, test generation for the EFSM-modeled specifications remains as an open research problem [13, 14, 51, 50, 58, 59, 66, 67, 71, 72, 75, 76]. The difficulty of automating test generation process for the EFSM models stems from the fact that choosing only feasible paths is undecidable for an EFSM [66]. The existence of the so-called *context variables*

in the EFSM models makes the automation of test generation for EFSM models a challenging task. In general, traversing an edge e_i of a directed EFSM graph depends on the values that the condition variables of e_i assume in the paths leading to e_i.

Realizing a sequence of transitions of an EFSM may not be feasible simply because one transition may require a condition which conflicts with the condition of another transition in the same path. On the other hand, all paths of an FSM graph are valid since the edge conditions do not impose restrictions over the paths. Therefore, efficient test generation algorithms that avoid edges with conflicting conditions in the same test sequence (i.e., generating only feasible test sequences) are needed for the specifications modeled as EFSMs.

Although in general, generating feasible test sequences from the EFSMs is an open research problem, this dissertation presents a method which enables the generation of only feasible test sequences from a class of EFSM models with the following properties:

- The specification consists of a single process and thus there are no communicating EFSMs.
- If the specification contains functions or procedures, the functionality of each function or procedure can be described within the process with a simple transformation.
- Pointers, recursive functions, and syntactically endless loops are assumed not to be present in the specification.
- All conditions and actions are linear.

The proposed methodology of test generation is depicted in Figure 1.1. The methodology aims the generation of feasible test sequences from a class of EFSMs by utilizing the exiting FSM-

based techniques. The method consists of the detection and elimination of *inconsistencies* in EFSM models. The final resulting EFSM graph has the property that all its paths are feasible. Therefore, the FSM-based test generation techniques can be used to generate the test sequences.

In this thesis, inconsistencies in EFSM models are defined. Algorithms for the detection and elimination of such inconsistencies are developed. *Symbolic execution*, used in software engineering, together with linear programming algorithms are utilized to detect inconsistencies. Graph splitting algorithms are introduced to handle the elimination of inconsistencies in EFSM models.

The rest of this dissertation is organized as follows. Background information is given in Chapter 2. The proposed methodology is outlined in Chapter 3. To illustrate the process of eliminating inconsistencies, various EFSM models with inconsistencies are analyzed in Chapter 4. Case studies are presented in Chapter 5. Concluding remarks and summary of results are presented in Chapter 6.

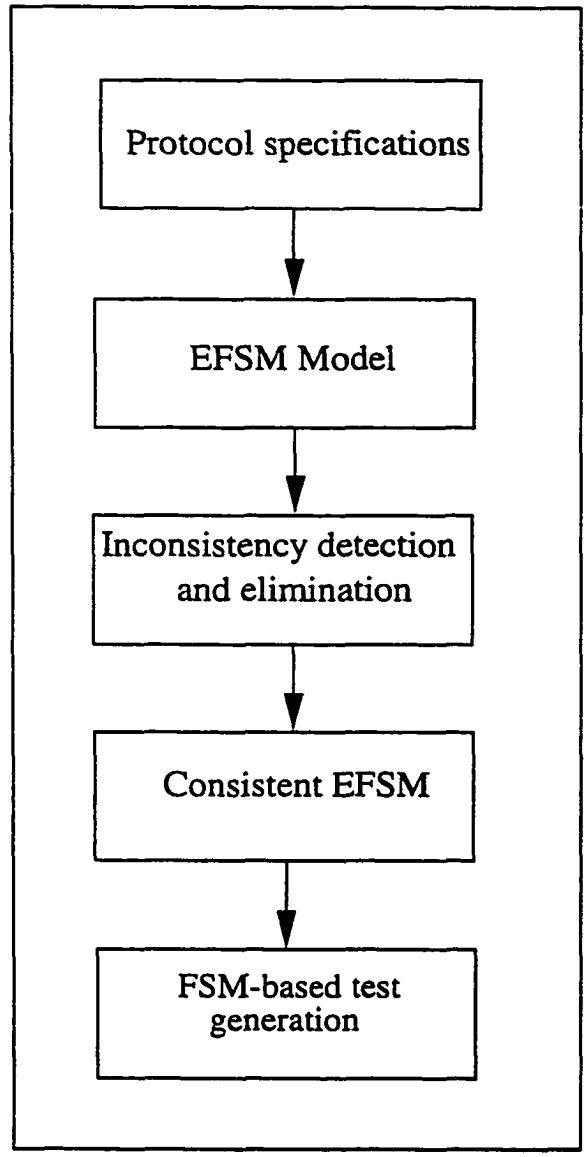

Figure 1.1: Proposed test generation method.

Chapter 2

Background Information

Formal specification and modeling techniques contribute towards the proper operation of an implementation before it is integrated with different components of a system, most likely manufactured by different vendors. To ensure the interoperability among the components of a heterogeneous system, each component must behave according to its specification [6]. Conformance testing is aimed to check whether an implementation under test (IUT) correctly represents its specification.

To give manufacturers the freedom of concealing the internal structures of their products, the IUT is presented as a block box. However, a precise model of the external behavior of the IUT must be available. The FSMs and the EFSMs which are used to model the external behavior of protocols can be represented as directed graphs.

This chapter presents a general descriptions of graphs, which are essential for modeling multi-

farious real-world problems, and how they can be used to generate effective tests. Several graph algorithms which are used in the following chapters of this thesis are also discussed. Methods for generating tests from the graph representation of protocols are outlined.

2.1 Graphs

Graphs play an important role in diverse areas including social sciences, linguistics, physical sciences, engineering, computer networks, and many computer science fields such as switching theory, artificial intelligence, formal languages, computer graphics, operating systems, and compilers [36]. When a graph is used to describe a problem, usually the essence of the problem is depicted while the details are left out. A graph is defined as follows:

Definition: 1 Graph, $G(V, E)$, *is a structure defined by two nonempty sets,* V *and* E. *The elements of* V *and* E *are called nodes (or vertices) and edges, respectively.*

Despite the fact that the number of elements of V and E can be infinite, all graphs discussed in this study have finite numbers of nodes and edges.

A pair of ordered nodes (v_i, v_j) of a *directed* graph, also called *a digraph*, describes an edge $e_k \in E$ which leaves v_i and terminates on v_j. The nodes v_i and v_j are called the *head* and the *tail nodes* of e_k, respectively. For simplicity, throughout this thesis, the directed graphs will referred to as graphs. As a convention, the nodes and edges of a graph are illustrated as circles and arrows, respectively. Figure 2.1 portrays an example of a graph. A node v_0 of G is identified as the *initial node* to represent the starting node of the graph.

If V, E, V' and E' are the sets of nodes and edges of the graphs G and G', respectively, G' is said to be a *subgraph* of G (expressed as $G' \subseteq G$) if:

$$V' \subseteq V \text{ and } E' \subseteq E : \forall\, e'_i \in E'\ tail(e'_i),\ head(e'_i) \in V' \tag{2.1}$$

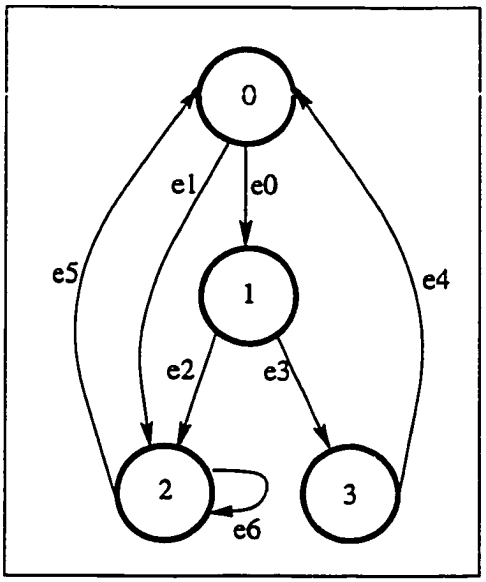

Figure 2.1: Directed graph.

Although the mapping between the nodes and the edges of a graph may be one-to-one (i.e., there is only one edge between two given pairs of nodes), graphs with *parallel edges* between certain pairs of nodes, known as *multigraphs*, are more common. Henceforth, the word *graph* also refers to a *directed multigraph*.

Node $v_i \in V$ is adjacent to node v_j if there is an edge $e_k \in E$ from v_i to v_j. For a graph of n nodes (i.e., $|V| = n$), an $n \times n$ matrix, called the *adjacency matrix* A, depicts the number of

edges from v_i to v_j. The elements of A are defined as:

$$A_{ij} = \begin{cases} m & \text{if there are } m \geq 1 \text{ edges from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

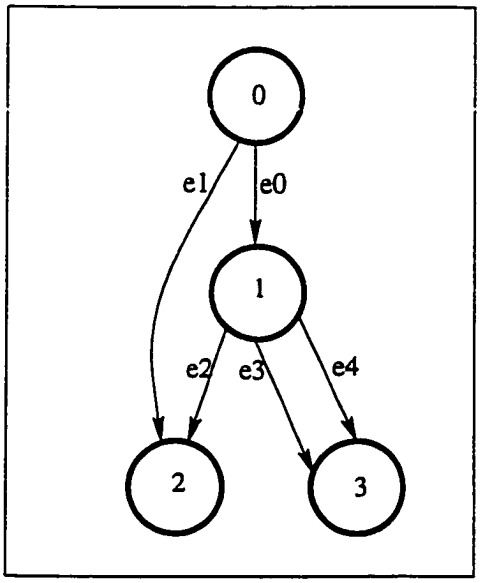

Figure 2.2: Loop-free graph

As an example, the adjacency matrix for the directed graph of Figure 2.1 is:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

A *path* in $G(V, E)$ is a non-null sequence of consecutive edges: $(v_1, v_2) \cdot (v_2, v_3) \cdot \ldots \cdot (v_{r-1}, v_r)$ where $\cdot$ denotes the concatenation of edges. A node v_i is said to be reachable from another node v_j if there is at least one path from v_j to v_i. A path in which all nodes are distinct

except the starting and ending nodes of the path (i.e., v_1 and v_r) forms a *loop*. Throughout this dissertation, the term *loop* refers to the *while/for* constructs of high-level formal specification languages such as VHDL, rather than the ordinary loops. If v_1 is the loop entry node, the nodes v_2 through v_{r-1} form the *loop body*. A *tour* of a graph is a sequence of edges that starts and ends at the same node. An edge $e_i \in E$ whose *head* and *tail* nodes are the same is called a *self-loop*. A *syntactically endless loop* is a loop defined by the path $(v_1, v_2) \cdot (v_2, v_3) \cdot \ldots \cdot (v_m, v_k)$, where $v_1 = v_k$, such that none of the conditional transfer statements in the loop leads to a node that is not part of the loop body. A graph with no loops, such as the graph shown in Figure 2.2, is called a *loop-free* or *acyclic* graph.

A *complete path* in G is a path whose starting node is the initial node and whose ending node is an exit node. A path is called a *simple path* if all nodes of the path are distinct.

In a loop-free graph, all paths are simple. Whether a graph is loop-free can easily be determined from the graph's adjacency matrix. The adjacency matrix is scanned to determine if there is a row whose entries are all zeros, (i.e., a node with no outgoing edges) [43]. By repeatedly deleting the rows whose entries are all zeros and the corresponding columns the adjacency matrix shrinks to zero for loop-free graphs [43].

A graph G is *strongly-connected* if each node of the graph is accessible from all other nodes of the graph. For all graphs considered in this study, a node is assumed to be identified as the *initial node* from which all other nodes are reachable, and vice versa.

The *path matrix* $P(n \times n)$ shows the number of different simple paths between the pairs of nodes of the graph and can be computed from A as follows [36]:

$$P = I + A + A^2 + A^3 + \ldots + A^n \tag{2.3}$$

where I is the identity matrix and n is the number of nodes. The entries of the k^{th} power of A (where $k \leq n$) indicate the number of different paths of length k between the pairs of nodes.

Often one is only interested in knowing whether a node is reachable from another node. In that case, the entries of the path matrix are calculated in a simpler manner than using (2.3) by employing the Warshall algorithm, shown in Figure 2.3, which uses Boolean matrix operations [36]. The adjacency matrix A is slightly modified such that each entry of A is either one or zero, depending on whether there is at least an edge from v_i to v_j.

The entries of the path matrix obtained with the Boolean matrix operations, thus, becomes:

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Each entry of the path matrix for the graph given in Figure 2.1 must have a value of one since each node is reachable from all other nodes whereas the path matrix of the graph of Figure 2.2 contains entries whose values are zeros, which signifies that certain nodes cannot be accessed from one another:

```
Warshall Algorithm:
n = number of nodes
for (all indices i < n ){
      for (all indices j < n){
            P_ij = A_ij;
      }
}
for (all indices k < n ){
      for (all indices i < n){
            for (all indices j < n){
                  P_ij = P_ij OR (P_ik AND P_kj);
            }
      }
}
```

Figure 2.3: Path computation algorithm.

$$P = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The number of incoming and outgoing edges of a node v_i are called the *indegree* $d_{in}(v_i)$ and the *outdegree* $d_{out}(v_i)$, respectively. If, for each node v_i, $d_{in}(v_i) = d_{out}(v_i)$ the graph is said to be *symmetric.*

2.1.1 Finite State Machines

FSM is arguably the most popular tool used for modeling communication protocols. Extensive research has been done in developing efficient test methodologies for complex communication

protocols modeled as FSMs [13, 14, 16, 34, 50, 51, 58, 59, 62, 66, 67, 71, 72, 75, 76]. Let us first define the FSM [53] model:

Definition: 2 *An* **FSM** *is a theoretical automaton which rests in a stable condition, called state, until an external stimulus is applied. Upon receiving the stimulus, the FSM generates an output (including null) and moves from its current state to a next state (which may be the same as the current state).*

The next state of an FSM is only a function of the current state and the current input. Formally an FSM is defined as:

$FSM = < S, I, O, \delta, \lambda >$ where

- S is a finite set of states.
- I is a finite set of inputs.
- O is a finite set of outputs.
- δ is a state transition function such that δ: $S \times I \rightarrow S$.
- λ is an output function such that λ: $S \times I \rightarrow O$.

The sets of inputs and outputs specified for a state of an FSM are called the *permissible input set* I and the *permissible output set* O of the state, respectively.

An FSM is called *deterministic* if for each input, $i \in I$, there is at the most one transition defined at each state of the FSM. A graph can be used to represent a *deterministic* FSM. The sets of nodes and edges of a graph $G(V, E)$ can represent the states and transitions of an FSM, respectively. An edge of an FSM graph G is denoted as:

$$(v_i, v_j;\ input_k/output_l) \tag{2.5}$$

where v_i, v_j, and $input_k/output_l$ are the *head* node, *tail* node, and the specified input/output pair for the edge, respectively. In all the figures throughout this dissertation, the conditions and actions of an edge are inclosed in parentheses "$(\cdots)$" and curly braces "$\{\cdots\}$", respectively. For some graphs, the conditions of an edge may also appear within two angled brackets "$\langle\cdots\rangle$". A positive integer C can be associated with an edge $(v_i, v_j;\ input_k/output_l)$ to denote the *cost* of the edge, which represents the time or the difficulty to realize the edge in a laboratory.

In general, a deterministic FSM can be used to model the control structure of a protocol [16]. However, for modeling the data portion of a protocol, a more powerful mechanism of modeling called the *extended-FSM* (EFSM) is used.

Definition: 3 *An* **EFSM** *is an FSM in which the next state and the generated output depend on its current state, the applied input, a set of variables, and Boolean expressions referencing the variables and input.*

The formal definition of an EFSM is:

$EFSM = < S, I, O, X, P, B, T >$ where

- S, I, and O are defined as in an FSM.
- X is a set of variables.
- P is a set of parameters associated with the input set (the set of variables and the parameters serve as a memory, M).
- B is a set of Boolean expressions which reference to a memory.

- $T = ST \cup IT$ is a set of transitions that consists of spontaneous transitions (ST) and input transitions (IT). Spontaneous transitions may make reference to a memory location and require no input.

Throughout the dissertation, the corresponding EFSM model to the *for* loop constructs of the high-level formal specification languages is shown in Figure 2.4.

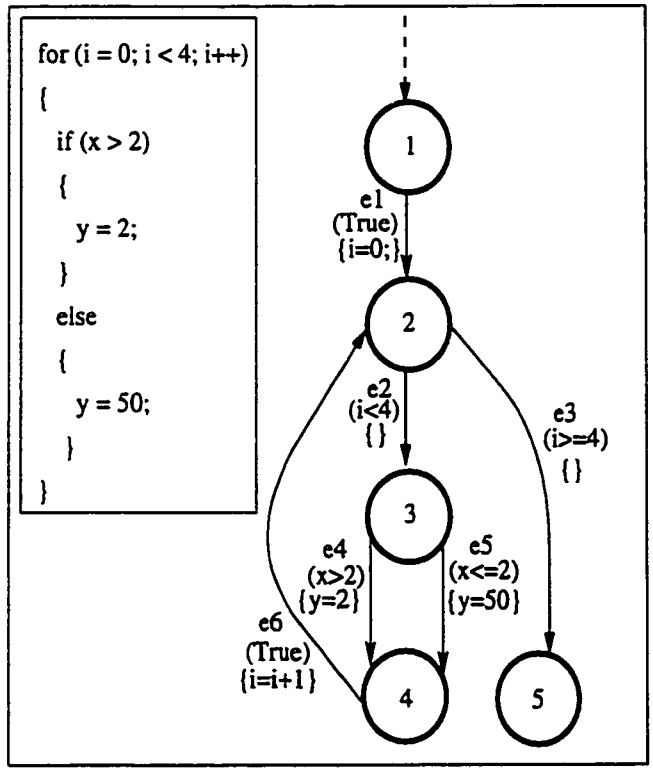

Figure 2.4: EFSM model of a *for* loop.

Because of the internal variables and the associated memory, the generation of test sequences automatically from the EFSMs is a challenging task. A test sequence generated without taking into account the effects of the internal variables on the conditions and actions of the EFSM transitions may become unrealizable in a test laboratory. The goal of this study is to propose a

method to generate feasible test sequences by analyzing the interdependencies among the actions and conditions of the EFSM models [27, 80, 81].

2.1.2 Definitions

Let us introduce the following terms used throughout this dissertation:

- $V_{v_i}^{reachable} \subseteq V$: set of nodes reachable from v_i without touching v_0, the initial node.
- $E_{v_i}^{reachable} \subseteq E$: set of outgoing edges of $V_{v_i}^{reachable}$.
- $E_{v_i}^{out} \subseteq E$: set of outgoing edges of v_i.
- $E_{e_i \rightsquigarrow e_j}^{out} \subseteq E$: set of edges in the paths between $tail(e_i)$ and $head(e_j)$.
- $E_{v_i}^{in} \subseteq E$: set of incoming edges of v_i.
- $E_{v_i}^{in(T)} \subseteq E_{v_i}^{in}$: set of incoming edges of v_i that have already been traversed.
- $E_{v_i}^{in(NT)} = E_{v_i}^{in} - E_{v_i}^{in(T)}$: set of incoming edges of v_i that have not been traversed.
- $Loop_{v_i} \in V$: node v_i is a loop entry/exit node.
- $V^{Loop_{v_i}} \subset V$: set of nodes that constitute the loop body whose entry/exit node is v_i.
- $E_{Loop_{v_i}}^{exit} \subset E$: set of outgoing edges of $Loop_{v_i}$ whose *tail* nodes are in the set $V_{Loop_{v_i}}^{reachable} - V^{Loop_{v_i}}$.
- $VAR = \{var_1, var_2, \cdots, var_m\}$: set of variables used in the edge conditions and actions of $G(V, E)$.
- $VAR_{v_i}^{con-used} \subseteq VAR$: set of variables used in the conditions of outgoing edges of $V_{v_i}^{reachable} \cup \{v_i\}$.

- $VAR_{v_i}^{dif-modified} \subseteq VAR$: set of variables modified differently in the paths leading to v_i.
- $v_{i(s)} \in V$: node v_i after it is split s number of times ($0 \leq s \leq s^*$).
- $e_{i(s)} \in E$: edge e_i after it is split s number of times ($0 \leq s \leq s^*$).
- $$T^{v_i} = \begin{cases} 1 & \text{if } v_i \text{ is touched} \\ 0 & \text{otherwise} \end{cases}$$
- $$\text{Cons}(e_k, e_r, v_i) = \begin{cases} 1 \text{ (i.e., consistent)}, & \text{if traversing } e_k \text{ or } e_r, \text{ where} \\ & tail(e_k) = tail(e_r), \text{ does not} \\ & \text{make } e_j \in E_{v_i}^{out} \text{ infeasible} \\ 0 \text{ (i.e., inconsistent)}, & \text{otherwise} \end{cases}$$

For simplicity, the notations $v_{i(s)}$ and v_i as well as $e_{i(s)}$ and e_i will be used interchangeably where appropriate.

2.1.3 Graph Traversal Algorithms

Typically, many problems require systematically scanning a graph. For example, it may be of interest to methodically discover all nodes that are reachable from a given node, v_i. The graph is then searched without any pre-plan (i.e., decisions are made along the search) while assuring the termination of the process as soon as all nodes reachable from v_i are visited. The depth-first (DF) and breadth-first (BF) graph traversal algorithms [21, 29]. terminate once the graph is completely traversed.

In this study, the DF and a modified breadth-first (MBF) graph traversal techniques are utilized

for the detection and elimination of *inconsistencies*. A brief discusion on the DF and BF graph traversal techniques and related algorithms are presented below.

2.1.3.1 Depth-first Graph Traversal

The DF graph traversal technique shown by the pseudo-code in Figure 2.5 is used for solving numerous graph problems. This technique has been known since the 19th Century when it was used for threading mazes. The strategy used in this technique is to search in the graph as deep as possible. Among the many applications of the DF search technique are finding the strongly-connected components of a graph and topologically sorting an acyclic (loop-free) graph [21]. In this dissertation, DF search is used for the analysis of the interdependencies among the condition variables of the EFSM models.

Let us briefly provide a description of the DF graph traversal of a graph. Suppose that the initial node v_0 is selected as the starting node.

One of the outgoing edges of v_0, $e_k = (v_0, v_i)$, is traversed to visit v_i, where v_i is one of the adjacent nodes of v_0. If there is at least one outgoing edge of v_0 which is not traversed, v_0 is placed in a *stack* and v_i becomes the new starting node. The traversal continues in this manner until a node without outgoing edges or a node which has been already visited is reached. In that case, the most recently placed node in the stack is selected to be the new starting node. The process continues until all nodes accessible from the initial node and associated edges are visited.

The following example illustrates the main steps of the DF graph traversal algorithm.

Depth-first Graph Traversal:

```
Let T^{v_i} = 1 if the node v_i is visited and 0 otherwise;
∀ v_i ∈ V, T^{v_i} = 0;
DF_DONE = False;
while (NOT DF_DONE){
    if ((T^{v_i} == 0) AND (E^{out}_{v_i} ≠ ∅){
        T^{v_i} = 1;
        traverse e_k ∈ E^{out}_{v_i}; E^{out}_{v_i} = (E^{out}_{v_i} − e_k);
        if (E^{out}_{v_i} ≠ ∅){
            PUSH (Stack, v_i);
        }
        if (T^{tail(e_k)} == 1){
            v_i = BACKTRACK();
        }
        else{
            v_i = tail(e_k);
        }
    }
    else if ((T^{v_i} == 0) AND (E^{out}_{v_i} == ∅)){
        T^{v_i} = 1;
        v_i = BACKTRACK();
    }
    else if ((T^{v_i} == 1) AND (E^{out}_{v_i} ≠ ∅)){
        traverse e_k ∈ E^{out}_{v_i}; E^{out}_{v_i} = (E^{out}_{v_i} − e_k)
        if (E^{out}_{v_i} ≠ ∅){
            PUSH (Stack, v_i);
        }
        if (T^{tail(e_k)} == 1){
            v_i = BACKTRACK();
        }
        else{
            v_i = tail(e_k);
        }
}
```

Figure 2.5: Depth-first graph traversal algorithm.

Example 1: Suppose that the graph of Figure 2.1 is to be traversed in a DF manner. Starting from v_0, let e_0 be the first edge to be traversed. The ending node of e_0 which is v_1 is not visited yet and becomes the new starting node. Since there is an outgoing edge of v_0 which has not been traversed, v_0 is put at the top of the stack. The edge $e_2 \in E_{v_1}^{out}$, where $tail(e_2) = v_2$, is traversed and v_1 is put in the stack since one of its outgoing edges is not traversed yet. After traversing e_5, v_2 is also put in the stack. At this point the stack contains the nodes of v_0, v_1, and v_2.

The ending node of e_5 has been already visited and according to the DF graph traversal algorithm, the backtracking function *BACKTRACK()* is invoked. The most recently placed node in the stack (i.e., v_2) is popped and e_6 is traversed. Since $tail(e_6) = v_2$ is a touched, node, v_1 is retrieved from the stack. The only non-traversed outgoing edge of v_1 (i.e., e_3) is then traversed. Since all outgoing edges of v_1 are traversed, it is not put in the stack again. The ending node of e_3 has not been visited yet and becomes the new starting node. The only outgoing edge of v_3 leads to v_0 which is a touched node. Thus the node at the top of the stack, v_0, is retrieved. The only remaining non-traversed edge e_1 is traversed. After v_0 is popped, the stack becomes empty and since e_1 is leading to a touched node, the DF graph traversal terminates.

In summary, a DF graph traversal for the directed graph shown in Figures 2.1 is:

e_0, e_2, e_5, e_6, e_3, e_4, e_1

The DF graph traversal algorithm assumes that each node of the graph is accessible from the starting node. Therefore, all nodes must be visited upon the termination of the algorithm.

However, if there are nodes that are not accessible from the starting node, the algorithm is invoked once for each unreachable subgraph. In this dissertation, all nodes are assumed to be accessible from the initial node and vice versa.

The functionality of the *BACKTRACK()* funtion is as follows. If upon invoking *BACKTRACK()* the stack is found empty, *True* is assigned to the variable DF_DONE to terminate the DF graph traversal. Otherwise, the node which was most recently placed in the stack is retrieved and the depth of the stack is decremented by one. On the other hand, the function *PUSH(Stack, v_i)* increments the depth of the stack by one and places v_i at the top of the stack.

The complexity of the DF algorithm is $O(V + E)$, where V and E are the number of nodes and edges of the graph, respectively. However, since $|V| \ll |E|$ for most of the graphs, the complexity of the algorithm reduces to $O(E)$ [21, 29].

2.1.3.2 Breadth-first Graph Traversal

The BF search is another widely used graph traversal technique. Concepts used in this technique are the basis for a number of important graph algorithms such as finding the shortest paths and constructing spanning trees [21]. In this dissertation, the BF shown in Figure 2.6 graph traversal technique is used for the detection and elimination of inconsistencies.

In BF search, the traversal starts with the initial node. All nodes with distance of length one from the initial node are visited. The next step is to visit all nodes of distance two from the initial node. The process is continued so that all nodes at distance d from the initial node are visited before any node at distance $d + 1$ is visited. In other words, the algorithm expands

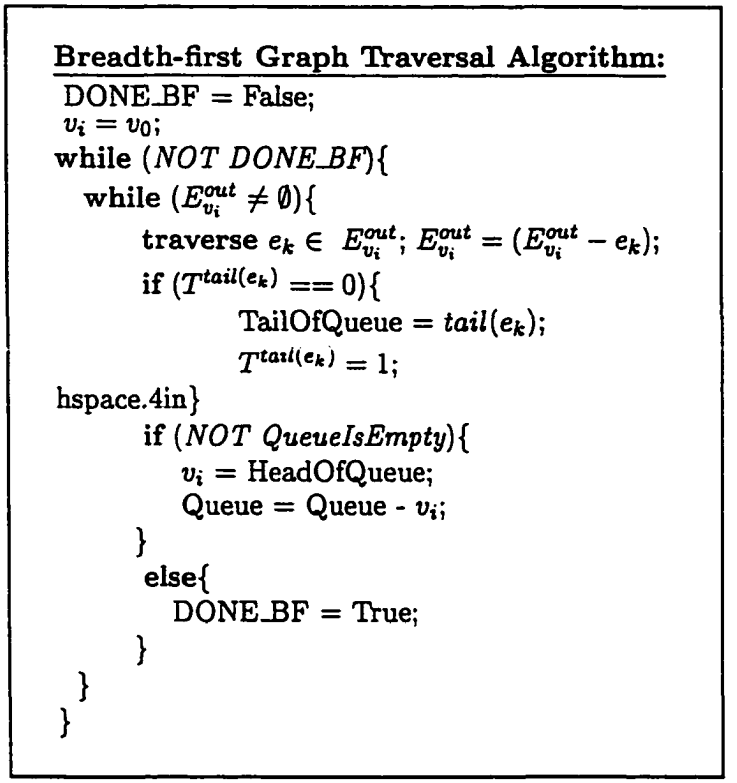

Figure 2.6: Breadth-first graph traversal algorithm.

the frontier between visited and unvisited nodes uniformly across the breadth of the frontier. Visited nodes are marked and placed in a first-in first-out queue.

After all adjacent nodes of the initial node are visited, the node at the head of the queue is selected to be the new initial node. This process continues until all edges and nodes of the graph are visited.

As for the DF algorithm, the complexity associated with BF algorithm is $O(V + E)$ which can be simplified as $O(E)$ for $|V| \ll |E|$ [21, 29].

Example 2: Let us assume that the directed graph of Figure 2.1 is being traversed in a BF

node v_i	traverse $e_k \in E^{out}_{v_i}$	queue
v_0	e_0	v_1
v_0	e_1	v_2, v_1
v_1	e_2	v_2
v_1	e_3	v_3, v_2
v_2	e_5	v_3
v_2	e_6	v_3
v_3	e_4	$\emptyset$

Table 2.1: BF graph traversal on Figure 2.1.

manner. The main steps of the graph traversal are depicted in Table 2.1. Each entry of the first column indicates the current node whose outgoing edges are being traversed. The traversed edges are in the second column. The contents of the queue are listed in the third column. As can be seen from Table 2.1, a BF graph traversal on the directed graph of Figures 2.1 results in: e_0, e_1, e_2, e_3, e_5, e_6, e_4.

2.2 Control and Data Flowgraphs

The control and data flowgraphs are both used to test general software and communication protocols. In general, FSMs and EFSMs can be directly derived from the control and data flowgraphs of the specifications. The control flowgraphs as well as the data flowgraphs can be constructed from the conventional flowchart whose basic elements consist of [5]:

- **decision points**: points at which the control flow diverges.
- **junction points**: points at which the control flow merges.
- **process blocks**: sequence of statements that are not interrupted either by junction points or decision points.

The main difference between a control flowgraph and the corresponding flowchart stems from the fact that the process blocks of the control flowgraph contain less details than the flowchart. With simple modifications on the flowchart of a specification, a control flowgraph consisting of a set of nodes and a set of edges can be constructed.

The controlling statements (such as *while*, *if*, *else*, and *else if*) check whether the conditions for traversing a path are met. Thus, decision points make the control to diverge. A combination of decision points and junction points form nodes of the data flowgraph. These nodes can be modeled as the states for an EFSM. The *predicates* used in the controlling statements have logical values of *true* or *false*. These predicates and their logical values can be related to the inputs of the EFSM. Moreover, performed tasks or actions in the processes of the flowgraph can be associated with the outputs of the EFSM.

Data flowgraphs are used when data objects are considered together with the control flowgraph. Each decision point and each assignment statement in the process blocks of the flowchart form a node for the data flowgraph. A data flowgraph node that corresponds to a block statement has one incoming edge and one or no outgoing edges.

The logical function at each decision point (i.e., predicate) is evaluated, based on the input values, to be either true or false. A predicate associated with a path is called *path-predicate* and consists of a concatenation of all of the predicates in the path and characterizes the necessary input values for the path to be traversed. Therefore, the feasibility of a path can be determined from its predicate expression.

Predicates associated with conditional statements whose decision points have only two branches are called *binary predicates*. The decision points with *multi-way* branches, (e.g., *case* statements in C language) can be transformed into equivalent *if-then-else* statements when possible, so that all predicates become binaries.

The test generation process becomes complicated when variables used for a predicate are process dependent. Such variables may be updated differently in the paths leading to the decision point. When a process dependent variable is used in a condition of an edge of a graph G, infeasible path(s) may be formed in G.

The corresponding data flowgraph of a specification (and thus its EFSM) may not be unique. Depending upon the conditions used in the decision points, there could be different data flowgraphs for the same specification. As an example, the data flowgraph representing the following conditional statement:

if *((x = A)* **AND** *(y = B))* **then** *z = 1;*

else *l = 2 ;*

can be modeled in two different ways as shown in Figure 2.7.

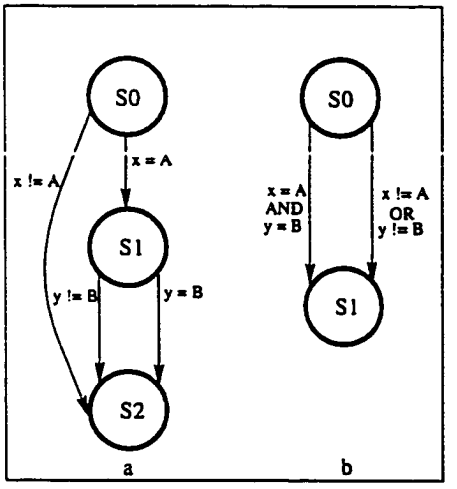

Figure 2.7: Two equivalent data flowgraphs.

One may consider decomposing the compound condition into two simpler conditional statements as shown in Figure 2.7.a. On the other hand, the compound conditional statement can be regarded as one single statement as in Figure 2.7.b.

2.3 Test Generation Methodologies

To detect discrepancies between a specification and its implementation caused by the errors in the implementation, development of efficient test generation methods is needed. In communication protocol engineering, conformance testing is used to detect possible mismatches between an implementation and its specification. Although the aim of testing is to detect errors and

increase the level of confidence on an implementation, none of the existing test methodologies can guarantee the absence of errors. A major challenge in conformance testing of complex communication systems is the development of methods that automatically generate test sequences with minimum lengths while maximizing the fault coverage.

As mentioned earlier, EFSM models include context variables. The interdependencies among these variables cause infeasible paths in the graph representation of the EFSM models and impair the automation of test generation from the EFSMs. On the other hand, FSM models do not include context variables and all paths of an FSM model are feasible. Therefore, it is easier to generate conformance tests for an FSM model than for an EFSM model.

The following sections provide an overview of the common approaches used to generate test sequences for FSM/EFSM-modeled specifications. Most of the examples in this dissertation are chosen from a formal description language called VHDL (Very High Speed Integrated Circuit Hardware Description Language) [4, 7], for which a brief discussion is provided later in this chapter. A short overview on symbolic execution, which is used to investigate the interdependencies among the EFSM action and condition variables, is also provided in Section 2.5.

2.4 Test Generation for Communication Protocols

In conformance testing, an IUT must meet certain requirements before a tester concludes that it is conforming to its specification. Such requirements are grouped as the *static conformance requirements* and the *dynamic conformance requirements*. The static conformance requirements

deal with the issues such as the capability of the IUT to support certain range of values whereas the dynamic conformance requirements are concerned with the observable behavior of the IUT while in operation.

Since exhaustive testing is impractical due to economical and time constraints, the following types of conformance tests are commonly used:

- **Basic interconnection tests:** to check if the main features of the IUT are implemented.
- **Capability tests:** to check if the IUT meets the static conformance requirements.
- **Behavior tests:** to check if the IUT satisfies the dynamic conformance requirements.
- **Conformance resolution tests:** to provide a definite yes/no answer to the questions regarding if a particular feature is correctly implemented.

During conformance testing of a state transition (s_i, s_j; $input_l/output_k$) of an FSM/EFSM, the three essential steps called the *basic test procedure steps* are as follows [53]:

1. The IUT is brought into the desired state of s_i.
2. The input $input_l$ required for the transition is applied to the IUT and the generated output(s) $output_k$ (if observable) are compared with the expected output(s).
3. It is verified that the IUT has moved from s_i to s_j as defined by the specification (if step 2 is successfully passed).

Bringing an IUT to a particular state may require many other transitions before reaching the desired state. This step addresses the so-called *controllability* issue in conformance testing.

The state verification step of the test procedure is referred to as the *observability* issue. When testing a state transition, applying an input to the IUT may generate the expected output. However, the implementation could have moved to a different state other than the expected next state. During the third step of the testing procedure, to verify that the IUT has made the specified transition into the defined next state, each state must be distinguishable from the others.

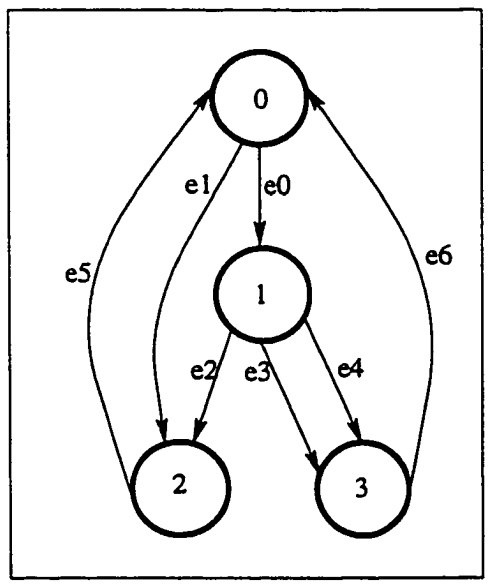

Figure 2.8: Asymmetric FSM graph.

The common techniques for new state verification to overcome the observability problem include unique input/output (UIO) sequences, distinguishing sequences (DS), characterizing sequences (W), and their variations. On the other hand, the transition tours (TT) technique addresses the issue of controllability [16, 53, 65, 78].

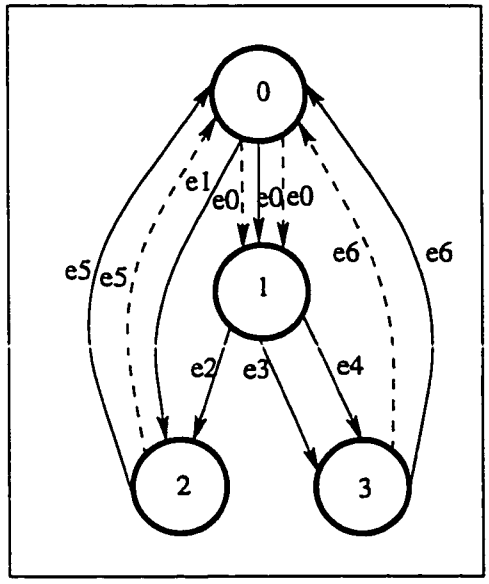

Figure 2.9: Augmented symmetric graph of the FSM graph given in Figure 2.8.

The fault coverage obtained by using the DS, W, and UIO methods are reported to be approximately the same. In the TT technique, the third step of the basic test procedure (i.e., new state verification) is not performed. Therefore, the fault coverage of this technique is lower than those of the other three techniques [69].

Among the FSM-based test generation methods used for generating effective test sequences for real-life protocols to remedy the problems of *controllability* and *observability* are the *Chinese postman* and *rural Chinese postman* methods [3, 78].

The Chinese postman problem is defined as *finding the minimum-length path of traversing a directed graph by covering each edge at least once.* The problem has a solution provided that the graph contains an *Euler tour.* An Euler tour is a sequence of edges that begins and ends at the same node, and, at the same time, contains each edge of the graph exactly once. A

strongly-connected graph G has an Euler tour if and only if

$$\forall\, v_i \in V:\ d_{in}(v_i) = d_{out}(v_i)$$

An augmented symmetric graph can be constructed from an FSM graph G that does not satisfy this condition by duplicating some of the edges of G. The duplications of the edges (if any) must be kept minimum so that the length of the resulting tour is minimum. Figure 2.9 shows the augmented symmetric FSM graph of Figure 2.8. An Euler tour for Figure 2.9: e_0, e_3, e_6, e_0, e_4, e_6, e_0, e_2, e_5, e_1, e_5.

For specifications possessing a special feature known as the *status feature* [78], each state of the FSM/EFSM is associated with a self loop whose input/output label is in the form of *status_enquiry/state_*s_n, where s_n is the state being verified. If the specification has the status feature, the solution to the Chinese postman algorithm, which is an Euler tour, yields a minimum-length test sequence for the IUT.

For specifications that do not have the status feature, the rural Chinese postman algorithm (which is a generalization of the Chinese postman algorithm) can be used to generate a minimum-length test sequence for the IUT [3]. For each state s_i, an input/output sequence called UIO sequence which uniquely identifies s_i from all other states is generated in the following manner. Starting from s_i, all input/output sequence of length one are checked for uniqueness. If there is at least one such sequence, a UIO sequence for the state is found. Otherwise, all input/output sequences of length two are checked for uniqueness. This process is continued until a UIO sequence is found for each state [65].

A concatenation of an edge and a UIO sequence of its ending node forms a *test edge* T. The

cost of a test edge is the total cost of its components. The *rural postman tour* is a tour in which each test edge is traversed at least once while other edges called *ghost edges* are traversed only when necessary. To form an Euler tour, the graph is symmetrized. Because of their costs, test edges are not duplicated [3]. A tour that covers all test edges and some of the ghost edges (if any) corresponds to a minimum-length test sequence.

2.4.1 Control and Data flow Analysis

The number of paths that may exist in a specification are prohibitively large. A practical path selection criterion which specifies a finite subset of the specification's paths must be defined. In software testing several path selection criteria have been proposed [18, 64]. In this section families of test selection criteria termed as the *path* and *data-flow* testing are discussed.

Path testing is the oldest of the structural testing techniques and is generally suitable for unit testing rather than the system testing. However, this technique is recognized to be important since almost all other testing strategies are based upon path testing [5].

Among the many path-testing criteria are:

1. **All-paths**: requires traversing all possible paths in the control flowgraph of the specification. Although this criterion is the strongest criterion, it is usually unachievable since the number of paths can be prohibitively large.

2. **Statement testing**: under the same test, each statement in the specification is executed at least once. This criterion is said to be the weakest of all testing criteria.

3. **Branch testing:** each branch of the control flowgraph of the specification is traversed at least once. If branch testing is applied, statement testing is automatically achieved.

Static data flow analysis (DFA) techniques are used to generate test sequences for EFSM models. Data-flow testing fills the gaps between all-paths, branch, and statement testing. It is used, but not limited, to detect data-flow anomalies that can cause errors. Data-flow anomalies include the use of a variable that was not declared, a variable that has been killed, and opening/closing a file that does not exit. The control flowgraph is used to select paths of interests. As will be shown, an EFSM graph can be constructed from the data flowgraph of a specification.

Let us provide some definitions used for the DFA [52, 64, 66]. A node v_j of the data flowgraph is called a

- *defining node* $d_{v_j}^{x_i}$ with respect to a variable x_i if the value of x_i is defined by the statement corresponding to v_j.
- *usage node* $u_{v_j}^{x_i}$ with respect to a variable x_i, if x_i is used in the statement corresponding to v_j either for a predicate (*p-use*) or for a computation (*c-use*).

A path of a data flowgraph is called:

- *definition-clear* path (*dc-path*) with respect to a variable x_i if there are defining and usage nodes, $d_{v_j}^{x_i}$ and $u_{v_k}^{x_i}$, where v_j and v_k are the initial and final nodes of the path, respectively, and no other node in the path is a defining node for x_i.
- *definition-use* path (*du-path*) with respect to a variable x_i if there are defining and usage

nodes, $d_{v_j}^{x_i}$ and $u_{v_k}^{x_i}$, where v_j and v_k are the initial and final nodes of the path. In other words, a path $P = e_1 \cdots e_m \dot{e}_j$ that does not contain a definition of x is a def-clear path with respect to x from tail(e_i) to tail(e_j)

A *du-path* which is not a *dc-path* is usually considered as an indication of errors.

For a *p-use* node v_j of a data flowgraph, $d_{out}(v_j) \geq 2$ whereas for a $d_{v_j}^{x_i}$ or *c-use* of $u_{v_j}^{x_i}$ node, $d_{in}(v_j) = 1$ and $d_{out}(v_j) \leq 1$.

Rapps and Weyuker [64] proposed a family of data-flow testing criteria that effects variable definitions and usages. A variable x is globally defined/used if x is used in a node other than the defining node of x. A *def/use graph* is a data flowgraph in which each edge is associated with one set *p-use*(e_i) and each node is associated with two sets *def*(v_i) and *c-use*(v_i). The set *p-use*(e_i) contains all variables used in the predicates of e_i. The sets *def*(v_i) and *c-use*(v_i) contain those globally defined and used variables at v_i, respectively. In addition, two more sets, $dcu(x, v_i)$ and $dpu(x, v_i)$, are defined for each node to construct the criteria. The set $dcu(x, v_i)$ represents the set of nodes $\{v_j\}$ such that $x \in$ *c-use*(v_j) and for which there is a def-clear path with respect to x from v_i to v_j. Similarly, $dpu(x, v_i)$ contains the set of nodes $\{v_j\}$ such that $x \in p - use(v_j)$ and for which there is a def-clear path with respect to x from v_i to v_j.

For a def/use graph G, a set of complete path P satisfies the:

1. **all-nodes** criterion: if every node v_i of G is included in P.

2. **all-edges** criterion: if every edge of G is included in P.

3. **all-defs** criterion: if for every node v_i of G and every $x \in def(v_i)$, P includes a *def-clear* path with respect to x from v_i to all elements of $dpc(x, v_i)$ or $dpu(x, v_i)$.

4. **all-p-use** criterion: if for every node v_i and every $x \in def(v_i)$, P includes a *def-clear* path with respect to x from v_i to all elements of $dpu(x, v_i)$.

5. **all-c-uses/some-p-uses** criterion if for every node v_i and every variable $x \in def(v_i)$, P includes some *def-clear* with respect to x from v_i to every element in $dcu(x, v_i)$. If $dcu(x, v_i)$ is empty, P must include a *def-clear* path with respect to x from v_i to some edge contained in $dpu(x, v_i)$.

6. **all-p-uses/some-c-uses** criterion: if for every node v_i and every variable $x \in def(v_i)$, P includes some *def-clear* with respect to x from v_i to every element in $dpu(x, v_i)$. If $dpu(x, v_i)$ is empty, P must include a *def-clear* path with respect to x from v_i to some edge contained in $dcu(x, v_i)$.

7. **all-uses** criterion: if for every node v_i and every $x \in def(v_i)$, P includes a *def-clear* path with respect to x from v_i to all elements of $dcu(x, v_i)$.

8. **all-du-paths** criterion: if for every node v_i and every $x \in def(x, v_i)$, P includes every *du-path* with respect to x. If there are multiple *du-paths* from a global definition to a given use, they must all be included in paths of P.

The all-du-use criterion generates fewer test cases than all-uses. Furthermore, all-du-use is suitable for programs containing loops where all-paths is not achievable.

Due to details presented in the data flowgraphs, a data flowgraph of a specification is usually

lengthy. A concise model of the data flowgraph of a specification, which is an EFSM, can be constructed as follows [52]:

1. Each maximum distinct sequence of nodes, $v_i, v_{i+1} \ldots, v_{i+l}$, of the data flowgraph in which $d_{in}(v_j) = d_{out}(v_j) = 1$, where $i \leq j \leq i + l$, forms an edge of the EFSM. The assignment statements in the sequence become the actions of the edge.

2. Each node v_i of the data flowgraph such that $d_{out} \geq 2$ forms a node of the EFSM graph.

3. A node v_i of the data flowgraph in which $d_{in}(v_i) = 0$ and $d_{out}(v_i) \geq 1$ can represent the initial node.

4. Each node of the data flowgraph such that $d_{in}(v_i) \geq 1$ and $d_{out}(v_i) = 0$ is an exit node.

2.4.2 Test Generation for EFSM models

The FSM-based test generation methods are not directly applicable to EFSM models. In this section, different approaches used to generate test sequences from EFSMs are discussed.

One approach to the problem of generating test sequences for the EFSM models is to transform the EFSM into equivalent FSM and then use the FSM-based test generation methods [14, 37, 50]. The size of the FSM that results from a transformed EFSM can be prohibitively large. For example, the number of states for the equivalent FSM is reported to be

$$S_{FSM} = S_{EFSM} \times D_{var_1} \times \ldots \times D_{var_n} \tag{2.6}$$

where S represents the number of states and D_{var_i} is the domain for the i^{th} variable [11, 37]. Such an approach may easily cause an explosion of the number of states, where the number of

states grows exponentially and, hence, executing a test sequence that covers all edges for the resulting FSM is not practical due to the time it requires.

Recently several test generation methods for the EFSM models appeared in the literature [13, 14, 50, 51, 58, 59, 66, 67, 75, 76]. Sarikaya *et al.* used the functional program testing approach to generate test sequences from the EFSMs [67]. The data flowgraph of the IUT is partitioned into smaller functional blocks representing various protocol functions. A set of tours is selected to test each functional block. Since the feasibility of these tours is not considered during the test generation, a number of infeasible tours may be generated.

Software data flow testing approaches have been used to generate tests for the communication protocols [66, 75, 76]. Ural applied the all-uses [64, 75] criterion, used for testing software written in block structured programming languages, to Estelle [9] specification of protocols. The specification is modeled as a directed graph which illustrates the relationship between definitions and usage of the variables. Before test sequences are generated, infeasible paths and data flow anomalies are identified. The all-uses criterion requires that for each variable var_j defined at a node v_i, there is a path P which includes a *def-clear* path with respect to the variable var_j from v_i to all uses of var_j. Recall that a path, $P = e_1 \cdot e_2 \cdots e_i$, is called a def-clear path with respect to var_j if var_j is not defined in the sub-path $e_2 \cdot e_3 \cdots e_{i-1}$. Several variants of this method appeared in literature. Salah *et al.* [66] applied the all-uses of the data-flow test selection criteria and *IO-def-chains* (defined as a sequence of du-pairs that exposes the effects an input variable on the output variables or predicates) to generate test sequences from the EFSMs, A set of complete paths of the data flowgraph G is selected to cover every maximal IO-def-chain

in G at least once. Clarke [18] indicated that none of the existing software test selection criteria guarantees the selection of only feasible paths. Therefore, the selection of complete feasible paths covering every maximal IO-def-chain may not be possible.

Miller and Paul [59] introduced a method to generate tests for both control and data for EFSM models. The control flow is tested in a manner similar to that of the FSM models and the test generation for the data portion is based on the assumption that the IUT is a "white box." All variables are assumed to be accessible to the tester, both to set as inputs and observe as outputs. Such assumption may not be possible for most of the protocols. A modified FSM is obtained where each transition of the EFSM is divided into several transitions corresponding to the different paths of the control flow while the number of states are kept the same. Test sequences are generated by using *def-ob-paths* and *conditional paths* defined as follows. A set of transitions $T_1, T_2, ..., T_n$, in which a variable var_i is assigned by an input parameter of T_1 and the value of var_i flows through other variables in $T_2, ..., T_{n-1}$ until it appears in an output parameter of T_n, is called *def-ob-path*. A transfer sequence may be needed from transition T_i to transition T_{i+1}. A conditional path $C = T_1, ..., T_n$ is a path in which a variable var_i is assigned by an input parameter of T_1 and the value of var_i flows through other variables of the path until it affects the enabling condition of T_n. For a def-ob-path and conditional path to be considered as part of a test sequence, each transition of these paths must not generate a context which causes another transition to be inexecutable.

Miller and Sanjoy [58] extended the concept of UIO sequences of the FSM models to EFSM models by proposing a new state verification method called *identifying sequences* (IS). The main

differences between the IS and UIO sequence is that the IS does not only depend on the state of the EFSM but also on the variables. The EFSM is assumed to have *stopping predicates.* A stopping predicate of a transition t enables it once and does not enable any successive transition. For an EFSM with the stopping predicate property, the EFSM stops after a transition t is executed and waits until a new set of inputs is provided. However, a typical EFSM may not have stopping predicates. To guarentee the executability of a transition sequence, it is not sufficient to satisfy only the predicates associated with the test sequence. Actions may affect the executability of the sequence by modifying the context variables. Therefore, the satisfiability of the predicate of a transition heavily depends on the actions of the other transitions already included in the sequence.

Chanson and Zhu [13] proposed a test generation method which considers both the control flow and data aspects of the EFSM models. For the control flow testing, a combination of selected subtours and characterizing sequences (i.e., the W method) are used. The new state verification is handled by using the so-called *cyclic characterizing sequence.* Each of such sequences contains a characterizing sequence for a state s_i and the sequence that brings the machine back to s_i. For the data flow coverage the all-du-path or the all-use is used. A set of paths $\mathbf{P}$ satisfying the required def-use association is collected. Each path $P_i \subseteq \mathbf{P}$ is concatenated to the shortest path from the initial state s_0 to the starting state of P_i. The shortest path from the ending state of P_i to s_0, followed by the cyclic characterizing sequence of s_0, is appended to the augmented path P_i. The feasibility of the selected tours is determined by using the *the constraint satisfaction problem* techniques used in the artificial intelligence.

If a tour containing a self-loop which influences the control flow is found to be infeasible, recurrence relations are used to calculate the number of times that the self-loop must be included in the tour so that the tour becomes feasible. The feasibility of the subtours is checked only after they are constructed. Furthermore, the presence of influencing self-loops may not be true for a number of EFSM models.

Li *et al.* [51] introduced a method for EFSM state verification called *Extended-UIO sequences.* For each state s_i of the EFSM, a specific preceding sequence from the initial state s_0, which enables the execution of the UIO sequence of s_i, must be constructed. A concatenation of the preceding sequence and the UIO sequence of the state forms the E-UIO sequence. To test a transition from s_i to s_j, two feasible walks W_1 and W_2 containing the same walk from the initial state to s_i are generated. W_1 consists of a transfer sequence of s_i and the UIO sequence of s_i. The last transition of W_1 is the transition under test. W_2 contains the E-UIO sequence of s_j. For states with multiple incoming transitions, the construction of W_1 and W_2 for each transition may not be achievable.

1. Since the E-UIO sequence of a state must contain a preceding sequence, each state could have E-UIO sequences as many as the number of incoming transition of E-UIO sequences.

2. Once a UIO is identified for a state, it is not decidable whether the UIO has a corresponding E-UIO sequences.

In the above method there are cases where the feasibility of a preceding sequence are modified to test certain transitions.

Lee and Yannakakis [50] provided a method to convert a class of EFSMs, where input variables are assumed to have finite domains, into equivalent FSMs. Combinations of a state and a variable value form a *configuration*. The configurations of the EFSM are partitioned into equivalent classes. The outgoing transitions from a state s_i are examined and partitioned into blocks such that all configurations in each block contain variables that are valid for the same predicate of the outgoing transition from s_i.

A transition t from block B to block C with action A is called *stable* only if the domain block of t is contained in the inverse image of $A : B \subseteq A^{-1}(C)$. To stabilize a transition its domain block is split. Each graph split could introduce new unstable transitions. Cheng and Krishnakumar [14] used this method to generate test sequences from hardware specification given in VHDL or Bestmap-C [14].

The number of possible combinations is $|S| \times |var_1| \times |var_2| \times \ldots \times var_m$, for an EFSM with m variables [50]. This method is not applicable to EFSMs whose parameters have infinite domains. Even when all variables have finite domains, the corresponding number of configurations can easily become prohibitively large.

Although the above-mentioned methods made significant contributions towards the test generation from the EFSMs, the inclusion of infeasible paths in the test sequences may be inevitable since the underlying models are EFSMs. Therefore, without a proper analysis of the interdependencies among the variables used in the actions and conditions of the EFSMs, considerable effort may be wasted on test generation since the infeasible portions of these test sequences will be discarded later.

For a restricted class of LOTOS expressions, called P-LOTOS, Higashino and Bochmann propose a method that provides solutions to a set of interrelated problems such as the test case derivation and the detections of nonexecutable branches, deadlocks, and nondeterminism [38]. The P-LOTOS expressions have the property that all variables are of integer and Boolean types and the operations are restricted to addition, subtraction, and comparisons. Furthermore, variable overloading is not supported. Inference rules are used to rename some of the variables in the event that a variable overloading occurs in a P-LOTOS expression.

For a given P-LOTOS expression, a tree called the extended labeled transition system (ELTS), which can be an infinite tree in the general case, is defined to represent the possible event sequences during the execution.

Let $t = \langle P_0, P_1 \cdots P_k \rangle$ be a LOTOS expression. An ELTS(t) for the expression is defined with the following properties:

- Each node of the tree has a label which is a behavior expression and the root node P_0 is the main processor.
- A node n of the ELTS(t) with the label B has a child n' whose label is B' provided that there exists a behavior expression B' satisfying the relation $B - \langle a, c \rangle \rightarrow B'$.
- For a node n and its child node n' of the ETLS(t), the edge $(n \rightarrow n')$ has two labels $Event(n \rightarrow n')$ and $Cond(n \rightarrow n')$ representing a and c, respectively.

The problems related to the detections of the deadlocks and non-determinism are beyond the scope of this thesis. Hence, we will only describe how the authors detect the infeasible edges

and generate the test cases.

The authors use linear integer programming to determine if a given branch of the ELTS is infeasible. Once an infeasible branch is found, the branch and all its descendants are deleted from the tree. After all infeasible paths are deleted from the ELTS, the resulting tree, called the reachability graph, is used to derive test cases.

As in the case of the ELTS, deleting infeasible branches and their descendents does not affect the other feasible paths when the number of parallel edges between each pair of nodes is one. In this thesis, our analysis is based on the fact that the number of parallel edges between a given pair of nodes can be more than one for most of the EFSM graphs.

Although an edge e_i of an EFSM graph may be infeasible due to the conditions and actions of certain edges in a path P_j, from the initial node, that leads to the $head(e_i)$, e_i may be feasible if certain other path P_k, also from the initial node, is traversed. The method proposed in this thesis analyzes, before deleting infeasible edges, the effect of actions on the condition variables. The method focuses first on the effects of variables differently modified by actions on condition variables.

If an edge e_i is found to be infeasible for some of the different values of its condition variables, a graph splitting algorithm is employed to resolve the conflicts among the edges whose actions and conditions conflict. After the graph is split, e_i will remain in one of the subgraphs. After the analysis of the effect of differently modified variables on on edge conditions is completed, the method checks if there are edges whose conditions cannot be satisfied at all due to the values of their condition variables. Such edges (if any) are deleted from the graph. Finally, in this

method, the interdependencies among the edge conditions are analyzed. Conflicts among the edges due to their conditions (if any) are resolved by employing a graph splitting algorithm.

In this thesis, variable overloading is allowed and variables can be of both integer and floating point type. The EFSM graph produced in this method has the property that all its paths are feasible. Therefore, the FSM-based techniques can be used to generate feasible test sequences from the resulting EFSM. Therefore, the method presented in this dissertation guarantees the generation of only feasible test sequences from a class of EFSMs.

2.5 Path Analysis and Symbolic Execution

In the field of software testing, program testing includes program verification (also known as program correctness) and program validation. In program verification, mathematical proofs using theorem-proving techniques are implemented to show that the program behaves as specified. The nature of the problem requires manually providing theorem correctness hypothesis which makes automating this approach impractical [17, 54]. Different program testing systems focus on data flow anomalies, path computations, and generating test data and verifying assertions for program paths.

One way of testing a system is to conduct path analysis and generate reliable test data. A test data is said to be reliable if it reveals the existing errors [49]. To traverse a given path, a set of inputs that satisfies all conditions of the path is needed. However, for an infeasible path, there is no data that can cause the path to be executed.

Symbolic execution is one of the widely used techniques in software engineering to generate tests and detect program errors such as missing paths, path-domain errors, and path-function errors [12, 17, 20, 47, 48, 49, 85]. In symbolic execution, variables assume symbolic values rather than actual values. When an assignment statement is symbolically executed, each variable on the right-hand side is replaced with its current symbolic value and after some simplifications the resultant expression becomes the new symbolic value for the left-hand side variable. Similarly, variables in the conditional statements are substituted with their symbolic values before these expressions are evaluated. The feasibility of the path is then determined by analyzing the symbolic values of the accumulated conditions.

The software testing tool DAVE interfaces symbolic execution with a data flow analysis system [17]. The program being tested is represented as an intermediate format to adopt different programming languages, simplify expressions during analysis, and detect parallelism and enhance optimization. The intermediate code is then symbolically executed. Directed graphs called *evolution graphs* are generated from the computations evolving from variables' symbolic values. Artificial constraints are created to simulate error conditions. For example, array subscripts are allowed to exceed their specified boundaries. The augmented constraint is solved and if the new constraint is inconsistent with the system of constraints, the path is found to be infeasible. If all constraints of a path are consistent, the final solution of the system of constraints will consist of the data that would cause execution of the path.

The inequality solver used by DAVE requires that all constraints are linear. DAVE uses the ALTRAN tool which detects non-linear constraints and manipulates expressions to put them in

a linear form, is used. Instead of solving the inequalities during the symbolic execution process, they are solved after the symbolic execution of the program is performed.

SADAT [85] is one of the systems that integrate symbolic execution with static/dynamic analysis systems of programs. This system incorporates techniques used by other systems such as DAVE, FACES, PET, RXVP, JAVS, DISSECT, and ATTEST. These systems perform static/dynamic analysis and symbolic execution as well. Complete testing would require executing all possible paths in the program. Since all possible paths cannot be covered, a subset of the paths that requires the execution of each decision-to-decision path at least once is aimed. Symbolic execution is performed on all variables, including those used in decisions, in the path. SADAT does not determine the feasibility of the paths. Hence, manual analysis for the symbolic values of the paths is needed [24, 85].

The final path predicate is composed of input variables, constants, and arithmetic and logical operators. The resultant path predicates can be used for input test data preparation and as a method of verifying that the program is implemented as specified. Certain statements are inserted in each decision-to-decision path to accumulate certain information, such as the frequency of execution of each decision-to-decision path. Such information can be used to identify dynamically dead codes and the number of executed iterations for each loop. Optimizations of the most frequently executed paths can be obtained by using this accumulated information. For SADAT, although static and dynamic analysis proved to be valuable, test data generation has some deficiencies for large programs and symbolic execution has shown deficiencies when there are loops and subroutine calls.

Symbolic execution tools may vary in their capabilities to detect errors. For example, DISSECT [47] reveals path-function errors more than the other types of errors and is not reliable for discovering path-domain errors. Regardless of their different features and capabilities, symbolic execution systems should be able to produce a path condition for each path, determine the feasibility of paths, and express output variables in terms of input variables and constants [24].

The deficiency of the symbolic execution is due to its path-oriented nature [12, 24, 47]. In this dissertation, symbolic execution is used as a tool in the detection and elimination of inconsistencies in EFSM models, not as a test generation tool.

2.6 VHDL

Very High Speed Integrated Circuit Hardware Description Language (VHDL) [4, 7] is an extension of the Very High Speed Integrated Circuit (VHSIC) program, funded by the U.S Department of Defense (DoD). VHDL was first proposed as a language in 1981. The motive was to enhance the interoperability of the different chips designed by different vendors [4, 7]. The language was also intended to be standardized so its documentation and testing would be easier. Among the features that make VHDL more suitable for describing digital systems is its support of timing constraints and logical descriptions. Both concurrent statements and processes are supported by VHDL.

VHDL is normally used to model digital systems at different levels of abstraction ranging from the algorithmic level to the gate level. Recently the language has been also used to model

communication protocols [45, 61].

Spurred by the fact that a number of manufacturers have commercially proven cost effective and highly efficient methods to implement hardware designs for VHDL specifications, VHDL is now widely used in the electronic industry. The hardware of a system specified in VHDL can be automatically sketched from the specification. Furthermore, one of the advantages of using VHDL as a formal description language is that VHDL specifications are synthesizable. Interoperability among components of a digital system can also be increased if all components are described in VHDL.

VHDL supports the top-down design methodology. Therefore, designers can define systems more accurately by describing the systems' behavior at a high level of abstraction. Details can be gradually included into the high level model of the system specified in VHDL. When a system is described in VHDL, the system's external interfaces, called *ports*, are defined separately from the system's internal implementation which is described in one or more different architectures.

The architecture of a system described in VHDL can be one of three types: structural, data-flow, or behavioral. Architectures that comprise more than one type are also acceptable in VHDL [7]. In the structural style of modeling the entity is defined as a set of interconnected simpler components. On the other hand, the data-flow modeling style uses concurrent statements from which the structure of the component can be implicitly deduced. Finally, in the behavioral modeling style the internal structure of the entity is not explicitly defined. The behavior of the entity is rather defined as a set of sequential statements that are specified in a *process* statement. Examples and case studies considered in this research use the behavioral modeling style.

VHDL specifications are normally modeled as EFSMs. Vemuri and Kalyanaraman [84] discussed test generation for the behavioral VHDL specifications by using symbolic execution approach. A constraint solver is used to obtain test data for a given path. In their method, infeasible paths of the behavioral VHDL specification are assumed not to be present. User annotations are attached to the control statements and variables to delimit the execution process and to set variables' ranges. All VHDL control statements, such as *loop*, *process*, and *wait* statements, are translated into equivalent *if-then-else* statements. Furthermore, spatial and temporal incarnations are used for choosing values that variables of the current path should resume.

Chapter 3

Generation of Feasible Test Sequences for EFSM Models

As the complexity of communication and computer systems increases, substantial effort is being directed towards the development of efficient test generation algorithms to minimize the cost associated with system failures. In conformance testing, discrepancies between an IUT and its specification are detected.

The FSMs and the EFSMs can be used to model the external behavior of protocols. In general, it is easier to generate tests for an FSM model than for an EFSM model since the FSM model does not include internal variables which may cause infeasible paths in the graph representation of the model. This relative simplicity of the FSMs allowed the development of methods to automatically generate feasible conformance tests [8, 9, 10, 11, 13, 14, 28, 39, 41, 50, 51, 53, 60,

62, 63, 65, 66, 67, 68, 74]. The automation of test generation from the EFSMs is mainly impaired by the existence of *inconsistencies* among the actions and conditions of the EFSM models. The specifications written in formal description languages such as VHDL and Estelle are typically complex enough to be modeled only as EFSMs [14, 66, 75, 81].

If the interdependencies among the conditions and actions of an EFSM model are not taken into consideration during the test generation process, the test sequences may contain infeasible paths. Due to the existence of such infeasible paths, considerable effort on test generation may be wasted since the infeasible portions of these tests will be discarded later. However, if an EFSM is *consistent* (or the EFSM is converted to a consistent EFSM), the FSM-based test generation methods can be used to generate conformance test sequences from the consistent EFSM [27, 79, 81].

In general, generating feasible test sequences from the EFSMs is an open research problem. This dissertation presents a method which enables the generation of only feasible test sequences from a class of EFSM models. It is assumed that the specification consists of a single process with linear actions and conditions. It is also assumed that pointers, recursive functions, and syntactically endless loops are not present in the specification.

The methodology presented here aims at the detection and elimination of inconsistencies caused by the interdependencies among the variables used in the actions and conditions of the EFSM models with the above-mentioned properties. After the inconsistencies caused by the action variables are eliminated, the method proceeds with the elimination of inconsistencies among the conditions of the EFSM (if any).

Although the examples presented in this dissertation are based on VHDL, the algorithms of this dissertation are applicable to all EFSMs with the aforementioned properties.

3.1 Inconsistencies

One of the major differences between the FSMs and EFSMs is due to the memory (i.e., the variables used in conditions) associated with the EFSMs. Unlike FSM graphs, the traversal of an edge $e_k = (v_i, v_j)$ of an EFSM graph mainly depends upon the conditions of the edges in the paths leading to v_i whereas for an FSM graph the traversal of an edge only depends on the current node and the specified input. In other words, from an EFSM graph the traversal of an edge from a node v_i may not be possible due to the conditions of the path(s) from v_0 to v_i while for an FSM graph all paths are feasible. The complexity of testing EFSM models increases when one or more variables used in the edge conditions can assume multiple values at the same node.

A comprehensive analysis of the inconsistencies among the actions and conditions of the EFSM models is given in the subsequent sections. Although the examples presented in this dissertation are simple, they are designed to depict the inconsistencies commonly present in real-life communication protocol specifications.

Let us represent the condition of an edge as

$$a_{00}x_0 + a_{01}x_1 + \ldots + a_{0(m-1)}x_{m-1} < op > d$$

and the of an action as

$$x_i = a_{01}x_0 + a_{11}x_1 + \ldots + a_{1(m-1)}x_{m-1} + d$$

where m, op, and d are the number of variables, an operator, and a constant, respectively.

Definition: 4 Condition inconsistency: *If there is no solution for the set of equations formed by the conditions of two edges e_i and e_j, where $head(e_j)$ can be reached from $tail(e_i)$ or $head(e_i) = tail(e_k)$, then e_i and e_j are said to have a condition inconsistency.*

As an example, consider the EFSM graph represented by a directed graph shown in Figure 3.1. A test sequence generated from this EFSM, which includes e_4 and e_5, cannot be realized since such a test sequence requires two conflicting conditions, $(c \leq 0)$ and $(c > 0)$, to be satisfied simultaneously.

Definition: 5 Action inconsistency: *If the set of equations formed by the actions of an edge e_i and the condition of another edge e_j, where $head(e_j)$ can be reached from $tail(e_i)$ or $head(e_j) = tail(e_i)$, has no solution, then the two edges of e_i and e_j are said to have an action inconsistency.*

In Figure 3.1, the action of e_0 assigns 1 to b. Later in the graph, the variable b is used in the conditions of e_8 and e_9. Since the set of equations formed by the action of e_0 and the condition of e_9:

$$(b = 1) \text{ AND } (b > 1)$$

does not have a solution, there is an action inconsistency between e_0 and e_9.

Definition: 6 Consistent EFSM: *An EFSM which is free of both action and condition inconsistencies is called a consistent EFSM.*

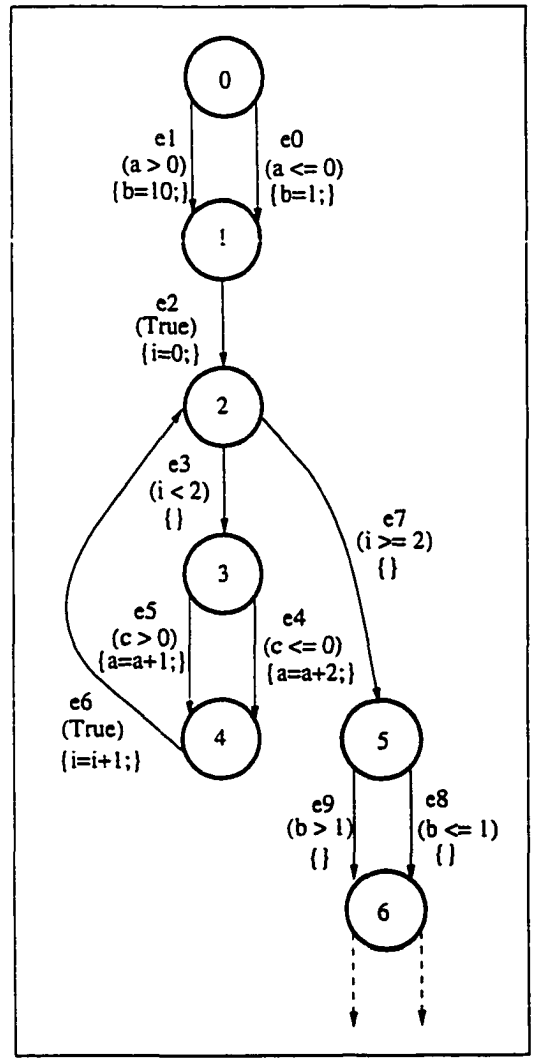

Figure 3.1: An EFSM graph with inconsistencies.

In a consistent EFSM, the variables used in the conditions and the actions do not impose any restrictions over the paths. Hence, all paths of a graph representing a consistent EFSM are feasible.

The first step towards the automatic generation of test sequences from an EFSM is to detect and eliminate inconsistencies (if any) in the EFSM models. The DF and a MBF graph traversals are used to detect inconsistencies. For the inconsistency elimination, a graph splitting technique based on symbolic execution and linear programming is introduced.

Once the inconsistencies are eliminated from an EFSM graph, the next step is to automatically generate realizable test sequences from the resulting consistent EFSM by using the test generation methods available for FSM models [8] - [14] , [28, 39, 41, 50, 51], [53] - [60], [62] - [69], [73] - [83].

3.2 Detection and Elimination of Inconsistencies

As mentioned earlier, variables used in the actions and conditions of an EFSM graph may contribute to the formation of infeasible paths in the EFSM graph. The algorithms presented in this chapter enable the automated generation of only feasible test sequences from a class of EFSMs. These algorithms eliminate the inconsistencies by creating new nodes and edges, which increases the size of the original graph. However, the inconsistency elimination algorithms create these new nodes and edges only when necessary. Therefore, although the well-known state explosion cannot be avoided for all EFSMs, the unnecessary growth of the state space is prevented. For the cases where the state explosion is unavoidable, the size of the new graph is constantly monitored as the algorithms eliminate the inconsistencies.

3.3 Action Inconsistencies

The elimination of inconsistencies from the EFSM models starts with the detection and removal of action inconsistencies (if any) and proceeds with the elimination of condition inconsistencies (if any). In this section, the detection and elimination of action inconsistencies are discussed.

Variables modified in the paths leading to a node v_i may cause action inconsistencies with the condition of another edge e_r, where $head(e_r) \in V_{v_i}^{reachable}$. Therefore, the effects of the variables modified differently by the actions of the paths leading to a node v_i on the conditions of the edges reachable from v_i need to be analyzed.

In general, the effects of edge actions on variables (i.e., variable modifications) can be represented in matrices. For an EFSM graph with m variables, $var_1, var_2, \ldots, var_m$, a pair of matrices $A(m \times m)$ and $\tilde{B}(m \times 1)$ are defined, where A and $\tilde{B}$ are called *the modification matrix* and *the modification vector*, respectively. Only one AUM pair, in which A and $\tilde{B}$ are initialized to the identity matrix and to a zero vector, respectively, is associated with the initial node of an EFSM graph.

The *accumulated effects* of the actions in the paths leading to a node v_i can be represented in a set of *Action Update Matrix* pairs

$$\text{AUM}(v_i, J) = \{A_{v_i,0}, \tilde{B}_{v_i,0}, A_{v_i,1}, \tilde{B}_{v_i,1}, \cdots, A_{v_i,J-1}, \tilde{B}_{v_i,J-1}\} \tag{3.1}$$

where $A_{v_i,k}$, $\tilde{B}_{v_i,k}$, and J are the k^{th} modification matrix, k^{th} modification vector ($0 \leq k < J$), and the number of the AUM pairs associated with v_i, respectively. The symbolic values of a variable var_r are represented in the r^{th} rows of AUM(v_i, J).

The number of AUM pairs associated with v_i solely depends on the number of different ways in which the actions of the edges leading to v_i modify variables. If the overall variable modifications of the actions of two paths leading to v_i are the same, only one AUM pair is sufficient to account for the effects of the actions in the two paths. Therefore, only unique AUM pairs are associated with v_i.

Example: 3 To describe the method of forming AUM pairs of a node, let us consider the EFSM of Figure 3.1. The AUM$(v_0, 1)$ is defined as:

$$A_{v_0,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_0,0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The AUM(v_1, J) associated with v_1 is based on the AUM$(v_0, 1)$ and the actions of the edges between v_0 and v_1 (i.e., e_0 and e_1). The first AUM pair of v_1 $(A_{v_1,0}, \tilde{B}_{v_1,0})$ is formed when e_0 is traversed. Because of the effects of the action of e_0 (i.e., $b = 1$), zeros are assigned to the elements of the 2^{nd} row of $A_{v_1,0}$ and 1 is assigned to the element of the 2^{nd} row of $\tilde{B}_{v_1,0}$.

$$A_{v_1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_1,0} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The values assumed by a, b, c, i, and *True* after e_0 is traversed can be determined from:

$$\tilde{V} = A_{v_1,0} * \tilde{V} + \tilde{B} \tag{3.2}$$

where $\tilde{V}(m\text{x}1)$ is the vector for the variables:

$$\begin{bmatrix} a \\ b \\ c \\ i \\ True \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \\ i \\ True \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Equation (3.2) yields a=a, b=*1*, c=c, i=i, and *True*=*True*, which implies that the value of b has changed while a, c, i, and *True* retain their values.

Similarly, due to the action associated with e_1, a new AUM pair $(A_{v_1,1}, \tilde{B}_{v_1,1})$ is formed for v_1:

$$A_{v_1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_1,1} = \begin{bmatrix} 0 \\ 10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where a=a, b=*10*, c=c, i=i, and *True*=*True*. Therefore, the two AUM pairs AUM$(v_1, 2)$ = $\{A_{v_1,0}, \tilde{B}_{v_1,0}, A_{v_1,1}, \tilde{B}_{v_1,1}\}$ formed for v_1 are:

$$A_{v_1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_1,0} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A_{v_1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_1,0} = \begin{bmatrix} 0 \\ 10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To form the AUM pairs for v_2, the action of e_2 (i.e., i=0) is applied to AUM(v_1,2). = $\{A_{v_1,0}, \tilde{B}_{v_1,0}, A_{v_1,1}, \tilde{B}_{v_1,1}\}$. As a result, two AUM pairs AUM(v_2,2) = $\{A_{v_2,0}$, $\tilde{B}_{v_2,0}$, $A_{v_2,1}, \tilde{B}_{v_2,1}\}$ are produced:

$$A_{v_2,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,0} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A_{v_2,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,1} = \begin{bmatrix} 0 \\ 10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In general, after traversing an edge $e_k = (v_i, v_j)$, new AUM pairs are formed for v_j by

applying the actions of e_k to the AUM(v_i, J). Depending upon the net modification on AUM(v_i, J) by the actions of e_k, the number of new AUM pairs formed for v_j may be less than or equal to that of v_i. This example will be continued later in this section and in Section 3.4.□

3.3.1 Detection of Action Inconsistencies:

The AUM pairs of a node are instrumental in the detection and elimination of action inconsistencies. Multiple AUM pairs at a node v_i indicate that certain variables are modified differently in the paths leading to v_i and may potentially cause action inconsistencies. Hence, the conditions of reachable edges from v_i must be investigated to decide if an inconsistency exists. The result may indicate that none of the conditions of the reachable edges from v_i uses the variables modified differently in the paths leading to v_i. In this case, there will be no action inconsistencies.

In this dissertation, a two-phase MBF graph traversal is designed to handle the detection of the action inconsistencies. For convenience, the two phases of the MBF graph traversal will be referred to as the P1-MBF and P2-MBF. Phase one of the MBF graph traversal, P1-MBF, can be viewed as the main graph traversal from which the P2-MBF, shown in Figure 3.4, may be invoked multiple times.

The main difference between the P1-MBF graph traversal and the conventional BF graph traversal can be described as follows. In the conventional BF graph traversal, all nodes of a distance k from the initial node v_0 are visited before any node of a distance $k+1$ from v_0 is visited. On the other hand, in the P1-MBF graph traversal, the traversal of $E^{exit}_{Loop_{v_i}}$ and the outgoing edges of

the $V^{Loop_{v_i}}$ are postponed until each edge in the sets ($E^{out}_{Loop_{v_i}}$ - $E^{exit}_{Loop_{v_i}}$) and ($E^{in}_{Loop_{v_i}} - E^{reachable}_{Loop_{v_i}}$) is traversed, respectively. During the P1-MBF graph traversal, the analysis of a loop (except for nested loops) which can be accessed through a yet to be traversed edge is avoided. Upon traversing an edge $e_k = (v_x, Loop_{v_j})$, it is checked if

$$\exists\ e_r = (v_x, v_y) \notin E^{out(T)}_{v_y} : (Loop_{v_j} \in V^{reachable}_{v_y}) \wedge (v_x, Loop_{v_j} \notin V^{Loop_{v_n}}) \tag{3.3}$$

The traversal of the edges of $E^{out}_{Loop_{v_j}}$ is postponed while (3.3) is satisfied. As can be seen from (3.3), an inner loop may be analyzed before traversing certain incoming edges of the outer loop entry/exit node.

In the P1-MBF graph traversal, upon traversing an edge $e_k = (v_x, v_y)$, it is checked if an action inconsistency between the edges in the paths leading to $v_y \in V$ and an edge $e_r \in E$: (v_w, v_z) reachable from v_y exists. The detection of action inconsistencies starts with checking the number of AUM pairs associated with v_y. If there is more than one, the effects of AUM(v_y, J) on the conditions of $E^{reachable}_{v_y}$ must be analyzed. Such analysis becomes complicated when the paths between v_y and v_w contain loops.

When a node with multiple AUM pairs is visited by using the P1-MBF, the P2-MBF graph traversal is initiated. P2-MBF aims to postpone the analysis of the effects of the AUM(v_y, J) on the conditions of $e_i \in E^{reachable}_{v_y}$, where there is a loop in the path(s) between v_y and tail(e_i), until this loop is completely analyzed. The P2-MBF graph traversal is designed to cope with the difficulty of analyzing the effects of the AUM(v_y, J) on the conditions of $E^{reachable}_{v_y}$ by traversing each edge of a loop body at most once. As a result, the traversal of the edges $E^{exit}_{Loop_{v_i}}$ is postponed until the loop whose entry/exit node is $Loop_{v_i}$ is completely analyzed by the P1-MBF graph

traversal. In addition, unlike the P1-MBF graph traversal, the loop is not advanced [40] during the P2-MBF graph traversal.

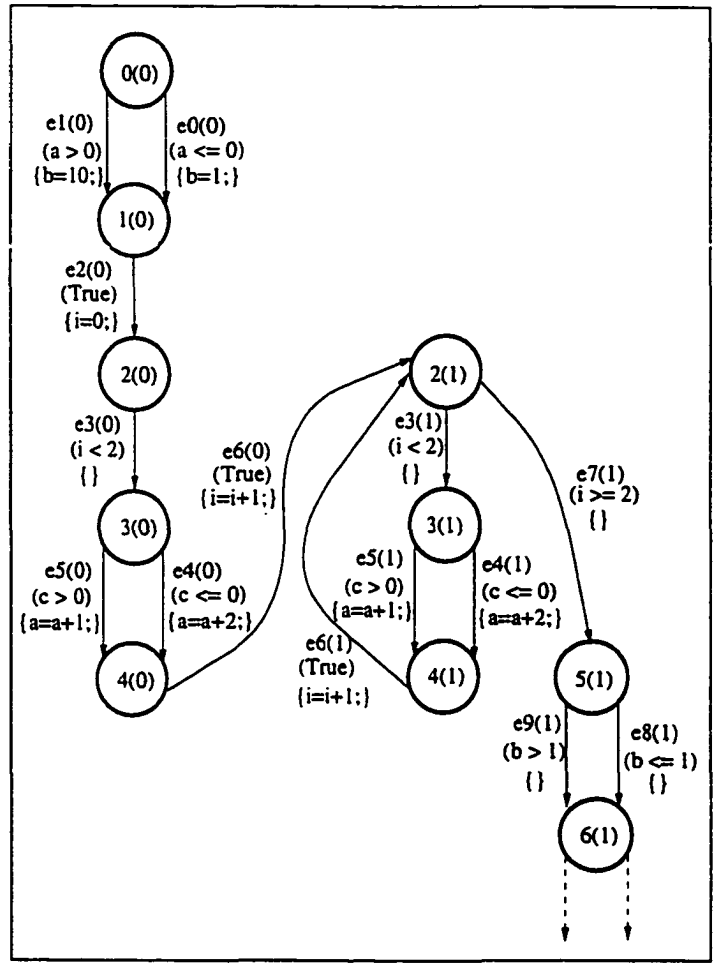

Figure 3.2: The EFSM graph of Figure 3.1 after the loop is advanced one iteration.

In the P2-MBF graph traversal, the traversal of a given path terminates if one of the following statements is true:

- an action inconsistency is detected
- the path contains a loop entry/exit node $Loop_{v_i}$ where

- all edges in $(E^{out}_{Loop_{v_i}} - E^{exit}_{Loop_{v_i}})$ are traversed or
- another entry/exit node $Loop_{v_j} \in V^{Loop_{v_i}}$ is visited by traversing an edge $e_k = (v_x, Loop_{v_j})$

The action inconsistency detection algorithm is given in Figure 3.3. The algorithm stops after finding the first action inconsistency. If there is no action inconsistency, the algorithms stops upon completing the traversal of the graph in the MBF manner.

3.3.2 Elimination of Action Inconsistencies

The next step after detection of an action inconsistency is to eliminate it. As part of the action inconsistency detection and elimination process, a graph splitting technique based on symbolic execution is used. For the variable interdependency analysis to be accomplished, the symbolic values of each variable at v_i are collected.

The elimination of an action inconsistency between two edges $e_k = (v_x, v_y)$ and $e_r = (v_w, v_z)$ of a loop-free graph is accomplished with the following algorithm. In this algorithm, the elimination of the action inconsistency between e_k and e_r is accomplished by placing these two edges in two separate subgraphs to prevent these two edges from being included in the same path. The two subgraphs are formed by splitting nodes and edges of $V^{reachable}_{v_y}$ and $E^{reachable}_{v_y}$ such that each subgraph contains either e_k or e_r. Figure 3.5 depicts the action inconsistency elimination algorithm.

The action inconsistency elimination algorithm shown in Figure 3.5 aims to avoid the creation of unnecessary duplicates of nodes and edges. For example, the effects of the AUM pairs on

Action Inconsistency Detection

input: (G, E);

output: $E^{conflict}$;

$\forall\, v_i \in V$ set $E_{v_i}^{delayed} = \emptyset$; $E_{v_i}^{out(T)} = \emptyset$; $T^{v_i} = 0$; $v_0 = v_i$; P1_DONE = False;

```
while (NOT P1_DONE){
    if (v_i == Loop_{v_i}){
        E_{v_i}^{delayed} = E_{Loop_{v_i}}^{exit};
    }
    while ((E_{v_i}^{out} - (E_{v_i}^{out(T)} ∪ E_{v_i}^{delayed})) ≠ ∅){
        traverse e_k ∈ (E_{v_i}^{out} - (E_{v_i}^{out(T)} - E_{v_i}^{delayed}));
        E_{v_i}^{out(T)} = E_{v_i}^{out(T)} ∪ {e_k};
        E_{tail(e_k)}^{in(T)} = E_{tail(e_k)}^{in(T)} ∪ {e_k};
            if (tail(e_k) == Loop_{v_j})
                if (head(e_k) ∈ V^{Loop_{v_j}}){
                    advance (Loop_{v_i});
                    exit();
                }
                else{
                    update AUM(tail(e_k), J);
                    if (new AUM pairs are formed for tail(e_k){
                        P2_MBF(v_i, V_{v_i}^{dif_modified});
                    }
                    if (∃ e_r = (v_x, v_y) ∉ E_{v_y}^{out(T)} : (Loop_{v_j} ∈ V_{v_y}^{reachable}) ∧ (v_x, Loop_{v_j} ∉ V^{Loop_{v_n}})){
                        if ((E_{tail(e_k)}^{in} - (E_{tail(e_k)}^{in(T)} ∪ {e_r = (v_x, Loop_{v_j}) : v_x ∈ V^{Loop_{v_j}}})) == ∅){
                            TailOfQueue2 = tail(e_k);
                        }
                    }
                }
            else if (T^{tail(e_k)} == 0){
                TailOfQueue1 = tail(e_k);
                    T^{tail(e_k)} = 1;
                    update AUM(tail(e_k), J);
            }
            else{
                    update AUM(tail(e_k), J);
                if (new AUM pairs are formed for tail(e_k){
                        P2_MBF(v_i, V_{v_i}^{dif_modified});
                    }
            }
    }
    if (HeadOfQueue1 == Null){
        if (HeadOfQueue2 == Null){
            P1_DONE = True;
        }
        else{
            v_i = HeadOfQueue2;
        }
    }
    else{
        v_i = HeadOfQueue1;
    }
}
```

Figure 3.3: Action inconsistency detection algorithm - P1-MBF.

P2-MBF Graph Traversal

input: v_x, $V_{vx}^{dif-modified}$;
output: $E^{conflict}$;
$\forall\, v_i \in V, E_{v_i}^{out(T)} = E^{conflict} = \emptyset, T^{v_i} = 0, v_i = v_x$;
$V^{dif-modified} = V_{v_i}^{dif-modified}$; P2_DONE = False;
while $(NOT\ P2\_DONE)\{$
  while $((E_{v_i}^{out} - (E_{v_i}^{out(T)} \cup E_{v_i}^{delayed})) \neq \emptyset)\{$
    traverse $e_k \in (E_{v_i}^{out} - (E_{v_i}^{out(T)} - E_{v_i}^{delayed}))$;
    $E_{v_i}^{out(T)} = E_{v_i}^{out(T)} \cup \{e_k\}$;
    $E_{tail(e_k)}^{in(T)} = E_{tail(e_k)}^{in(T)} \cup \{e_k\}$;
    if $(T^{tail(e_k)} == 0)\{$
      TailOfQueue = $tail(e_k)$;
      $T^{tail(e_k)} = 1$;
    $\}$
    $E^{check} = E_{tail(e_k)}^{out}$;
    while $(E^{check} \neq \emptyset)\{$
      select $e_c \in E^{check}$;
      $E^{check} = E^{check} - \{e_c\}$;
      if $((V_{e_c}^{con-used} \cap V^{dif-modified}) \neq \emptyset)\{$
        if (e_c is infeasible)$\{$
          $E^{conflict} = \{e_c \cup e_k\}$
          P2_DONE = True;
        $\}$
      $\}$
    $\}$
  $\}$
  if (HeadOfQueue == Null)$\{$
    P2_DONE = True;
  $\}$
  else$\{$
    v_i = HeadOfQueue;
    if $(v_i == Loop_{v_i})\{$
      $E_{v_i}^{delayed} = E_{Loop_{v_i}}^{exit}$;
    $\}$
  $\}$
$\}$
return $E^{conflict}$;

Figure 3.4: P2-MBF algorithm.

the outgoing edges of v_w due to the incoming edges of v_y are carefully analyzed to prevent unnecessary future graph splits. Only the edges of $E^{in}_{v_y} = (E^T_{v_y} \cup E^{NT}_{v_y})$ that do not conflict with the edges in $E^{out}_{v_w}$ should be duplicated during the splitting. During the graph splitting, it is not apparent if placing copies of the edges in $E^{NT}_{v_y}$ in the same subgraph with $e_k = (v_x, v_y)$ will cause action inconsistencies. Copies of such edges are temporarily included in the same subgraph with e_k.

The effects of the actions of $E^{depend}_{v_w}$ on the outgoing edges of $v_w \in V^{reachable}_{tail(e_k)}$ must be analyzed later when each edge in $E^{depend}_{v_w}$ is traversed, where $E^{depend}_{v_w}$ is the set of edges temporarily included in the subgraph containing e_k. When $e'_{r(s^*)} \in E^{depend}_{v'_{w(s^*)}}$ is traversed, it is checked if the actions of $e'_{r(s^*)}$ are inconsistent with the conditions of the outgoing edges of $v'_{w(s^*)}$. If they are, $e'_{r(s^*)}$ is removed from the graph. Recall that $v'_{w(s^*)}$ is one of the copies of $v_{w(s)}$ whose outgoing edges conflicted with $e_{k(s)}$.

The graph splitting is slightly different when the tail(e_k) is a loop entry/exit node as will be described later.

Since the graph topology changes with the creation of the new edges and nodes, the action inconsistency detection algorithm restarts after each graph splitting. As defined in Section 2.1.2, a path from v_i to v_j should not include v_0. Hence, the inconsistency removal algorithm does not split v_0.

Action Inconsistency Elimination

input: $e_k = (v_x, v_y)$ and $e_r = (v_w, v_z)$, where an action inconsistency exists between e_k and e_r

goal: to eliminate the inconsistency between e_k and e_r by creating two subgraphs such that each subgraph contains either e_k or e_r, but not both

begin

Split $v_{y(s)}$ into two nodes as: $v'_{y(s)}$ and $v'_{y(s^*)}$;

Split $\forall\ v_{i(s)} \in V^{reachable}_{v_{y(s)}}$ into two nodes as:

$v'_{i(s)} \in V^{reachable}_{v'_{y(s)}}$ and $v'_{i(s^*)} \in V^{reachable}_{v'_{y(s^*)}}$;

Split $\forall\ e_{m(s)} = (v_{i(s)}, v_{j(s)}; L_{e_{m(s)}}) \in E^{reachable}_{v_{y(s)}}$ into two edges as:

$e'_{m(s)} = (v'_{i(s)}, v'_{j(s)}; L_{e_{m(s)}}) \in E^{reachable}_{v'_{y(s)}}$ and

$e'_{r(s^*)} = (v'_{i(s^*)}, v'_{j(s^*)}; L_{e_{m(s)}}) \in E^{reachable}_{v'_{y(s^*)}}$;

Split $\forall\ e_{m(s)} = (v_{i(s)}, v_{y(s)}; L_{e_{m(s)}}) \in E^{in(T)}_{v_{y(s)}}$ such that

$\text{Cons}(e_{m(s)}, e_{k(s)}, v_{w(s)}) = 1$, into two edges as:

$e'_{m(s)} = (v_{i(s)}, v'_{y(s)}; L_{e_{m(s)}}) \in E^{in}_{v'_{y(s)}}$ and

$e'_{m(s^*)} = (v_{i(s)}, v'_{y(s^*)}; L_{e_{m(s)}}) \in E^{in}_{v'_{y(s^*)}}$ where $v_{w(s)}$ is the node whose outgoing edges are inconsistent with $e_{k(s)}$;

Create a duplicate of $\forall\ e_{m(s)} = (v_{i(s)}, v_{y(s)}; L_{e_{m(s)}}) \in E^{in(T)}_{v_{y(s)}}$ such that

$\text{Cons}(e_{m(s)}, e_{k(s)}, v_{w(s)}) = 0$ as:

$e'_{m(s)} = (v_{i(s)}, v'_{y(s)}; L_{e_{m(s)}}) \in E^{in}_{v'_{y(s)}}$;

Split $\forall\ e_{m(s)} = (v_{i(s)}, v_{y(s)}; L_{e_{m(s)}}) \in E^{in(NT)}_{v_{y(s)}}$ into two edges as:

$e'_{m(s)} = (v_{i(s)}, v'_{y(s)}, L_{e_{m(s)}}) \in E^{in(NT)}_{v'_{y(s)}}$ and

$e'_{m(s^*)} = (v_{i(s)}, v'_{y(s^*)}, L_{e_{m(s)}}) \in E^{depend}_{v'_{w(s^*)}}$;

end

Figure 3.5: Action inconsistency elimination algorithm.

3.3.3 Elimination of Action Inconsistencies from the EFSM Graphs with Loops

The basic concepts used to eliminate inconsistencies from the loop-free graphs are extended to handle the removal of inconsistencies from the EFSM graphs with loops. As for the loop-free

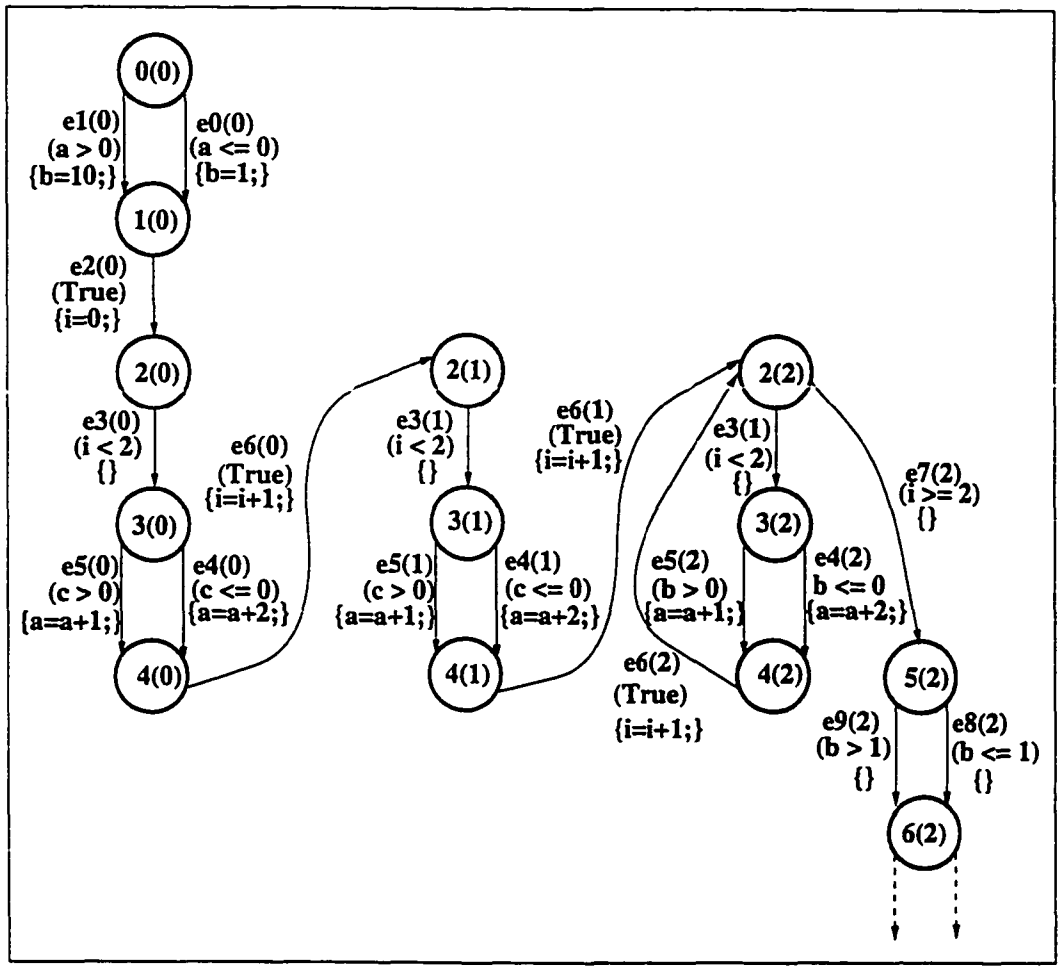

Figure 3.6: The EFSM graph of Figure 3.1 after the loop is advanced two iterations.

graphs, the action inconsistency detection is performed by using a two-phase MBF graph traversal. For simplicity, only loops with single entry/exit nodes are considered in this dissertation. As mentioned before, the exit criteria for all the loops considered in this study can be reached in a finite number of steps.

Let us suppose that in a graph with loops, when $e_k = (v_x, v_y)$ is traversed an action inconsistency is detected with one of the edges in $E_{v_y}^{reachable}$. One of the following cases is true:

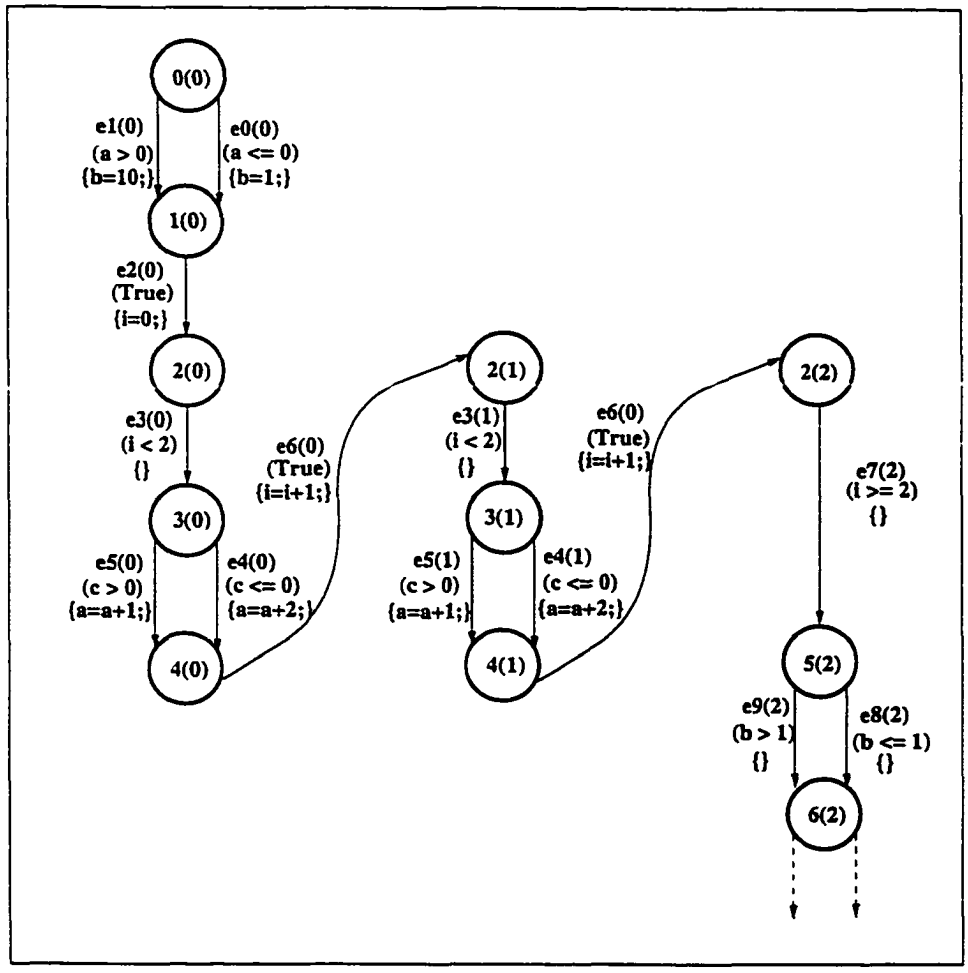

Figure 3.7: The EFSM graph of Figure 3.6 after the infeasible outgoing edge of $v_{2(2)}$ is removed.

- **Case 1:** $v_x \notin V^{Loop_{v_i}}$.
- **Case 2:** $v_x, v_y \in V^{Loop_{v_i}}$ and $y \neq i$ (i.e., tail(e_k) is not the loop entry/exit node).
- **Case 3:** $v_x, v_y \in V^{Loop_{v_i}}$ and $y = i$ (i.e., tail(e_k) is the loop entry/exit node).

The method of graph splitting for Cases 1 and 2 is similar to that of the loop-free graphs except that if $Loop_{v_i}$ is split, a copy of each edge $e_r \in E^{out}_{Loop_{v_i}}$ is included in all new subgraphs. For Case 3, the following steps are taken to eliminate action inconsistencies:

1. The loop is advanced one iteration by duplicating all nodes of $V^{reachable}_{Loop_{v_i}}$ and the edges of $E^{reachable}_{v_y}$ as: $V^{reachable}_{v_{y(s)}}$, $V^{reachable}_{v_{y(s^*)}}$, $E^{reachable}_{v_{y(s)}}$, and $E^{reachable}_{v_{y(s^*)}}$, respectively. Since a loop must have only one entry node, $\forall\ e_r = (v_x, v_y) \in E^{in}_{Loop_{v_i}}$: $v_x \notin V^{Loop_{v_i}}$ are not duplicated.

2. The AUM pairs of $Loop_{v_i}$ are not updated.

3. The ending node of $e_{k(s)}$ is changed to $Loop'_{v_{i(s^*)}}$.

4. To determine the exit condition(s) of the loop, the feasibility of the conditions of each edge in $E^{out}_{Loop'_{v_{i(s)}}}$ is investigated. If the loop exit criterion is satisfied, the conditions of the edges in $(E^{out}_{Loop'_{v_{i(s)}}} - E^{exit}_{Loop'_{v_{i(s)}}})$ become infeasible. Otherwise, the conditions of edges in $E^{exit}_{Loop'_{v_{i(s)}}}$ are infeasible. An edge whose condition is found to be infeasible is removed from the graph.

Similar to the case of loop-free graphs, the above steps are repeated after each graph split.

Let us introduce the following definitions for any node $v_i \in V$:

- $\text{AUM}(v_i, J)[v_i \overset{SE}{\rightsquigarrow} v_x]$: the resulting $\text{AUM}(v_x, J)$ after applying symbolic execution on $\text{AUM}(v_i, J)$ in the paths between v_i and $v_x \in V$.
- $E^{conf(e_x \in E^{in}_{v_i},\ e_r)}_{v_i} \subseteq E^{in(T)}_{v_i}$: set of traversed incoming edges of v_i, where each edge conflicts with e_r after obtaining $\text{AUM}(v_i, J)[v_i \overset{SE}{\rightsquigarrow} head(e_r)]$.
- $E^{conf(e_x \in E^{out}_{e_k \rightsquigarrow e_r},\ e_r)}_{e_k \rightsquigarrow e_r}$: set of edges in the paths from $tail(e_k)$ to $head(e_r)$, where each edge conflicts with e_r after obtaining $\text{AUM}(tail(e_k), J)[tail(e_k) \overset{SE}{\rightsquigarrow} head(e_r)]$.

- $E^{inf(v_i,head(e_r))}_{v_i \rightsquigarrow head(e_r)}$ set of edges in the paths between v_i and $head(e_r)$ whose conditions are infeasible after obtaining AUM$(v_i, J)[v_i \overset{SE}{\rightsquigarrow} head(e_r)]$.

After an action is detected and eliminated, the resulting graph $G'(V', E')$ is characterized as:

$$V' = (V - V^{reachable}_{v_{y(s)}} - \{v_{y(s)}\}) \cup V'_I \cup V'_{II}, \text{ where} \tag{3.4}$$

$$V'_I = (\{v'_{y(s)}\} \cup V^{reachable}_{v'_{y(s)}}) \text{ and } V'_{II} = (\{v'_{y(s^*)}\} \cup V^{reachable}_{v'_{y(s^*)}})$$

$$E' = (E - E^{reachable}_{v_{y(s)}} - E^{in}_{v_{y(s)}}) \cup E'_I \cup E'_{II}, \text{ where} \tag{3.5}$$

$$E'_I = (E^{reachable}_{v'_{y(s)}} - E^{inf(v'_{y(s)}, tail(e_r))}_{v'_y \rightsquigarrow tail(e_r)}) \cup E^{in(NT)}_{v'_{y(s)}} \cup (E^{in(T)}_{v'_{y(s)}} - E^{conf(e_x \in E^{in}_{v'_{y(s)}}, e_r)}_{v_{y'(s)}}) \text{ and}$$

$$E'_{II} = (E^{reachable}_{v'_{y(s^*)}} - E^{conf(e_x \in E^{out}_{e_k \rightsquigarrow e_r}, e_r)}) \cup E^{in}_{v'_{y(s^*)}} \cup (\{e_k\} \cup E^{depend}_{v'_{w(s^*)}})$$

Due to the removal of edges with infeasible conditions from $G'(V', E')$, unreachable subgraphs may result. A simple DF graph traversal can be used to eliminate unreachable subgraphs from $G'(V', E')$. Furthermore, if the condition of an edge e_i cannot be satisfied due to the action of another edge e_j, it is removed from the graph as will be described in Section 3.4.

Example 3 (Continued): The EFSM graph of Figure 3.1 will be used to demonstrate the process of detecting and eliminating inconsistencies in EFSM models. The main graph

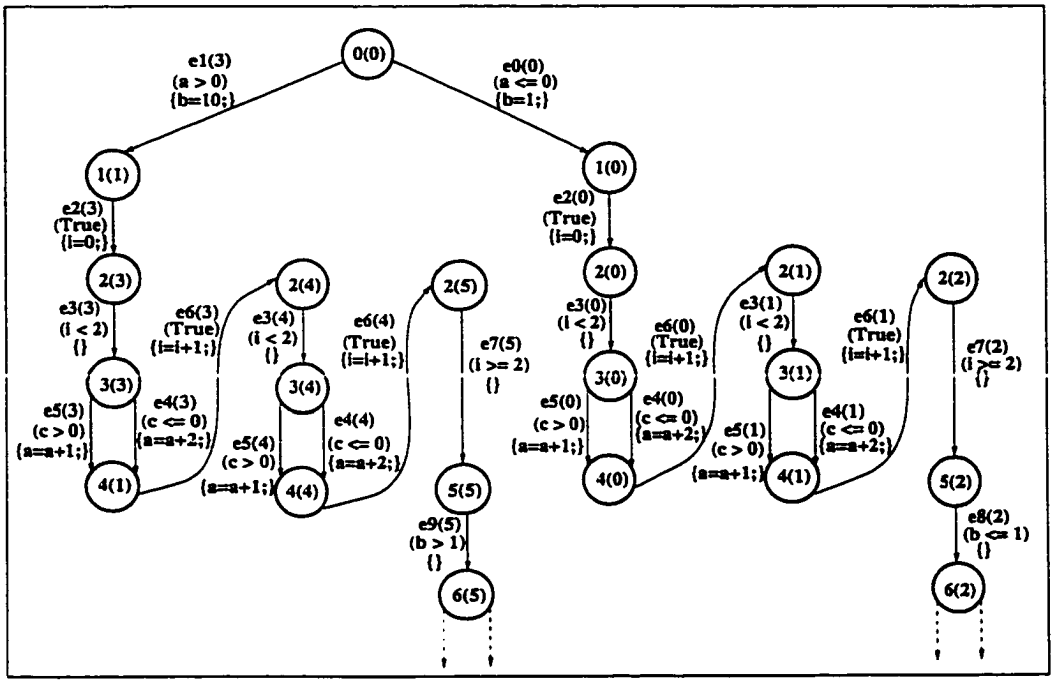

Figure 3.8: The resulting EFSM graph after splitting the graph of Figure 3.7 due to e_1 action.

traversal is performed by using the P1-MBF from which the P2-MBF may be invoked multiple times.

As described in Section 3.3, the following two AUM pairs are created for v_1 after the outgoing edges of v_0 are traversed:

$$A_{v_1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \hat{B}_{v_1,0} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A_{v_1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \bar{B}_{v_1,1} = \begin{bmatrix} 0 \\ 10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

which indicates that the two incoming edges of v_1 modify differently the variable b. Hence, the P2-MBF graph traversal, where v_1 is considered as the starting node, is initiated. P2-MBF traverses the edge of e_2, e_3, e_4, e_5 and e_6. Since these edges do not use b in their conditions explicitly or implicitly, the P2-MBF terminates. Notice that, during the P2-MBF graph traversal of Figure 3.1, e_7 is not traversed since it is in $E^{exit}_{Loop_{v_2}}$.

The P1-MBF continues until the loop is completely analyzed without detecting any inconsistencies as shown in Figures 3.2 through 3.7. As stated earlier, the inconsistency detection and elimination algorithms restart after each graph split. The P2-MBF is invoked again upon visiting v_1 of Figure 3.7. The P2-MBF graph traversal proceeds until $v_{5(2)}$ is visited. At this point, an action inconsistency is detected between the edges leading to $v_{1(0)}$ and the outgoing edges of $v_{5(2)}$. As a result, the graph of Figure 3.7 is split as shown in Figure 3.8 to prevent the conflicting edges to be accessible from one another.

At this point all action inconsistencies due to the variables modified differently in the paths leading to a node are eliminated from the EFSM graph. However, as can be seen from Figure 3.8, there are condition inconsistencies in the resulting EFSM graph (e.g., the edges of $e_{4(0)}$ and $e_{5(1)}$ conflict since their conditions require that $c \leq 0$ and $c > 0$, respectively). This example will be will be further analyzed in Section 3.4 where the

detection and elimination of condition inconsistencies are addressed.□

3.4 Condition Inconsistencies

In this section, the detection and elimination of condition inconsistencies (if any) is pursued, which is the next step after the action inconsistencies are eliminated from the EFSM model.

A test sequence generated from an EFSM should avoid including two or more edges with conflicting conditions. The conditions of the edges of a test sequence constitute a system of constraints. When an edge $e_k = (v_i, v_j)$ is included in the test sequence, a new set of constraints is formed. Hence, the condition of e_k, together with the edge conditions already included in the sequence, determine not only the feasibility of the sequence but also whether other outgoing edges of $V_{v_j}^{reachable}$ can be included in the same test sequence. For example, a test sequence generated from the EFSM graph of Figure 3.8, which contains $e_{4(0)}$ and $e_{5(1)}$ is infeasible since such a test sequence would require that $(c \leq 0)$ and $(c > 0)$ simultaneously.

Algorithms available for solving linear programming problems can be used in deciding whether a certain path predicate is feasible [70]. Let us first briefly describe the linear programming problem.

A linear programming problem consists of:

- An objective function that needs to be optimized.
- A set of decision variables whose values are to be found.
- A set of constraints that the values for the decision variables should satisfy.

Formally a linear programming problem is defined as:

$$\begin{aligned} &\text{Minimize} \quad M\mathbf{x} \text{ subject to} \\ &A\mathbf{x} = \mathbf{b}, \text{ where} \\ &\mathbf{x} \geq 0 \end{aligned} \tag{3.6}$$

where A, $\mathbf{b}$, M and $\mathbf{x}$ are the coefficient matrix for the set of the constraints, the vector containing the constants associated with the constraints, the coefficient matrix for the objective function, and the vector representing the decision variables, respectively. Notice that the operators of (3.6) can be easily changed to the form of $\leq$ or $\geq$.

The simplex algorithm can be used to decide if a set of linear constraints can be satisfied simultaneously [1, 70]. The algorithm consists of two phases. The first phase starts with a *basic feasible solution* and determines if an optimal solution for the original linear programming exists. Depending upon certain properties of the solution found in phase one, the feasibility of the original problem is determined. If the problem is found to have a solution, phase two of the simplex algorithm is initiated to find an optimal solution.

In this dissertation, the existence of a condition inconsistency between two edges of an EFSM graph is determined by formulating a simplified linear programming problem, where the cost function is skipped. The cost function of the simplex algorithm is not significant if the aim is to decide whether the traversal of a path is feasible. Therefore, phase one of the simplex algorithm is sufficient to achieve the goal of detecting condition inconsistencies.

Since the edge conditions may have different operator, certain modifications may be needed before the linear programming problem, formed by the edge conditions of a path, is solved. All

operators of the conditions should have one of the operators of $\leq$, $\geq$, or $=$. For example, the modifications needed to convert all operators to $\leq$ can be summarized as:

1. If the operator of the condition is $=$, a very small real number δ is added to the constant of the condition.
2. If the operator of the condition is $<$, δ is subtracted from the constant of the condition.
3. If the operator of the condition is $\geq$, all the coefficients and the constant of the condition are multiplied by -1.
4. If the operator of the condition is $>$, all the coefficients and the constant of the condition are multiplied by -1 and δ is added to the constant of the condition.
5. If the operator of the condition is $\neq$, the actions of steps 2 and 4 should be done since the constraint may have a solution in one of the two regions specified by the operators $<$ and $>$.

In Section 3.3 edge actions were represented as matrices. In a similar fashion, the edge conditions in a path from the starting node v_0 to a node v_i can be represented in matrices. A triplet of matrices are defined as C $(m \times p)$, $\tilde{OP}$ $(p \times 1)$, and $\tilde{D}$ $(p \times 1)$, where m is the number of variables, p is the number of conditions in the path from v_0 to v_i, C is the coefficient matrix, $\tilde{OP}$ is the operator vector containing the relations of $=, <, >, !=, \cdots$, etc., and $\tilde{D}$ is the scalar vector containing the scalar values of the conditions in the path.

The AUM pairs discussed in Section 3.3 are applied to the edge conditions of the EFSM graph. A single condition of an edge $e_k = (v_i, v_j)$ is in the form of $\tilde{C} * \tilde{V}(\tilde{OP})\tilde{D}$. The condition of e_k will be

modified based on the symbolic values of the variables var_0 through v_{m-1}, which are represented by the AUM(v_i, J). As described in Section 3.3, the current values of the variables including all the modifications represented by an AUM pair of v_i are in the form of: $\tilde{V} = A_{i,k} * \tilde{V} + \tilde{B}_{i,k}$. Substituting $\tilde{V}$ values in an edge condition will result in $\tilde{C}(A_{i,k} * \tilde{V} + \tilde{B}_{i,k})(\tilde{OP})\tilde{D}$, which simplifies as $\tilde{E} * \tilde{V}(\tilde{OP})f$, where $\tilde{E} = \tilde{C} * A_{i,k}$ is an m-element vector and f is a scalar. During the DF graph traversal, an edge $e_k = (v_i, v_j)$ whose condition is infeasible based on the AUM pairs of v_i is deleted from the graph. The values assumed by the variables used in the condition of e_k can be determine from:

$$C * \tilde{V} = C * (A_{v_2,k} * \tilde{V} + \tilde{B}_{v_2,k}) \tag{3.7}$$

where C is the coefficient matrix for the condition of e_k and $0 \leq k < J$ for J is the number of AUM pairs associated with v_i.

The accumulated different conditions of the paths leading to v_i can be represented in a set of *Accumulated Condition Matrix* (ACM) triplets: ACM(v_i, J) = ($C_{v_i,0}$, $\tilde{OP}_{v_i,0}$, $\tilde{D}_{v_i,0}$, $C_{v_i,1}$, $\tilde{OP}_{v_i,1}$, $\tilde{D}_{v_i,1}, \cdots, C_{v_i,J-1}, \tilde{OP}_{v_i,J-1}$, $\tilde{D}_{v_i,J-1}$), where $C_{v_i,k}$, $\tilde{OP}_{v_i,k}$, $\tilde{D}_{v_i,k}$, and J are the k^{th} coefficient matrix, k^{th} operator matrix, k^{th} scalar value matrix ($0 \leq k < J$), and the number of the ACM triplets associated with v_i, respectively.

Example 3 (Continued): Let us describe the process of forming ACM triplets for the nodes of an EFSM graph. Suppose that the EFSM graph of Figure 3.1 is being traversed in a DF manner. The DF graph traversal of this graph starts with the edge of e_0, whose condition can be represented by the following ACM triplet:.

$$C = [1\ 0\ 0\ 0\ 0],\ \vec{OP} = [\leq],\ \tilde{D} = [0].$$

The *True* condition of e_2, which is the next edge to be traversed in the DF manner, is represented by the triplet of:

$$C = [0\ 0\ 0\ 0\ 1],\ \vec{OP} = [=],\ \tilde{D} = [1].$$

Hence, the ACM$(v_2, 1)$ formed by traversing the edges of e_0 and e_2 is constructed by appending the two triplets associated with these two edges as shown below:

$$C_{v_2,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \vec{OP}_{v_2,0} = \begin{bmatrix} \leq \\ = \end{bmatrix} \quad \tilde{D}_{v_2,0} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The triplets associated with the condition of e_3 is appended to ACM$(v_2, 1)$ to form ACM$(v_3, 1)$ when e_3 is traversed:

$$C_{v_3,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \vec{OP}_{v_3,0} = \begin{bmatrix} \leq \\ = \\ < \end{bmatrix} \quad \tilde{D}_{v_3,0} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

which implies that $a \leq 0$, $True$=1, and $i < 2$.

As mentioned before, if an edge $e_r = (v_i, v_j)$ is found to be infeasible due to the AUM pairs of v_j, the edge is deleted from the graph. The feasibility of an edge can be determined from the AUM pairs of its *head* node. For example, the current values of the variables used in the condition of e_3 can be found by applying the two AUM pairs associated with v_2 on the condition of e_3 as defined by 3.7, where C is the coefficient matrix for the condition of e_3, k assumes the values of 0 through 1, and

$$A_{v_2,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,0} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A_{v_2,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,1} = \begin{bmatrix} 0 \\ 10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Using $A_{v_2,0}$ and $\tilde{B}_{v_2,0}$ in (3.7) gives:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ i \\ True \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ i \\ True \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right)$$

which simplifies as: $i = 0$.□

3.4.1 Detection of Condition Inconsistencies

The condition inconsistency detection is performed by traversing the graph in a DF manner. Two different approaches may be taken to detect and eliminate condition inconsistencies. In the first approach, all ACM triplets of a node v_y are collected and then each of ACM(v_y, J) is compared with the conditions of edges in $E_{v_y}^{out}$.

In an alternative approach, which is used in the algorithms of this dissertation, the condition inconsistencies can be detected and eliminated by focusing on the ACM triplets of a node v_y, ACM(v_y, J), one at a time. Let $\mathrm{DF}_{v_0 \rightsquigarrow v_y}$ be the path between v_0 and v_y traversed for the first time that v_y is visited in the DF graph traversal. Let ACM(e_r) be the triplet representing the condition of $e_r \in E_{v_y}^{reachable}$. Furthermore, let $VAR_{e_i}^{con-used} \in VAR$, where $e_i \in E$, be the set of variables used in the conditions of e_i. Once a node v_y is visited by traversing an edge $e_k = (v_x, v_y)$, each edge $e_r = (v_w, v_z)$, where $v_w \in V_{v_y}^{reachable}$, such that

$$VAR_{e_r}^{con-used} \cap VAR_{e_k}^{con-used} \neq \emptyset \tag{3.8}$$

is identified. The consistency between the conditions of e_k and e_r is then checked by appending ACM(e_r) to the ACM(v_y, p).

Let us introduce the following definition:

$$\mathrm{Feas}(\mathrm{ACM}(v_y,p),\, e_r) = \begin{cases} 1, & \text{if ACM}(v_y,p)\#\text{ACM}(e_r) \text{ has a solution} \\ 0, & \text{otherwise} \end{cases} \tag{3.9}$$

where # denotes appending the two ACMs.

If the new constraints formed by ACM(v_y, p)#ACM(e_r) has a solution, e_r is said to be *consistent*

with the edges whose conditions constitute ACM(v_y, p) and thus the condition of e_r is removed from the constraints to continue the condition inconsistency detection. However, if the set of constraints represented by ACM(v_y, p)#ACM(e_r) does not have a solution, e_r is *inconsistent* with the edges whose conditions constitute ACM(v_y, p). The condition inconsistency detection algorithm is presented in Figure 3.9. The algorithm stops when a condition inconsistency is found or, when there is no such inconsistency, upon the completion of the DF graph traversal.

3.4.2 Elimination of Condition Inconsistencies

The elimination of the condition inconsistency between two edges of $e_k = (v_x, v_y)$ and $e_r = (v_w, v_z)$ is accomplished by splitting the nodes $\{v_i \in V_{tail(e_k)}^{reachable} : v_i \rightsquigarrow v_w\}$, with their incoming and outgoing edges, into two subgraphs such that each subgraph contains either e_k or e_r but not both. The outgoing edges of the nodes $\{v_i \in V_{tail(e_k)}^{reachable} : v_i \rightsquigarrow v_w\}$ will be referred to as $E_{v_y \rightsquigarrow v_w}^{out}$. The algorithm given in Figure 3.10 eliminates the condition inconsistency between two edges $e_k = (v_x, v_y)$ and $e_r = (v_w, v_z)$.

During the inconsistency elimination, the consistency of the edges of $E_{v_y}^{NT}$ with the outgoing edges of v_w needs to be checked. Since the consistency of the edges of $E_{v_y}^{in(NT)}$ with $E_{v_w}^{out}$ cannot be decided during the graph splitting, copies of these edges are included temporarily in the same subgraph with $e_k = (v_x, v_y)$. The influence of $E_{v_w}^{depend}$ on the conditions of the outgoing edges of $v_w \in V_{tail(e_k)}^{reachable}$ must be analyzed later when each edge in $E_{v_w}^{depend}$ is traversed, where $E_{v_w}^{depend}$ is the set of edges which are temporarily included in the subgraph containing e_k. An edge $e'_{r(s^*)} \in E_{v'_{w'(s^*)}}^{depend}$ whose condition is found to be inconsistent with the condition of an outgoing

Condition Inconsistency Detection

begin
  input: $G(V, E)$
  output: E^{conf}
  goal: to detect a condition inconsistency
  $E^{conf} = \emptyset$;
  $v_x = v_0$;
  $V = V - \{v_x\}$;
  DONE_DF = False;
  while $((NOT\ DONE\_DF) \wedge (V \neq \emptyset))\{$
    while $((E^{out}_{v_x} \neq \emptyset) \wedge (NOT\ DONE\_DF))\{$
      traverse $e_k = (v_x, v_y) \in E^{out}_{v_x}$;
      $E^{out}_{v_x} = E^{out}_{v_x} - \{e_k\}$;
      if $((\exists\ e_r = (v_w, v_z) \in E^{reachable}_{v_y}$ where
        $(VAR^{con-used}_{e_k} \cap VAR^{con-used}_{e_r} \neq \emptyset) \wedge$
        $(\text{Feas}(\text{ACM}(v_y, p)e_r) = 0)))\{$
        $E^{conf} = \{e_k\} \cup \{e_r\}$;
        DONE_DF = True;
      }
    }
    select a new node $v_x \in V$ as determined by the DF;
    $V = V - \{v_x\}$;
  }
  return E^{conf};
end

Figure 3.9: Condition inconsistency detection algorithm.

edge of $v'_{w(s)}$ is removed from the graph.

After a condition inconsistency is detected and eliminated, the resulting graph $G'(V', E')$ is defined as:

$$V' = (V - \{v_i \mid v_i \in V^{reachable}_{v_{y(s)}} \wedge v_i \rightsquigarrow v_w\} - \tag{3.10}$$

Condition Inconsistency Elimination

input: $e_k = (v_x, v_y)$ and $e_r = (v_w, v_z)$, where a condition inconsistency exists between e_k and e_r

goal: to eliminate the inconsistency between e_k and e_r by creating two subgraphs such that each subgraph contains either e_k or e_r, but not both

begin

Split $v_{y(s)}$ into two nodes as: $v'_{y(s)}$ and $v'_{y(s^*)}$;

Split $\forall\, v_{i(s)} \in V^{reachable}_{v_{y(s)}} : v_i \rightsquigarrow v_w$ into two nodes as:

$v'_{i(s)} \in V^{reachable}_{v'_{y(s)}}$ and $v'_{i(s^*)} \in V^{reachable}_{v'_{y(s^*)}}$;

Split $\forall\, e_{m(s)} = (v_{i(s)}, v_{j(s)}; L_{e_{m(s)}}) \in E^{reachable}_{v_{y(s)}}$: $tail(e_{r(s)}) \rightsquigarrow v_w$ into two edges as:

$e'_{m(s)} = (v'_{i(s)}, v'_{j(s)}; L_{e_{m(s)}}) \in E^{reachable}_{v'_{y(s)}}$ and

$e'_{m(s^*)} = (v'_{i(s^*)}, v'_{j(s^*)}; L_{e_{m(s)}}) \in E^{reachable}_{v'_{y(s^*)}}$;

Split $\forall\, e_{m(s)} = (v_{i(s)}, v_{y(s)}; L_{e_{m(s)}}) \in (E^{in(T)}_{v_{y(s)}} - \{e_k\})$ into two edges as:

$e'_{m(s)} = (v_{i(s)}, v'_{y(s)}; L_{e_{m(s)}}) \in E^{in(T)}_{v'_{y(s)}}$ and

$e'_{m(s^*)} = (v_{i(s)}, v'_{y(s^*)}; L_{e_{m(s)}}) \in E^{in}_{v'_{w(s^*)}}$ where $v_{w(s)}$ is the node whose outgoing edges are inconsistent with $e_{k(s)}$;

Split $\forall\, e_{m(s)} = (v_{i(s)}, v_{y(s)}; L_{e_{m(s)}}) \in E^{in(NT)}_{v_{y(s)}}$ into two edges as:

$e'_{m(s)} = (v_{i(s)}, v'_{y(s)}, L_{e_{m(s)}}) \in E^{in(NT)}_{v'_{y(s)}}$ and

$e'_{m(s^*)} = (v_{i(s)}, v'_{y(s^*)}, L_{e_{m(s)}}) \in E^{depend}_{v'_{w(s^*)}}$;

end

Figure 3.10: Condition inconsistency elimination algorithm.

$\{v_{y(s)}\}) \cup V'_I \cup V'_{II}$, where

$$V'_I = (\{v'_{y(s)}\} \cup V^{reachable}_{v'_{y(s)}}) \text{ and } V'_{II} = (\{v'_{y(s^*)}\} \cup V^{reachable}_{v'_{y(s^*)}})$$

$$E' = (E - E^{out}_{v_y \rightsquigarrow v_w} - E^{in}_{v_{y(s)}}) \cup E'_I \cup E'_{II}, \text{ where} \tag{3.11}$$

$$E'_I = E^{in(NT)}_{v'_{y(s)}} \cup E^{in(T)}_{v'_{y(s)}} \cup E^{reachable}_{v'_{y(s)}} \text{ and}$$

$$E'_{II} = E^{depend}_{v'_{w(s^*)}} \cup E^{in(T)}_{v'_{y(s^*)}} \cup \{e_k\} \cup (E^{reachable}_{v'_{y(s^*)}} -$$
$$\{e_i \in E^{reachable}_{v'_{y(s^*)}} : \text{Feas}(\text{ACM}(v_y, p)\#\text{ACM}(e_i)) = 0\})$$

As for the action inconsistency detection and removal case, the algorithms restart after each graph split.

Example 3 (Continued): The action inconsistencies are eliminated from the EFSM graph prior to handling the condition inconsistencies as shown in Figures 3.2 through 3.8. Hence, the analysis of condition inconsistencies starts with the graph of Figure 3.8. Due to the edge conditions of the edges of $e_{4(0)}$, $e_{4(1)}$, $e_{5(0)}$, $e_{5(1)}$, $e_{4(3)}$, $e_{4(4)}$, $e_{5(3)}$, and $e_{5(4)}$, the graph shown in Figure 3.8 contains condition inconsistencies.

A DF graph traversal on the graph of Figure 3.8 begins with the edge of $e_{0(0)}$. The ACM triplet of $v_{1(0)}$ resulting from the traversal of $e_{0(0)}$ is ACM($v_{1(0)}$, 1):

[1 0 0 0 0] [≤] [0]

The conditions of all outgoing edges of $v_i \in V^{reachable}_{v_0}$ are consistent with the condition of e_0.

The DF graph traversal of Figure 3.8 proceeds with the edges of $e_{2(0)}$, $e_{3(0)}$, $e_{4(0)}$, $\cdots$. The ACM triplet resulting from the traversal of the path $e_{0(0)} \cdot e_{2(0)} \cdot e_{3(0)} \cdot e_{4(0)}$ is:

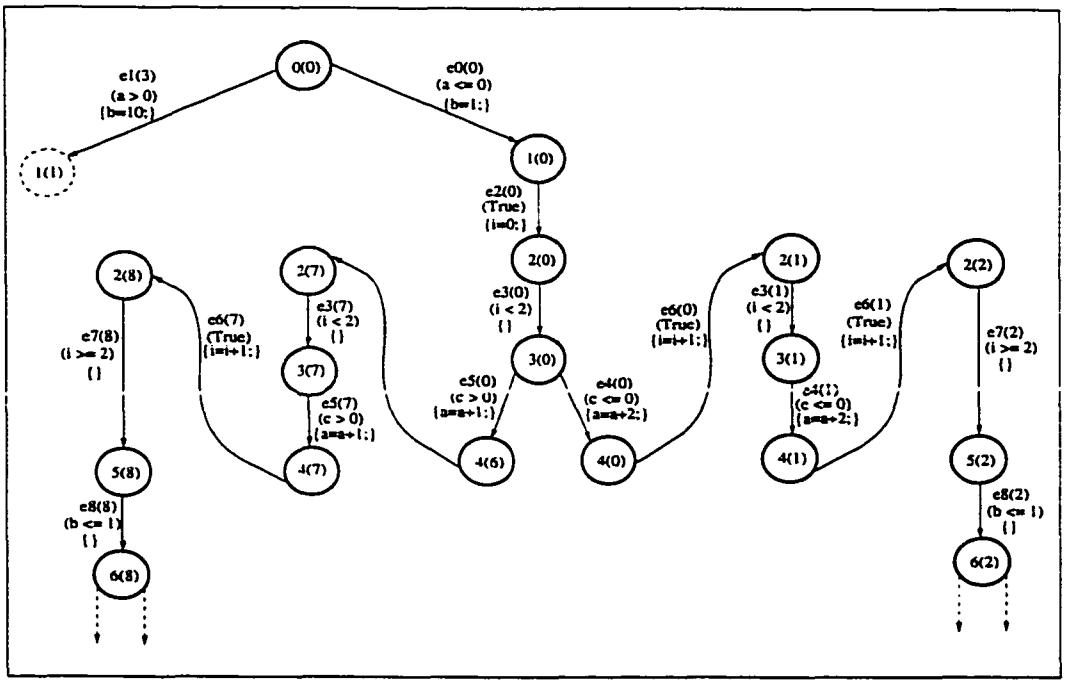

Figure 3.11: The EFSM graph after the graph of Figure 3.8 is split due to the condition of $e_{4(0)}$ (the subgraph starting from node $v_{1(0)}$ is shown).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} \leq \\ = \\ < \\ \leq \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

At this point of the graph traversal, it is found that the edges $e_{4(1)}$ and $e_{5(1)}$ use the variable c in their conditions, which is also used in the condition of $e_{4(0)}$. Hence, it is checked if each of the systems of constraints formed by appending ACM($e_{4(1)}$) or ACM($e_{5(1)}$) to ACM($v_{4(0)}, 1$) has a solution.

Since the system of constraints formed by ACM($v_{4(0)}, 1$)# ACM($e_{5(1)}$):

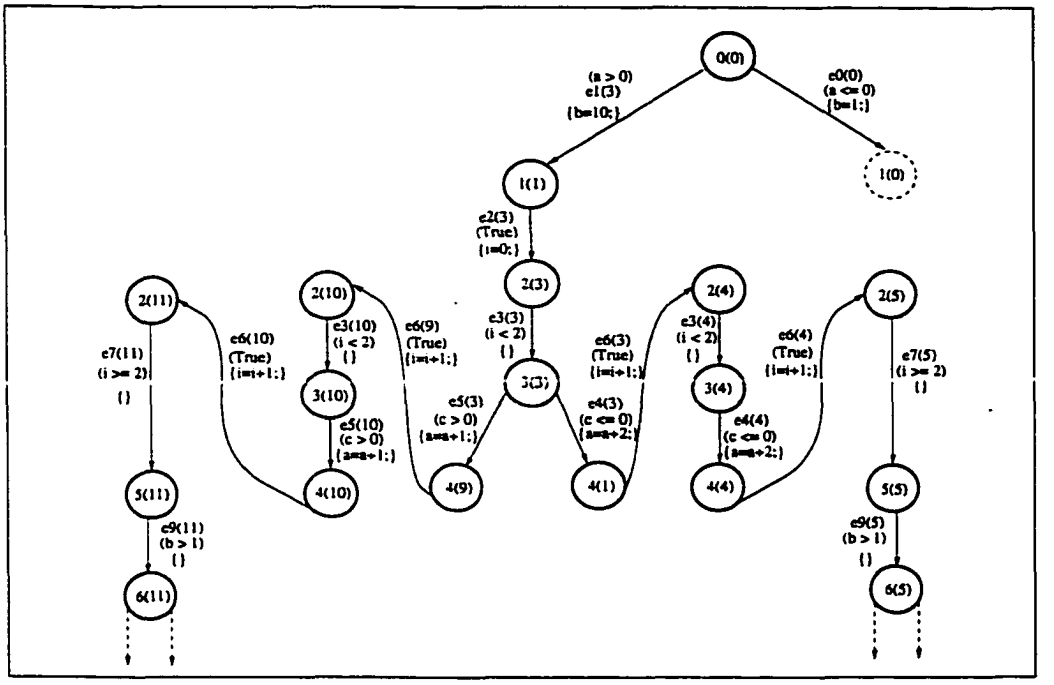

Figure 3.12: The EFSM graph after the graph of Figure 3.12a is split due to the condition of $e_{4(3)}$ (the subgraph starting from node $v_{1(1)}$ is shown).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} \leq \\ = \\ < \\ \leq \\ > \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

does not have a solution, there is a condition inconsistency between $e_{4(0)}$ and $e_{5(1)}$. To eliminate this inconsistency, the graph of Figure 3.8 is split such that $e_{4(0)}$ and $e_{5(1)}$ are placed into two separate subgraphs as shown in Figure 3.12a.

Similarly, when $e_{4(3)}$ of the graph of Figure 3.12a is traversed, the linear programming problem formed by $\text{ACM}(v_{4(3)}, 1)\#\text{ACM}(e_{5(4)})$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} > \\ = \\ < \\ \leq \\ > \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

has no feasible solution. As a result, the EFSM graph of Figure 3.12a is split to eliminate the condition inconsistency between $e_{4(3)}$ and $e_{5(4)}$ as described in the condition inconsistency removal algorithm. Figure 3.12b shows the final EFSM graph which contains no inconsistencies. Notice that the subgraph of Figure 3.12b which starts from the node $v_{1(0)}$ is identical to the subgraph starting from $v_{1(0)}$ of Figure 3.12a. □

3.5 Algorithm Implementation

The inconsistency detection and elimination algorithms presented in this thesis have been implemented as a software package written in C language. The size of the software package is about 12,000 lines of algorithmic C code (the package does not contain any graphical features). As its input, the software package reads a user specified file containing the description of an EFSM graph with the properties of the class of the EFSMs considered in this thesis. The output of the software package is a file, whose name is also provided by the user, where the final resultant EFSM graph is stored after all inconsistencies are eliminated.

Before the detection of inconsistencies, the software package checks if graph is strongly connected (i.e., each node is reachable from the others). First, the action inconsistencies are eliminated

from the graph (if any). Upon the detection and elimination of an action inconsistency, the software restarts the analysis, with the resultant graph being used as an input graph for the next iteration, to determine if there are other action inconsistencies remaining in the EFSM graph. This process continues until there is an iteration in which the MBF graph traversal terminates without detecting an action inconsistency.

The routine which handles the condition inconsistencies is invoked after all of the action inconsistencies are eliminated from the graph (if any). Similar to the action inconsistency detection and elimination process, the output EFSM graph obtained after detection and elimination of a condition inconsistency is used as an input for the next iteration.

Appendix A contains the formats used in the software package for the original EFSM graph input and the resultant EFSM graphs after each of the action and condition inconsistencies are eliminated.

3.6 Complexity of the Algorithms

The complexity of the action inconsistency detection and elimination is contributed by a two-phase MBF graph traversal and constructing the number of AUM pairs for each node, for each edge for each AUM pair.

The complexity for the two-phase MBF graph traversal is $(O(E^2)$. For each node v_i, the number of AUM pairs is $\sum_1^{|V|-1} |E^{v_j \rightarrow v_i}| \times |\mathrm{AUM}(v_j, J)|$ (where $|E^{v_j \rightarrow v_i}|$ is the number of edges from v_j to v_i) such that $\exists\ e_k = (v_j, v_i)$.

The complexity for the condition inconsistency detection and elimination is bounded by the number of AUM pairs of each node and executing the linear programming for each edge. Linear programming takes $\min(m^2, S^2)$ steps where m is the number of variables and S is the number of constraints [1].

Therefore, for the general case, the complexity of algorithms for handling the action inconsistencies is exponential with respect to the number of simple paths (i.e., the number AUM pairs). Similarly, the condition inconsistency elimination can be exponential with respect to the number of graph splits. However, based on our experience with several protocols (even with nested and/or concatenated loops), the complexity of both algorithms and, hence, the size of the consistent graph are bounded by the number of different values each condition variable assumes.

Chapter 4

Examples of EFSM Classes

The inconsistency detection and elimination algorithms presented in Chapter 3 are applied to several EFSM graphs with different topologies. The examples presented in the following sections are selected to represent the common EFSM models of the specifications given in high-level formal specification languages such as VHDL. Three types of EFSM graphs containing simple, nested, and concatenated loops [2, 5] are considered.

The loop body of a simple loop cannot include other loops whereas the loop body of a nested loop contains one or more loops. Two loops whose entry/exit nodes are $Loop_{v_i}$ and $Loop_{v_j}$ are said to be concatenated if:

$$\exists\, e_k : (e_k \in E^{exit}_{Loop_{v_i}}) \wedge (e_k \in E^{in}_{Loop_{v_j}}) \tag{4.1}$$

Each of the two concatenated loops can be a simple, nested, or concatenated to another loop.

As mentioned earlier, all loops (i.e., *while/for* loop constructs of high-level formal specification

languages) in this study are assumed to have single entry/exit nodes. Furthermore, syntactically endless loops [64] are not considered.

In the following discussion, the variable *True* does not appear in the AUM pairs of the nodes since it does not contribute to the formation of inconsistencies.

4.1 EFSM Graphs with Simple Loops

In this section, the process of eliminating inconsistencies from EFSM graphs with simple loops is considered. To illustrate the concepts, the inconsistency detection and elimination algorithms are applied to the EFSM graph of Figure 4.1.

The set of variables used in the edge actions and conditions is $VAR = \{a, b, i, j, x\}$. The graph contains action inconsistencies. For example, due to the action of e_0, the path consisting of the edges of e_0,e_2,e_3,e_5,e_6,e_3,e_5,e_6, and e_7 is infeasible.

The P1-MBF graph traversal of the graph in Figure 4.1 starts with the traversal of e_0 whose ending node is v_1. The action of e_0 assigns 2 to x. Therefore, at this point of the graph traversal v_1 has only one AUM pair (i.e., AUM$(v_1, 1) = \{A_{v_1,0}, \tilde{B}_{v_1,0}\}$:

$$A_{v_1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_1,0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

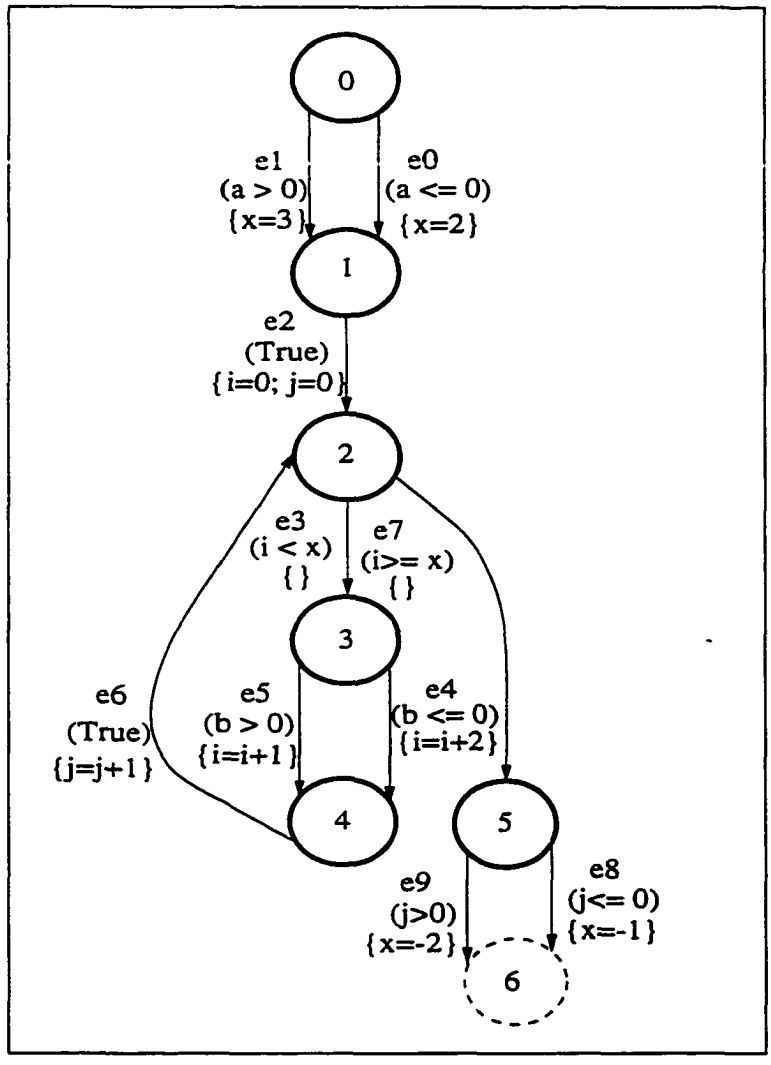

Figure 4.1: An EFSM graph with a loop.

Based upon the above AUM pair the values assumed by a, b, i, j, and x after e_0 is traversed are determined as:

$$\begin{bmatrix} a \\ b \\ i \\ j \\ x \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} a \\ b \\ i \\ j \\ x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

which results in $a = a,\ b = b,\ i = i,\ j = j$, and $x = 2$.

Since it is not known yet whether any of the variables $var_i \in VAR$ can assume more than one value, the effects of the action of e_0 on the conditions of the rest of the edges are not investigated until multiple AUM pairs are associated with v_1.

The next edge of the graph traversal e_1 assigns 3 to x and terminates on v_1. Recall that due to the action of e_0 one AUM pair in which x assumes 2 was associated with v_1. The application of the action of e_1 on AUM$(v_0, 1)$ produces a new AUM pair which makes AUM$(v_1, 2)$ $= \{A_{v_1,0}, \tilde{B}_{v_1,0}, \{A_{v_1,1}, \tilde{B}_{v_1,1}\}$:

$$A_{v_1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_1,0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

$$A_{v_1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_1,1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 3 \end{bmatrix}$$

Once the number of AUM pairs associated with v_1 becomes more than one, P2-MBF is invoked to detect action inconsistencies among the edges leading to v_1 and the outgoing edges of the nodes reachable from it. For v_1 of Figure 4.1:

$$VAR_{v_1}^{dif-modified} = \{x\} \text{ and } VAR_{v_1}^{con-used} = \{b, i, x\}.$$

where

$$VAR_{v_1}^{dif-modified} \cap VAR_{v_1}^{con-used} = \{x\}$$

The above result establishes a necessary condition for action inconsistencies between the edges leading to v_1 and those reachable from v_1. Two edges, e_3 and e_7, that use x in their conditions are identified. Symbolic execution is employed (i.e., AUM$(v_1, 2)[v_1 \overset{SE}{\rightsquigarrow} v_2]$: = AUM$(v_2, 2)$) to take into consideration the actions in the paths between v_1 and the *head* node of e_3 and e_7, which is v_2. As a result, the assignment of zero to i and j by the actions in the paths between v_1 and v_2 is shown in the AUM$(v_2, 2) = \{A_{v_2,0}, \tilde{B}_{v_2,0}, A_{v_2,1}, \tilde{B}_{v_2,1}\}$:

$$A_{v_2,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_2,0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

$$A_{v_2,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \bar{B}_{v_2,1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 3 \end{bmatrix}$$

Since the actions of the edges leading to v_2 modify x differently, e_7 becomes infeasible and action inconsistency between e_7 and e_0 is detected. As a result, v_2 and all reachable nodes (except the initial node) and edges from v_1 are split. The resulting graph is shown in Figure 4.2. By splitting the graph of Figure 4.1, thus including e_0 and e_1 into two separate subgraphs where any path connecting the two subgraphs must include the initial node, the action inconsistencies between e_1 and e_7 is eliminated.

The MBF graph traversal is restarted since the graph topology is changed by the creation of the new edges and nodes. The graph of Figure 4.2 still contains action inconsistencies. For example, the actions of $e_{4(0)}$ and $e_{5(0)}$ modify i differently and cause action inconsistencies among $e_{4(0)}, e_{5(0)}, e_{3(0)}$, and $e_{7(0)}$. Similarly, the actions of $e_{4(1)}$ and $e_{5(1)}$ also modify i differently and cause action inconsistencies among $e_{4(1)}, e_{5(1)}, e_{3(1)}$ and $e_{7(1)}$. The detection and elimination of the inconsistencies among these edges are described below.

The P1-MBF graph traversal of the graph of Figure 4.2 continues with the edges of $e_{0(0)}$, $e_{1(1)}$, $e_{2(0)}$, $e_{2(1)}$, $e_{3(0)}$, $e_{3(1)}$, $e_{4(0)}$, $e_{5(0)}$, ... Upon traversing $e_{5(0)}$, two AUM pairs are created for $v_{4(0)}$. Each of the two paths leading to $v_{4(0)}$, namely $e_{0(0)}.e_{2(0)}.e_{3(0)}.e_{4(0)}$ and $e_{0(0)}.e_{2(0)}.e_{3(0)}.e_{5(0)}$ (where $e_i.e_j$ means e_i followed by e_j), contributes to the formation of a unique AUM pair for

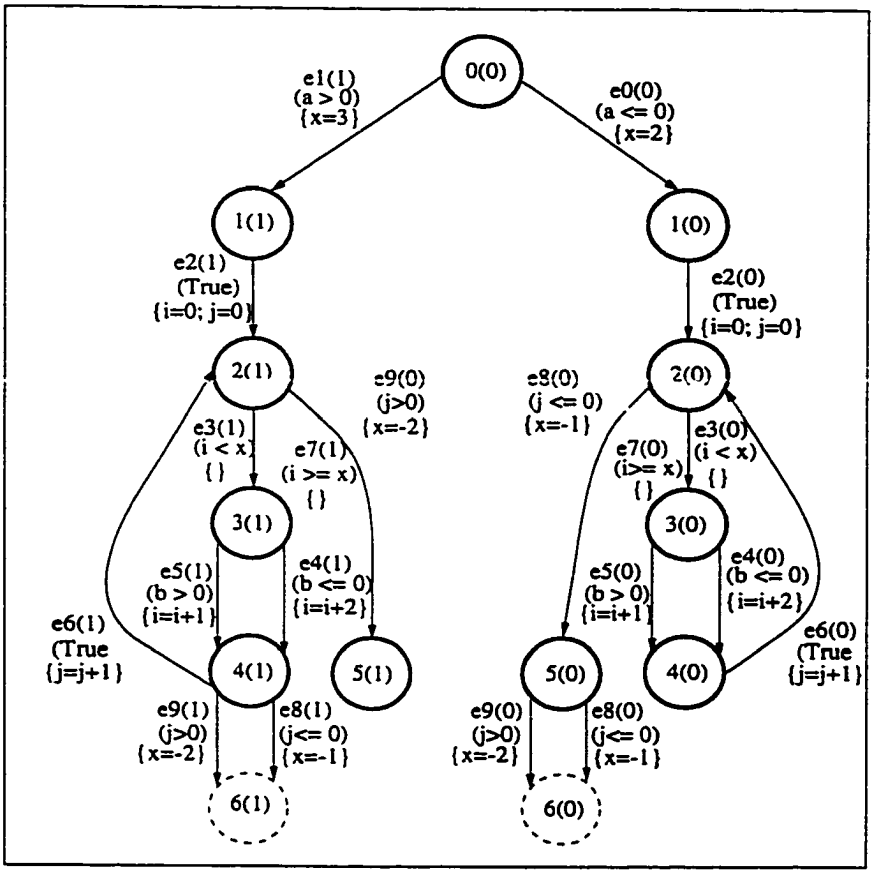

Figure 4.2: The EFSM graph after the graph of Figure 4.1 is split due to e_1 effects.

$v_{4(0)}$:

$$A_{v_{4(0)},0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \hat{B}_{v_{4(0)},1} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 2 \end{bmatrix}$$

$$A_{v_{4(0)},1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_{4(0)},1} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 2 \end{bmatrix}$$

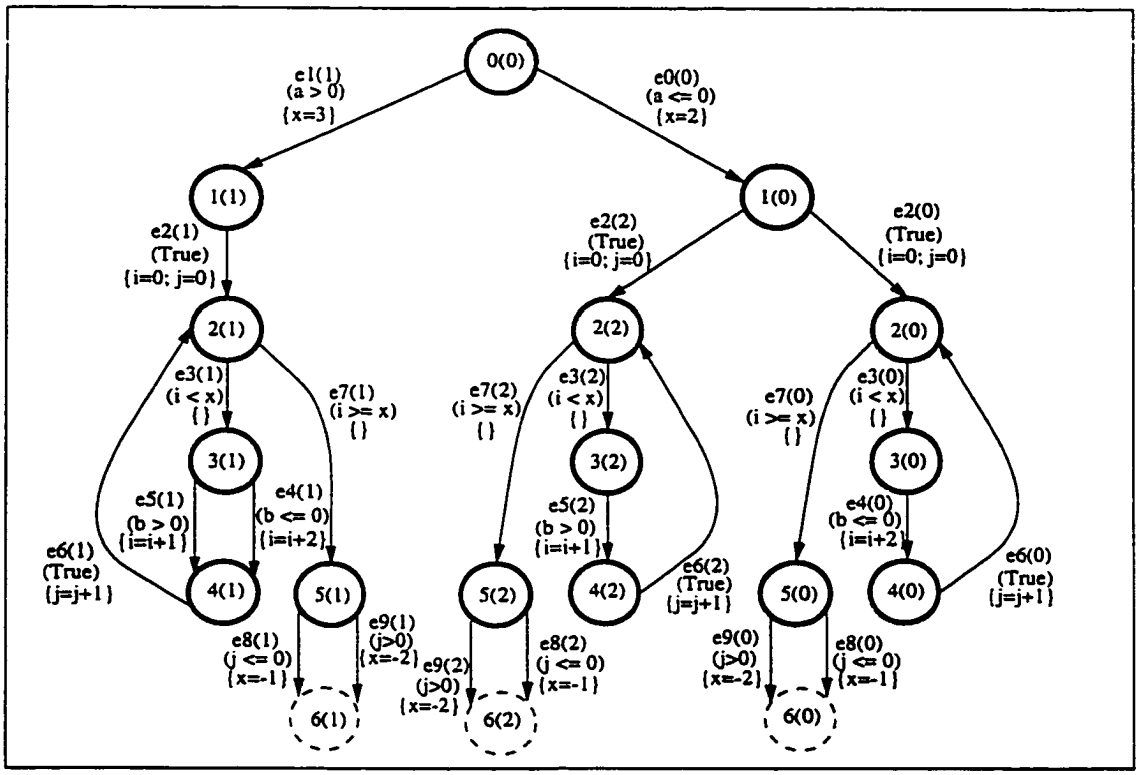

Figure 4.3: The EFSM graph after the graph of Figure 4.2 is split due to $e_{5(0)}$ action.

As a result, P2-MBF is invoked when $v_{4(0)}$ is reached by traversing $e_{5(0)}$. Upon traversing $e_{6(0)}$ in the P2-MBF graph traversal, the existence of action inconsistencies between the edges leading to $v_{4(0)}$ and the outgoing edges of $v_{2(0)}$ is detected. Particularly, the condition of $e_{7(0)}$ is inconsistent with the action of $e_{5(0)}$. Hence, the subgraph accessible from $v_{4(0)}$ is split as shown

in Figure 4.3. Similarly, the two AUM pairs resulting from the paths $e_{1(1)} \cdot e_{2(1)} \cdot e_{3(1)} \cdot e_{4(1)}$ and $e_{1(1)} \cdot e_{2(1)} \cdot e_{3(1)} \cdot e_{5(1)}$ of Figure 4.3 are:

$$A_{v_{4(1)},0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_{4(1)},0} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 3 \end{bmatrix}$$

$$A_{v_{4(1)},1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \tilde{B}_{v_{4(1)},1} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 3 \end{bmatrix}$$

The nodes and edges reachable from $v_{4(1)}$ are split when $v_{4(1)}$ is visited since an action inconsistency is found between the edges leading to $v_{4(1)}$ and $e_{7(1)}$. Figure 4.4 shows the resulting graph after this inconsistency is eliminated.

The P1-MBF graph traversal of the graph of Figure 4.4 continues as: $e_{0(0)}$, $e_{1(1)}$, $e_{2(0)}$, $e_{2(2)}$, $e_{2(1)}$, $e_{2(3)}$, $e_{3(0)}$, $e_{3(2)}$, $e_{3(3)}$, $e_{4(0)}$, $e_{5(2)}$, $e_{4(1)}$,$e_{5(3)}$, e_6, ... After traversing $e_{6(0)}$ it is found that $v_{2(0)}$ is a loop entry/exit node $Loop_{v_{2(0)}}$. Therefore, the loop is advanced once by duplicating the nodes and edges of the loop body. Since $e_{7(0)}$ is infeasible (i.e., the maximum number of the loop iterations is not reached yet), it is removed from the graph. As a result, $v_{5(0)}$, $e_{8(0)}$, and $e_{9(0)}$ become unreachable from v_0 and must be removed also from the graph. The resulting graph is shown in Figure 4.5.

In a similar fashion, the loops whose entry/exit nodes are $v_{2(1)}, v_{2(2)}$, and $v_{2(3)}$ are advanced after traversing the edges of $e_{6(1)}$, $e_{6(2)}$, and $e_{6(3)}$, respectively. (See Figure 4.6 for the resulting graph.)

The graphs appearing in Figures 4.7-4.11 show successive graph splitting due to advancing loops. The subgraphs shown in Figures 4.10 and 4.11 are free of action inconsistencies caused by the variables which can assume multiple values at a given node. However, the edges of $e_{8(4)}$, $e_{8(8)}$, $e_{8(9)}$, and $e_{8(11)}$, in these subgraphs, are infeasible. These edges are eliminated during the condition inconsistency detection. As can be seen from graph shown in Figures 4.10, and 4.11, there are no condition inconsistencies. However, when the nodes $v_{5(4)}$, $v_{4(8)}$, $v_{5(9)}$, and $v_{4(11)}$ are reached by traversing the edges of $e_{7(4)}$, $e_{7(8)}$, $e_{7(9)}$, and $e_{7(11)}$, respectively, the edges of $e_{8(4)}$, $e_{8(8)}$, $e_{8(9)}$, and $e_{8(11)}$ are found to be infeasible. These infeasible edges are deleted from the graph.

The final consistent EFSM graph is presented in Figure 4.13. The two subgraphs that are not shown in this figure can be seen in Figure 4.12.

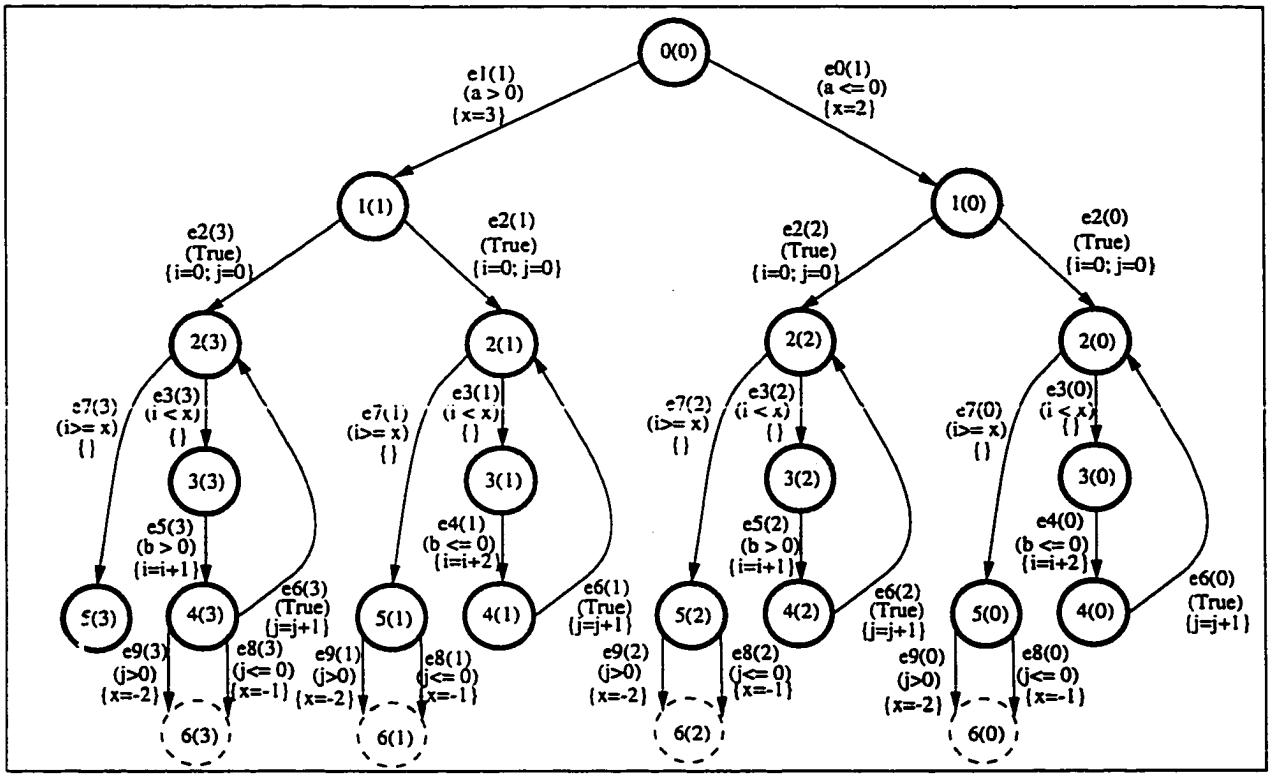

Figure 4.4: The EFSM graph after the graph of Figure 4.3 is split due to $e_{5(1)}$ action.

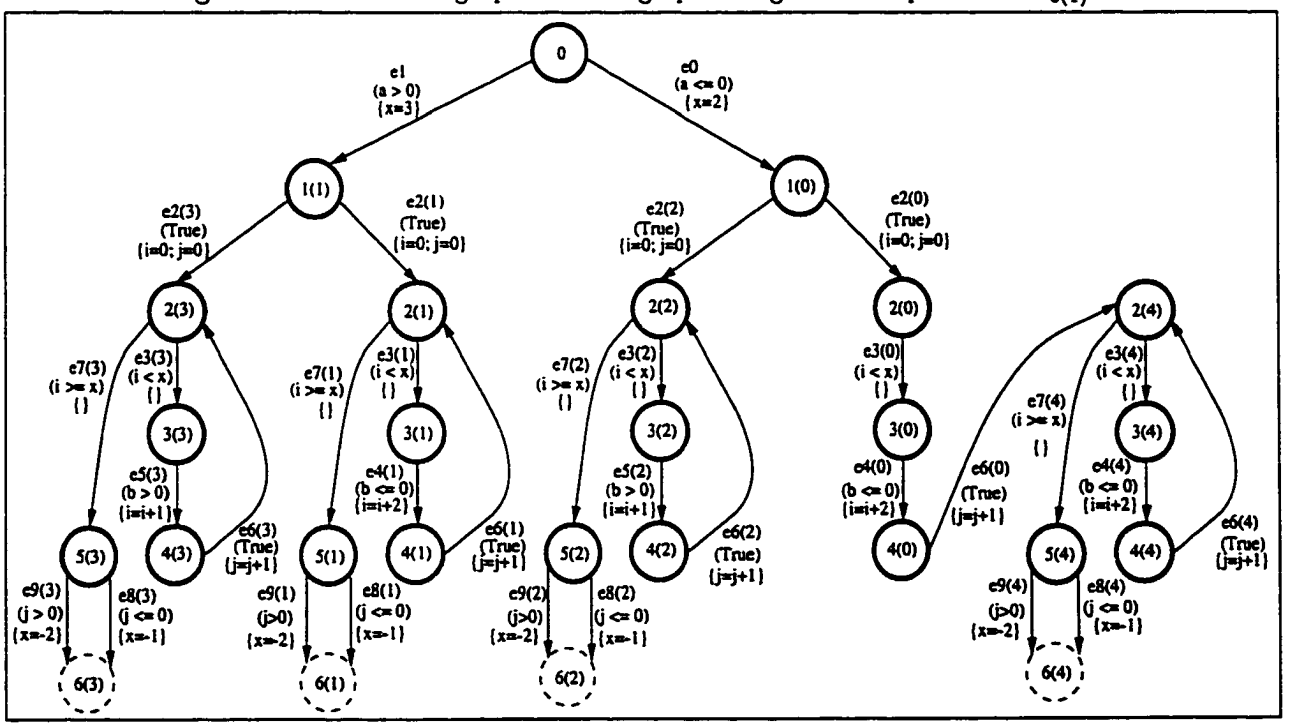

Figure 4.5: The EFSM graph after the graph of Figure 4.4 is split due to $e_{6(0)}$ action.

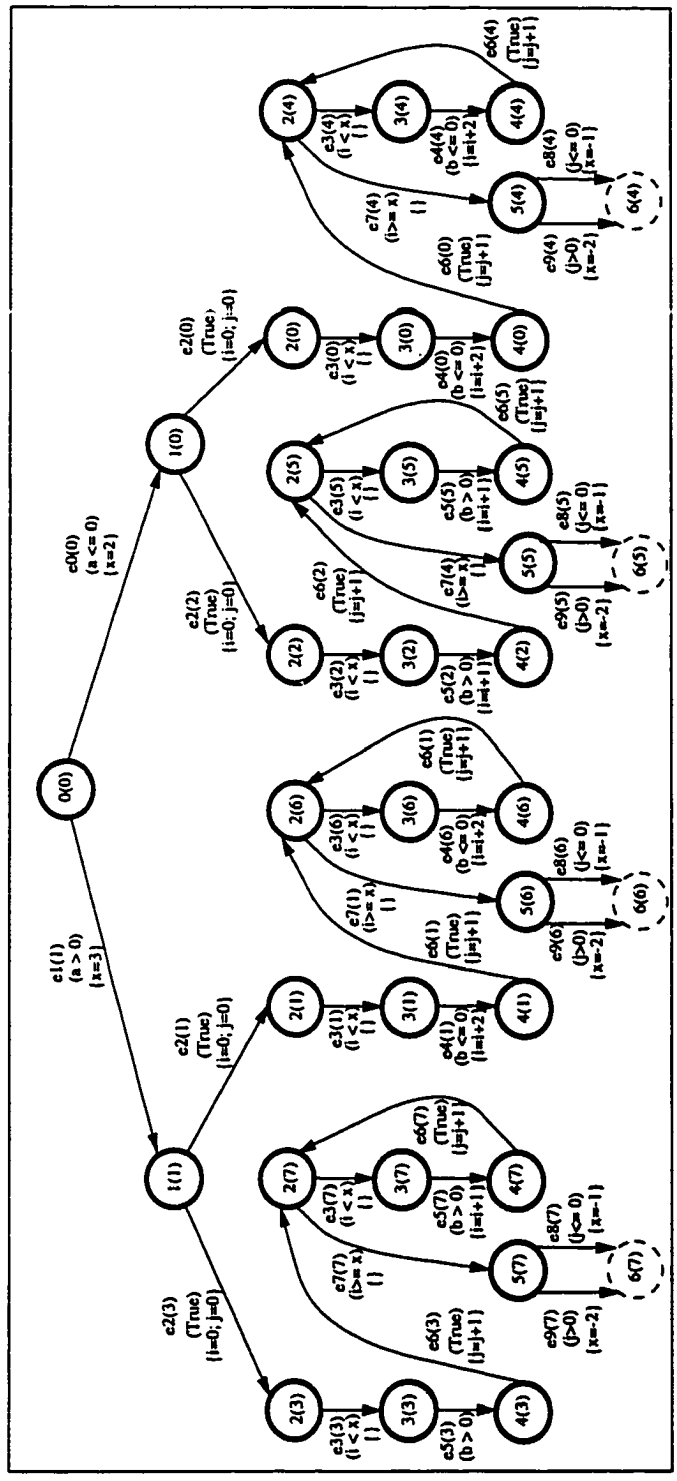

Figure 4.6: The EFSM graph after the graph of Figure 4.5 is split due to $e_{6(2)}$, $e_{6(1)}$ and $e_{6(3)}$ effects.

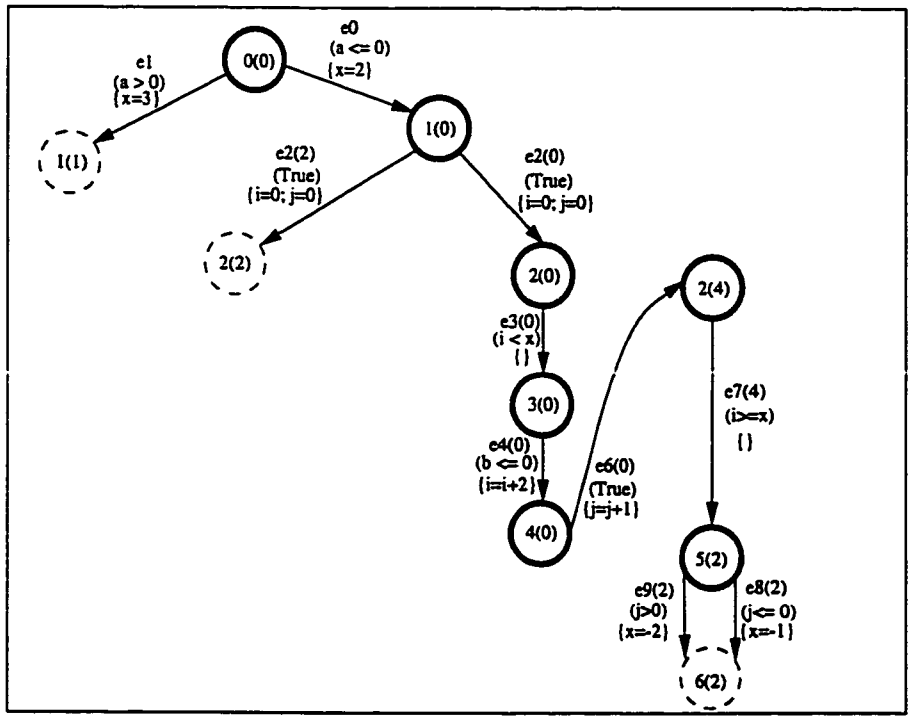

Figure 4.7: The EFSM graph after the graph of Figure 4.6 is split due to $e_{6(4)}$ effects.

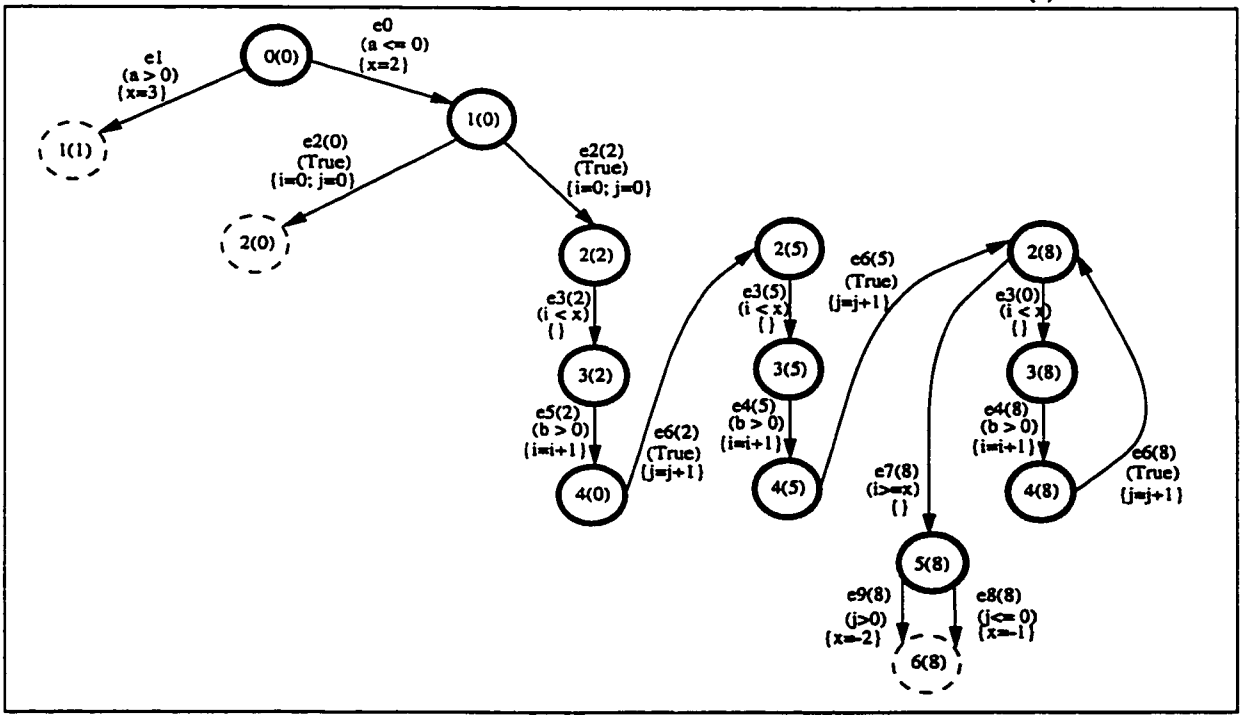

Figure 4.8: The EFSM graph after the graph of Figure 4.7 is split due to $e_{6(5)}$ effects.

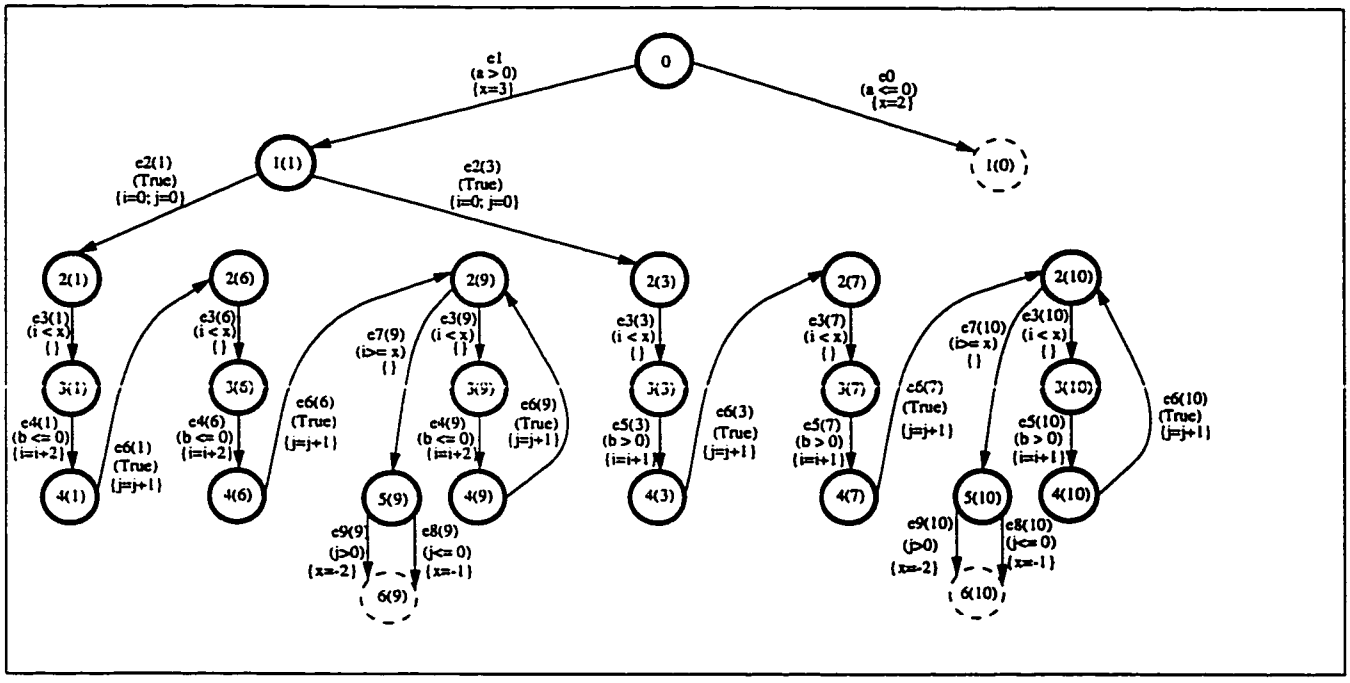

Figure 4.9: The EFSM graph after the graph of Figure 4.8 is split due to $e_{6(5)}$ and $e_{6(7)}$ effects.

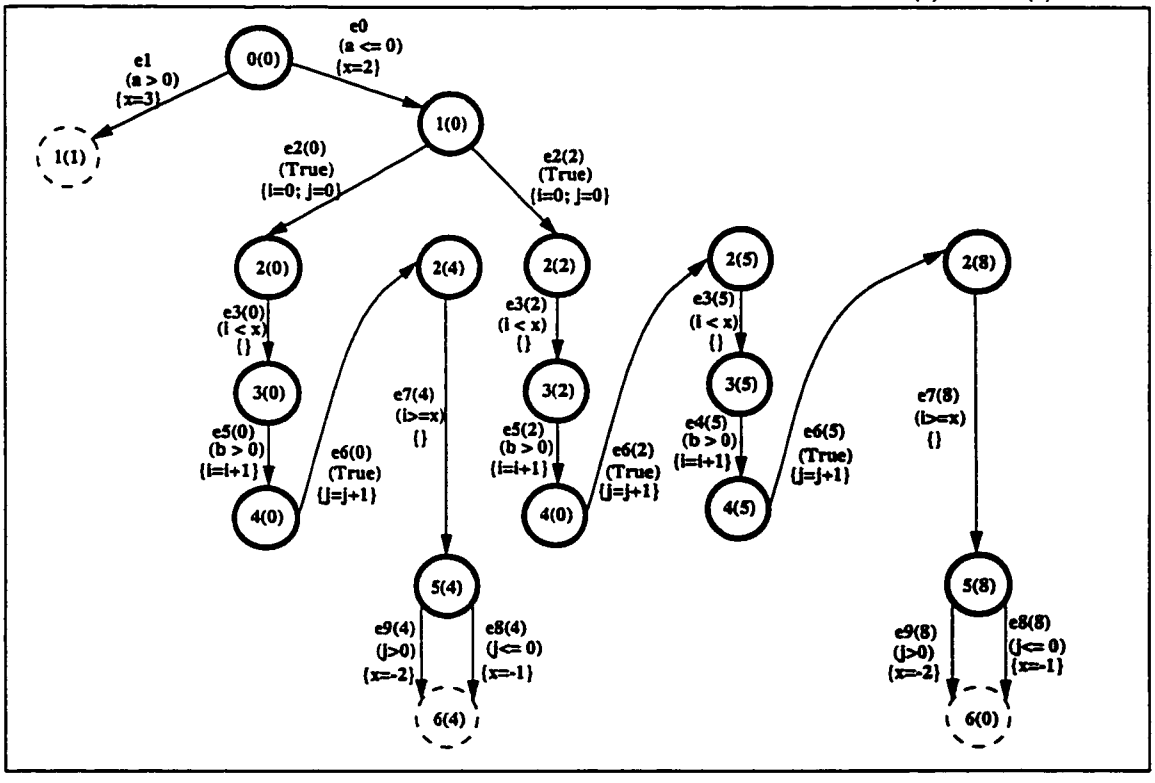

Figure 4.10: EFSM graph after the graph of Figure 4.9 is split due to $e_{6(8)}$ effects.

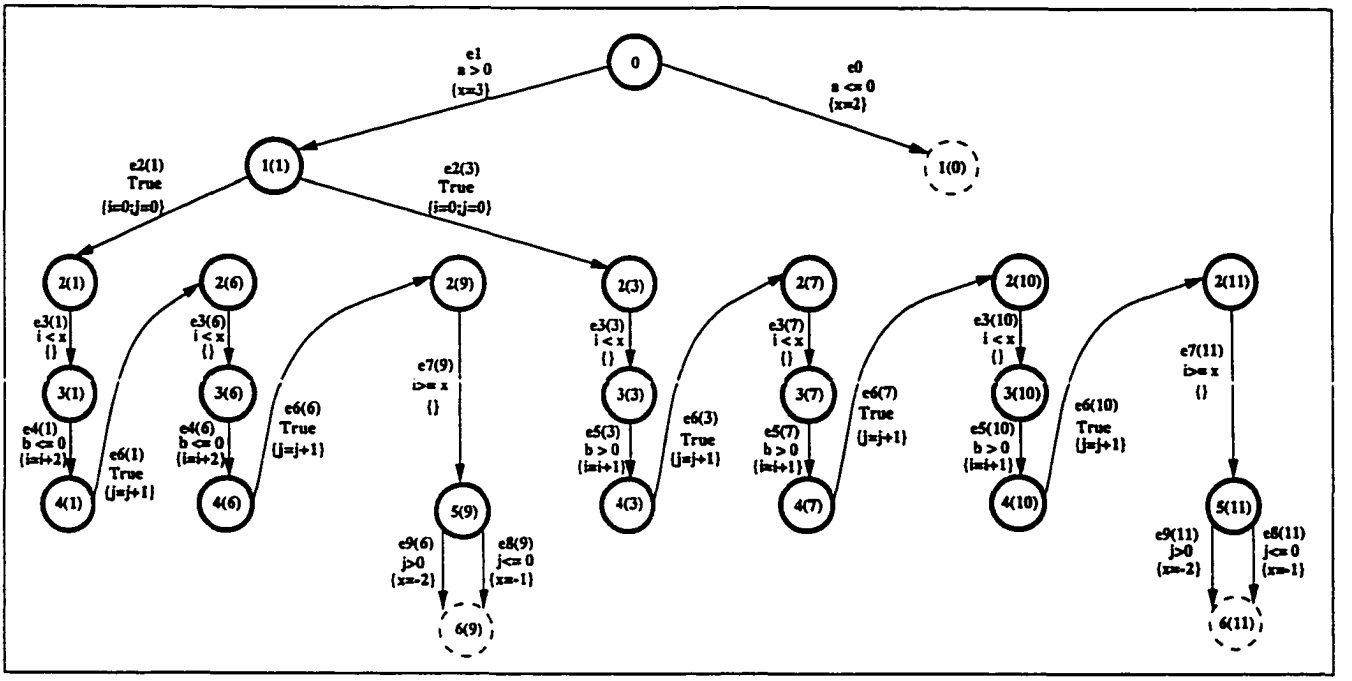

Figure 4.11: The EFSM graph after the graph of Figure 4.10 is split due to $e_{6(9)}$ and $e_{6(11)}$ effects.

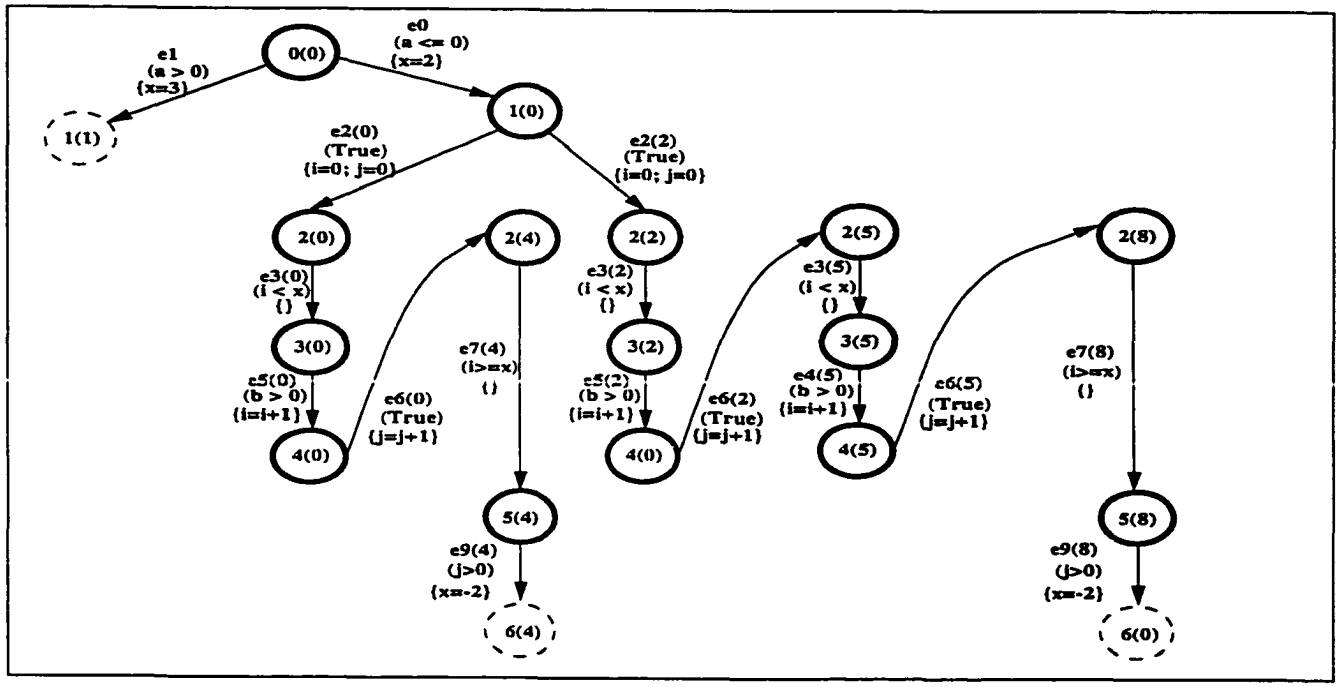

Figure 4.12: The EFSM graph after $e_{8(4)}$ and $e_{8(8)}$ of the graph of Figure 4.10 are deleted.

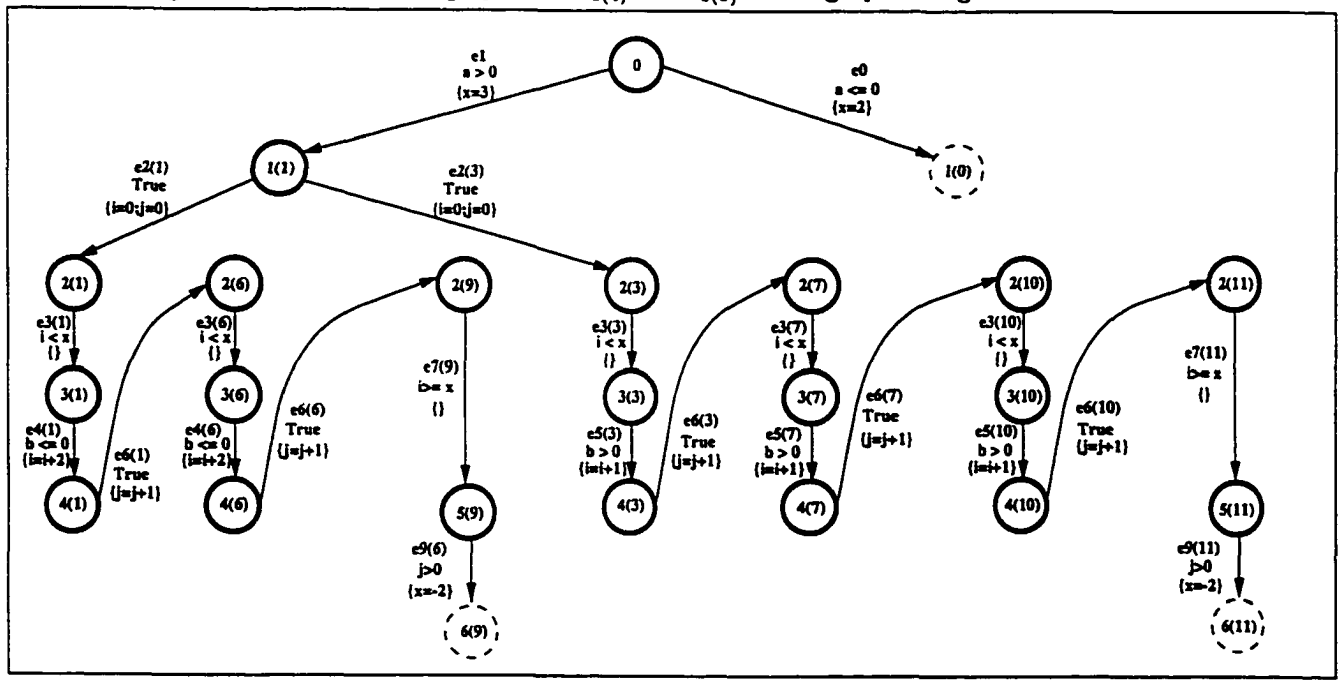

Figure 4.13: The EFSM graph after $e_{8(9)}$ and $e_{8(11)}$ of the graph of Figure 4.11 are deleted.

4.2 EFSM Graphs with Nested Loops

The application of the inconsistency detection and elimination algorithms to the EFSM graphs with nested loops is demonstrated in this section. As mentioned earlier, the detection and elimination of action inconsistencies are performed before interdependencies among the condition variables are analyzed. According to the MBF graph traversal, the analysis of the nested loops starts with the most inner loop. Recall that a loop is analyzed by advancing its iterations during the P1-MBF graph traversal. After all action inconsistencies are eliminated from the graph, the detection and elimination of condition inconsistencies are proceeded.

The set of variables used in the actions and conditions of the EFSM graph of Figure 4.14 is: $\{b, c, d, i, j, n, x\}$. The following AUM pair, in which the value of the variables i and n are modified to 0 and -100, respectively, is created for the node v_2 when e_0 is traversed:

$$A_{v_2,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \end{bmatrix}$$

The traversal of the edge e_1, which is the next edge in the P1-MBF graph traversal, creates a new AUM pair for v_1 (i.e., the *tail* node of e_1):

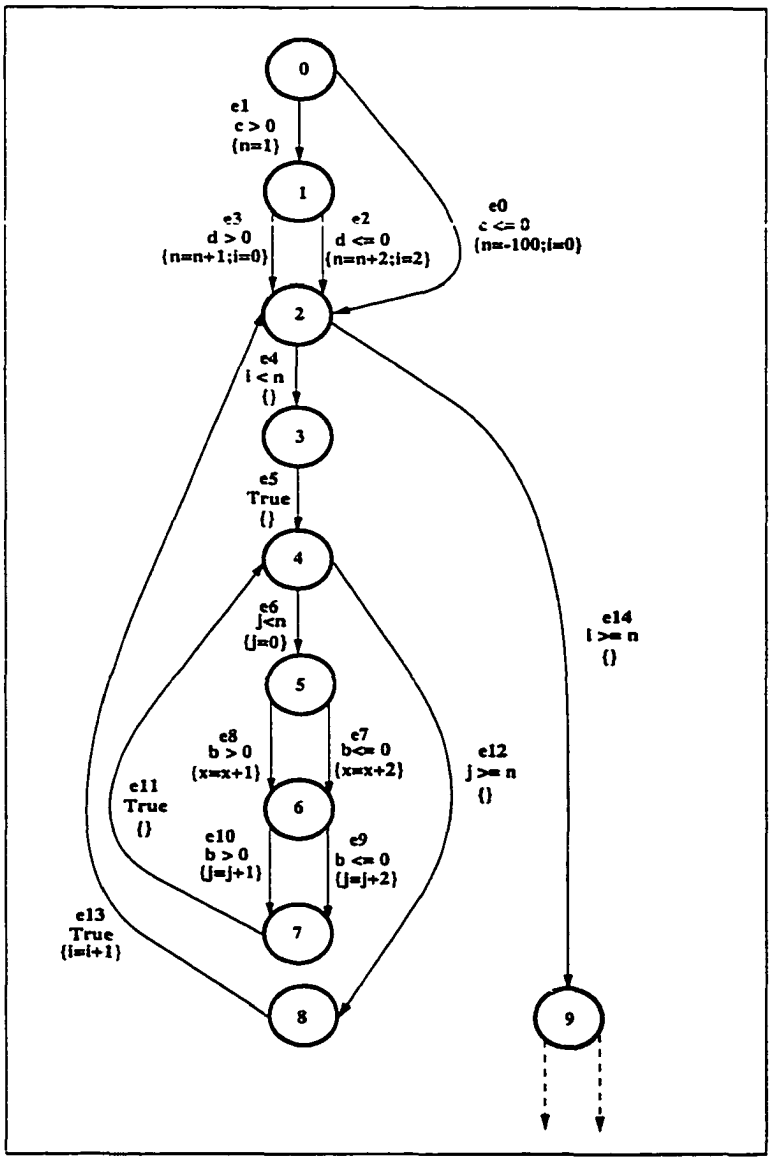

Figure 4.14: An EFSM with nested loops.

$$A_{v_{1(0)},0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{1(0)},0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

The number of unique AUM pairs associated with v_2, after e_2 is traversed, becomes two:

$$A_{v_2,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \end{bmatrix}$$

$$A_{v_2,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_2,1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

Once the above two AUM pairs are created for v_2, the P2-MBF graph traversal is initiated. P2-MBF of the MBF graph traversal, which starts from node v_2, should continue until an action

inconsistency is detected or both of the incoming edges of v_7 are traversed. The number of iterations of the loop whose entry/exit node is v_2 depends on the variables i and n (i.e., i and n are used in the condition of the outgoing edges of v_2). As can be seen from AUM(v_2, 2) these two variables are modified differently in the paths leading to v_2. Hence, the graph is split as shown in Figure 4.15. Notice that due to the action of e_0, e_4 cannot be traversed with e_0. Therefore, e_4 is deleted from the subgraph containing $e_{0(0)}$. As a consequence, all the nodes and edges reachable from $v_{2(0)}$, except $v_{9(0)}$ and $e_{14(0)}$, are deleted from the graph since these edges and nodes become unreachable from the initial node.

Upon traversing $e_{3(0)}$ of the graph in Figure 4.15, multiple AUM pairs are formed for $v_{2(0)}$:

$$A_{v_{2(0)},0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{2(0)},0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \end{bmatrix}$$

$$A_{v_{2(0)},1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{2(0)},1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \end{bmatrix}$$

Based on the above two AUM pairs, an action inconsistency is found to exist between $e_{3(0)}$

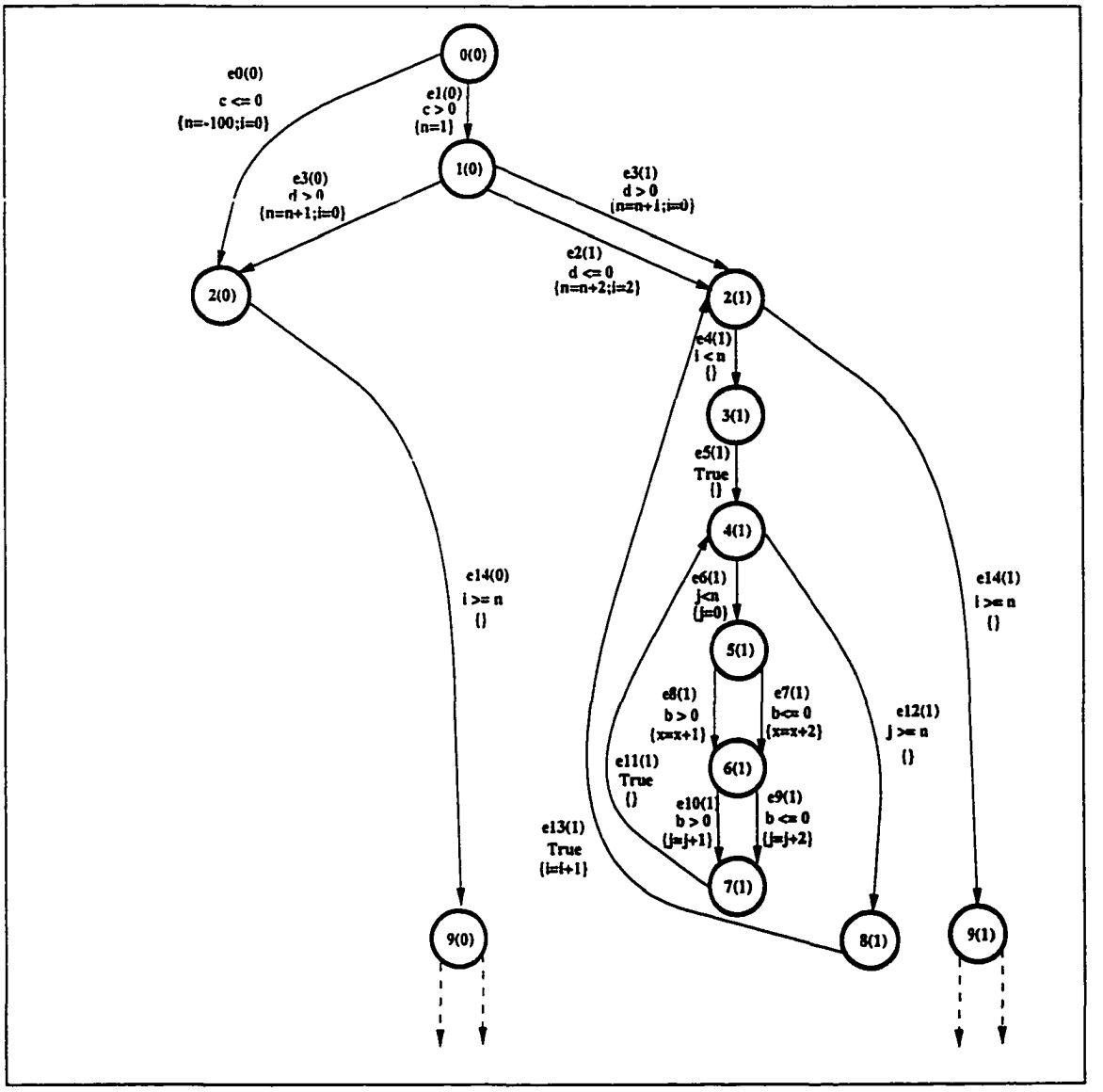

Figure 4.15: The EFSM graph after the graph of Figure 4.14 is split due to e_2 action.

and $e_{14(0)}$. To eliminate this inconsistency, $v_{2(0)}$ and the subgraph reachable from it are split as shown in Figure 4.16. The aim is to prevent $e_{3(0)}$ and $e_{14(4)}$ from being accessible from one another. As can be seen from the graph of Figure 4.16, $e_{14(2)}$ is deleted from the subgraph that contains $e_{3(0)}$. The node $v_{2(0)}$ is removed from the graph as shown in Figure 4.17 since the initial node becomes $v_{2(0)}$.

The MBF graph traversal of the graph in Figure 4.17 continues with the edges of $e_{0(0)}$, $e_{1(0)}$, $e_{14(0)}$, $e_{2(1)}$, $e_{2(1)}$, $e_{3(1)}$, $\cdots$. After $v_{2(1)}$ is reached by traversing $e_{3(1)}$, $\mathrm{AUM}(v_{2(1)}, 2) = \{A_{v_{2(1)},0}, \tilde{B}_{v_{2(1)},0}, A_{v_{2(1)},1}, \tilde{B}_{v_{2(1)},1}\}$:

$$A_{v_{2(1)},0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{2(1)},0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

$$A_{v_{2(1)},1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{2(1)},1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \end{bmatrix}$$

are created for $v_{2(1)}$. Since the conditions of $e_{4(1)}$ and $e_{14(1)}$ use the variables i and n which are

modified differently in the paths leading to $v_{2(1)}$, $v_{2(1)}$ and the subgraph reachable from it are split. The resulting graph appears in Figure 4.18.

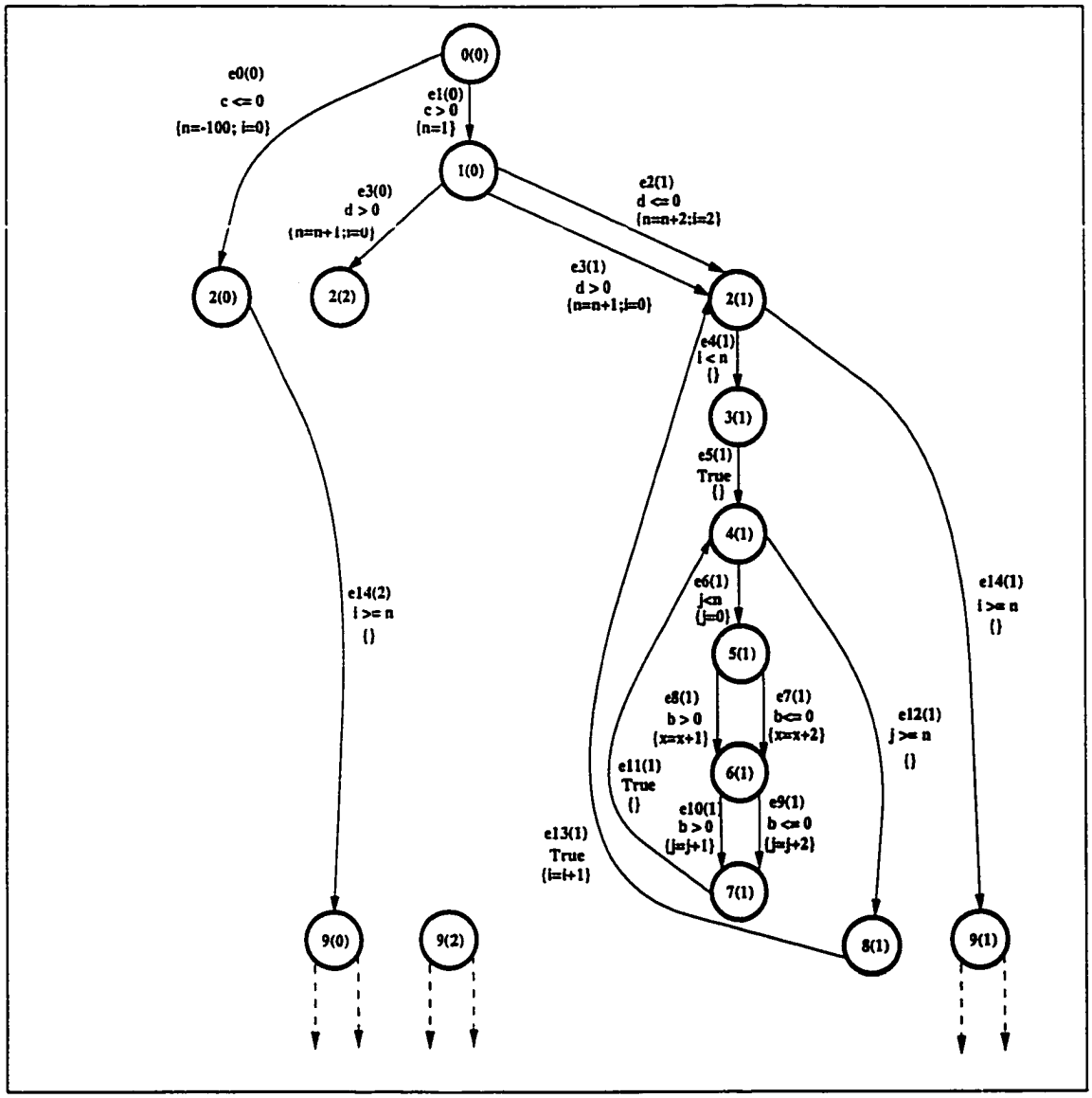

Figure 4.16: The EFSM graph after $e_{3(1)}$ is removed from the graph of Figure 4.15

The number of unique AUM pairs associated with $v_{7(1)}$ of the graph of Figure 4.18 becomes two

when $e_{10(1)}$ is traversed:

$$A_{v_{7(1)},0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{7(1)},0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 2 \\ 3 \\ 0 \end{bmatrix}$$

$$A_{v_{7(1)},1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \tilde{B}_{v_{7(1)},1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 1 \\ 3 \\ 0 \end{bmatrix}$$

The action inconsistency detection algorithm invokes the P2-MBF graph traversal to determine whether an edge $e_i \in E^{reachable}_{v_{7(1)}}$ becomes infeasible due to the variables modified differently in the paths leading to $v_{7(1)}$ of the graph of Figure 4.18.

An action inconsistency is found to exist among the edge of $e_{9(1)}$, $e_{10(1)}$, $e_{4(4)}$, and $e_{12(4)}$. The variable j, which can assume two different values at $v_{7(1)}$, is used in the conditions of $e_{6(1)}$ and $e_{12(1)}$. To eliminate this inconsistency, $v_{7(1)}$ and the subgraph reachable from it are split as shown in Figure 4.19. Similarly, the graph in Figure 4.20 shows the resultant graph after the action inconsistency due to the action of $e_{10(2)}$ is eliminated.

The subgraphs reachable from the nodes $v_{2(1)}$, $v_{2(2)}$, $v_{2(3)}$ and $v_{2(4)}$ of Figure 4.20 will basically

go through similar graph splits since there is a close similarity in these subgraphs' structures and their edge actions and conditions. The subgraph reachable from $v_{2(0)}$ of Figure 4.20 does not contain inconsistencies and will not involve further graph splitting. For simplicity, the rest of the discussion will focus on the subgraph reachable from $v_{2(1)}$ of the graph in Figure 4.20. The creation AUM pairs of the nodes will not also be shown.

Figures 4.21 and 4.24 depict the resulting EFSM graphs after the loop whose entry/exit node is $v_{4(1)}$ is advanced once and twice by traversing $e_{11(1)}$ and $e_{11(6)}$ of the graphs in Figures 4.19 and 4.24, respectively. Similarly, Figure 4.27 shows when the same loop is completely analyzed.

As can be seen from the graph appearing in Figure 4.37, there exists condition inconsistencies among several edges of the graph. Consider the edges $e_{8(1)}$ and $e_{9(1)}$. The ACM triplets obtained when the graph is traversed in a DF manner, starting from the initial node to $v_{6(1)}$, form a system of equations. By appending the condition of $e_{9(1)}$ to such an ACM, the following ACM triplets are created:

$$C_{v_{6(1)},0} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \hat{OP}_{v_{6(1)},0} = \begin{bmatrix} > \\ \leq \\ < \\ < \\ \leq \\ > \end{bmatrix} \quad \tilde{D}_{v_{6(1)},0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Since the system of equations formed by the above ACM triplets does not have a solution, there is a condition inconsistency between $e_{8(1)}$ and $e_{9(1)}$. Therefore, the subgraph starting from $v_{6(1)}$ is split such that $e_{8(1)}$ and $e_{9(1)}$ cannot be accessible from one another. The subgraph contain-

ing $e_{8(1)}$ is removed from the graph since the initial node cannot be reached from $tail(e_{8(1)})$. Similarly, the condition of the edge $e_{8(5)}$ conflicts with the condition of $e_{7(1)}$. The elimination of the condition inconsistency between these two edges will lead to the removal of $e_{8(5)}$ from the graph. The resultant EFSM graph, after the condition inconsistencies among $e_{7(1)}$, $e_{8(1)}$, $e_{9(1)}$, and $e_{8(5)}$ are eliminated, is depicted in Figure 4.42.

The final consistent EFSM graph appears in Figure 4.45. In this graph, only two subgraphs are shown for simplicity. The remaining subgraphs can be seen in Figures 4.42 through 4.44.

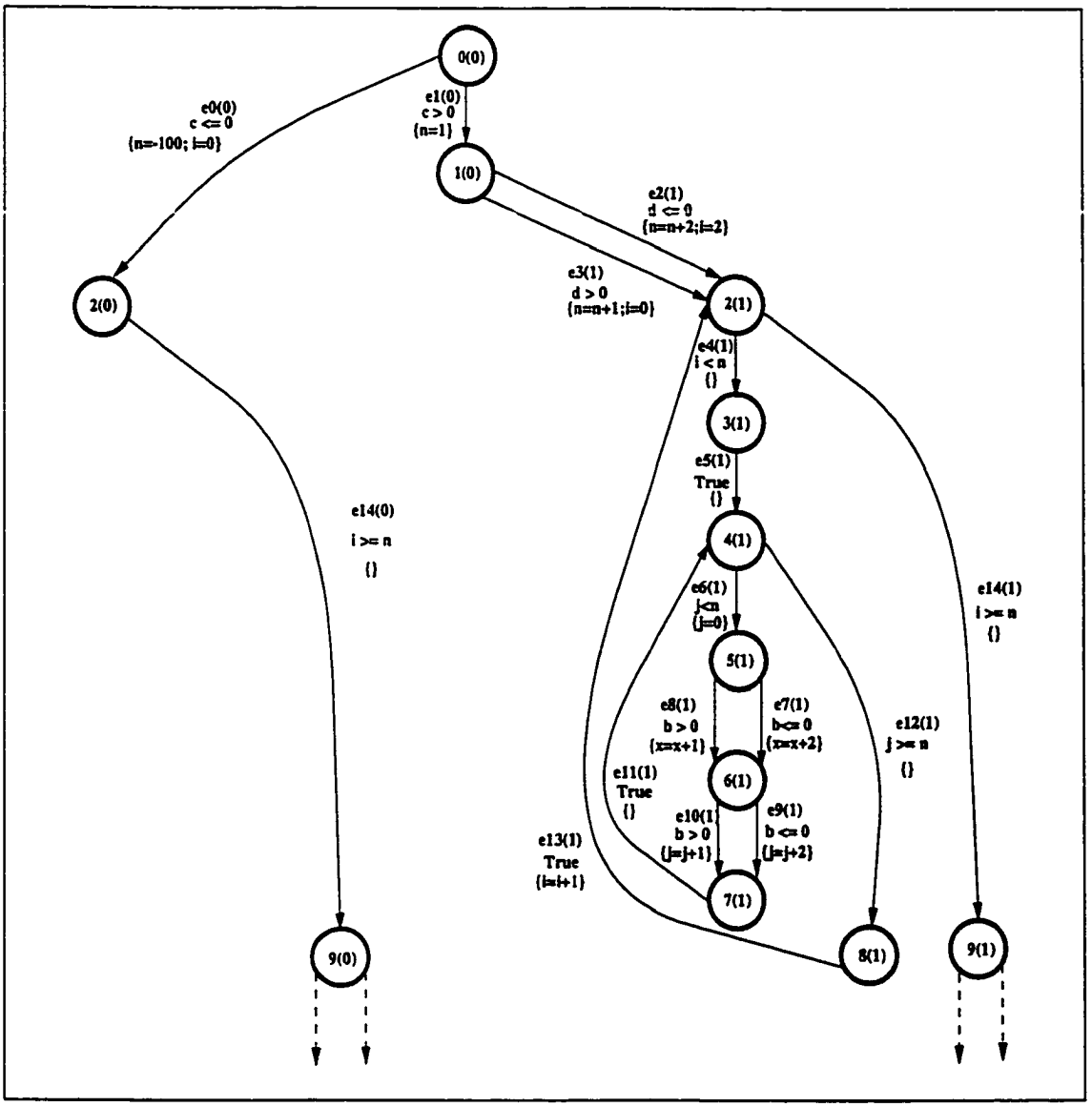

Figure 4.17: The EFSM graph after the graph of Figure 4.16 is split due to $e_{3(0)}$ action.

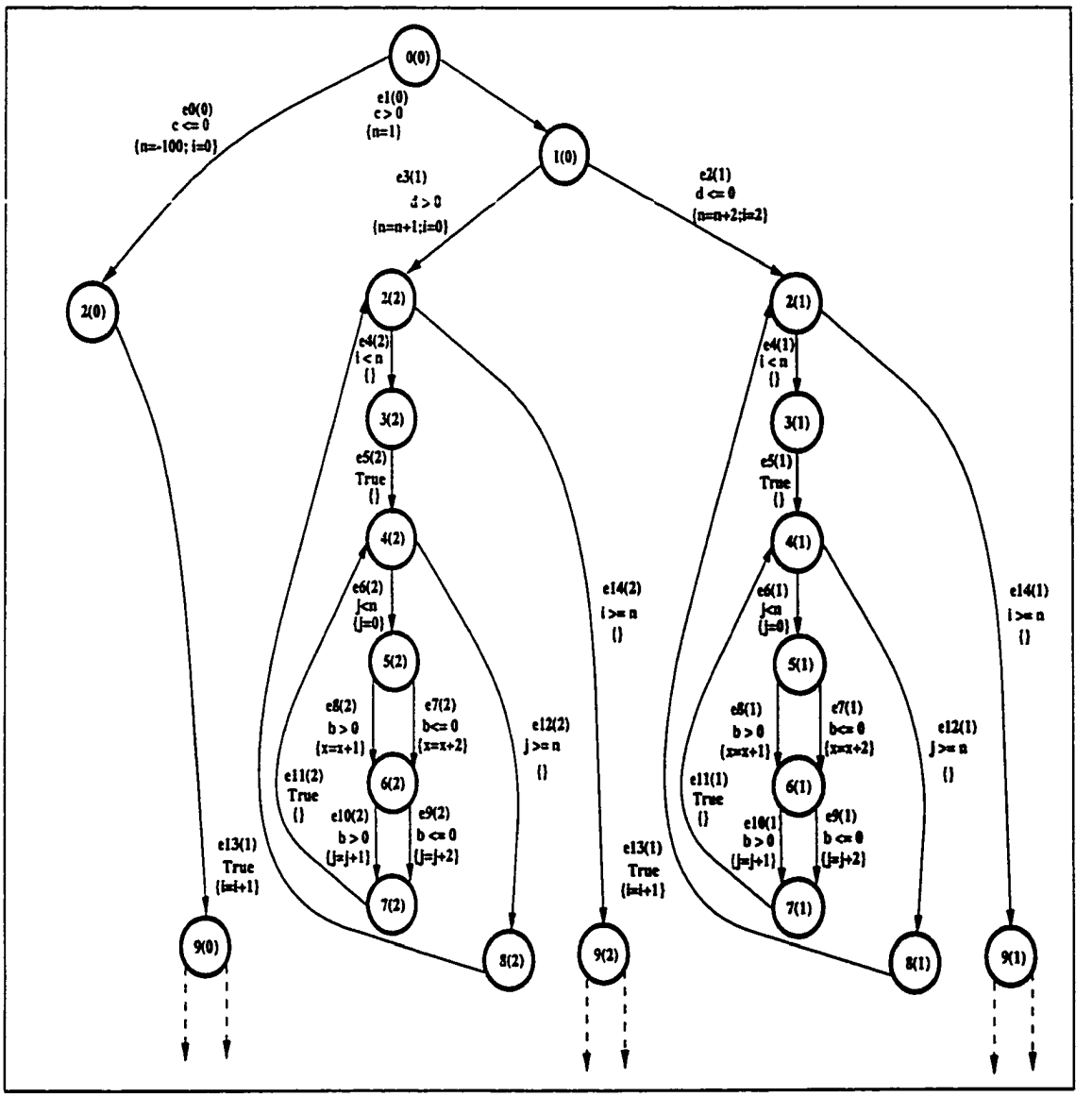

Figure 4.18: The EFSM graph after the graph of Figure 4.17 is split due to $e_{10(0)}$ action.

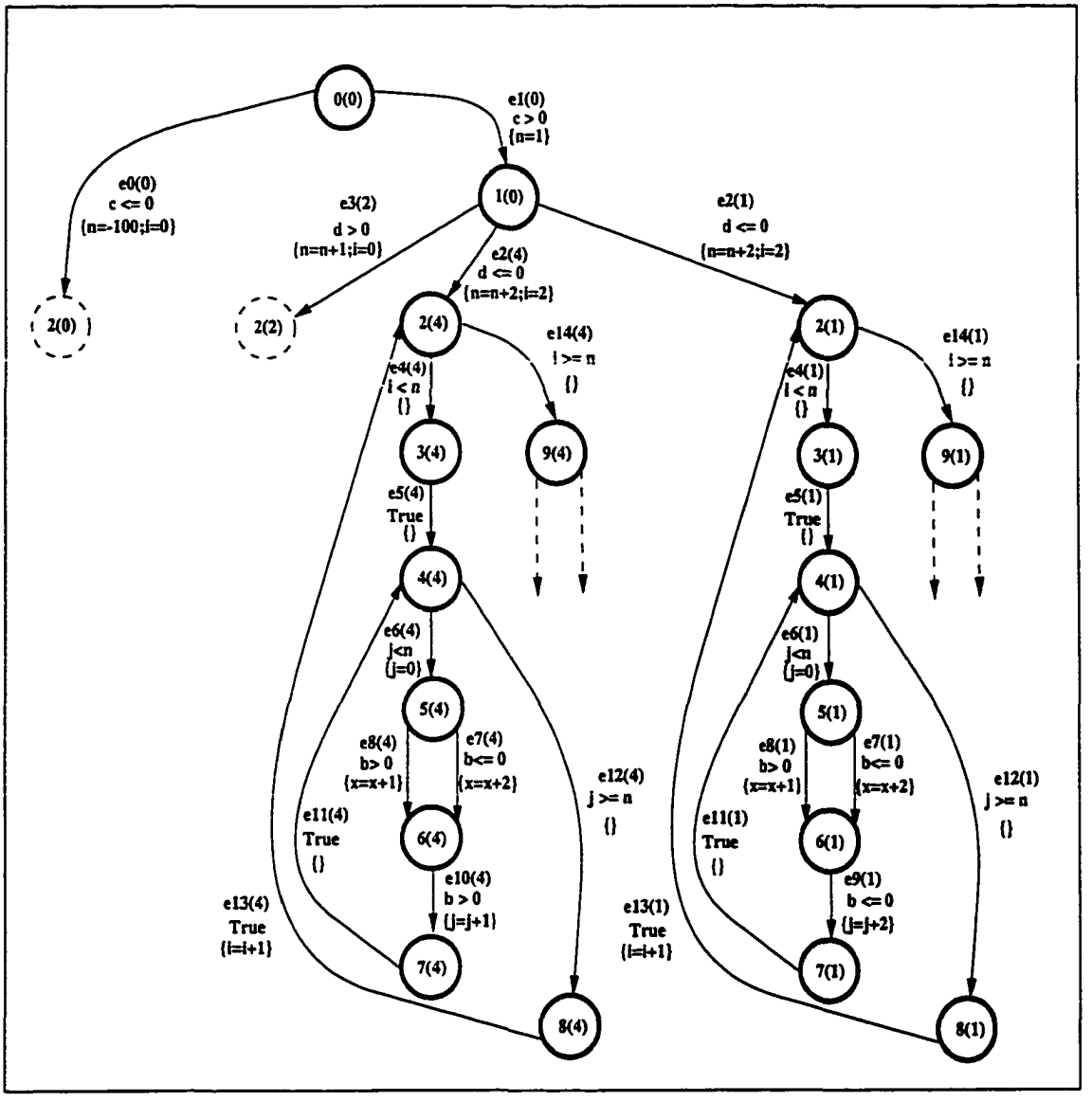

Figure 4.19: The EFSM graph after the graph of Figure 4.18 is split due to $e_{10(1)}$ action.

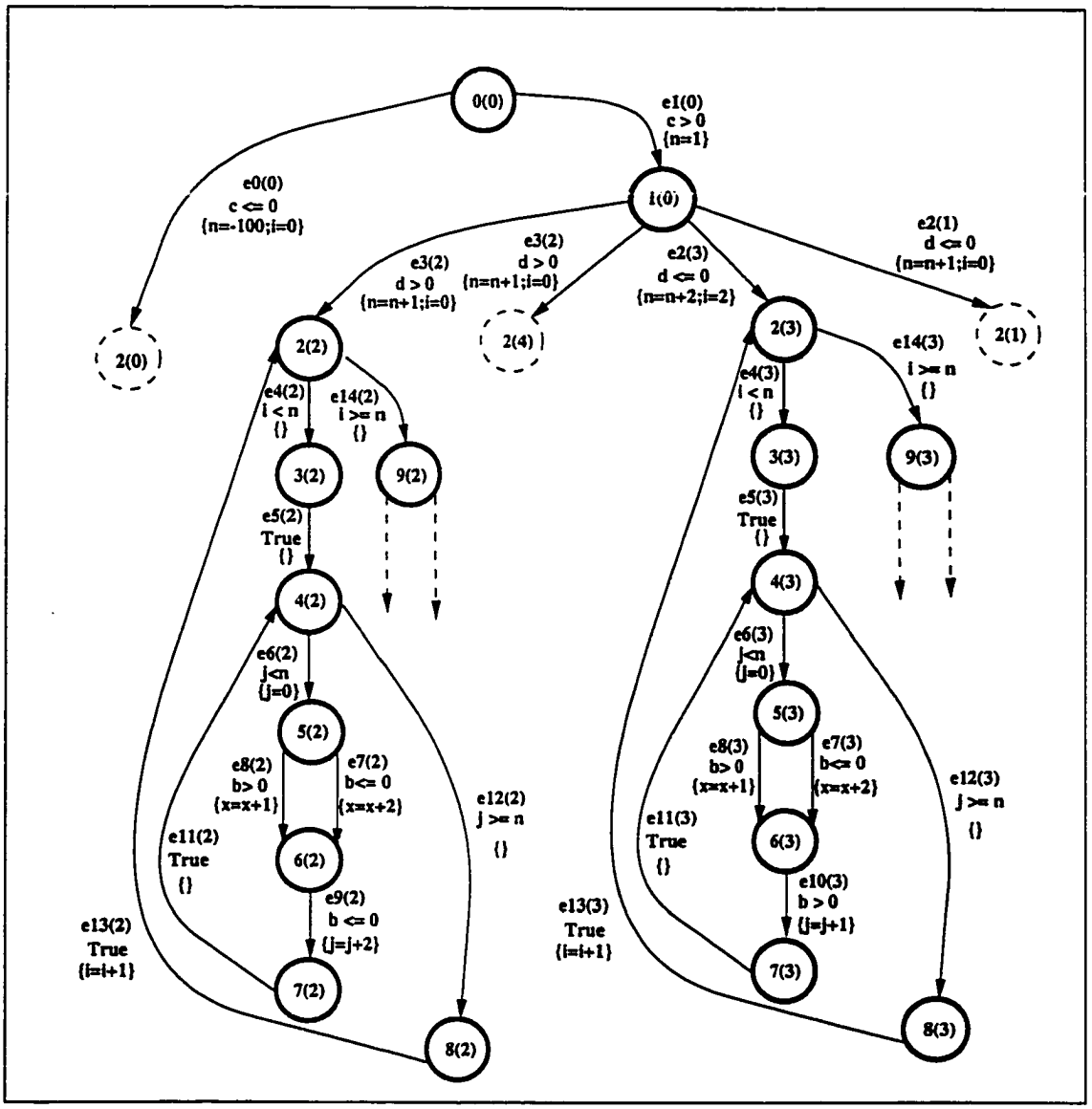

Figure 4.20: The EFSM graph after the graph of Figure 4.19 is split due to $e_{10(2)}$ action.

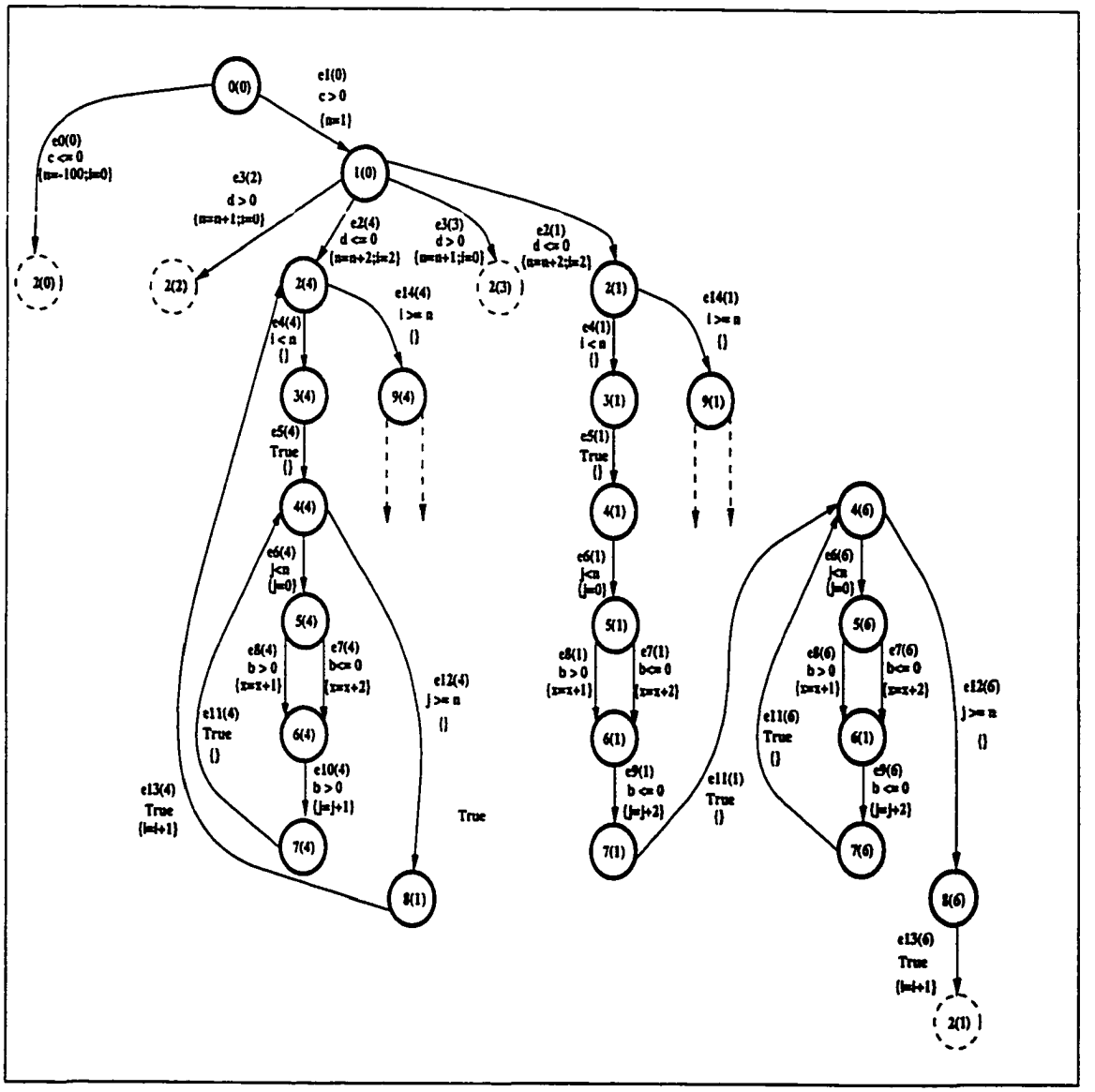

Figure 4.21: The EFSM graph after the loop with $Loop_{v_{4(1)}}$ of Figure 4.20 is advanced one iteration.

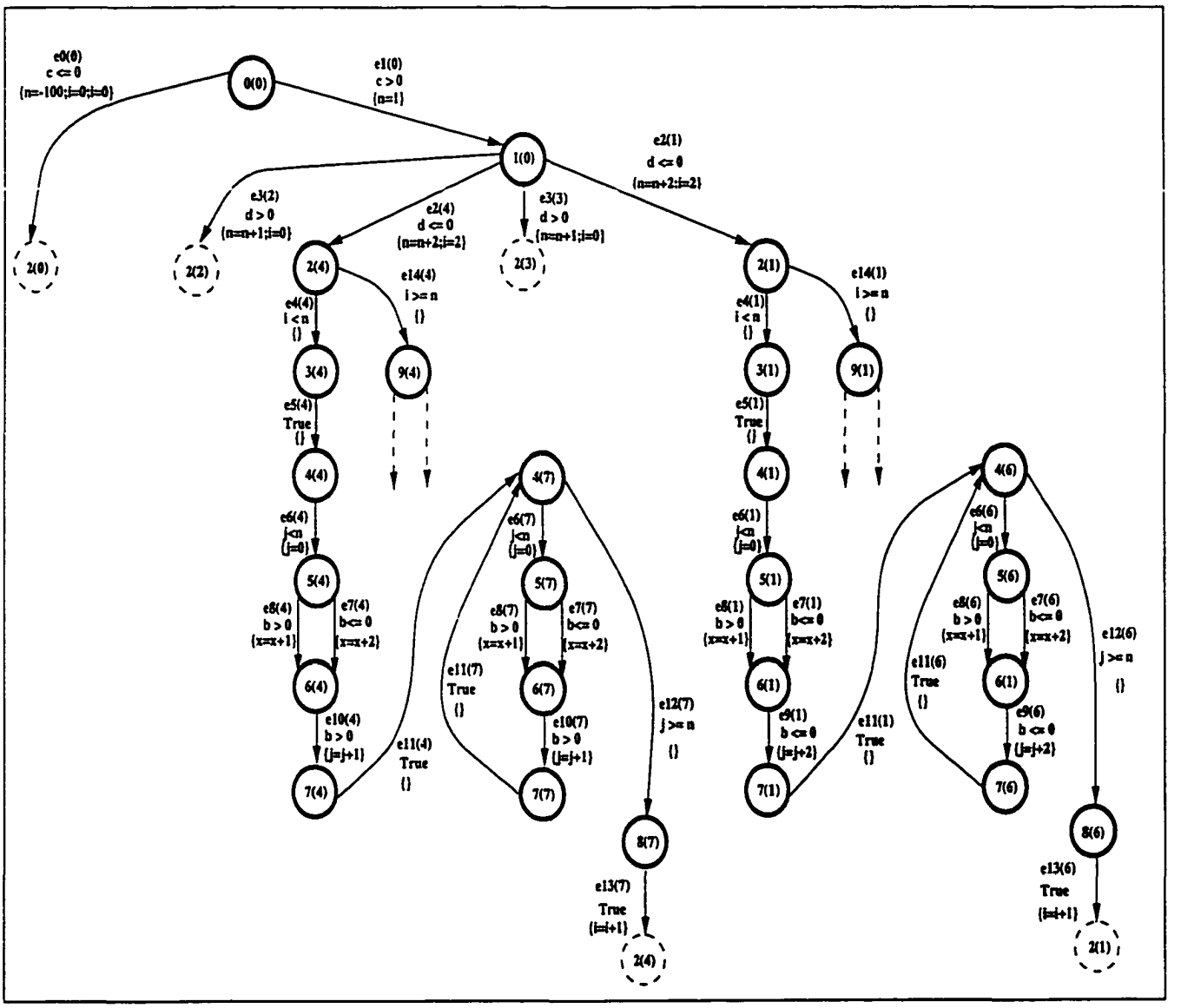

Figure 4.22: The EFSM graph after the loop with $Loop_{v_{4(4)}}$ of Figure 4.20 is advanced one iteration.

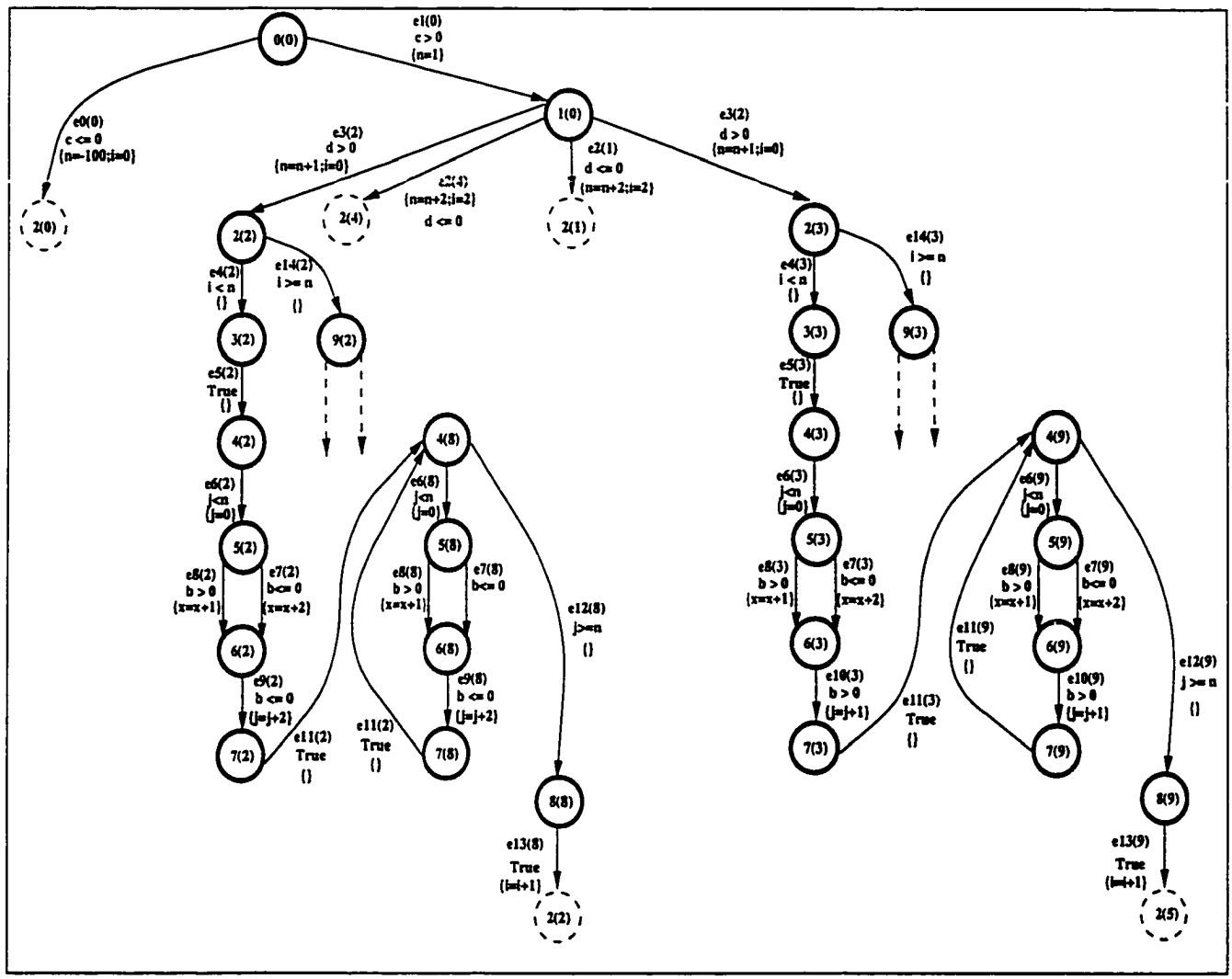

Figure 4.23: The EFSM graph after each loop with $Loop_{v_{4(2)}}$ and $Loop_{v_{4(3)}}$ of Figure 4.20 is advanced one iteration.

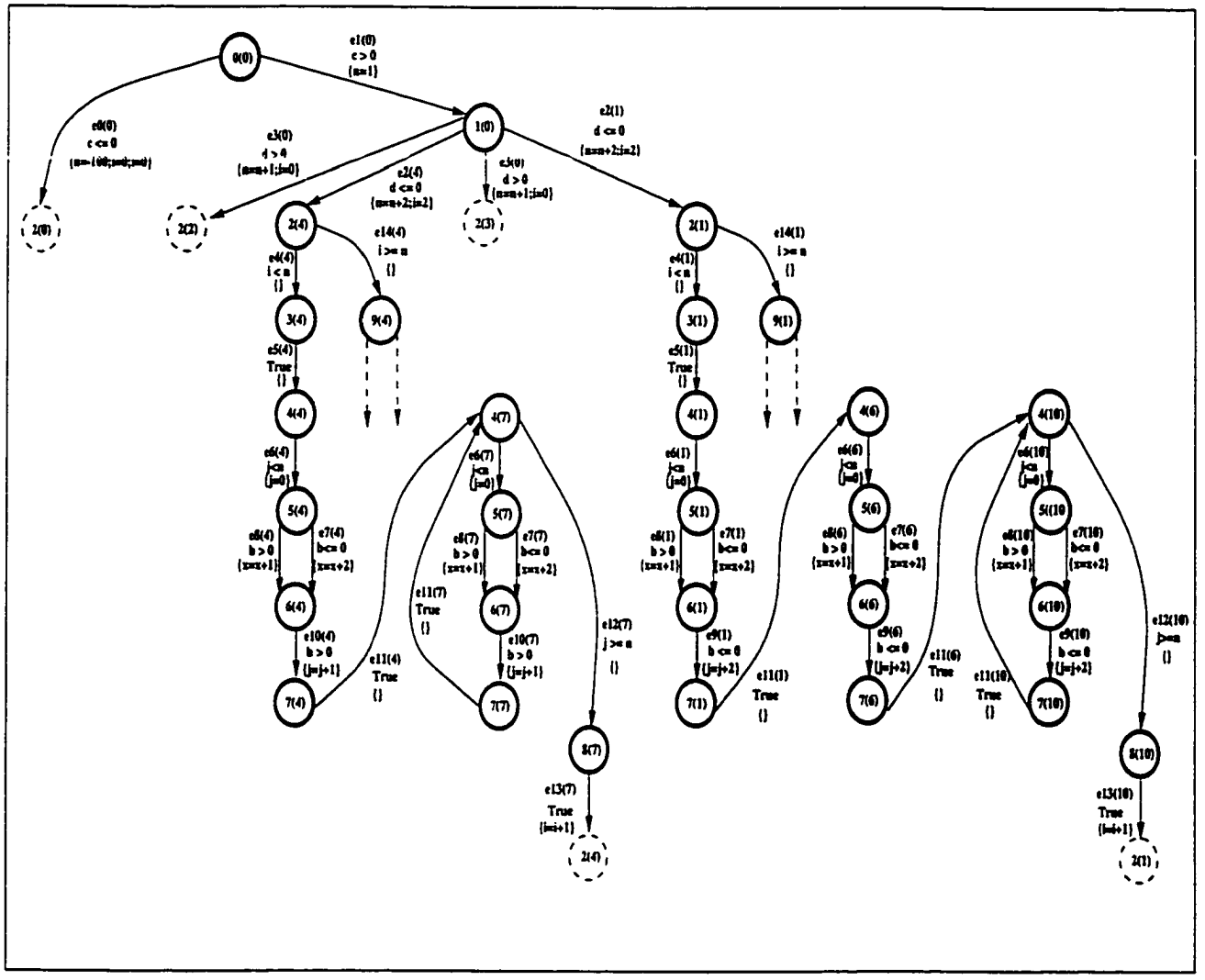

Figure 4.24: The EFSM graph after the loop with $Loop_{v_{4(1)}}$ of Figure 4.20 is advanced two iterations.

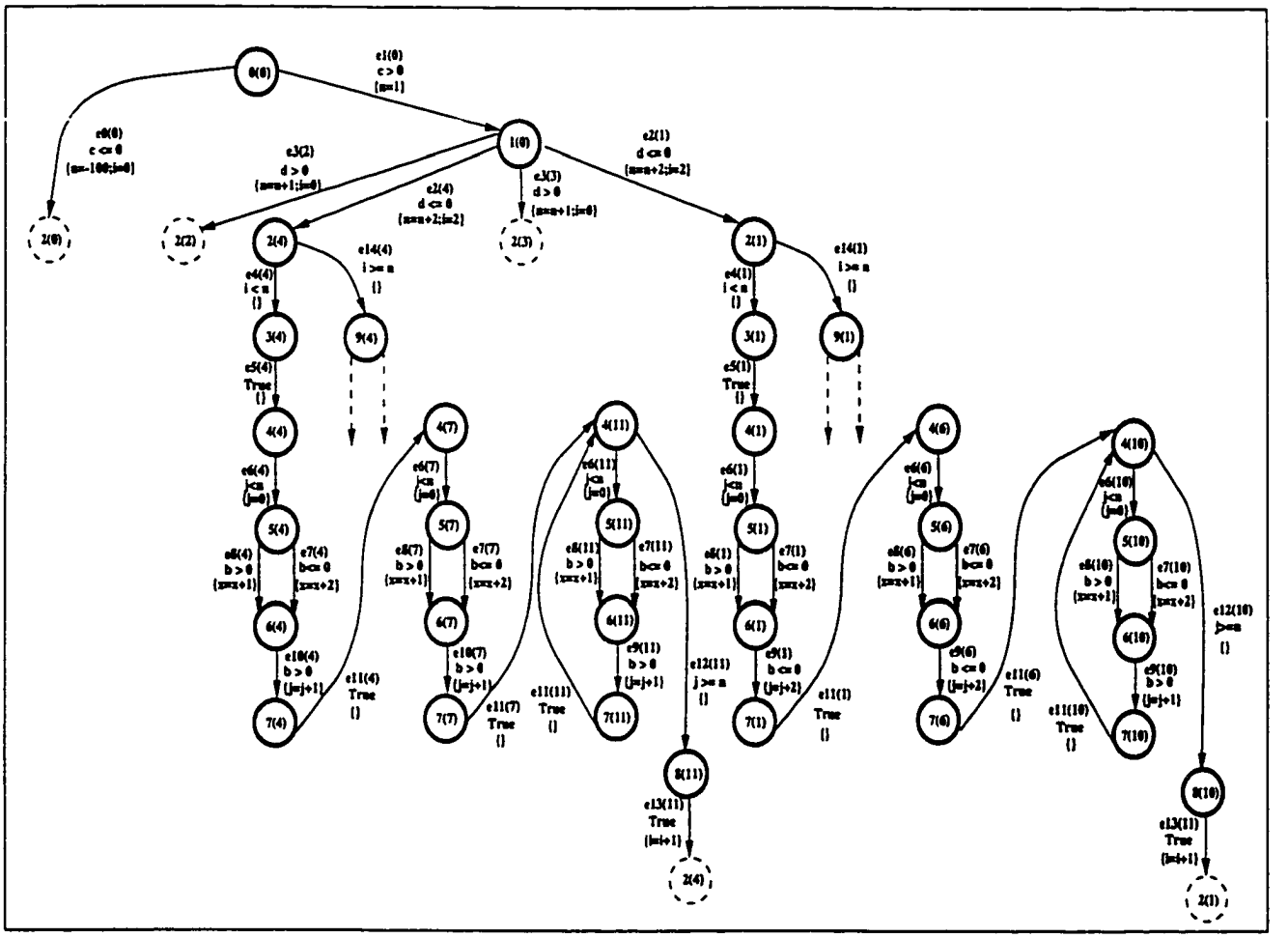

Figure 4.25: The EFSM graph after the loop with $Loop_{v_{4(4)}}$ of Figure 4.20 is advanced one iteration.

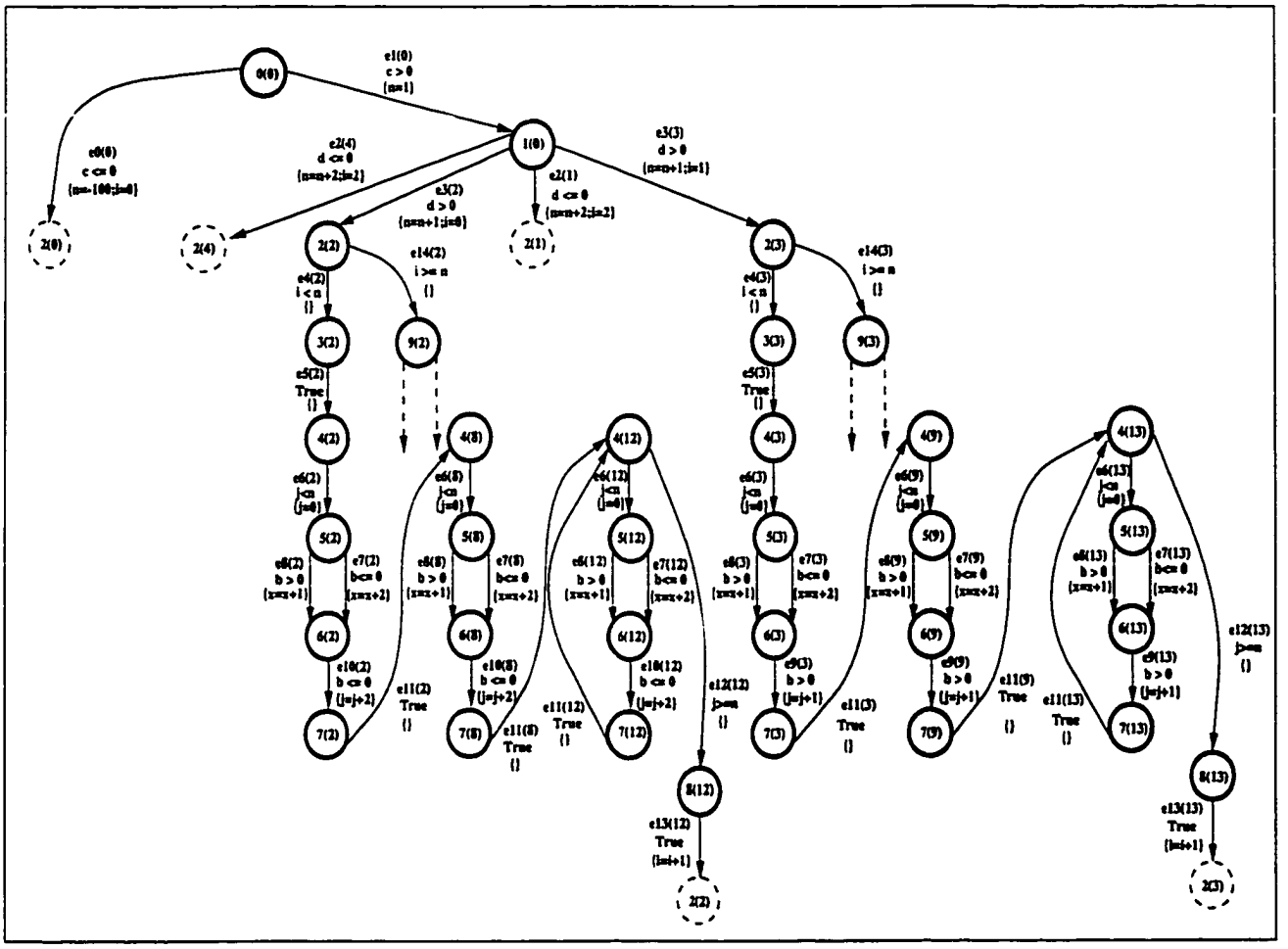

Figure 4.26: The EFSM graph after the loop with $Loop_{v_{4(3)}}$ of Figure 4.20 is advanced two iterations.

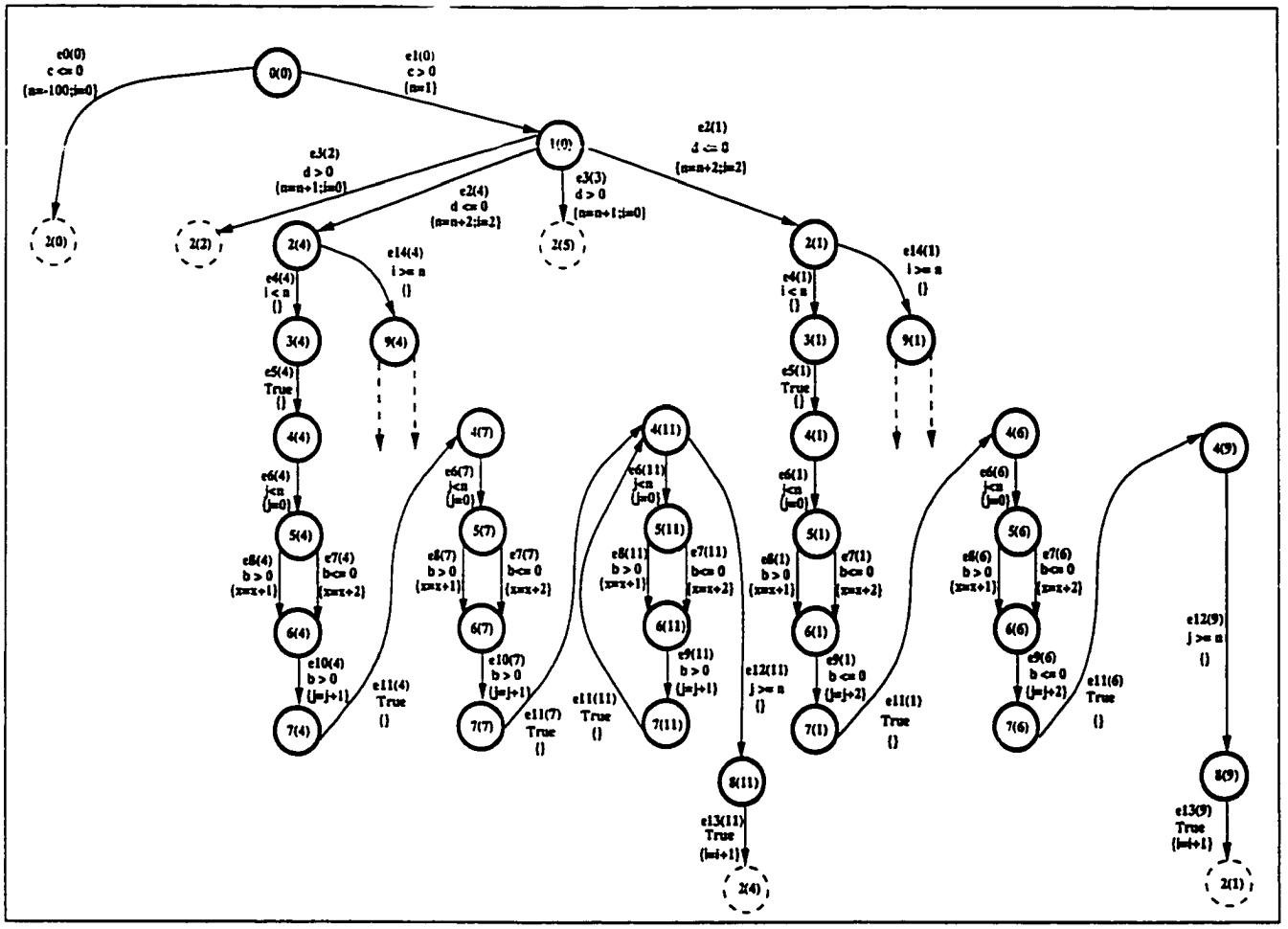

Figure 4.27: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{4(9)}$ of the graph in Figure 4.26.

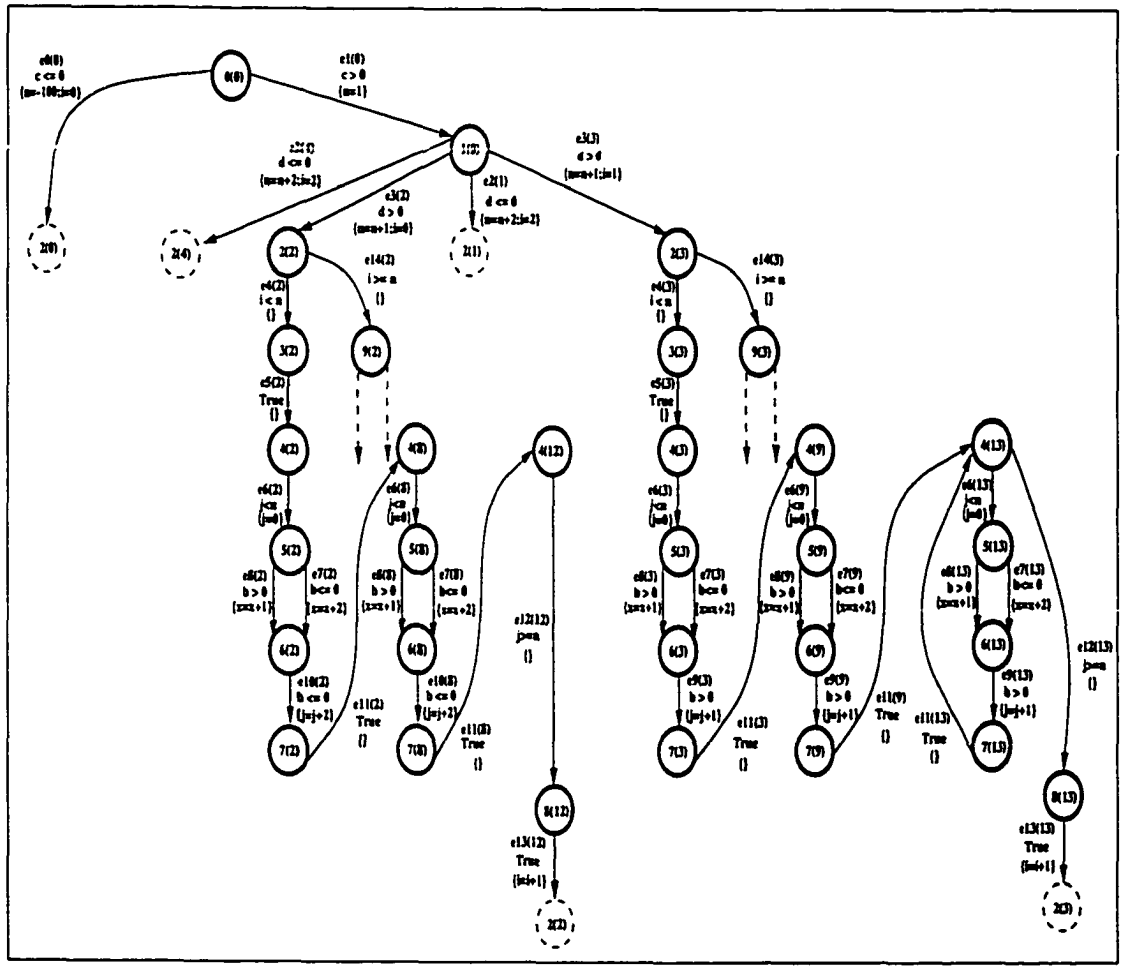

Figure 4.28: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{4(12)}$ of the graph in Figure 4.27.

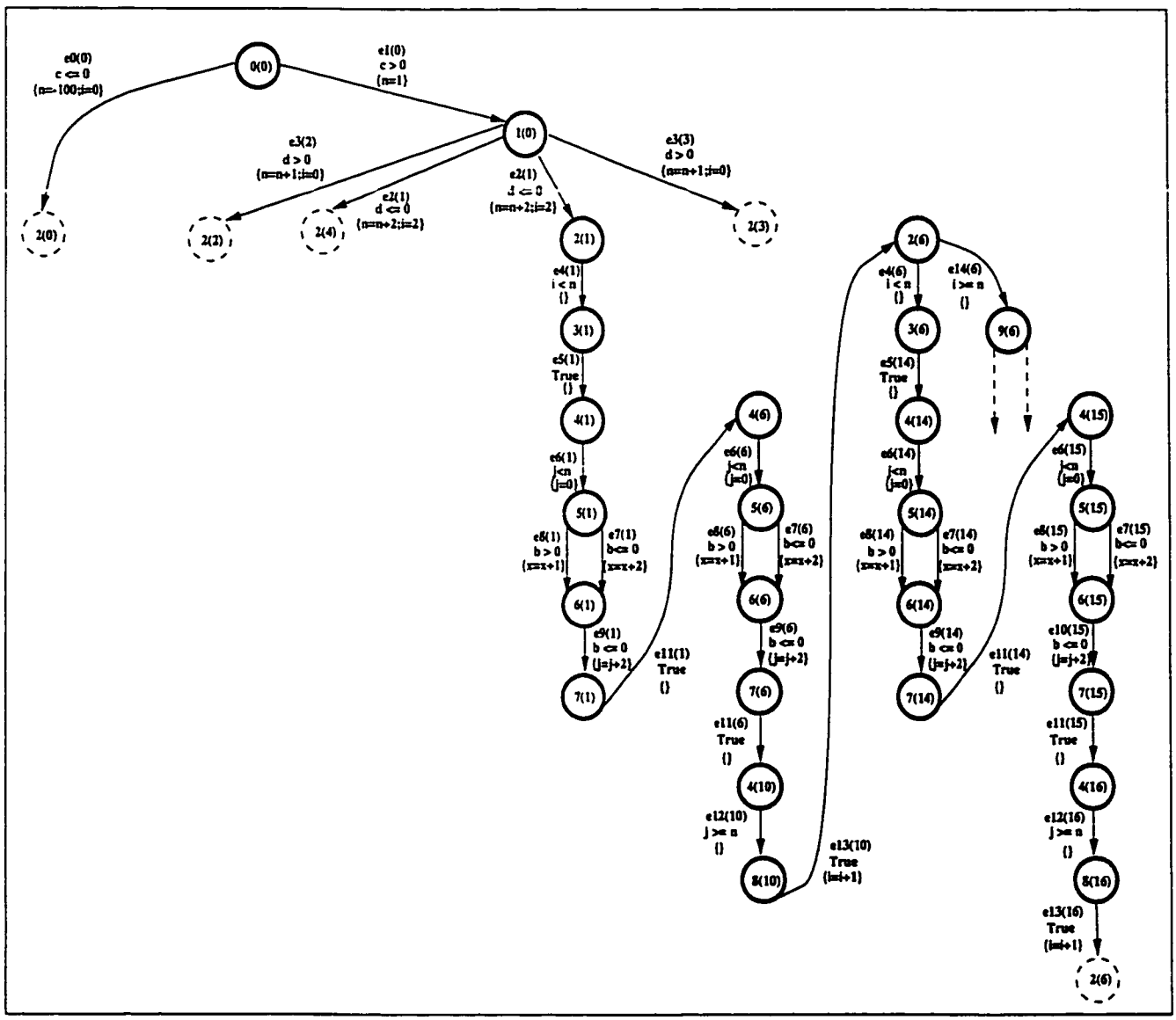

Figure 4.29: The EFSM graph after the loop with $Loop_{v_{2(1)}}$ of Figure 4.28 is advanced one iteration.

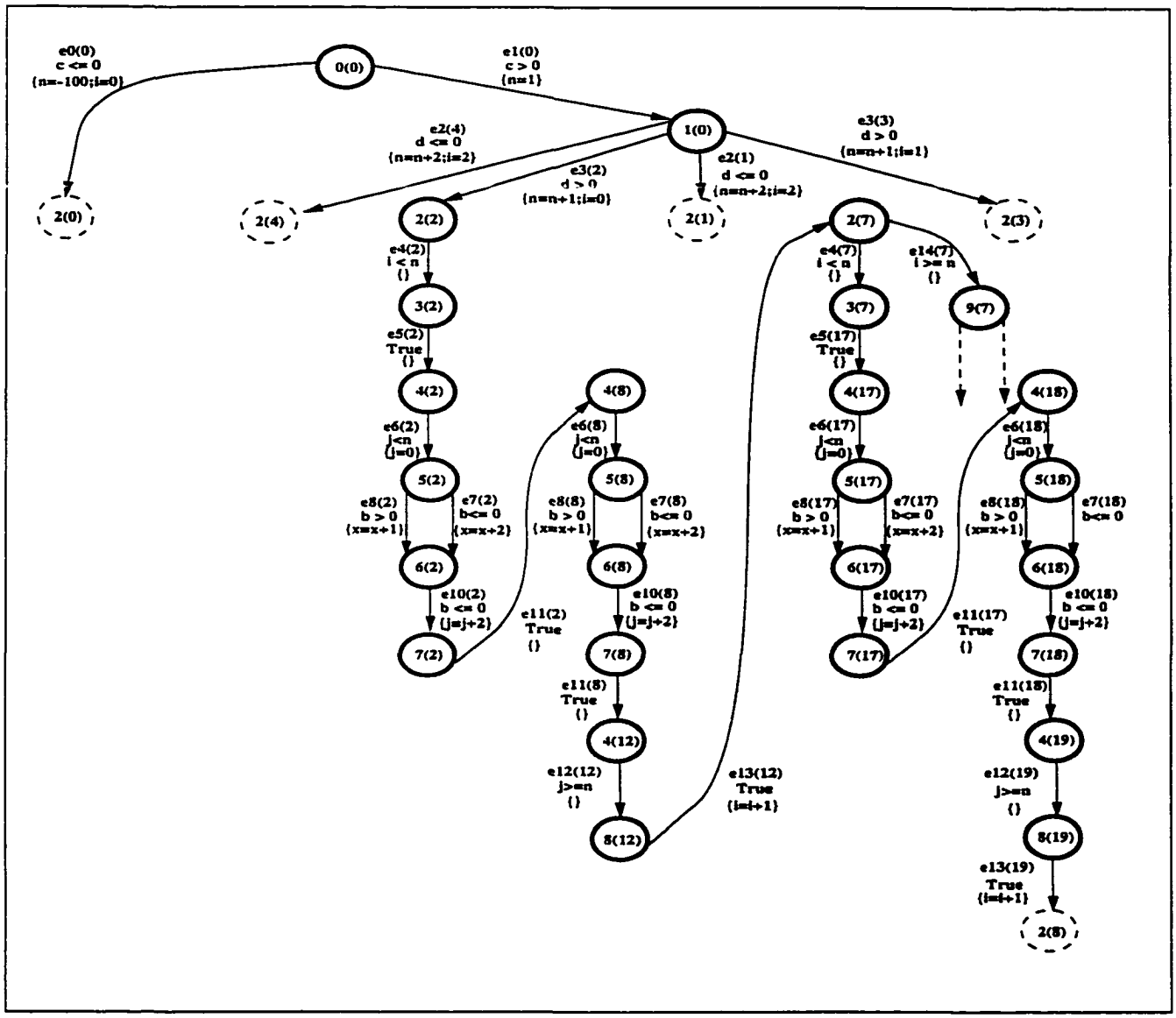

Figure 4.30: The EFSM graph after the loop with $Loop_{v_{2(2)}}$ of Figure 4.29 is advanced one iteration.

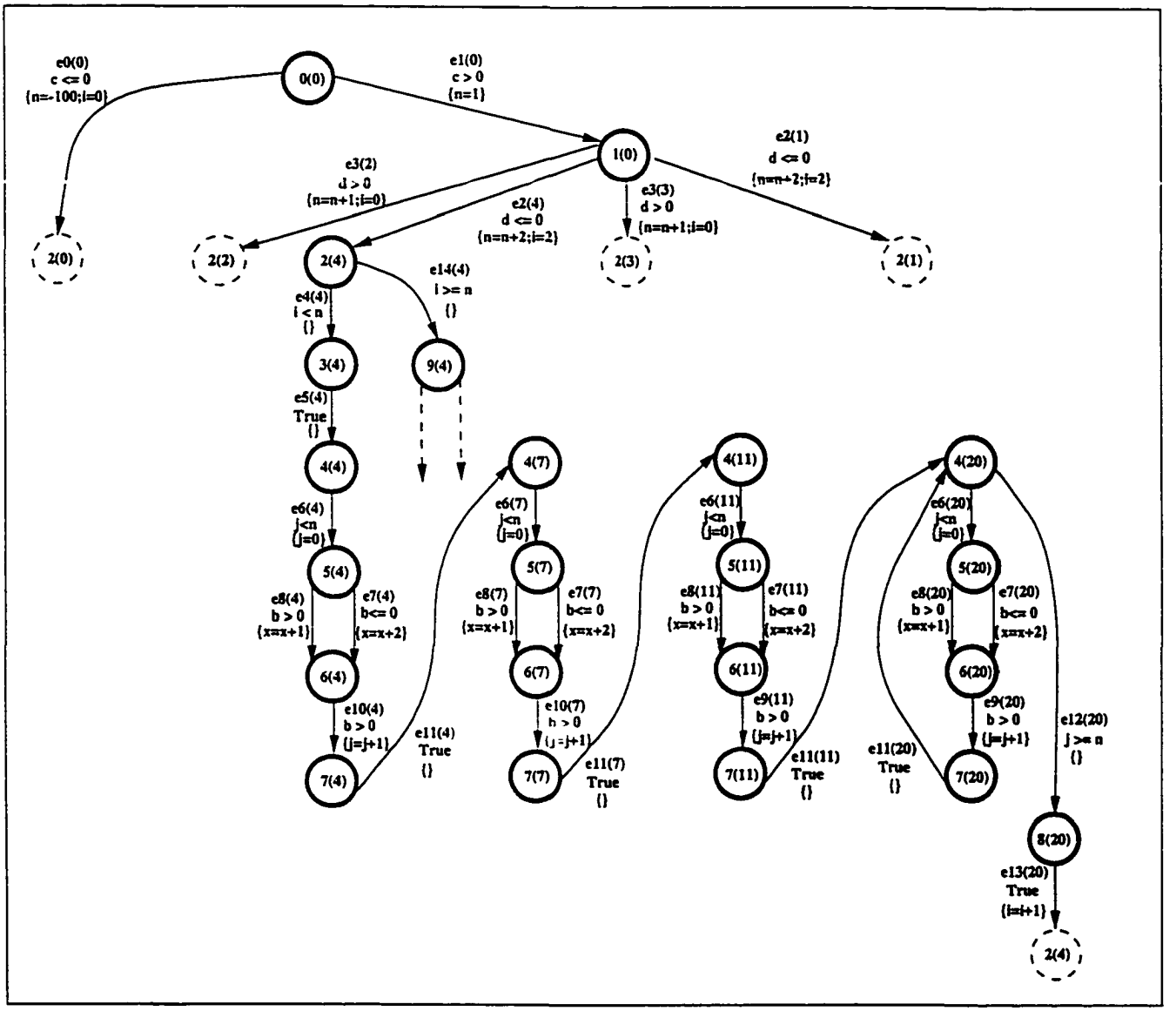

Figure 4.31: The EFSM graph after the loop with $Loop_{v_{4(4)}}$ of Figure 4.30 is advanced three iterations.

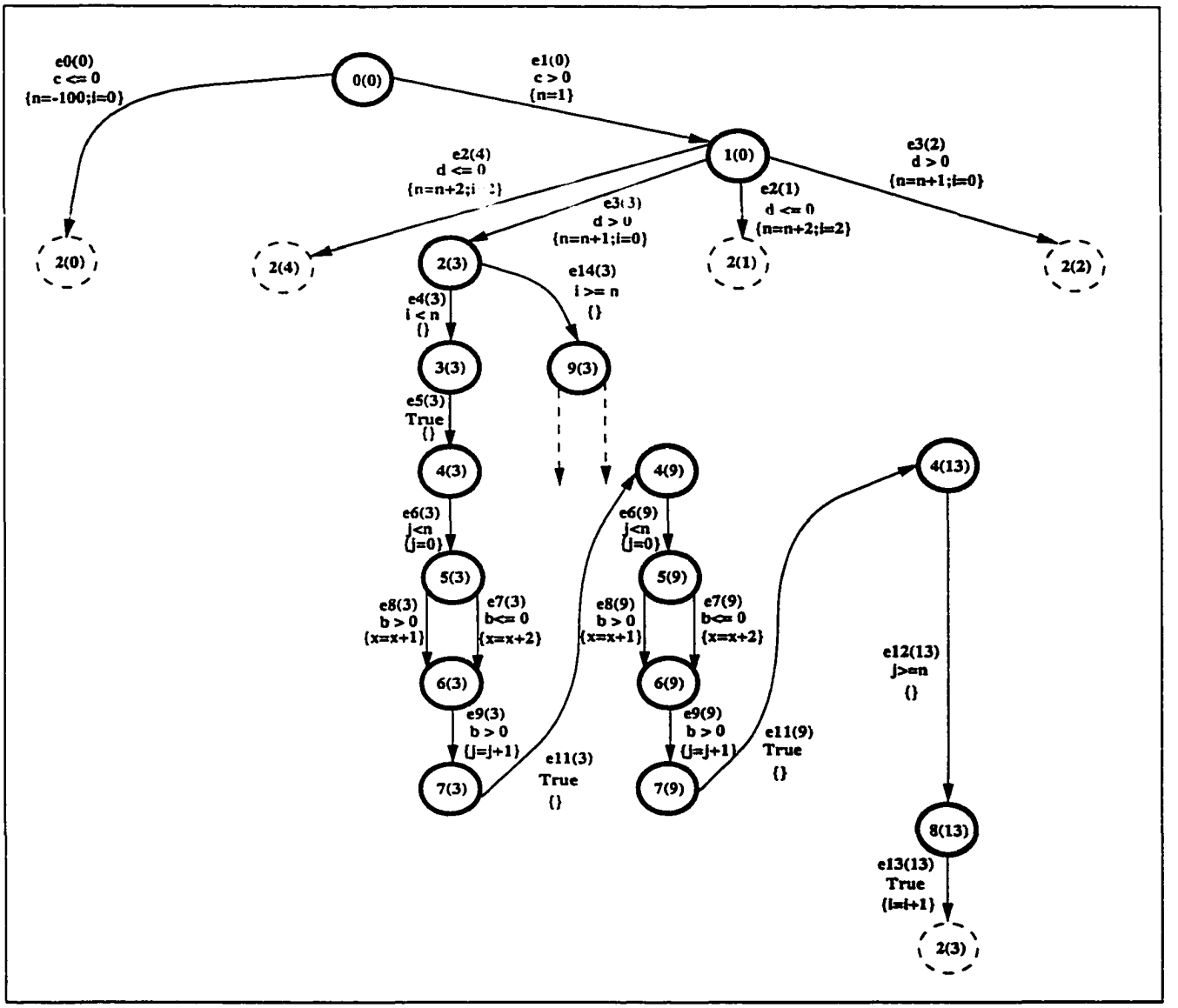

Figure 4.32: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{4(13)}$ of the graph in Figure 4.31.

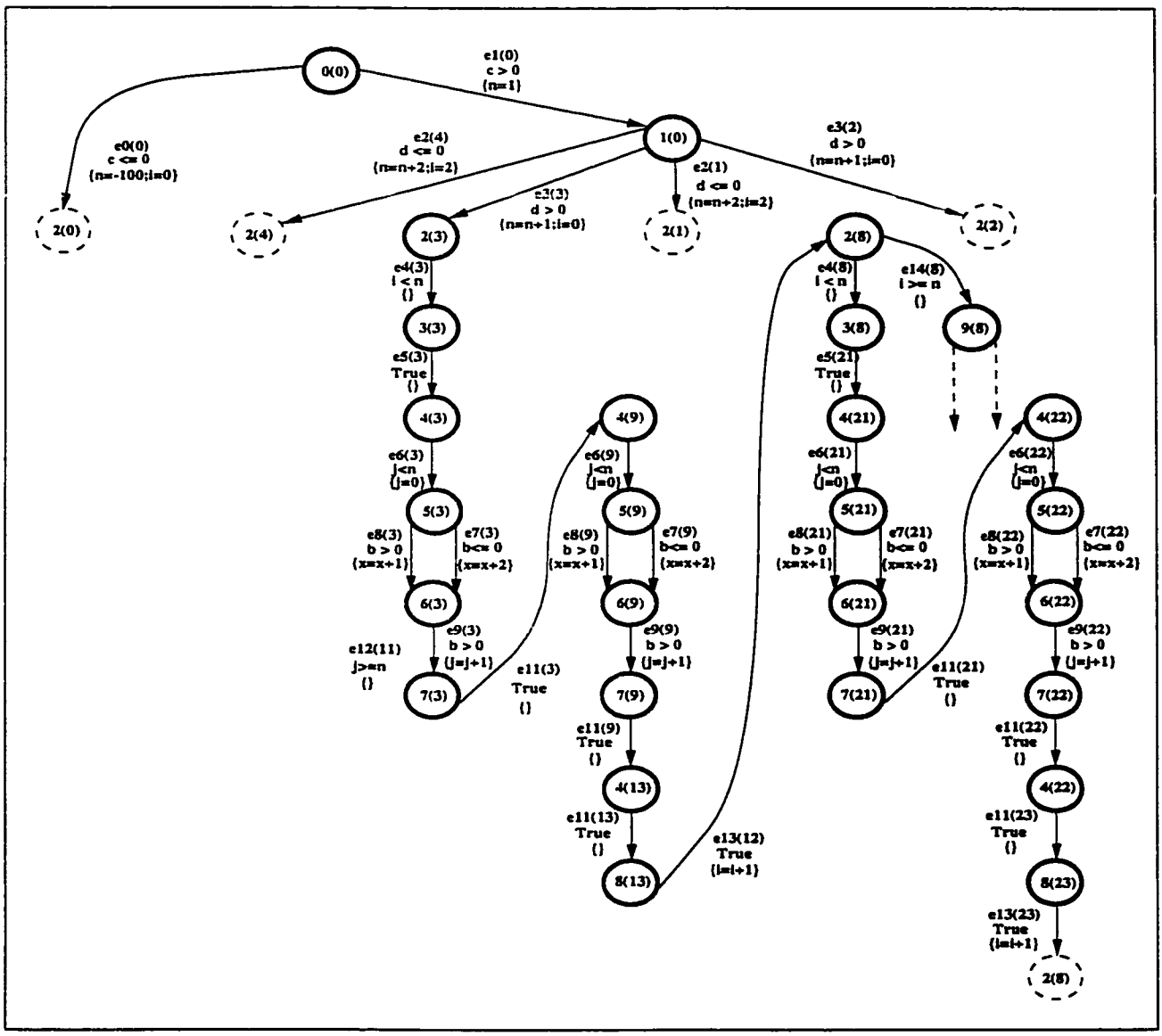

Figure 4.33: The EFSM graph after the loop with $Loop_{v_{2(3)}}$ of Figure 4.32 is advanced one iteration.

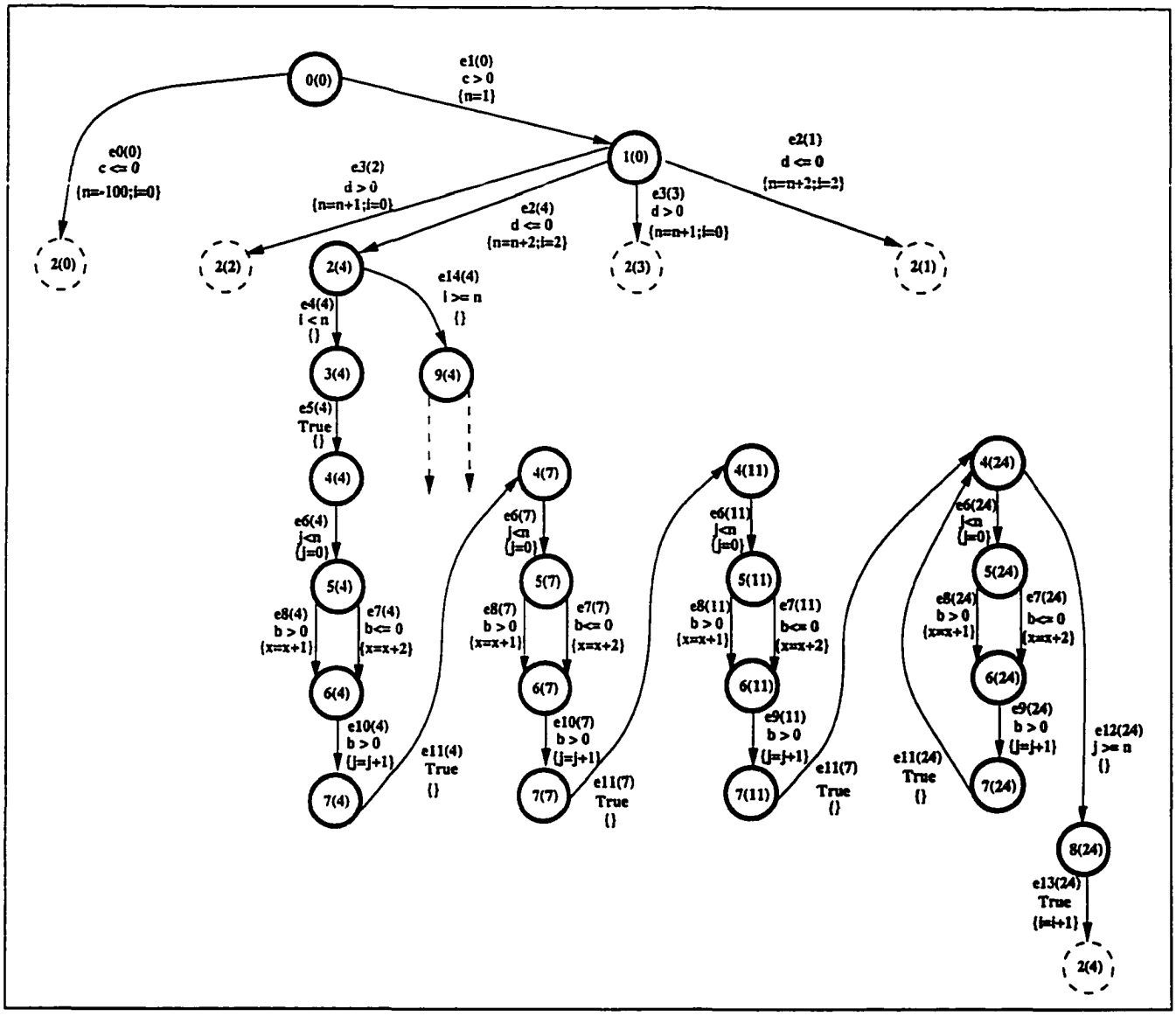

Figure 4.34: The EFSM graph after the loop with $Loop_{v_{4(4)}}$ of Figure 4.33 is advanced three iterations.

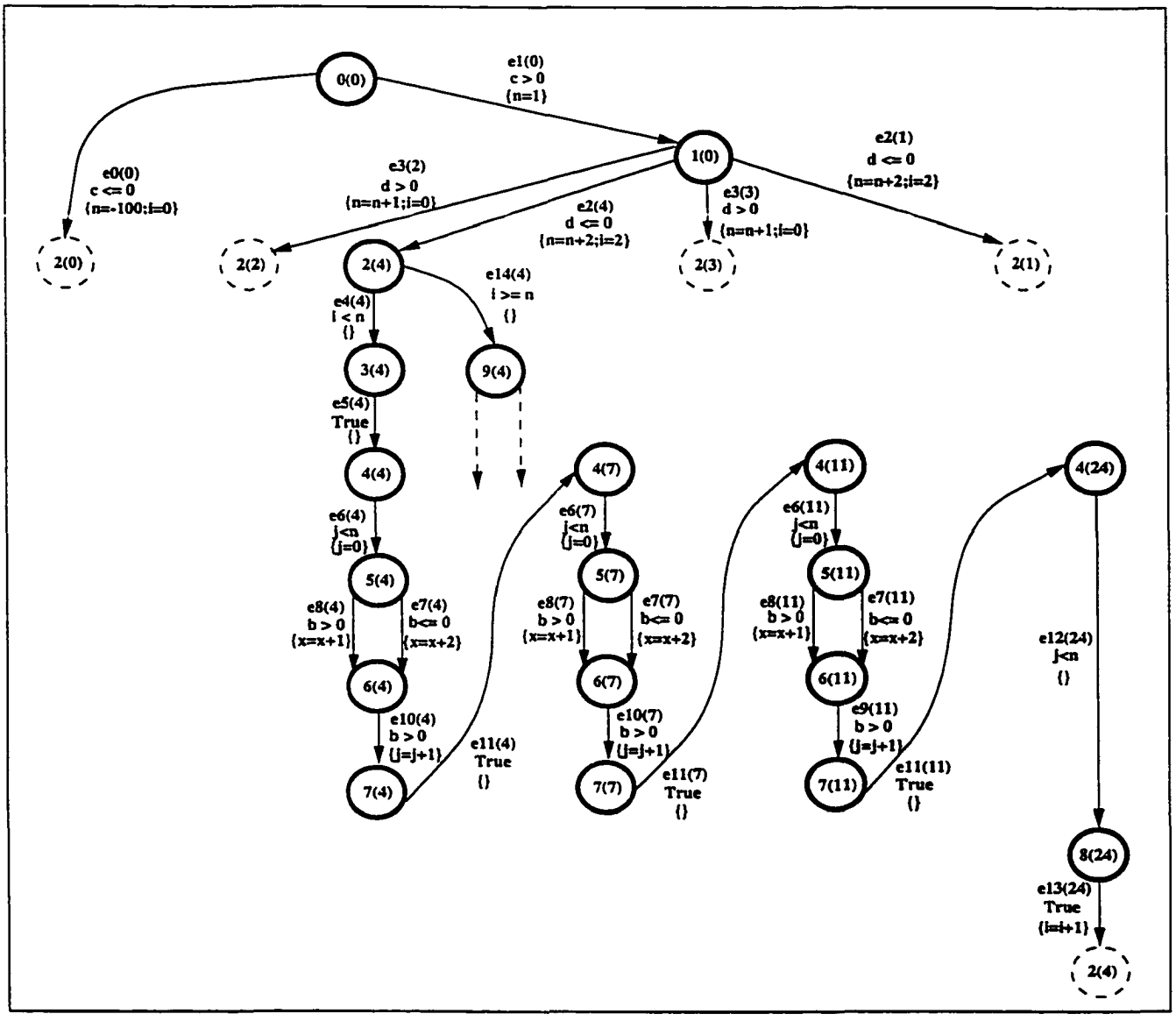

Figure 4.35: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{4(24)}$ of the graph in Figure 4.34.

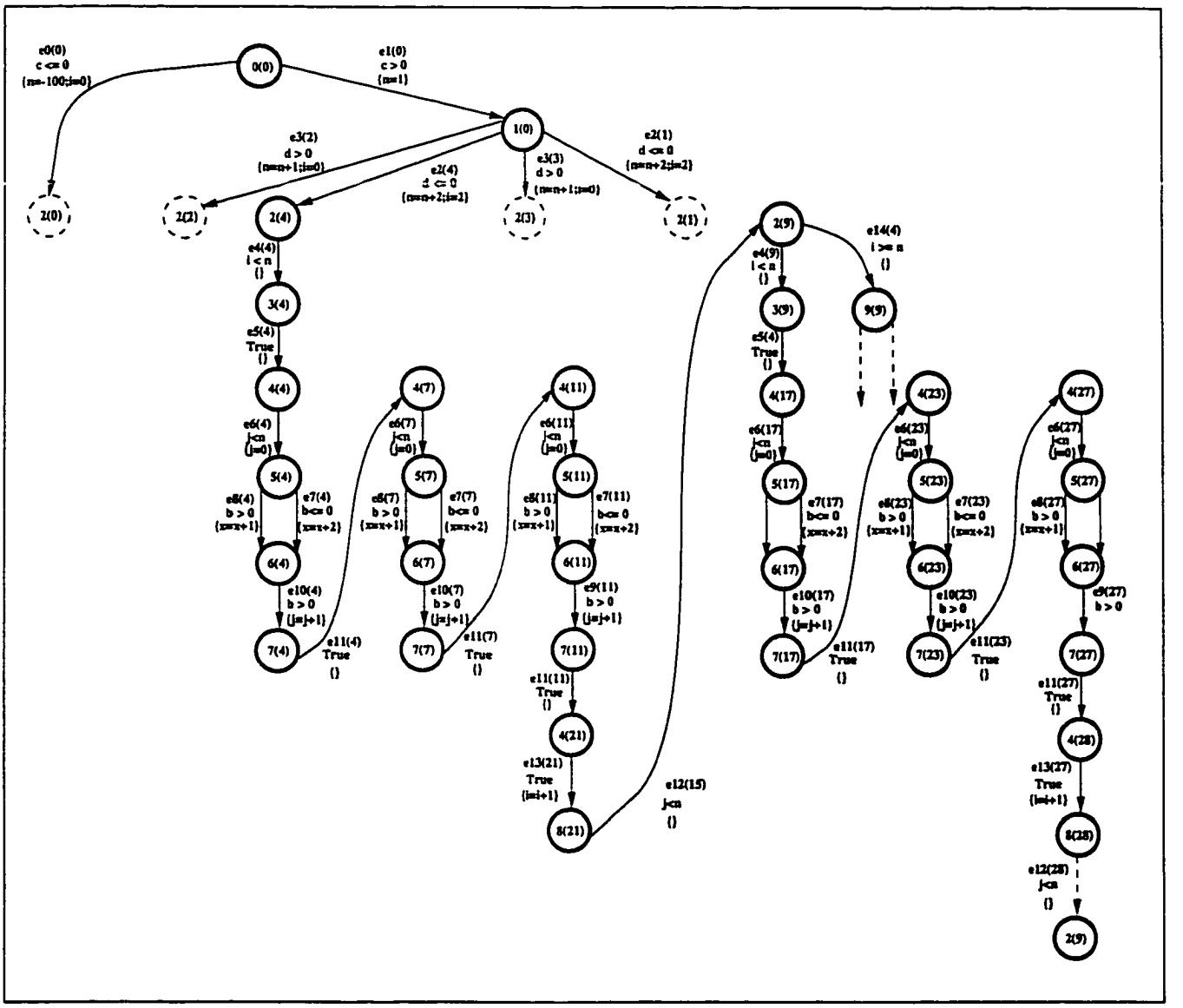

Figure 4.36: The EFSM graph after the loop with $Loop_{v_{4(4)}}$ of Figure 4.35 is advanced one iteration.

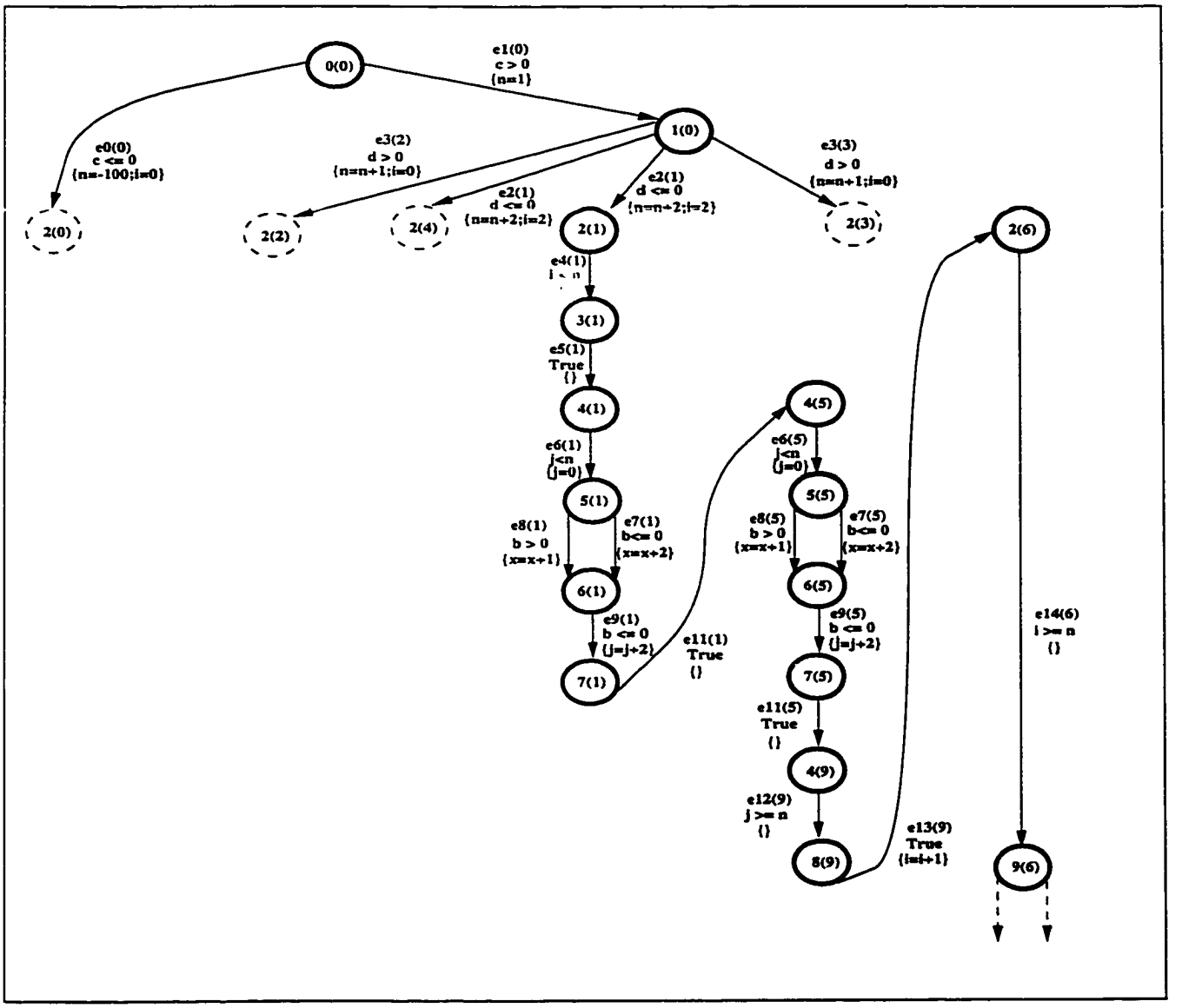

Figure 4.37: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{2(6)}$ of the graph in Figure 4.36.

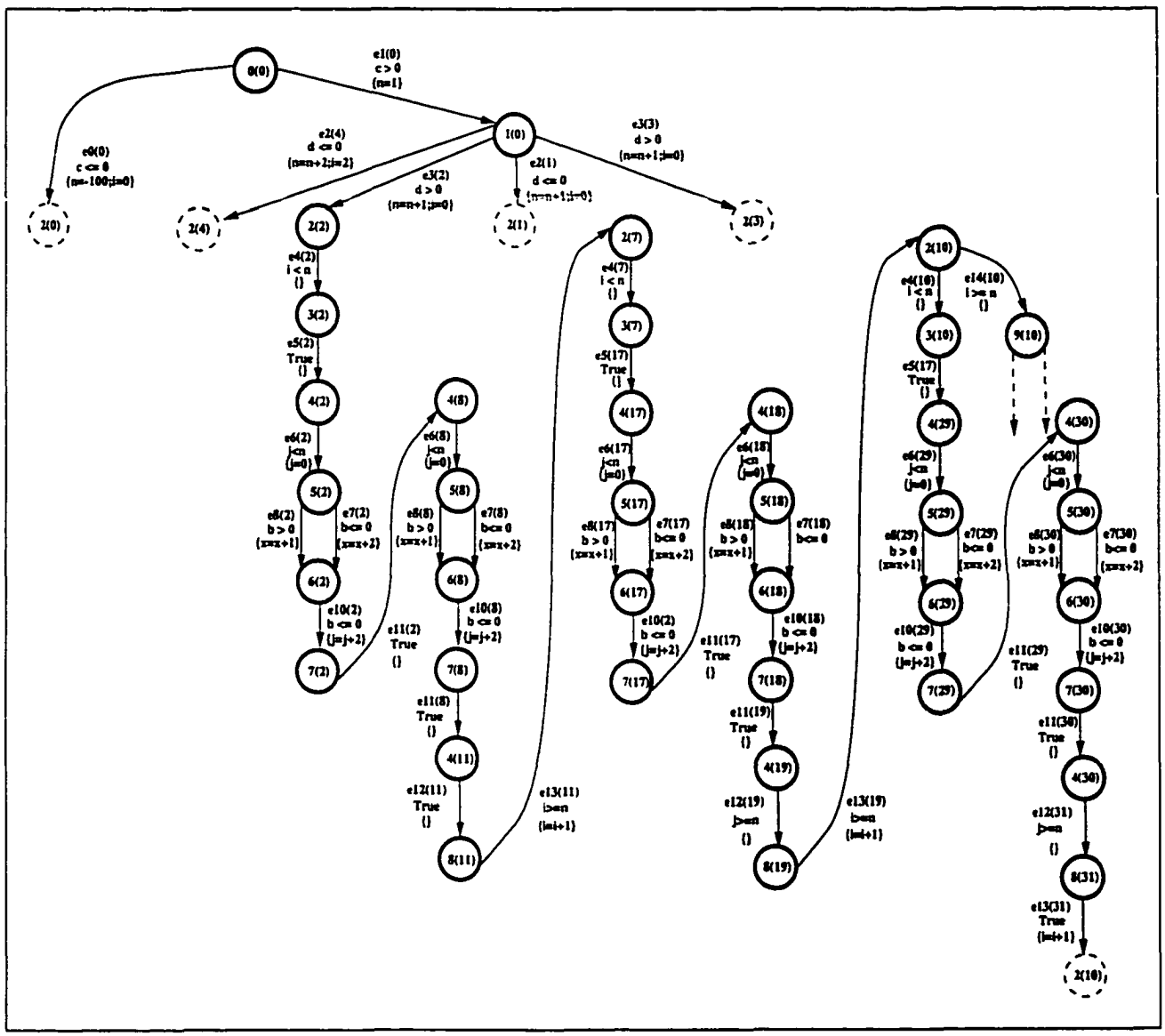

Figure 4.38: The EFSM graph after the loop with $Loop_{v_{2(2)}}$ of Figure 4.37 is advanced two iterations.

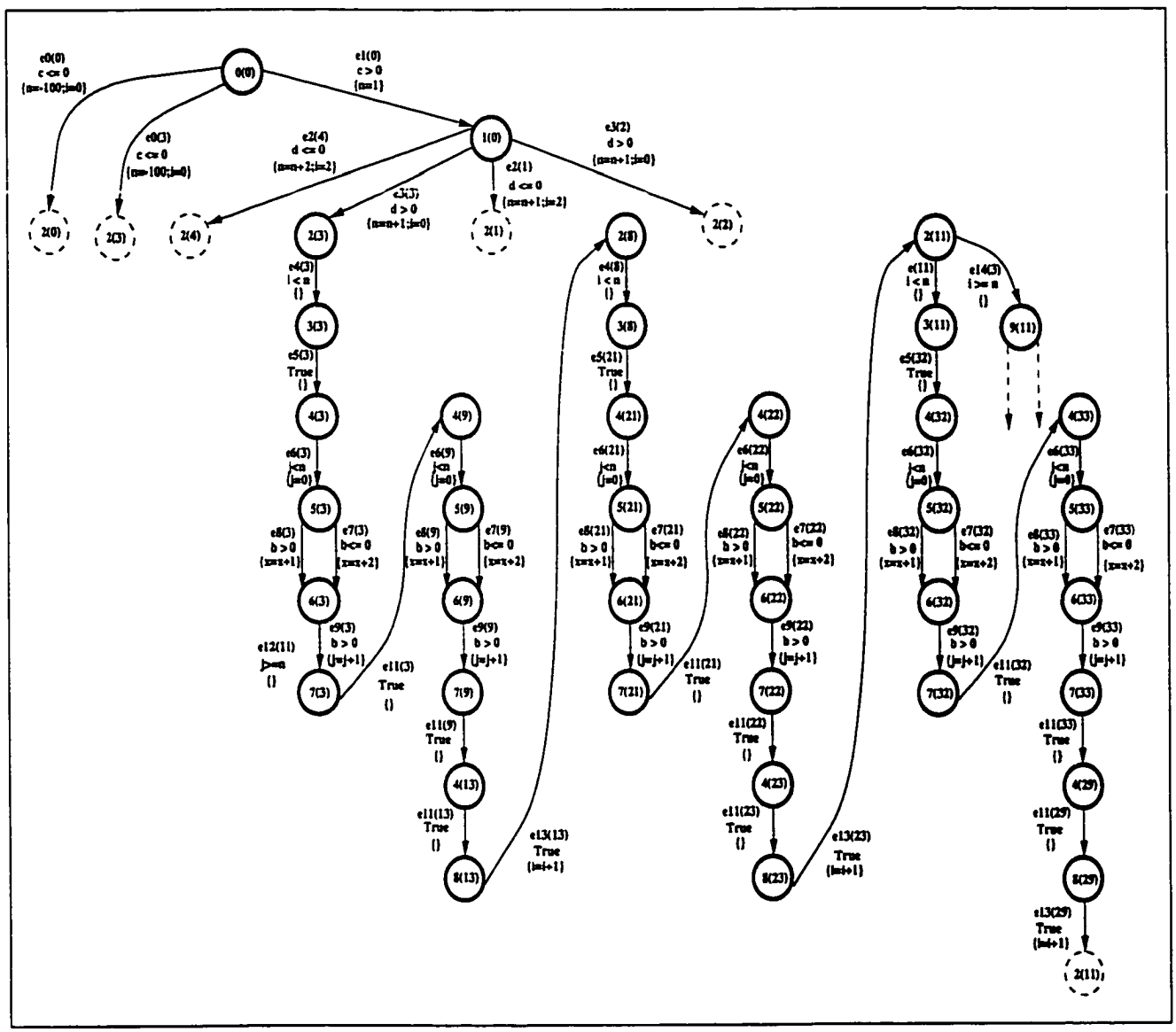

Figure 4.39: The EFSM graph after the loop with $Loop_{v_{2(3)}}$ of Figure 4.38 is advanced two iterations.

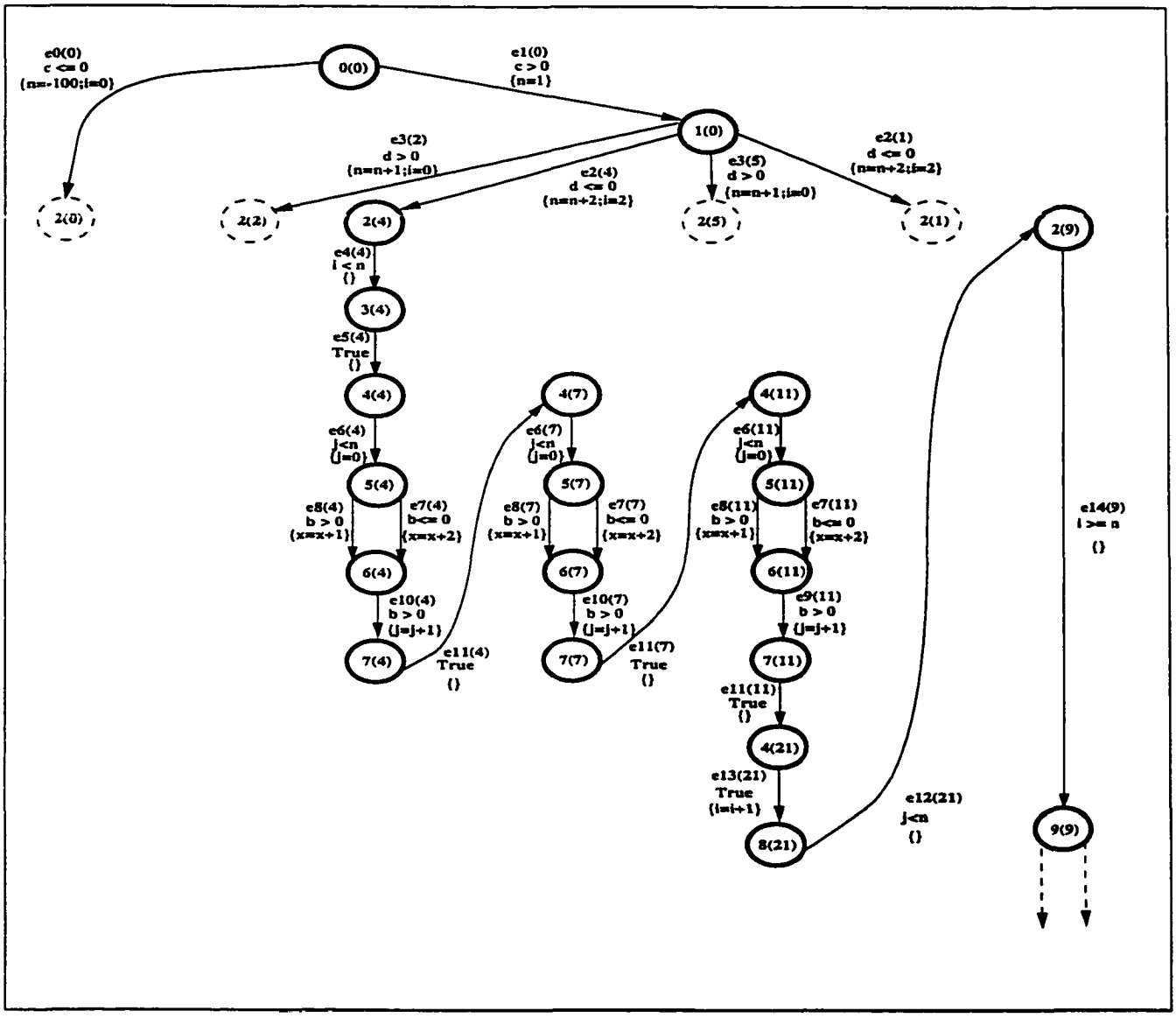

Figure 4.40: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{2(9)}$ of the graph in Figure 4.39.

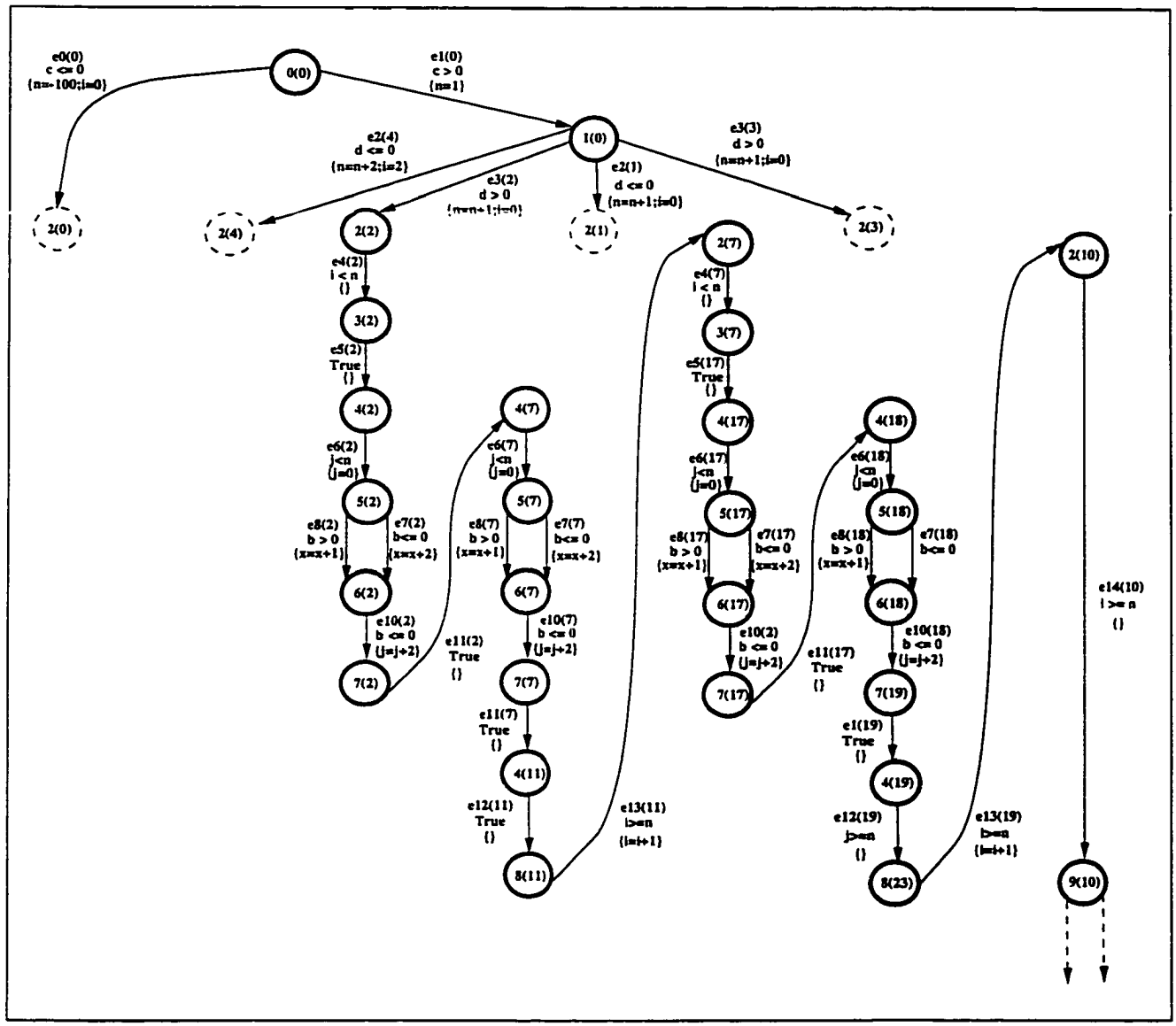

Figure 4.41: The EFSM graph after the infeasible edges are removed from the subgraph reachable from $v_{2(10)}$ of the graph in Figure 4.40.

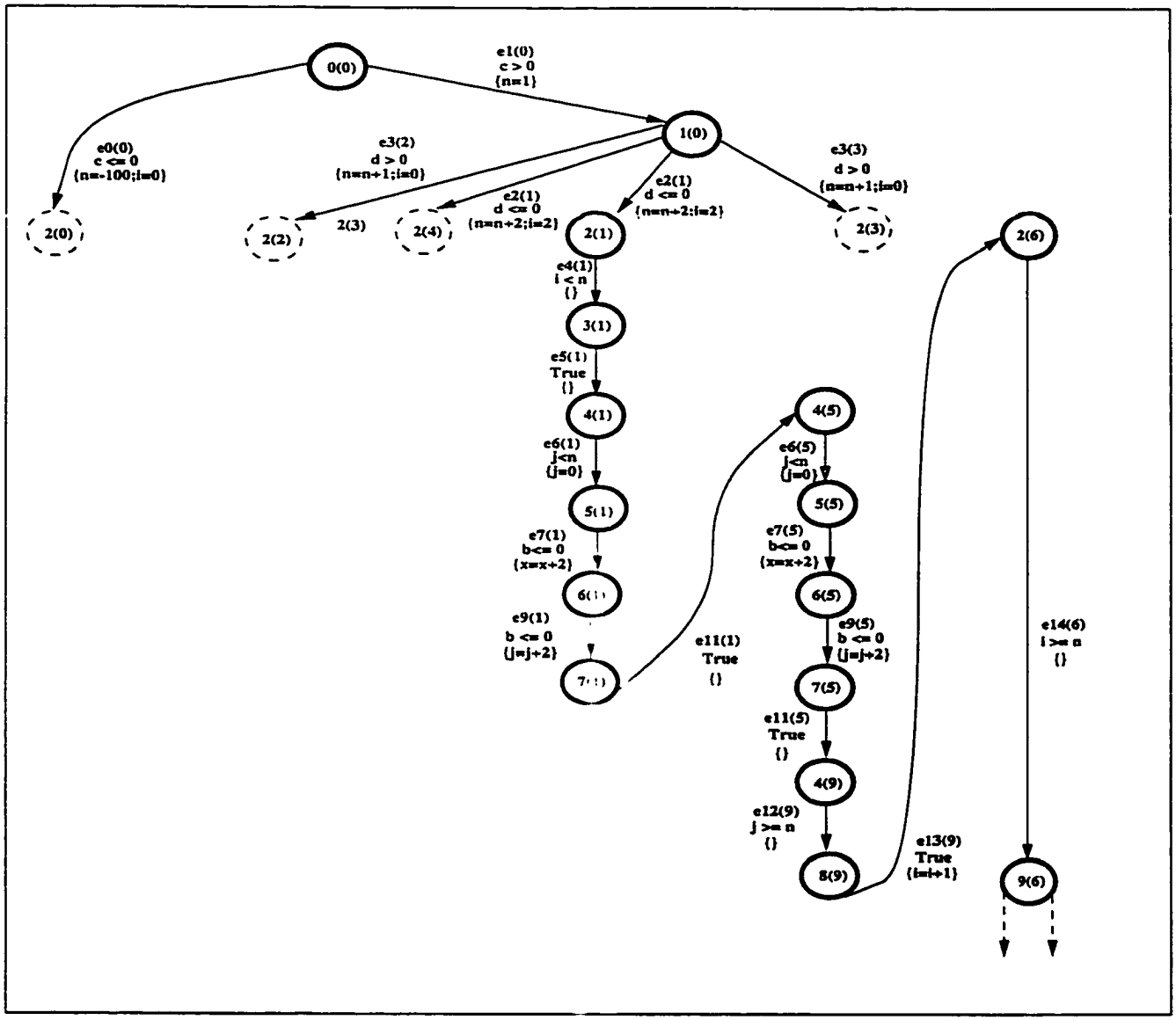

Figure 4.42: The EFSM graph after the condition inconsistency between $e_{8(1)}$ and $e_{9(1)}$ is eliminated from the EFSM graph of Figure 4.41.

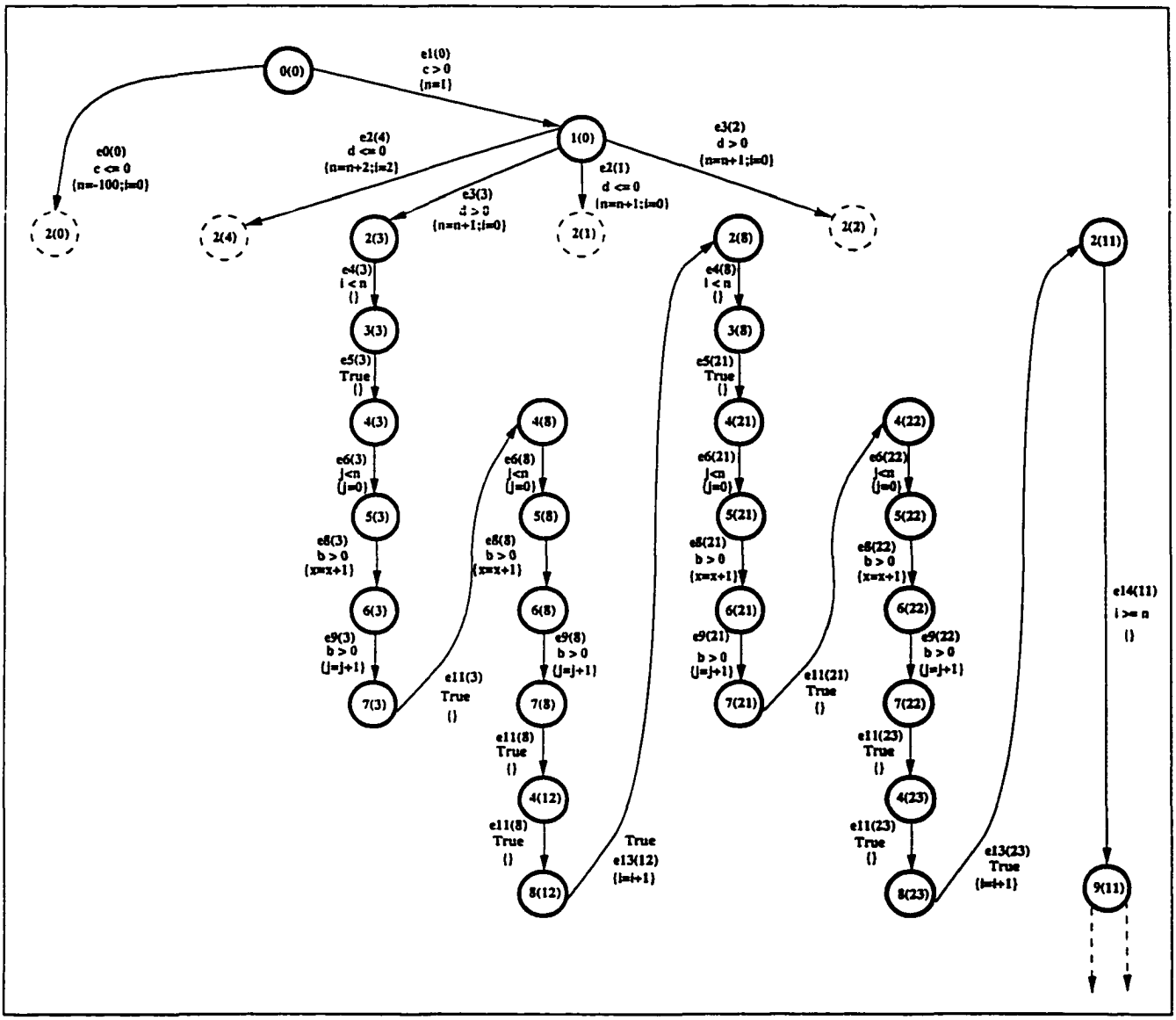

Figure 4.43: The EFSM graph after the condition inconsistency between $e_{7(3)}$ and $e_{9(3)}$ is eliminated from the EFSM graph of Figure 4.42.

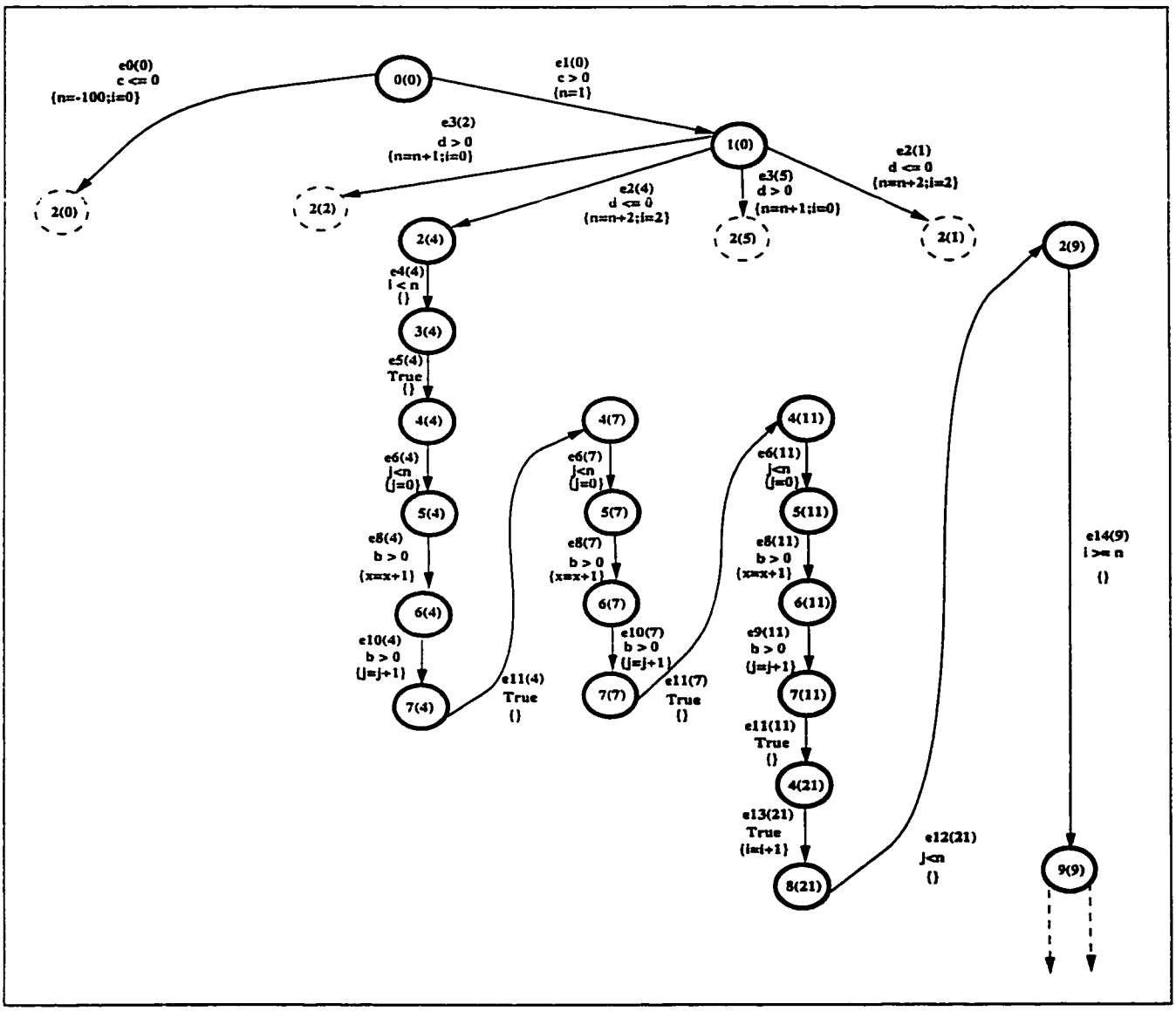

Figure 4.44: The EFSM graph after the condition inconsistency between $e_{7(4)}$ and $e_{10(4)}$ is eliminated from the EFSM graph of Figure 4.43.

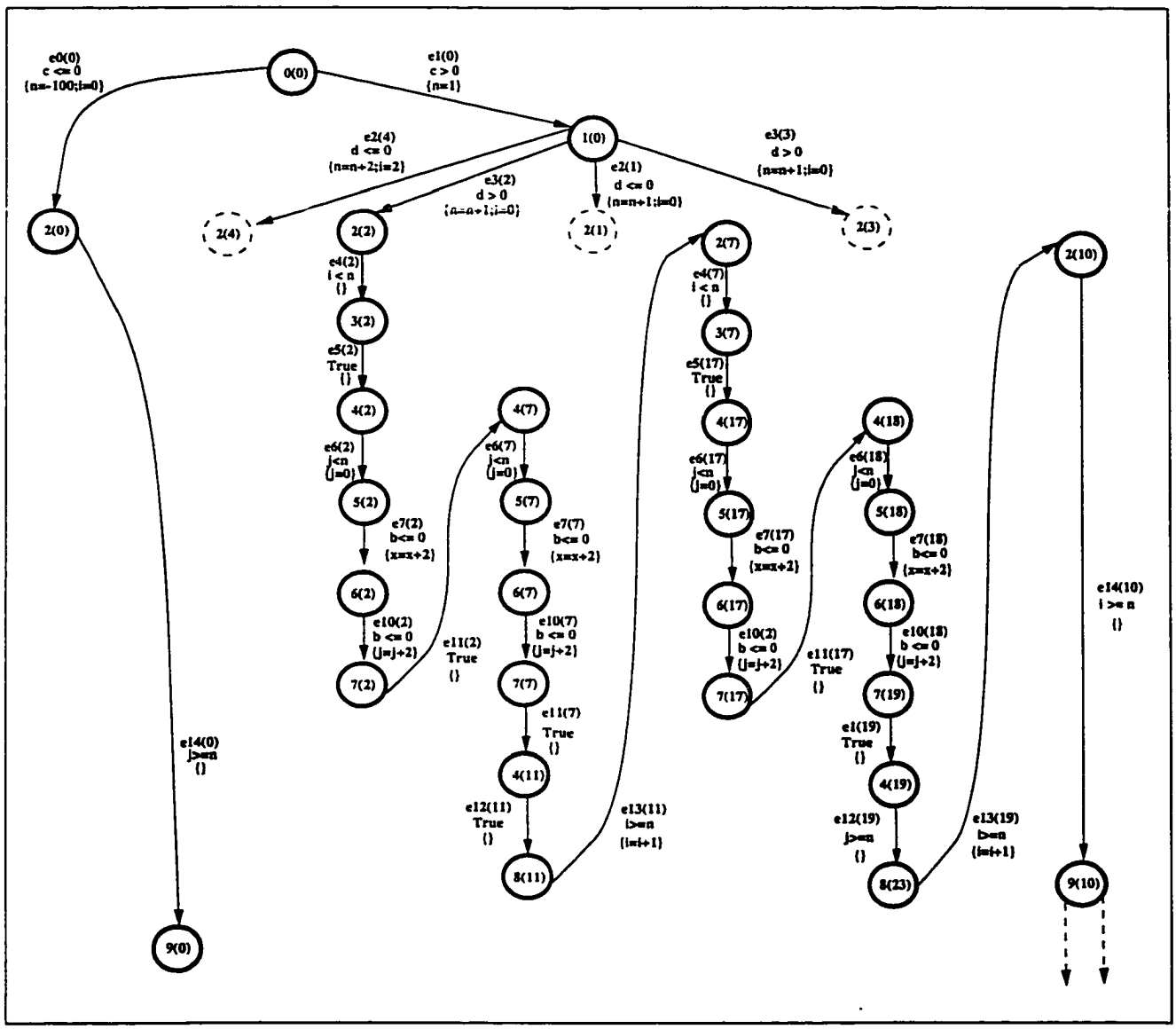

Figure 4.45: The EFSM graph after the condition inconsistency between $e_{8(2)}$ and $e_{10(2)}$ is eliminated from the EFSM graph of Figure 4.44.

4.3 EFSM Graphs with Nested and Concatenated Loops

In the previous two sections, the process of detecting and eliminating inconsistencies from the EFSM graphs with simple and nested loops was demonstrated. In this section, the elimination of inconsistencies from the EFSM graphs with both nested and concatenated loops will be discussed. The inconsistency detection and elimination algorithms are applied to the EFSM graph of Figure 4.46, which contains both nested and concatenated loops.

Let us assume that two loops $Loop_1$ and $Loop_2$ with the entry/exit nodes of $Loop_{v_i}$ and $Loop_{v_j}$, respectively, are concatenated such that:

$$\exists\, e_k : (e_k \in E^{exit}_{Loop_{v_i}}) \wedge (e_k \in E^{in}_{Loop_{v_j}})$$

According to the MBF graph traversal, none of the outgoing edges of the loop body of $Loop_2$ is traversed until $Loop_1$ is completely analyzed.

The application of the action inconsistency detection to the EFSM graph appearing in Figure 4.46 indicates the existence of an action inconsistency among the edges leading to v_1 and its outgoing edges. The EFSM graph of Figure 4.47 shows the resulting graph after this inconsistency is eliminated.

Figures 4.48 and 4.49 depict the resulting EFSM graphs after each of the loops with the entry/exit nodes of $v_{2(0)}$ and $v_{2(1)}$, of the graph of Figure 4.48, is advanced once.

For simplicity, only the subgraph starting from $v_{1(0)}$ of Figure 4.4 will be analyzed. Notice that the nested loop with the entry/exit node of $v_{1(0)}$ is analyzed as described in Section 4.3. The resulting graph when this loop is completely analyzed is shown in Figure 4.53

In a similar fashion, the analysis of the loop with the entry/exit node of $v_{6(0)}$ is performed as shown in Figures 4.52 through 4.57.

The subgraph starting from $v_{1(0)}$ of the graph of Figure 4.57 does not contain any action inconsistencies. However, the graph contains condition inconsistencies. For example, a path that includes the edges $e_{5(0)}$ and $e_{4(2)}$ will not be feasible. Upon reaching $v_{4(0)}$ in the DF graph traversal, a condition inconsistency is detected between $e_{5(0)}$ and $e_{4(2)}$. The graph in Figure 4.58 shows the resulting graph when this condition inconsistency is eliminated. Notice that the elimination of this inconsistency removes also the condition inconsistencies among several other edges such as $e_{14(2)}$ and $e_{13(3)}$.

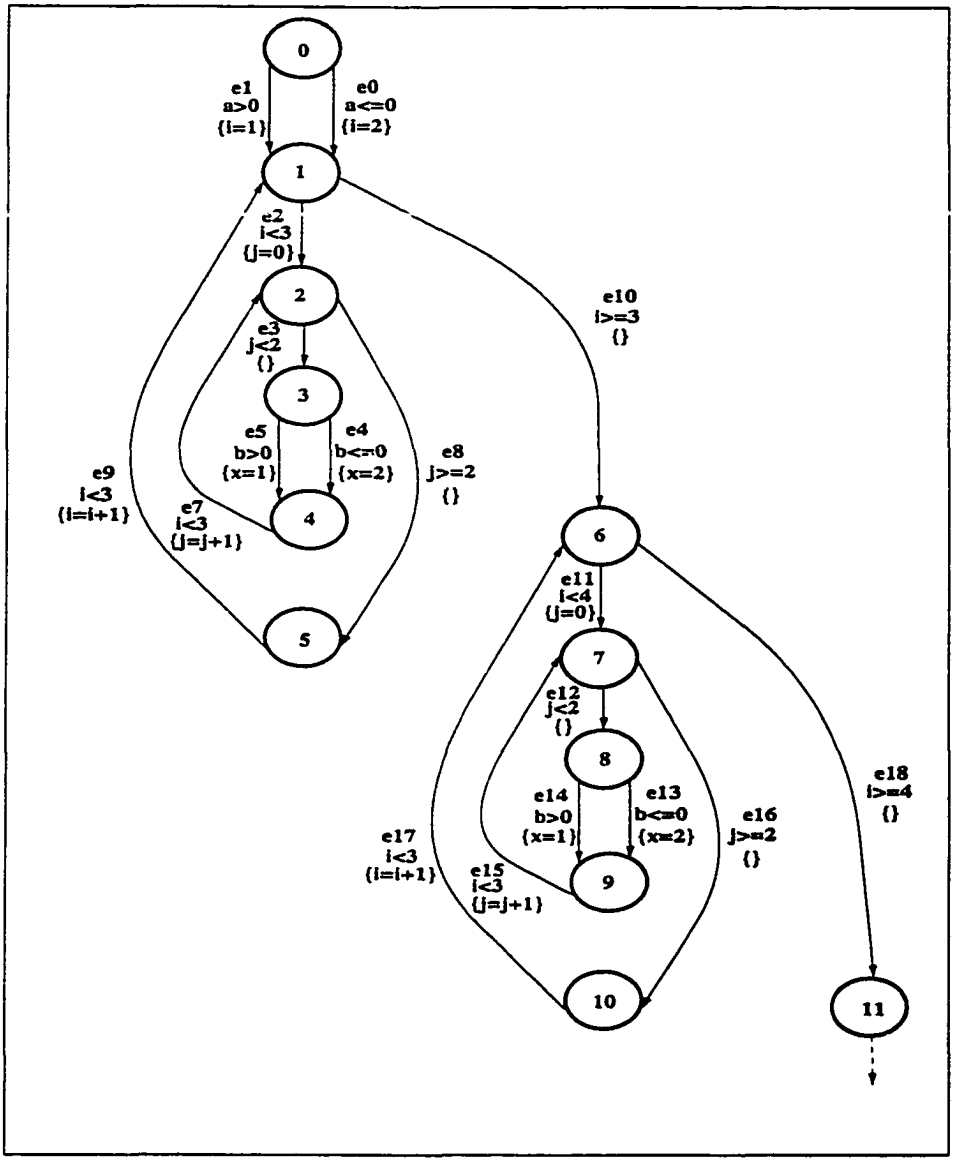

Figure 4.46: An EFSM graph with nested and concatenated loops.

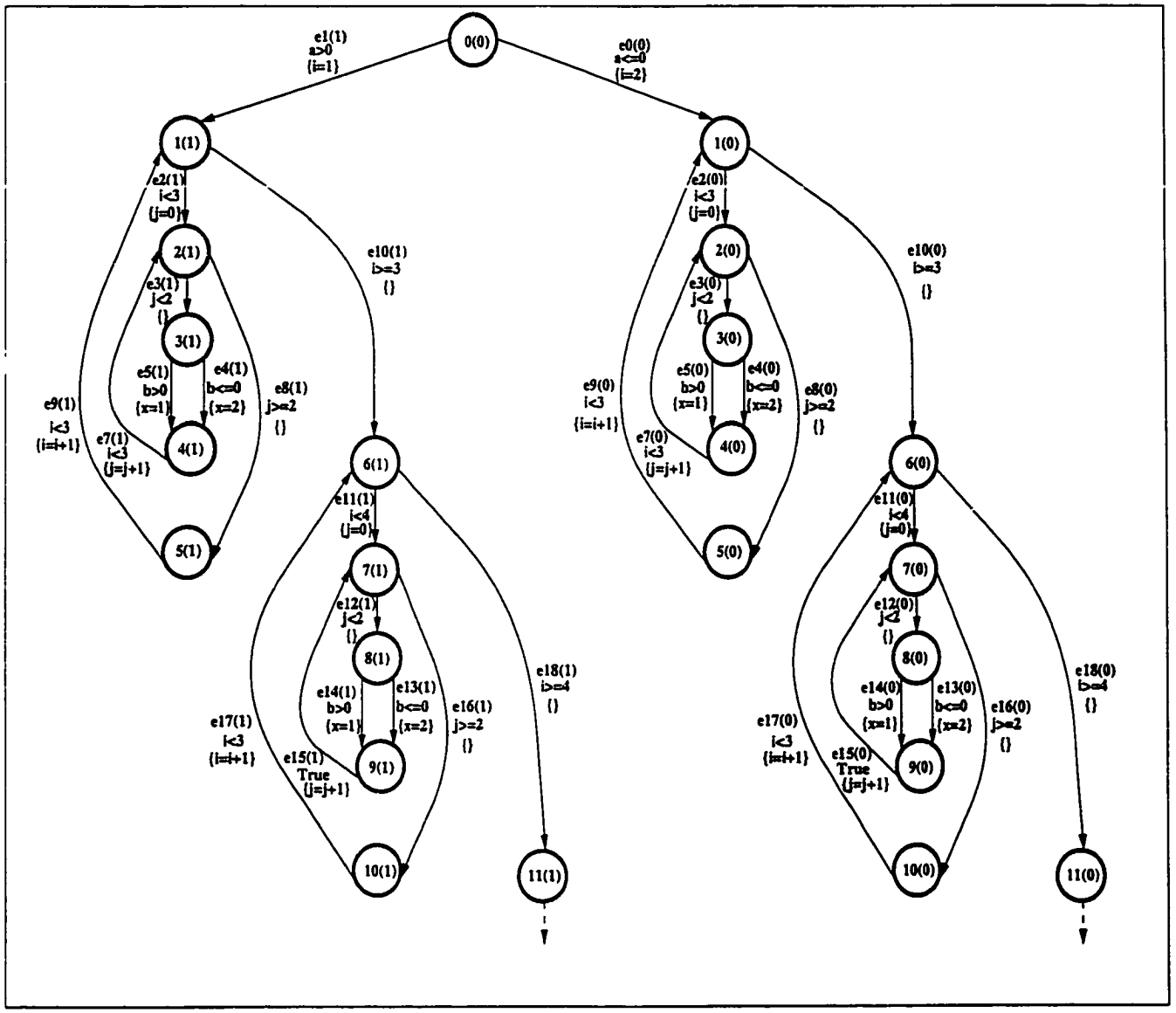

Figure 4.47: The EFSM graph after the graph of Figure 4.46 is split due to e_1 action.

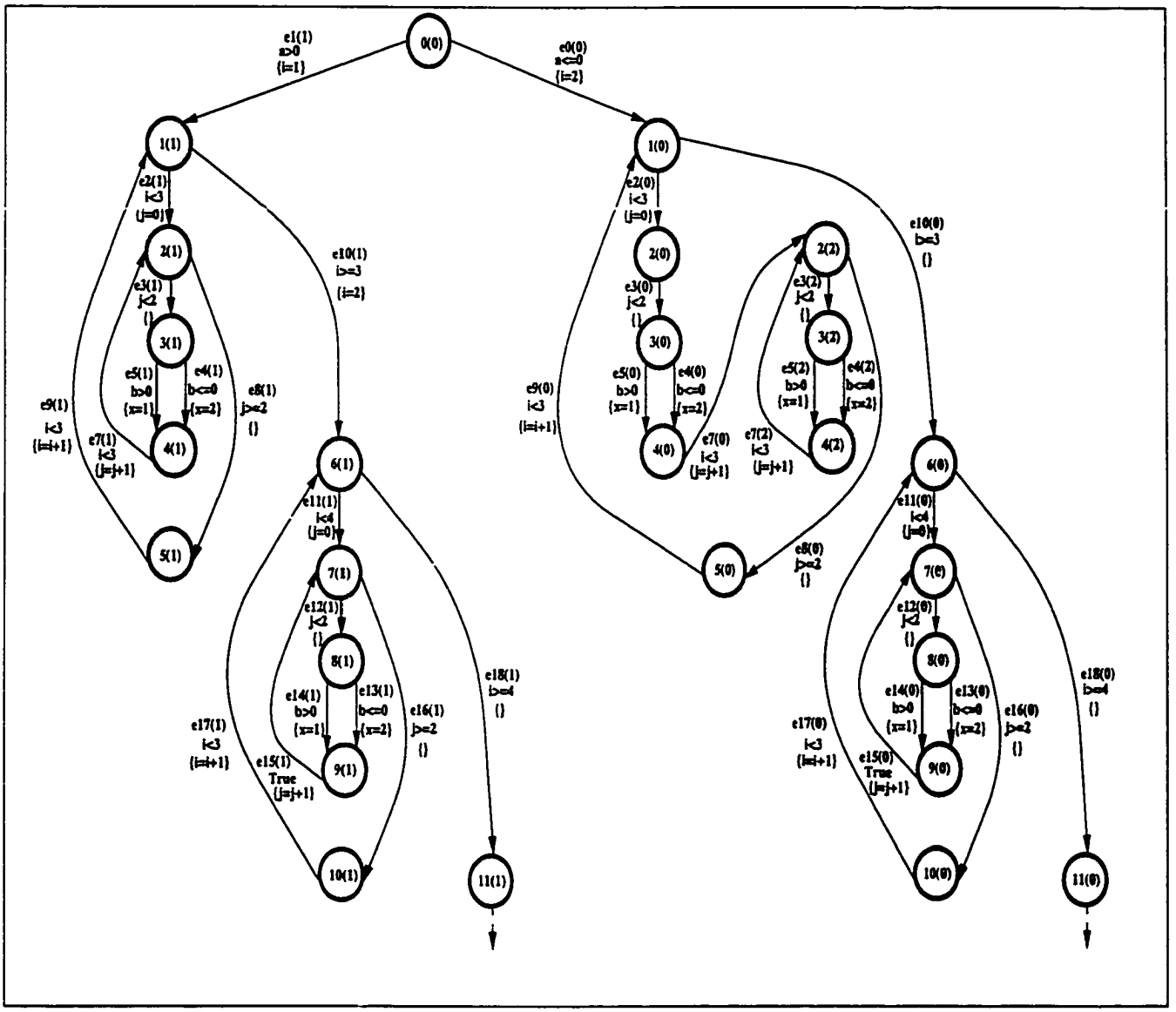

Figure 4.48: The EFSM graph after the loop with the entry/exit node of $v_{2(0)}$ of the graph of Figure 4.47 is advanced once.

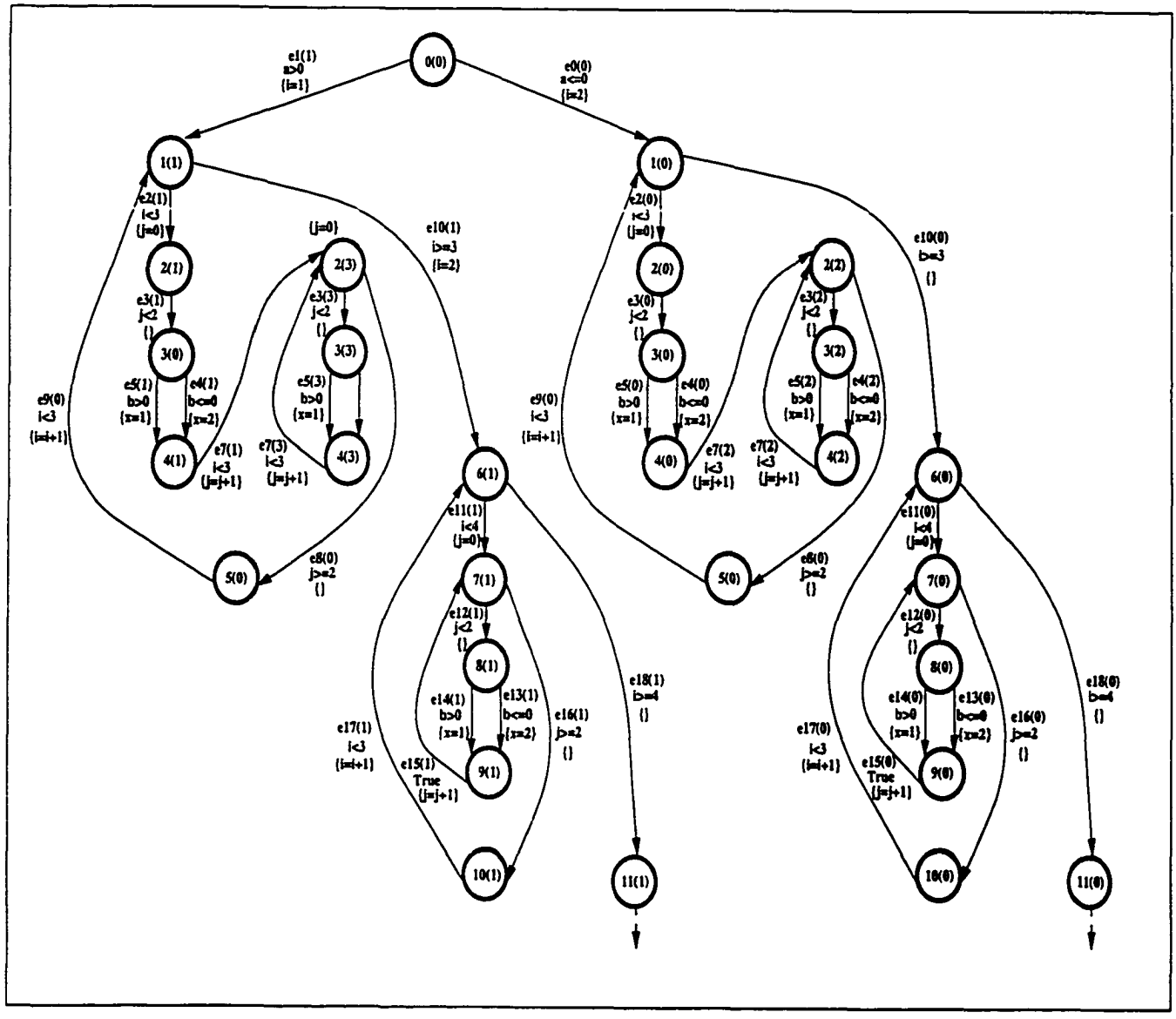

Figure 4.49: The EFSM graph after the loop with the entry/exit node of $v_{2(1)}$ of the graph of Figure 4.48 is advanced once.

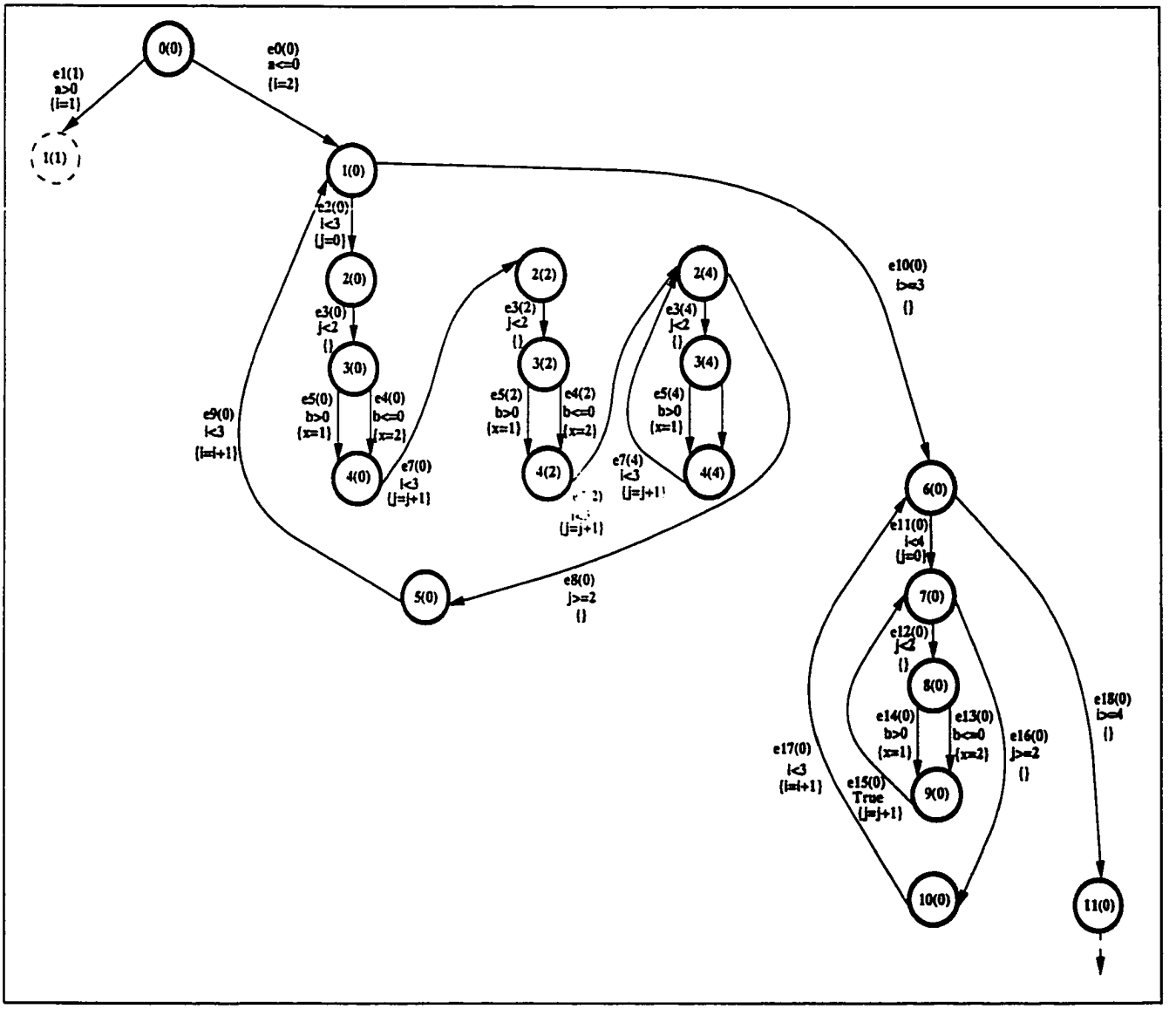

Figure 4.50: The EFSM graph after the loop with the entry /exit node of $v_{2(0)}$ of the graph of Figure 4.49 is advanced twice.

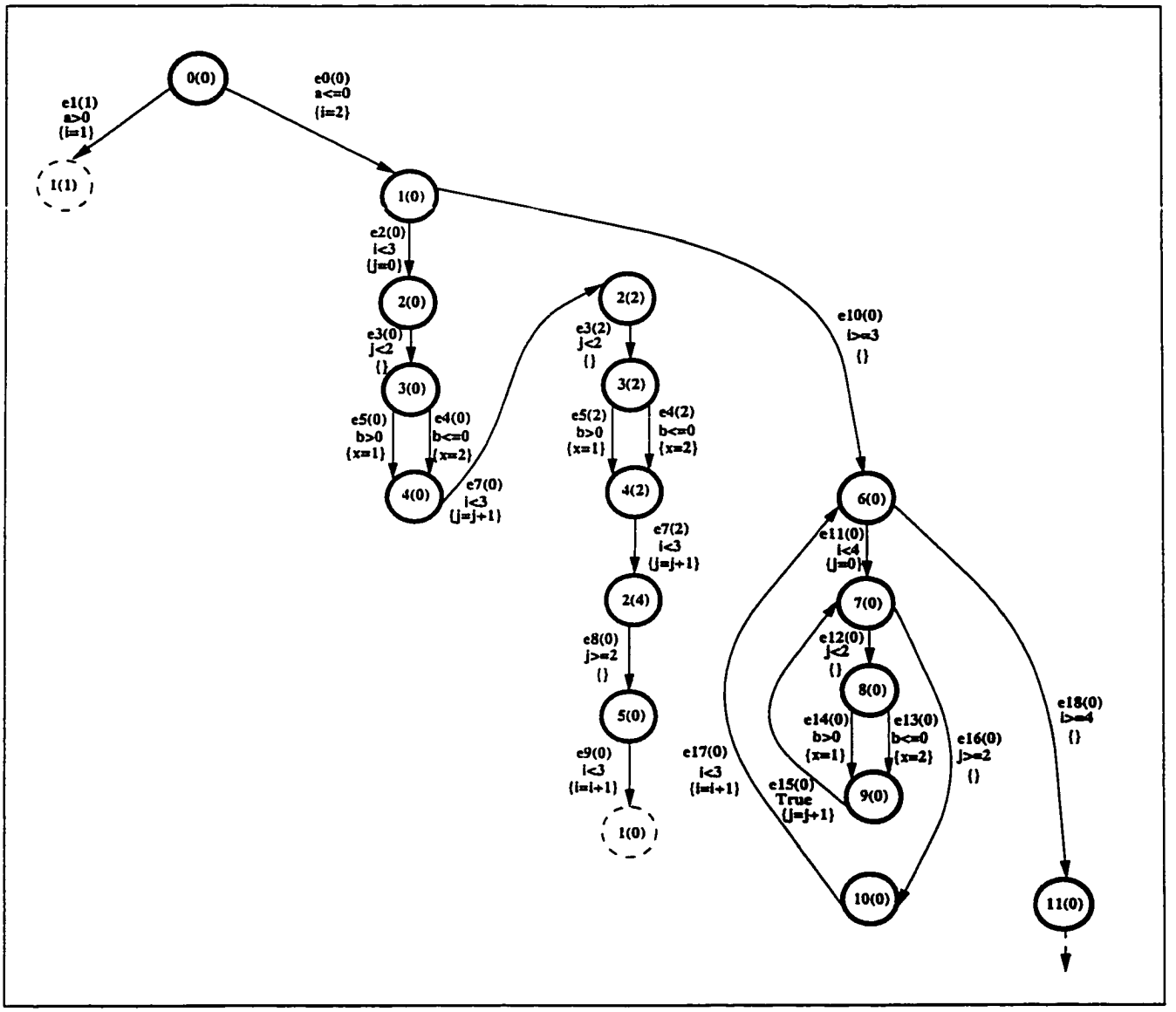

Figure 4.51: The EFSM graph after the loop with the entry/exit node of $v_{2(0)}$ of the graph of Figure 4.50 is completely analyzed.

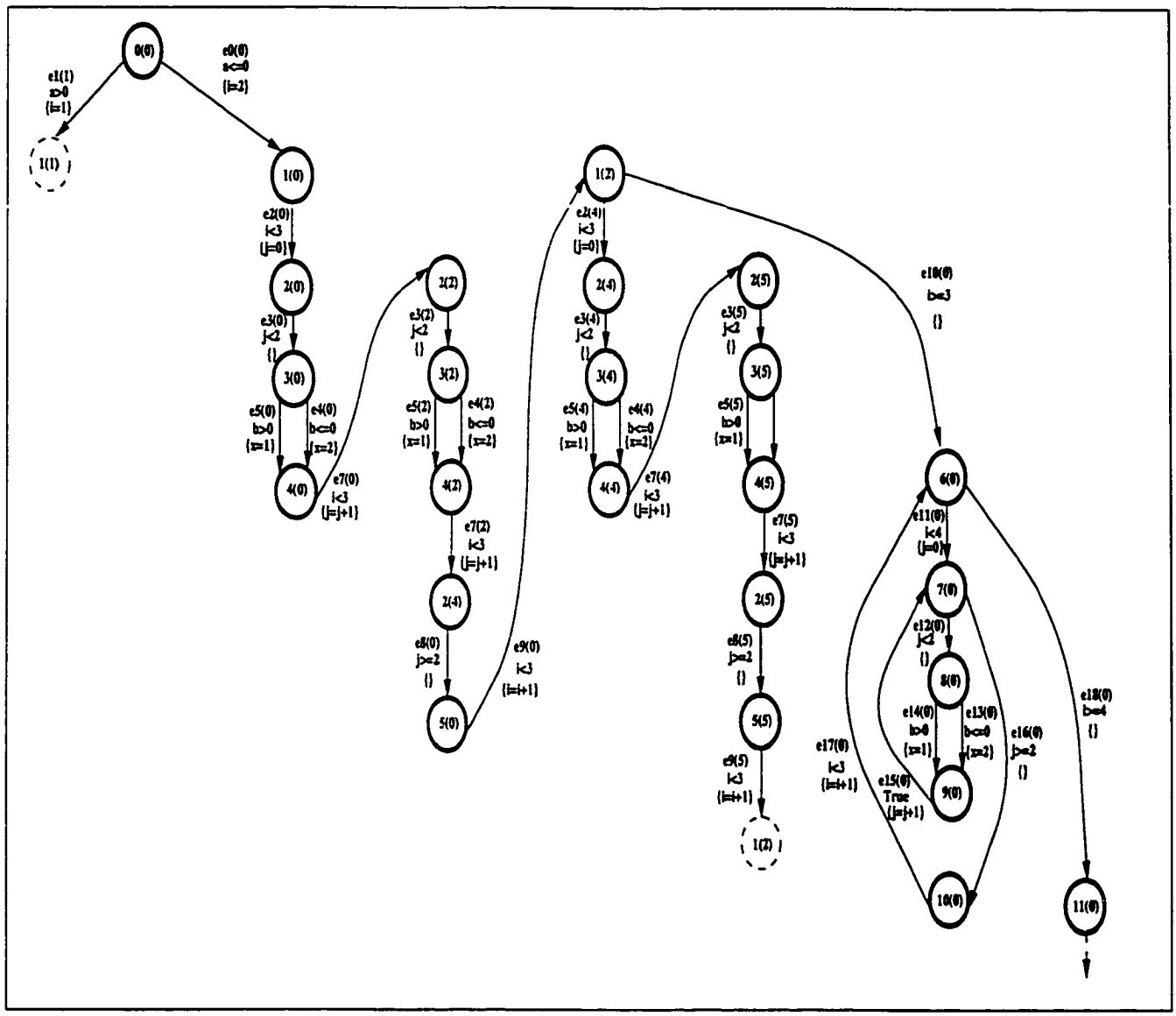

Figure 4.52: The EFSM graph after the loop with the entry/exit node of $v_{1(0)}$ of the graph of Figure 4.51 is advanced once.

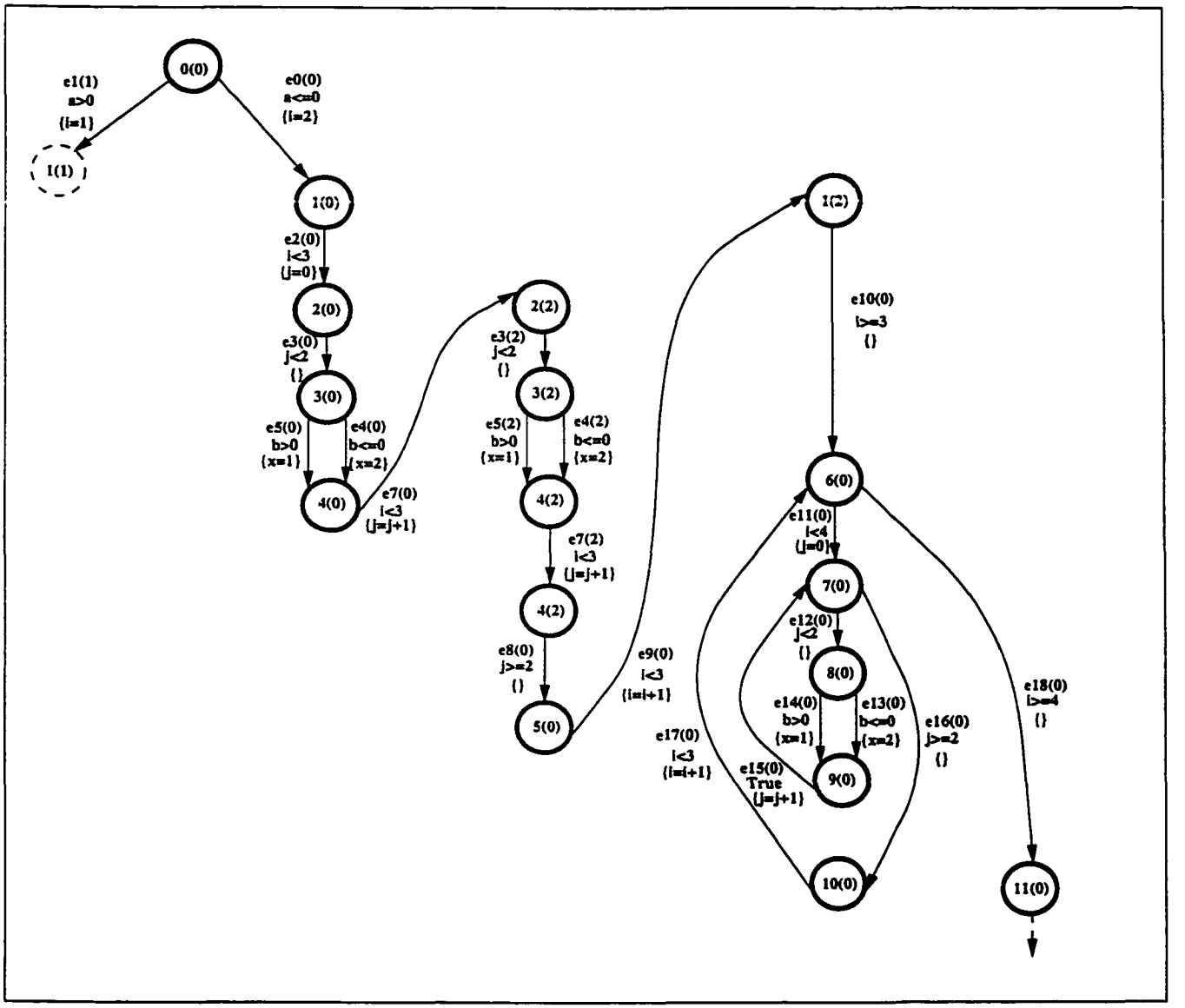

Figure 4.53: The EFSM graph after the loop with the entry/exit node of $v_{2(0)}$ of the graph of Figure 4.52 is completely analyzed.

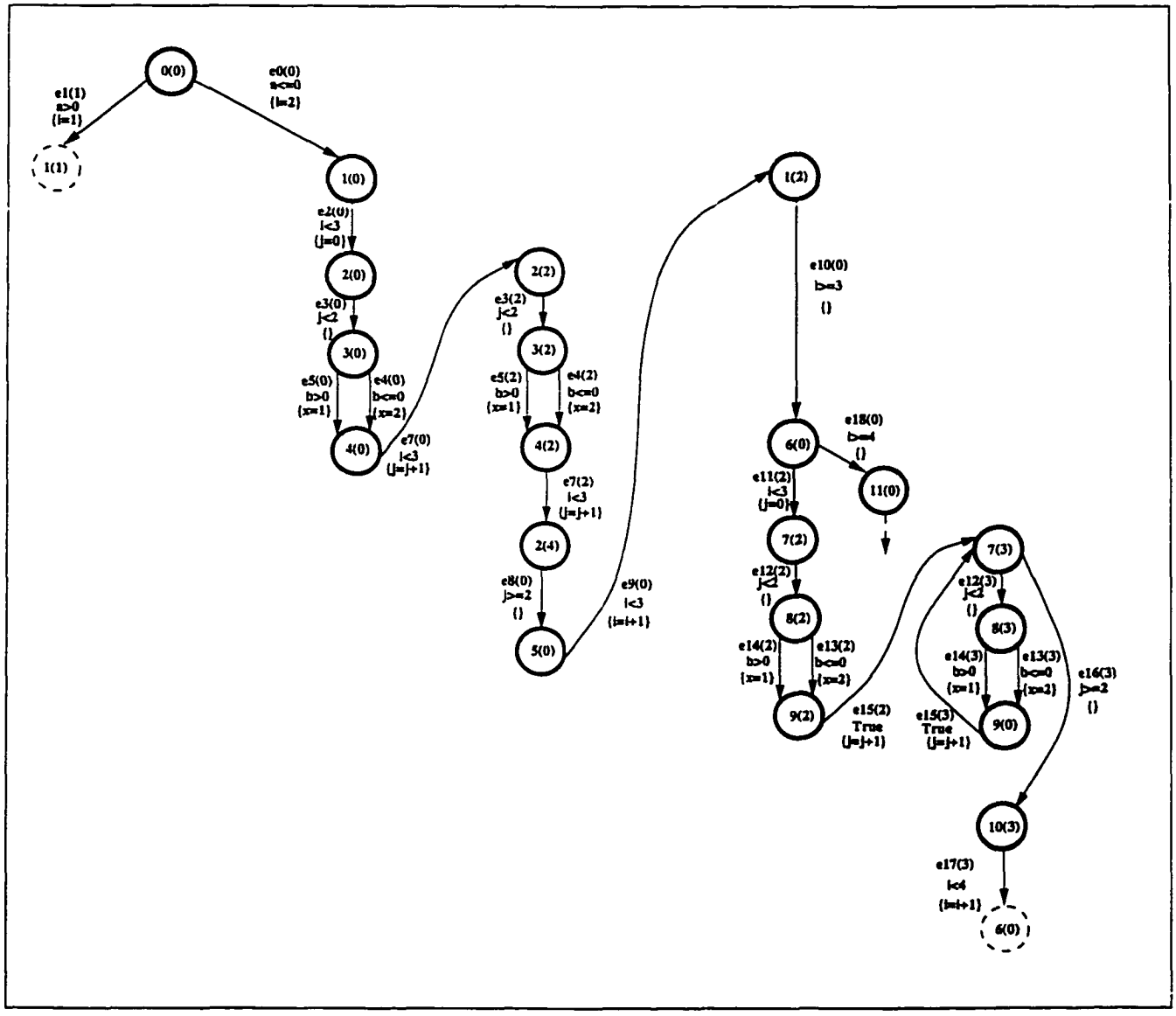

Figure 4.54: The EFSM graph after the loop with the entry/exit node of $v_{7(2)}$ of the graph of Figure 4.53 is advanced once.

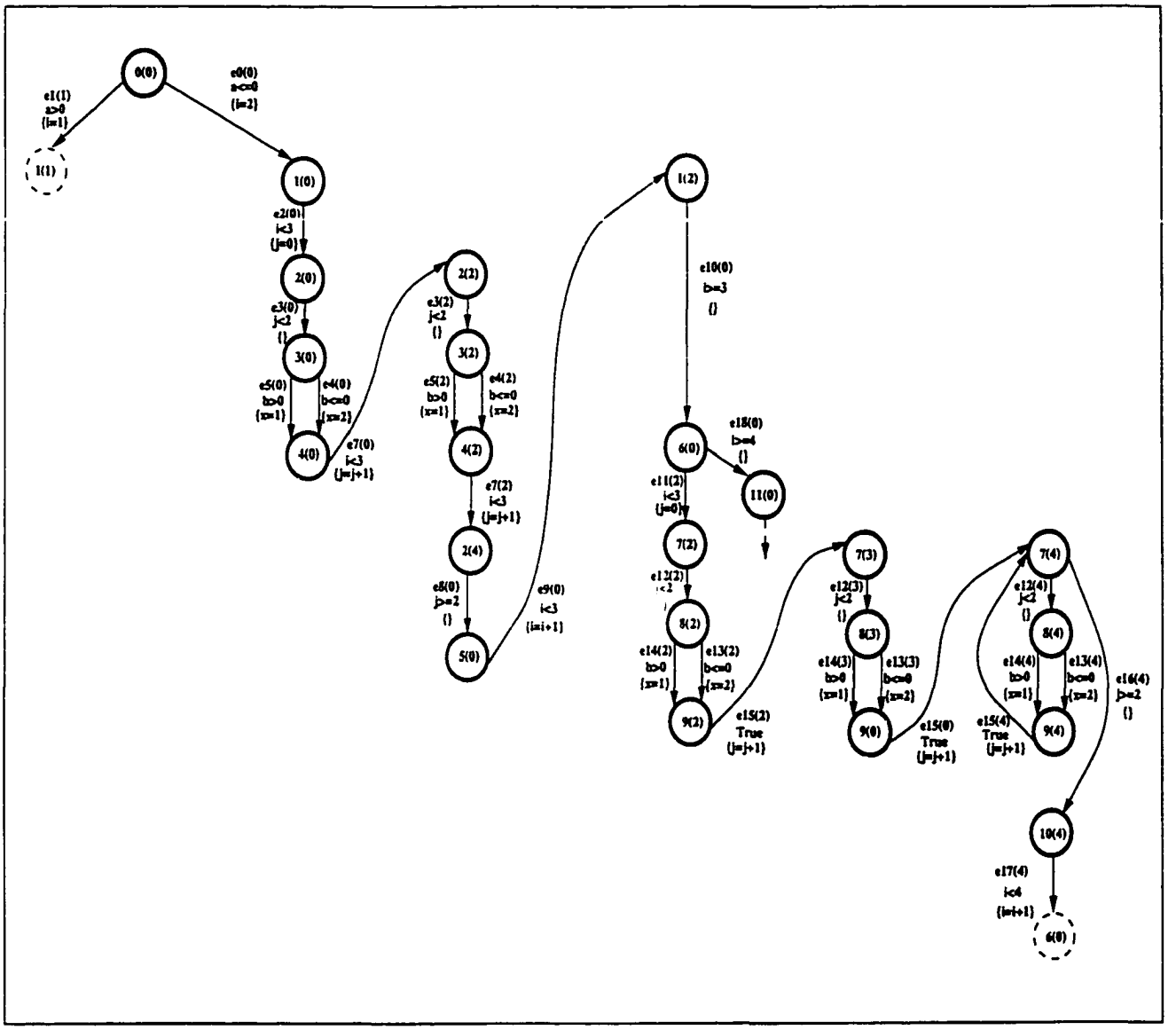

Figure 4.55: The EFSM graph after the loop with the entry/exit node of $v_{7(2)}$ of the graph of Figure 4.54 is advanced twice.

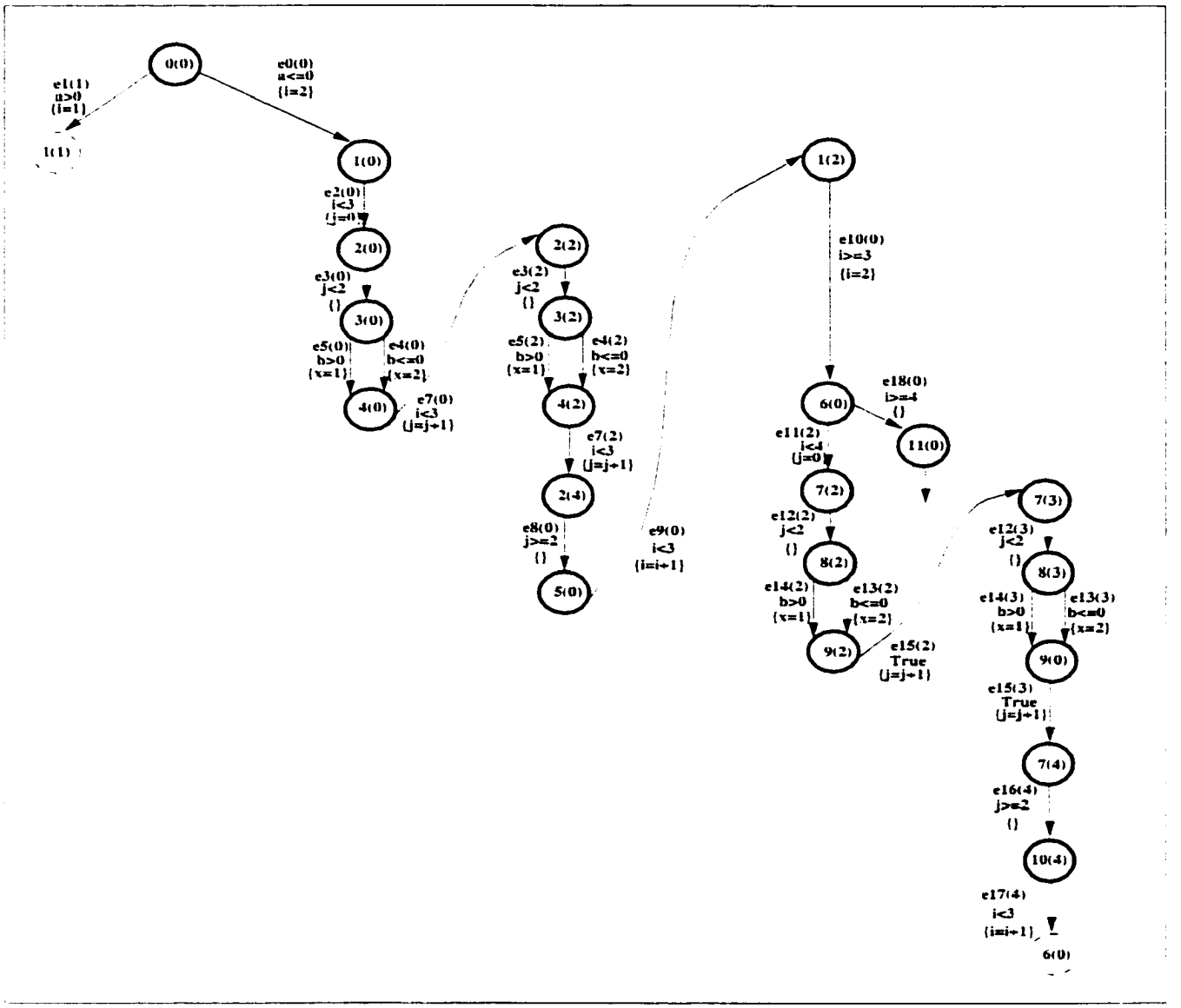

Figure 4.56: The EFSM graph after the loop with the entry/exit node of $v_{7(2)}$ of the graph of Figure 4.55 is completely analyzed.

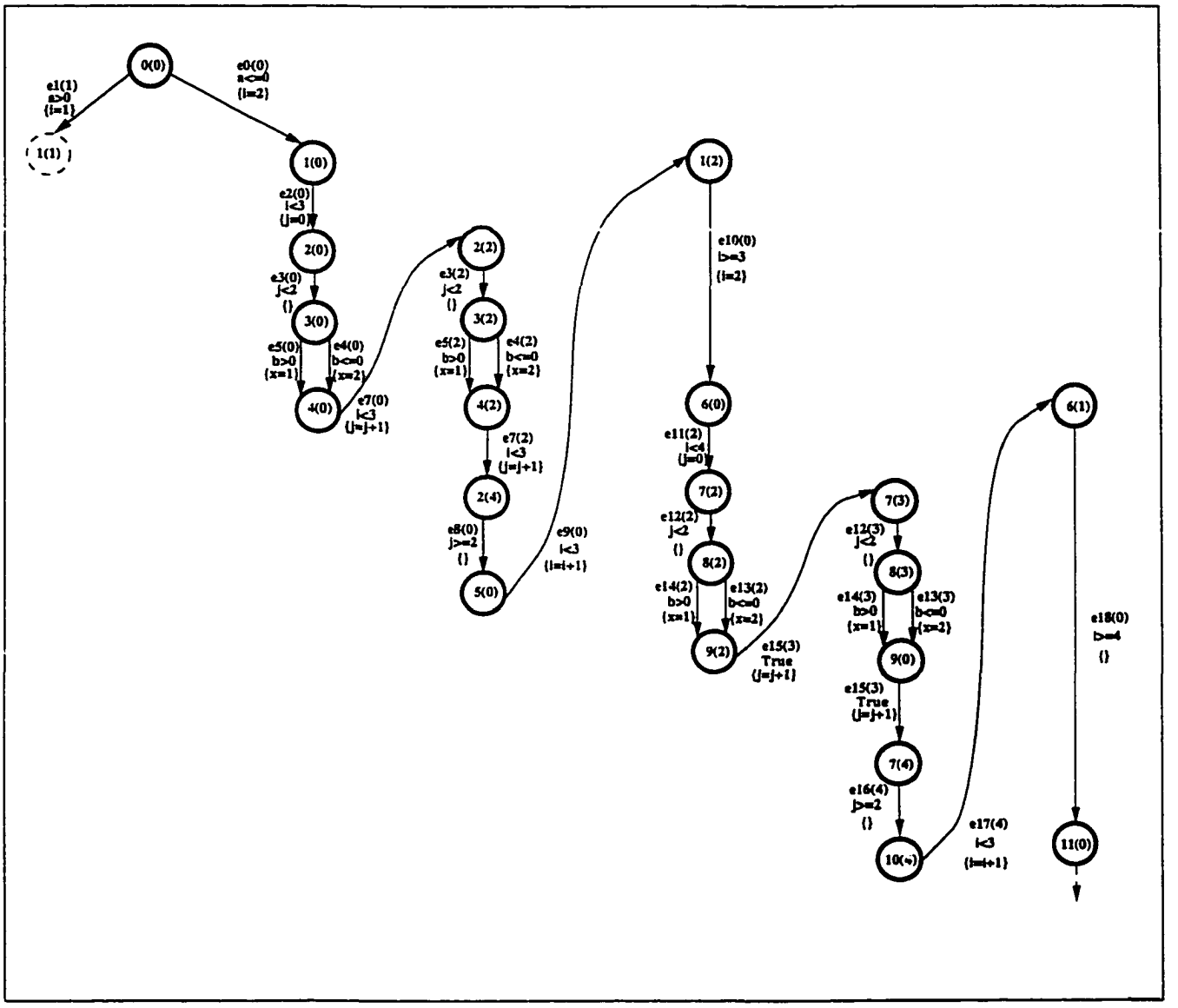

Figure 4.57: The EFSM graph after the loop with the entry/exit node of $v_{6(0)}$ of the graph of Figure 4.56 is completely analyzed.

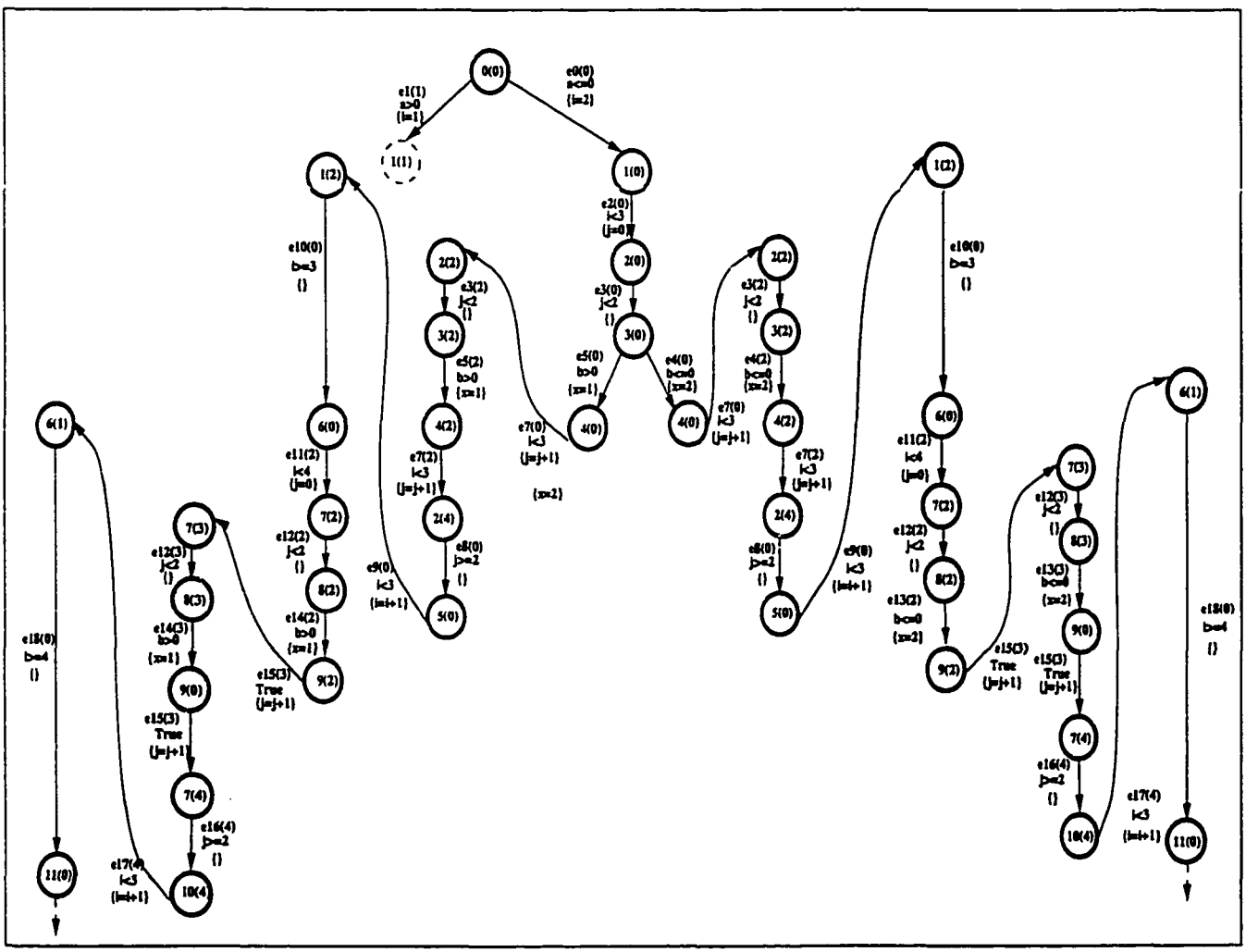

Figure 4.58: The EFSM graph after the condition inconsistencies were eliminated from the EFSM of the graph of Figure 4.57.

Chapter 5

Application of Inconsistency Detection and Elimination Algorithms

The inconsistency detection and elimination algorithms presented in Chapter 3 enable the application of the existing FSM-based conformance test generation methods to EFSM models. These algorithms were applied to the FSM model of a VHDL specified protocol called the Local Proxy of the Adaptive Computing Architecture (ACA). The ACA and the Local Proxy is briefly discussed. The resulting consistent EFSM model after applying the algorithms will be given. Other relevant conformance testing issues such as the the effects of reachability analysis, redundancies in specifications, and re-structural of the specifications on test generation will be pointed out.

The application of the inconsistency detection and elimination algorithms to protocols with conflicting timers is discussed later in this chapter.

5.1 The Adaptive Computing Architecture and the Local Proxy

The requirements of defense networking and computing present significant challenges to current architectures for distributed computing. In particular, the mix of distributed computing, networks which provide variable bandwidth and reliability and mission critical applications which must use these networks demands new application architectures. The Adaptive Computing Architecture (ACA) has been proposed as a framework based on ODP bindings that supports policy-based adaptive management of network resources [22]. The ACA maintains three key kinds of information about the system: adaptive management policies, interface specifications (with QoS requirements) and network topology and QoS measures. Figure 5.1 is a high-level logical depiction of the ACA. The **Adaptive QoS Manager** establishes and manages bindings between application and service objects. The AQM is responsible for managing enterprise resource usage according to the current policies by mediating new requests for service and actively managing resource usage by existing bindings. If network resources are not available (and policy dictates), the AQM can pre-emptively shut down or adjust an existing binding to optimize its resource usage. Similarly, network load can be reduced by inserting filter objects within a binding. The **Policy Service** stores the adaptation policies that specify *how* and *when* the AQM should adapt to changing resource availability. Adaptation policies typically fall into one of three categories: i) usage preferences that are specific to a particular kind of application, ii)

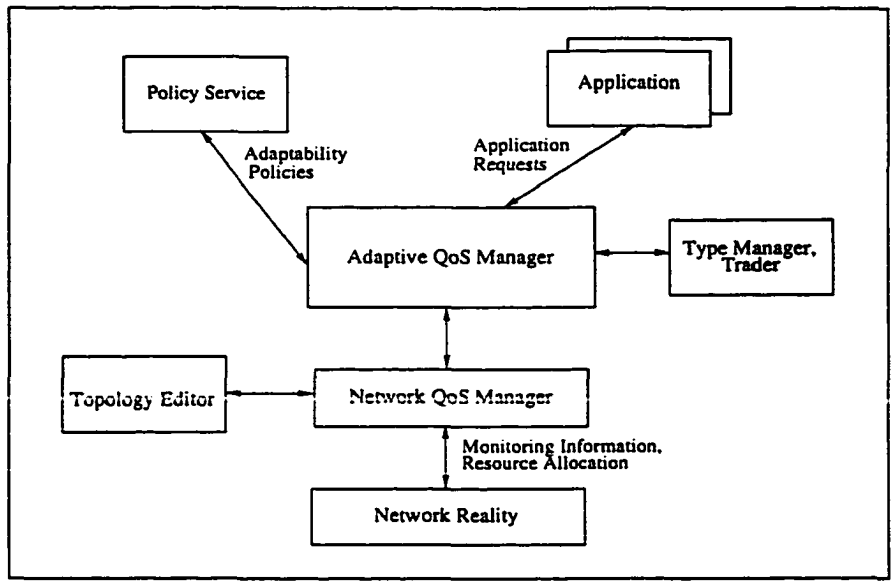

Figure 5.1: The adaptive computing architecture.

policies that address the trade-off of cost versus performance, iii) policies that are modal and/or sensitive to user identity. The **Network QoS Manager** holds knowledge of the state of the network, consisting of two distinct parts. The nominal network topology and its associated QoS measures are expressed using the Network QoS Specification Language (NQSL). The second part deals with the current network state. The NQM must be informed of the current network load by management interfaces on critical links and gateways. The NQM also uses the information that it gathers to perform route selections and preemption on the basis of required QoS parameters. The **Trader** and **Type Manager** are supporting ODP services.

In order to validate the ACA it is desirable to test it on a larger network than that used by this initial prototype. DSTO is building an Experimental C3I Technology Environment (EXC3ITE). Therefore, we have chosen to model the EXC3ITE network as a realistic exemplar heterogeneous defence network to support a simulation of the ACA. The simulation model of

the ACA represents an *implementation* of the architecture. The AQM is implemented as two distributed components. The first part is a local component that is present on all workstations. This part of the AQM implements policies local to the workstation and provides an interface between applications and the architecture. Second, aspects of the AQM which have wider impact form part of a LAN component. The local component is modeled as two related processes: Local Proxy and Local Proxy2. The main function provided by the Local Proxy process is to manage a set of Local Proxy2 processes which in turn re-direct application requests to the LAN Proxy component and performs local monitoring for the application.

The behavioral model of a VHDL specification can be used as a formal description for a communication protocol [45, 61]. The behavioral model of the Local Proxy component has been specified in VHDL, where the architecture is represented in a single process. An EFSM representation of the Local Proxy is constructed from its VHDL behavior description (since internal variables are used in the specification, a simpler FSM model could not be utilized). The main functionality of the local Proxy component is depicted when the component is in its *listen* mode. Only a portion of the EFSM model of the Local Proxy, that portrays the component's *listen* mode, is presented in Figure 5.3. In the specification, when the *else* part of an *if* statement is missing, a "complementary" *else* statement with a *null* output is created. The complementary edges of Figure 5.3 include e_{26}, e_{28}, e_{36}, and e_{42}.

5.1.1 Test Generation for the Local Proxy

The VHDL specification for the Local Proxy of the ACA is an EFSM which can be used to automatically generate conformance tests. An EFSM model requires the detection and removal

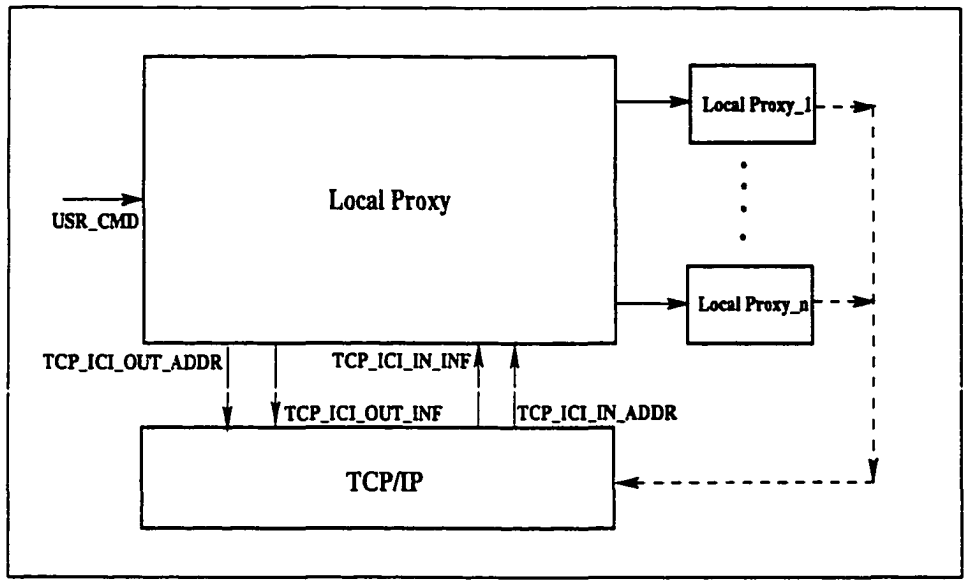

Figure 5.2: The Local Proxy of ACA.

of the inconsistencies among the conditions and actions of its specification (if any).

Several action and condition inconsistencies were found when the inconsistency detection algorithms were applied to the EFSM graph of the Local Proxy. As mentioned earlier, finding inconsistencies in an EFSM model simply indicates that some edges in the graph representation of the EFSM cannot be traversed together with certain other edges. For example, a test sequence with the edges e_{27}, e_{37}, and e_{43} requires *TCP_ICI_IN_INF_TYPE* to be *ESTAB*, *SG_FWD*, and *CLOSE/ABORT*, respectively. Since the input signal *TCP_ICI_IN_INF_TYPE* is not updated in the walk containing these edges, executing such a test sequence is not feasible.

Once the inconsistencies are detected in the Local Proxy, they can be eliminated from the EFSM model by the inconsistency elimination algorithms. Action inconsistencies from the EFSM model of the Local Proxy. eliminated first. If the conditions of outgoing edges of a node v_i

become infeasible due to the variables modified in the paths leading to v_i, v_i and all nodes accessible from it are split into parallel subgraphs. The number of times that a node is split is determined by the number of different values for the variables causing the inconsistencies at v_i. The second step is to eliminated the condition inconsistencies (if any). During the removal, path conditions up to node v_i are accumulated. If the conditions of the outgoing edges $v_j \in V_{v_i}^{reachable}$ conflict with the accumulated path conditions, v_i and each node v_x accessible from v_i such that $(v_x \in V_{v_i}^{reachable}) \wedge v_j \in V_{v_x}^{reachable})$ are split into two subgraphs. The number of times that a node is split depends upon the number of condition sets for the variables of the outgoing edges of v_i.

During the node splits, a node that can be accessed from v_i is split only if there is a path between the two nodes, without any intermediate read condition(s) for the variable(s) that are causing inconsistencies. For example, in the Local Proxy, because of the "wait" statement in e_{49} (i.e., a "read" type of statement), the nodes accessible from v_{28} are not split. After infeasible edges were deleted, Figure 5.4 shows the final graph, which is free of inconsistencies, obtained by the inconsistency detection and elimination algorithms.

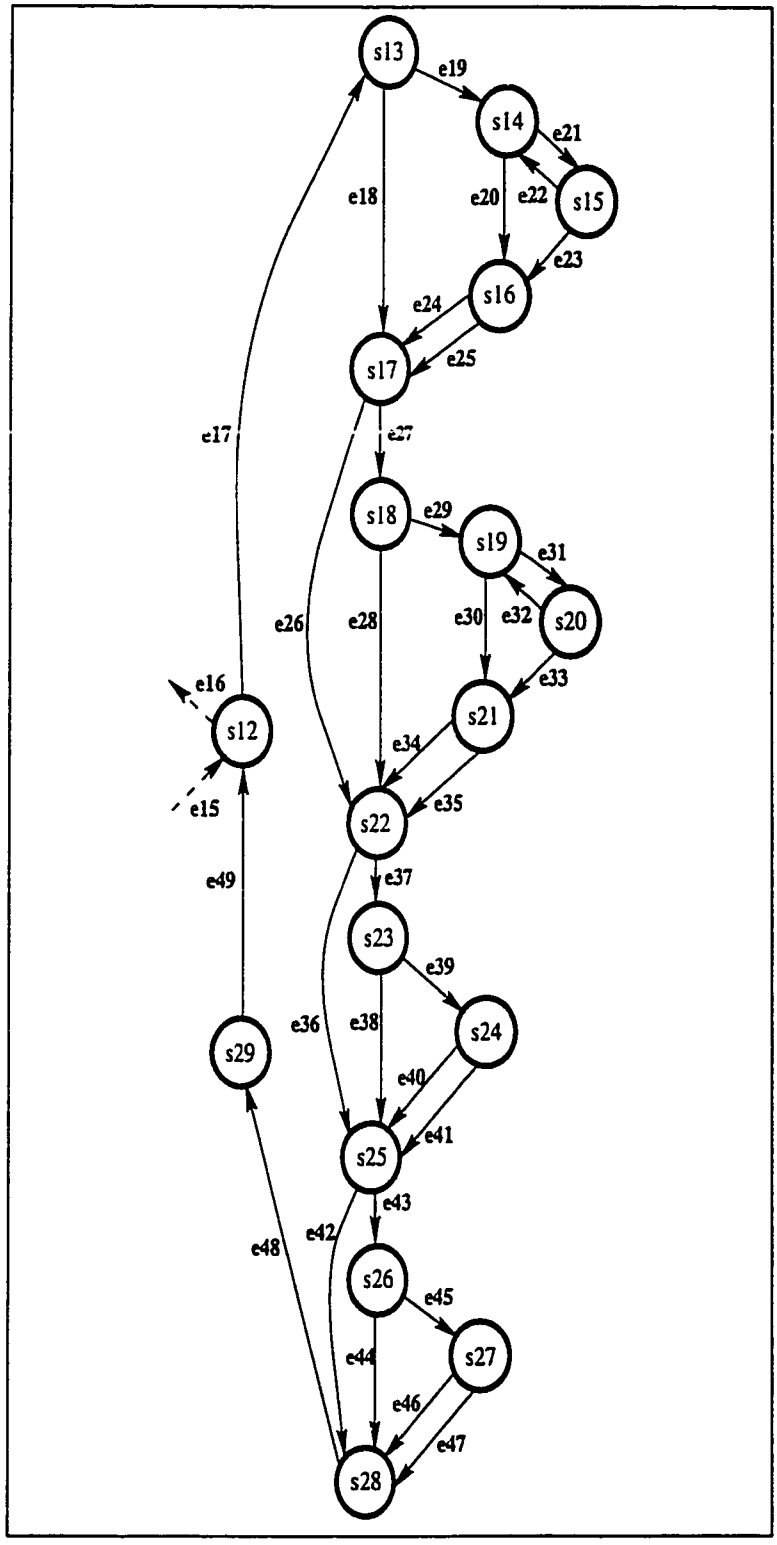

Figure 5.3: The EFSM model of the Local Proxy.

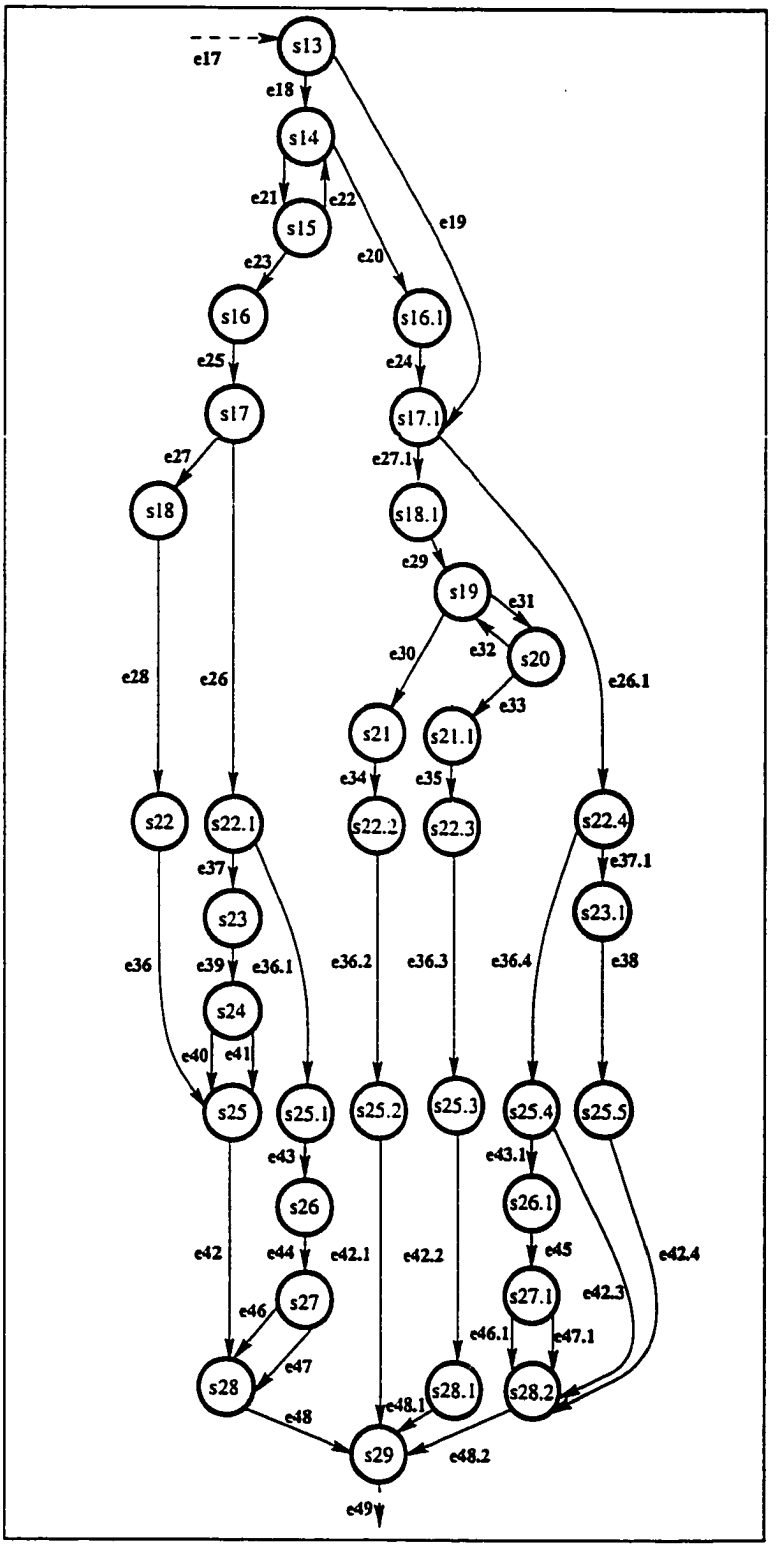

Figure 5.4: EFSM model of the Local Proxy after inconsistencies were eliminated.

The conformance tests for the Local Proxy are generated by using the Rural Chinese Postman (RCP) method, which combines the rural postman tours and the unique input/output (UIO) sequences [3]. The RCP method is developed for FSM models, and, therefore, cannot address the issue of inconsistencies. However, after the inconsistencies are eliminated, the EFSM model of a specification effectively becomes an FSM model which then can be used with the RCP method. A minimum length test sequence generated by the RCP method for the Local Proxy consists of 185 steps. (During this early step of the case study, the test sequence is generated without using the UIO sequences.)

5.1.2 Refining the Local Proxy

During the initial phase of a protocol design, it is important that the external functionality of the protocol is well-defined while the internal functionality is considered as a black-box. This hierarchical approach enables a successful process cycle of design, development and conformance testing. Within this framework, conformance test generation for the Local Proxy produced several specification changes. Various recommendations were presented to the protocol designers that enhanced the completeness of the Local Proxy specification and improved its testability. The following sections highlight the observations regarding the testability of the Local Proxy.

5.1.2.1 Redundancies in Specifications

UIO sequences are planned to be used for the state verification during conformance testing. It has been shown that, if a state of an FSM/EFSM does not have an equivalent state, it possesses

a UIO sequence [65]. Two states are said to be equivalent if the permissible input set for one is a subset for the permissible input set of the other and the corresponding outputs and next states are the same [53]. Two states are also *k-equivalent* if the pair cannot be distinguished with a sequence of length k. The main reason that UIO sequences cannot be used for FSMs/EFSMs containing *equivalent* states is due to the inability to detect transfer errors. A *transfer error* on an edge from state v_i to v_j can cause the implementation under test (IUT) to move to a different, but *equivalent*, state from v_j. Failure to identify a state may allow for non-conforming IUTs to successfully pass the conformance tests.

To describe the protocol behavior precisely, a protocol designer may repeat certain portions of the specification several times. Such redundancies will eventually lead to the formation of equivalent states, impairing the testability of the IUT. Therefore, balancing the clarity of the specification with its testability is an important issue to be addressed at the design stage.

In the process of forming UIO sequences for the Local Proxy, it is observed that some of the states produce identical outputs for the same inputs, and their next states are *2 or 3-equivalent.* For example, the input/output set for v_{20} is identical to that of v_{15} and the sets of their adjacent nodes ($\{v_{19}, v_{21}\}$ for v_{20} and $\{v_{14}, v_{16}\}$ for v_{15}) contain *2* or *3-equivalent* nodes. Therefore, nodes v_{19}, v_{20}, and v_{21} can be merged with v_{14}, v_{15}, and v_{16}, respectively. Since the number of nodes and edges will be reduced after the merge, the length of test sequence will be also shorter.

For testing purposes, the behavior of an EFSM is transformed to an FSM by duplicating some of the nodes and edges of the EFSM graph. The number of states could dramatically increase if proper methods of expansion are not employed. The removal of the redundant states is also

helpful in reducing the number of states in the final EFSM graph.

5.1.2.2 Reachability

Conformance testing methods require that each node of the directed graph of the FSM/EFSM model of the specification can be reached from any other node. During the inconsistency removal process, the EFSM graph is split into subgraphs. Infeasible transitions are dropped from the new subgraphs. A node with no incoming edges becomes unreachable and is removed from the graph. The final graph with no inconsistencies, shown in Figure 5.4 for the Local Proxy, has no unreachable nodes.

As indicated in Section 5.1, the EFSM model of the Local Proxy contains complementary edges (i.e., the edges, with *null* outputs, created for *if* statements whose corresponding *else* statements were not present in the VHDL specification) such as e_{26}, e_{28}, and e_{36}. The actions of these complementary edges of Local Proxy are further discussed with the designers to clarify if they were inserted correctly. During these discussions, it is observed that some of the complementary edges were the only edges connecting some subgraphs to the rest of the EFSM graph. For example, if e_{26} and e_{28} (complementary edges) were removed from Figure 5.4, $v_{22.1}$ and v_{18} would become nodes without incoming and outgoing edges, respectively.

This observation revealed that certain actions were left unspecified in the Local Proxy. As an example, let us consider the actions of the complementary edge e_{28}. The messages received from TCP/IP are of three types: *connection established, segment received*, and *connection closed/aborted*. The Local Proxy takes appropriate actions dictated by the type of the mes-

sage received from TCP/IP. The actions of e_{28}, corresponding to the TCP/IP message signaling for *connection established* were needed to be specified when $TCP\_ICI\_IN\_INF\_TYPE = ESTAB$ and $conn\_idx \neq NEGAT\_1$.

The modification made to the Local Proxy specification based on the reachability analysis demonstrates the need for integrating the protocol development with the conformance test generation process. This integration will allow for the removal of costly mistakes from the specification at an early stage of the development, before they propagate into many different implementations possibly combined with other errors.

5.1.2.3 Structure of EFSMs

Due to the interdependencies among the action and condition variables, automated test generation for EFSM modeled protocols becomes a difficult process. The graph of EFSM model might be split multiple times to remove the inconsistencies, where each split node may increment the length of the test sequence. The complexity of splitting nodes and edges of the EFSM graph (and hence the length of test sequence) can be reduced at the design stage by excluding some of the inconsistencies from the specification, without tradeoffs.

In general, a protocol can be specified in several different ways leading to different FSMs/EFSMs all of which accomplish the same task(s). The EFSM model of a VHDL specification consists of interconnected data flow subgraphs. The topological interconnection among these subgraphs has a direct impact on the length of tests generated for the EFSM. A cascade of data flow subgraphs that use the same conditional variables can be interconnected such that the number of infeasible

paths are minimum (provided that the controlling variables are not updated in these data flow subgraphs). By minimizing the infeasible paths, the test sequence length can be shortened.

Figure 5.5 shows two fragments of different VHDL specifications, which are equivalent in terms of their functionalities. Their corresponding EFSM models, however, differ significantly.

```
if (x = 1) then A := b;
else if (x = 2) then A := c;
else null;
end if;
end if;
```

(a)

```
if (x = 1) then A := b;
else null;
end if;
if (x = 2) then A := c;
else null;
end if;
```

(b)

Figure 5.5: Examples of two VHDL specifications.

All paths for the EFSM model of Figure 5.5.a are valid and thus the EFSM is consistent. On the other hand, the EFSM model of Figure 5.5.b contains condition inconsistency and an infeasible path. The EFSM model of Figure 5.5.b, therefore, requires mechanisms to remove the inconsistency, which increases the complexity of the test generation process.

A simple but important example that supports this case can be found in the Local Proxy. As stated in Section 3, it is not possible to include e_{27}, e_{37}, and e_{43} in the same test sequence. This problem will be resolved when the inconsistencies are eliminated from the EFSM model. The work of splitting some nodes, however, could have been prevented if the tail nodes for e_{28}, e_{34}, e_{35}, e_{38}, e_{40}, and e_{41} were combined as v_{28}. Such combination of tail states can be achieved by using the format depicted in Figure 5.a for conditions of outgoing edges of v_{17}, v_{22}, and v_{25}

(which currently use the format of Figure 5.b).

Therefore, the conformance testers, where possible, provide recommendations for the sections of the EFSM graph that requires heavy splitting due to inconsistencies. Restructuring the specification may reduce the complexity of the test generation process, while reducing the test sequence length.

5.2 Conflicting Timers Problem

The messages exchanged among the peer entities of a communication system may require delivery of services within certain time, measured with timers. The FSM models of these systems include transitions with timing conditions.

The timing parameters of an FSM model have significant influence on its behavior. Functional errors in protocols with real-time requirements can be caused by the conflicting actions and conditions involving timers. A test sequence generated for an FSM model in which time variables affect the behavior of the protocol may become infeasible if the timing constraints are not efficiently analyzed during the test generation. Issues related to the so-called *conflicting timers problem* have been studied in more detail by Uyar *et al.* [30, 31, 32, 33, 83].

As an example, among the several timers defined for the the Datalink Layer of the MIL-STD 188-220 are the $BUSY$ and ACK timers. It is observed that for the *FRAME_BUFFERED* state, a buffered frame cannot be transmitted if either timer is running. On the other hand, if ACK timer expires while the $BUSY$ timer is inactive, a buffered frame is retransmitted. However, if

the ACK timer expires while the $BUSY$ timer is running, no output is generated [32].

Although the timers problem is generally an open research problem, most of the time variables are linear and their values implicitly increase with the time. These two features simplify the conflicting timers problem. In this special case, it is expected that an efficient solution to the conflicting timers problem can be found [30].

The inconsistency detection and elimination algorithms presented in this thesis are used to solve the conflicting timers problem. The application of these algorithms is expected to eliminate the redundancies of manual state expansion and significantly shorten the test sequences while providing the same fault coverage [32]. An FSM graph whose actions and conditions use timing variables is depicted in Figure 5.6. The edge conditions and actions of this graph are defined as follows:

$$e_{on} : \langle 1 \rangle \{T_1 = 0; T_2 = 0; f_1 = -\infty; f_2 = -\infty; t_{0,1} = 1; t_{1,1} = 1; t_{2,1} = 2; L_0 = 1\}$$

$$e_{off} : \langle T_1 == 0 \ \wedge \ T_2 == 0 \rangle \{\}$$

$$e^2_{7,1} : \quad \langle L_2 == 2 \ \wedge \ f_2 \geq 3.7 \ \wedge \ (3.7 - f_2 < 5.5 - f_1) \ \wedge \ T_2 == 1 \rangle$$
$$\{L_2 = 1; T_2 = 0; f_1 = f_1 + 1; f_2 = -\infty\}$$

$$e^2_{7,2} : \quad \langle L_2 == 2 \ \wedge \ f_2 < 3.7 \ \wedge \ (3.7 - f_2 < 5.5 - f_1) \ \wedge \ T_2 == 1 \rangle$$
$$\{L_2 = 1; T_2 = 0; f_1 = f_1 - f_2 + 4.7; f_2 = -\infty\}$$

$$e^1_{8,1} : \quad \langle L_2 == 2 \ \wedge \ f_1 \geq 5.5 \ \wedge \ (5.5 - f_1 < 3.7 - f_2) \ \wedge \ T_1 == 1 \rangle$$
$$\{L_0 = 1; T_1 = 0; f_2 = f_2 + 1; f_1 = -\infty\}$$

$$e^1_{8,2} : \quad \langle L_2 == 2 \ \wedge \ f_1 < 5.5 \ \wedge \ (5.5 - f_1 < 3.7 - f_2) \ \wedge \ T_1 == 1 \rangle$$

$$\{L_0 = 1; T_1 = 0; f_2 = f_2 - f_1 + 6.5; f_1 = -\infty\}$$

e_2 : $\langle L_0 > 0 \ \wedge \ f_1 < 5.5 \ \wedge \ f_2 < 3.7 \rangle$

$$\{f_1 = f_1 + 1; f_2 = f_2 + 1; L_1 = 1; T_1 = 1; f_1 = 0; T_2 = 0; f_2 = -\infty\}$$

e_4 : $\langle L_1 > 0 \ \wedge \ f_1 < 5.5 \ \wedge \ f_2 < 3.7 \rangle$

$$\{L_2 = 1; f_1 = f_1 + 1; f_2 = f_2 + 1; T_2 = 1; f_2 = 0\}$$

$e_{0,1}$: $\langle t^s_{0,1} > 0 \ \wedge \ (T_1 == 0 \ \wedge \ T_2 == 1) \ \wedge$

$(t^s_{0,1} < 5.5 - f_1) \ \wedge \ (t^s_{0,1} < 3.7 - f_2) \rangle$

$$\{L_0 = 0; f_1 = f_1 + t^s_{0,1}; f_2 = f_2 + t^s_{0,1}; t^s_{0,1} = 0\}$$

$e_{1,1}$: $\langle t^s_{1,1} > 0 \ \wedge \ 3t^s_{1,1} < 5.5 - f_1 \ \wedge \ 3t^s_{1,1} < 3.7 - f_2 \rangle$

$$\{L_1 = 0; f_1 = f_1 + 3t^s_{1,1}; f_2 = f_2 + 3t^s_{1,1}; t^s_{1,1} = 0\}$$

$e_{2,1}$: $\langle t^s_{2,1} > 0 \ \wedge \ (T_1 == 1 \ \wedge \ T_2 == 1) \ \wedge$

$(t^s_{2,1} < 5.5 - f_1) \ \wedge \ (t^s_{2,1} < 3.7 - f_2) \rangle$

$$\{L_2 = 0; f_1 = f_1 + t^s_{2,1}; f_2 = f_2 + t^s_{2,1}; t^s_{2,1} = 0\}$$

$e^1_{2,1}$: $\langle (T_1 == 1 \ \wedge \ (1 < 5.5 - f_1 < t^s_{2,1})) \ \wedge \ t^s_{2,1} > 0 \ \wedge$

$(5.5 - f_1 < 3.7 - f_2) \ \wedge \ (T_1 == 1 \ \wedge \ T_2 == 1) \rangle$

$$\{L_2 = 2; f_1 = f_1 + 1; f_2 = f_2 + 1; t^s_{2,1} = t^s_{2,1} - 1\}$$

$e^2_{2,1}$: $\langle (T_2 == 1 \ \wedge \ (1 < 3.7 - f_2 < t^s_{2,1})) \ \wedge \ t^s_{2,1} > 0 \ \wedge$

$(3.7 - f_2 < 5.5 - f_1) \ \wedge \ (T_1 == 1 \ \wedge \ T_2 == 1) \rangle$

$$\{L_2 = 2; f_1 = f_1 + 1; f_2 = f_2 + 1; t^s_{2,1} = t^s_{2,1} - 1\}$$

The following edges, called observer edges, are designed to

assure the proper traversal of the self-loops.

$s'_{0,1} : \langle L_0 < 2 \wedge t^s_{0,1} == 0 \rangle \{\}$

$s''_{0,1} : \langle L_0 < 2 \wedge (t^s_{0,1} > 0 \wedge (T_1 == 1 \vee T_2 == 0)) \rangle \{\}$

$s_0 : \langle 1 \rangle \{L_0 = 2\}$

$s'_{1,1} : \langle L_1 < 2 \wedge t^s_{1,1} == 0 \rangle \{\}$

$s_1 : \langle 1 \rangle \{L_1 = 2\}$

$s'_{2,1} : \langle L_2 < 2 \wedge t^s_{2,1} == 0 \rangle \{\}$

$s''_{2,1} : \langle L_2 < 2 \wedge (t^s_{2,1} > 0 \wedge (T_1 == 1 \vee T_2 == 0)) \rangle \{\}$

$s_1 : \langle 1 \rangle \{L_2 = 2\}$

For example, in Figure 5.6, the actions of $e^1_{7,1}$ set T_2 to 0. Since the time condition of e_5 requires that $\langle T_2 == 1 \rangle$, the action of $e^1_{7,1}$ causes inconsistency with the condition of $e^2_{2,1}$. Similarly, a test sequence that includes both $e_{0,1}$ and $e^1_{2,1}$ contains condition inconsistency—$e_{0,1}$ requires that $\langle T_1 == 0 \rangle$ and $e^1_{2,1}$ that $\langle T_1 == 1 \rangle$. Hence, a test sequence generated without considering such inconsistencies may be infeasible.

The application of inconsistency elimination algorithms to this FSM produces an FSM graph in which the generation of only feasible test sequences is guaranteed. Notice that the inconsistency detection and elimination algorithms of Chapter 3 deal with the elimination of inconsistencies in EFSM graphs of the specifications given in high-level languages such as VHDL. For the general FSM/EFSM graphs, these algorithms are slightly modified. Such modifications can be

summarized as follows. When a node v_i is visited, based on the available AUM pairs of v_i, the outgoing edges of v_i are classified as traversable and non-traversable. During the inconsistency detection, an edge e_i is traversed only if its condition can be satisfied by the current AUM pairs of $head(e_i)$. Furthermore, if a subgraph starting from node the v_i is split due to an action inconsistency, copies of all outgoing edges v_i are included in the subgraphs containing $v'_{i(s)}$ and $v''_{i(s)}$ with the non-traversable edges marked. Upon the completion of the graph traversal, all non-traversable edges are deleted from the graph.

Once node v_0 is visited by traversing e_{on}, it is found that none of the outgoing edges of v_0, except e_2, is traversable. Henceforth, a non-traversable edge will be drawn with a dashed-arrow. As stated earlier a node v_i drawn with a dashed-circle represents a subgraph that starts from v_i.

Figures 5.8 through 5.11 show successive graph splits due to the inconsistencies among the time-actions and conditions of the graph which appears in Figure 5.6. The resulting graph when inconsistencies due to the time variables are eliminated is depicted in Figure 5.12. Notice that, for simplicity, the observer edges and nodes [32] are not shown in the final graph.

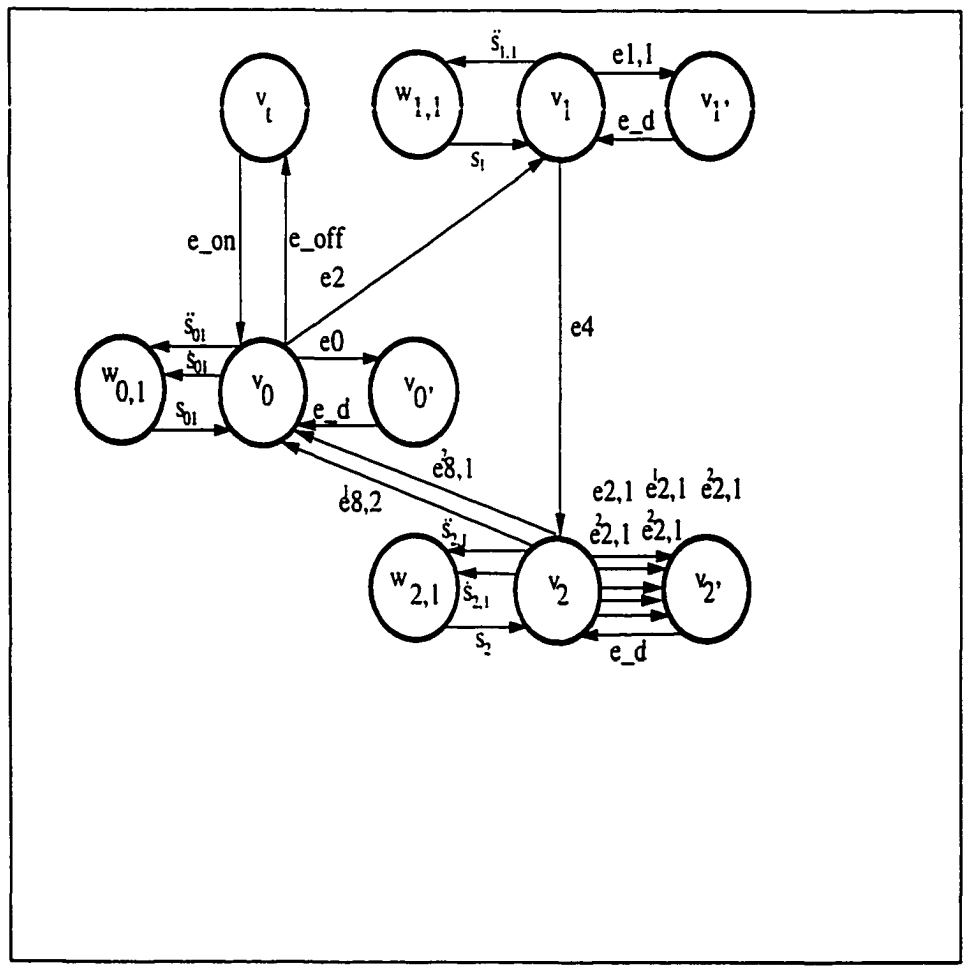

Figure 5.6: An FSM graph with timing parameters.

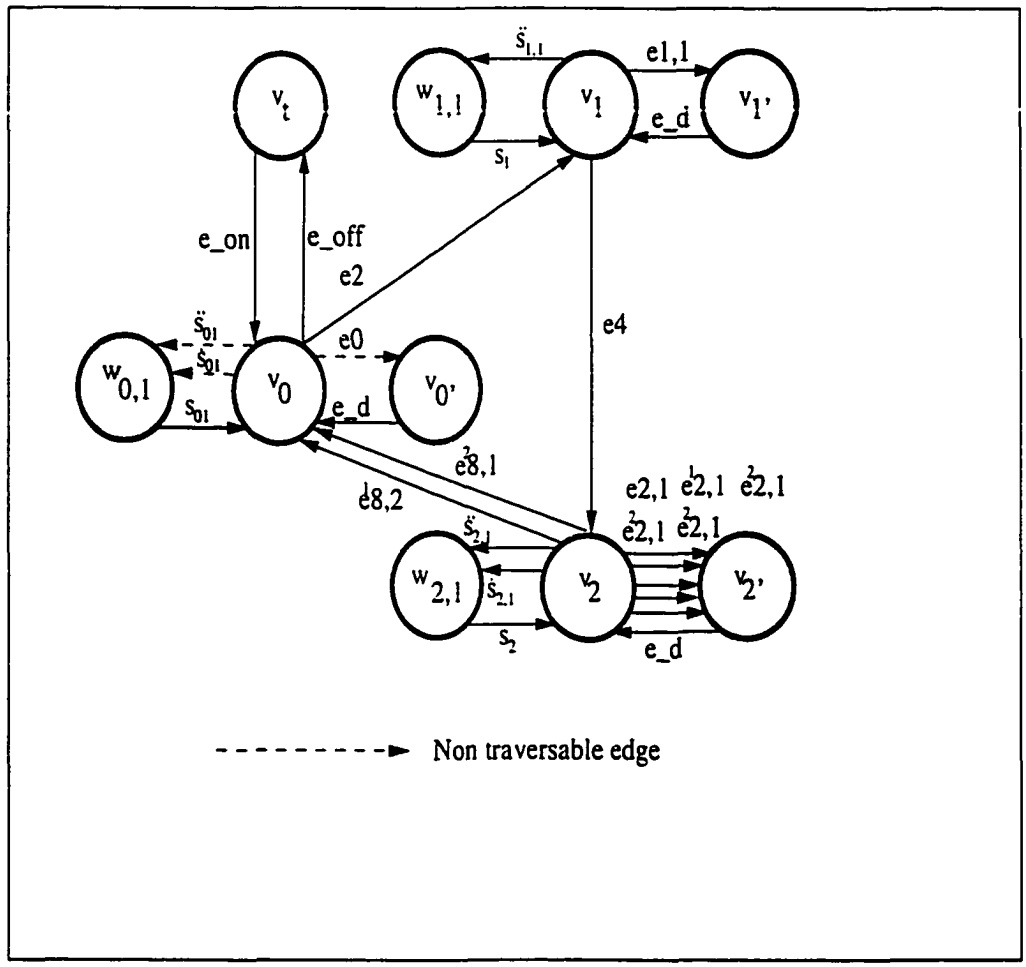

Figure 5.7: The FSM graph of Figure 5.6 with the non-traversable outgoing edges of v_0 marked.

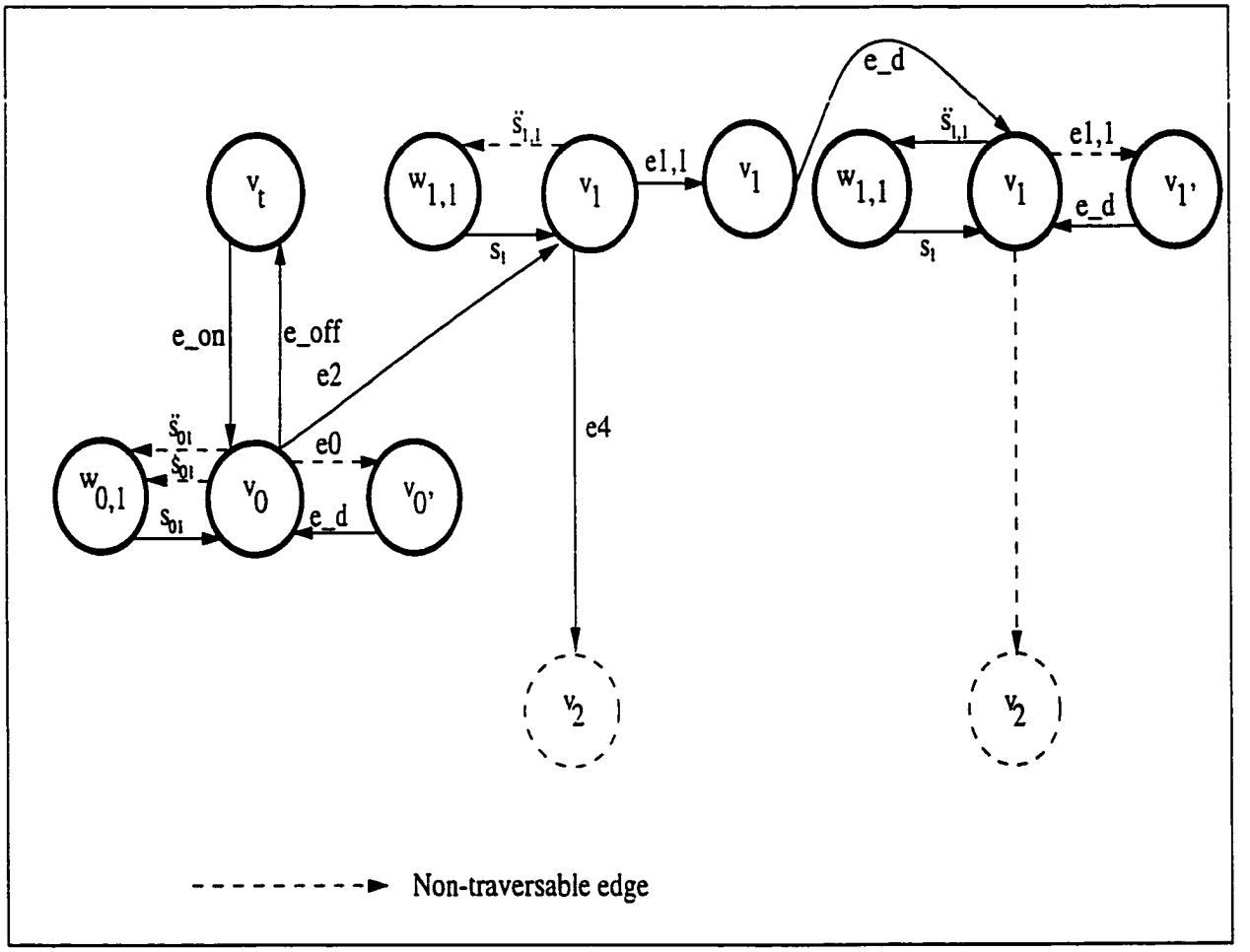

Figure 5.8: The FSM graph after the graph of Figure 5.6 is split due to the effects of $e_{1,1}$.

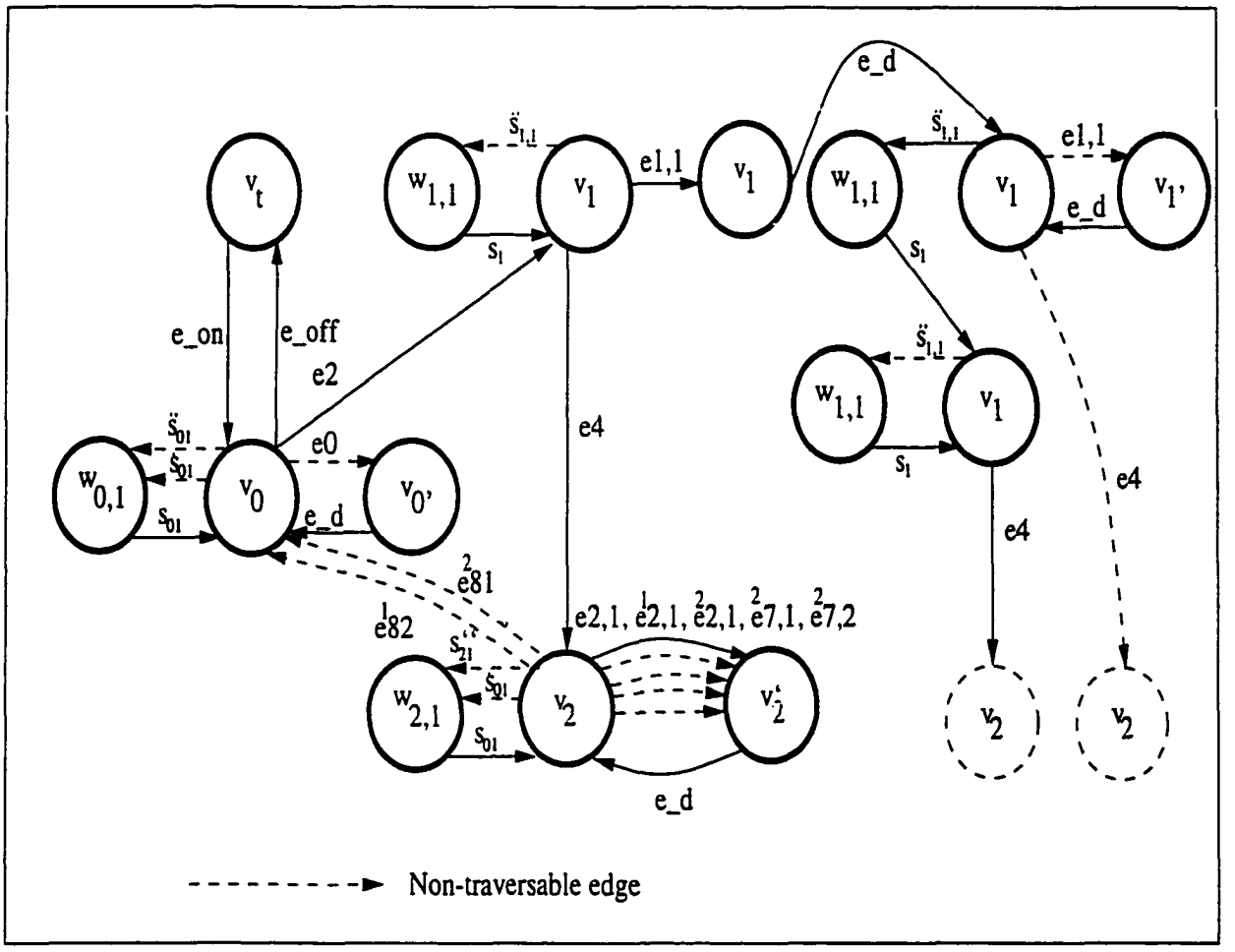

Figure 5.9: The FSM graph of Figure 5.8 with the non-traversable outgoing edges of v_0 marked.

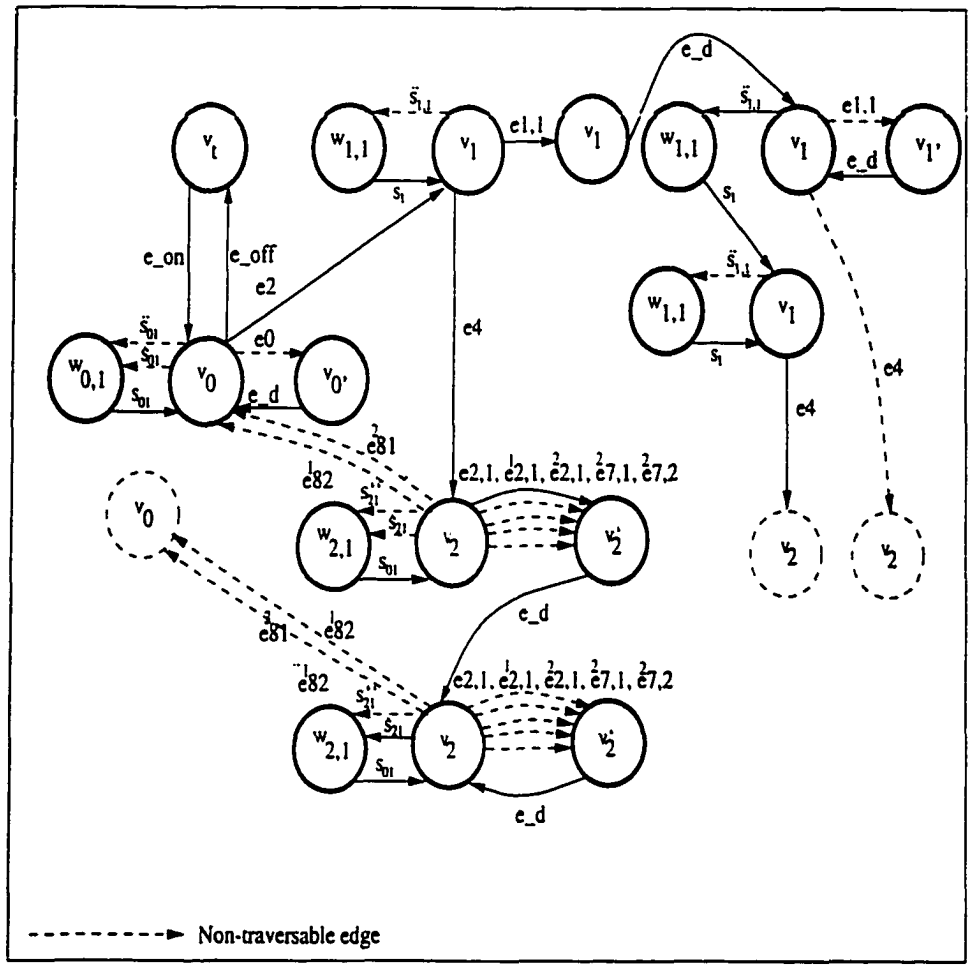

Figure 5.10: The FSM graph after the graph of Figure 5.9 is split due to the effects of $e_{2,1}$.

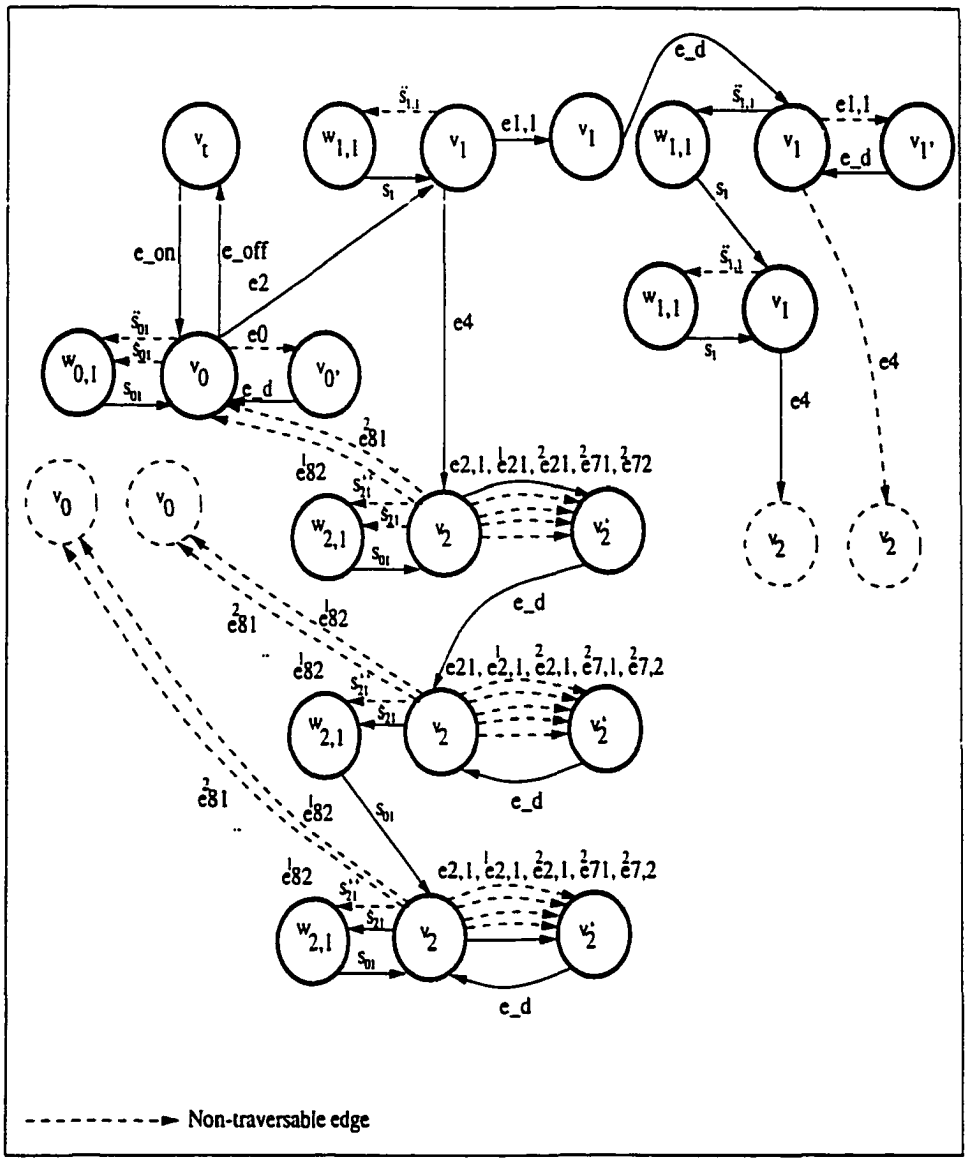

Figure 5.11: The FSM graph after the graph of Figure 5.10 is split due to the effects of s_2.

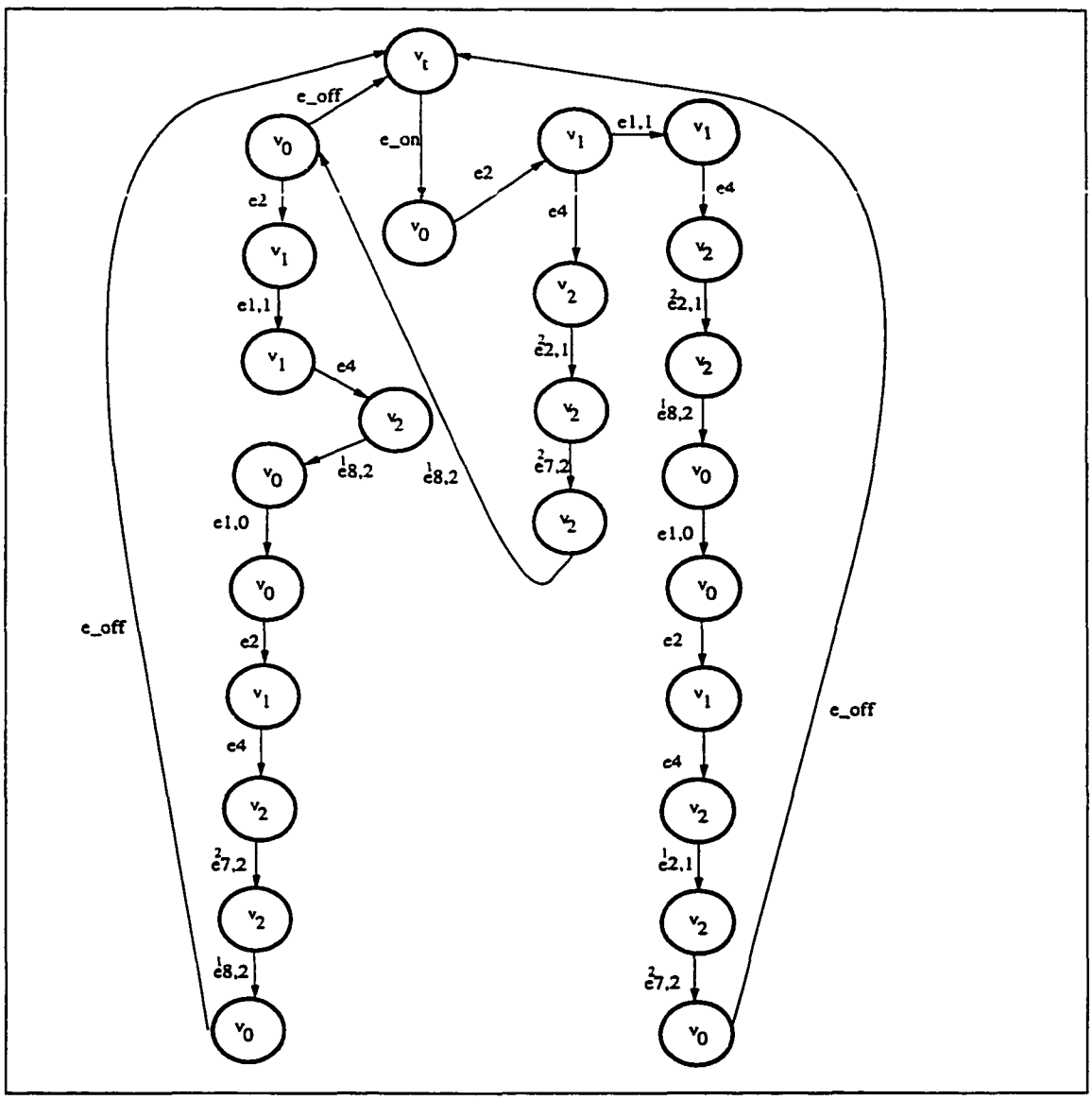

Figure 5.12: The resulting FSM graphs after timing conflicts are removed.

Chapter 6

Conclusions

The generation of only feasible conformance test sequences from the EFSMs is complicated by the interdependencies among the action and condition variables of the EFSMs. The inconsistencies caused by such interdependencies impair the direct application of the FSM-based test generation methods to the EFSM models.

Recently, several methods for generating test sequences for EFSM models appeared in the literature. However, for most of these methods, the inclusion of infeasible paths in the test sequences may be inevitable since the underlying models are EFSMs. Furthermore, the methods that convert EFSMs to equivalent FSMs are applicable only to EFSMs with limited variable domains.

Although, in general, generating feasible test sequences from the EFSM is an open research proble, this dissertation presents a method that enables the generation of realizable test sequences from a class of EFSMs. It is assumed that the specification consists of a single process

with linear actions and conditions. It is also assumed that pointers, recursive functions, and syntactically endless loops are not present in the specification. *Inconsistencies* in EFSM models are defined. Algorithms for the detection and elimination of these inconsistencies from the EFSM models are developed.

The detection and elimination of inconsistencies from the EFSM models are accomplish in two phases. In the first phase, inconsistencies caused be the action variables are detected and eliminated from the graphs. The detection of the action inconsistencies requires accounting for the variable modifications in the paths leading to a given node. The matrix representation of the edge actions allows the effects of the edge actions on the variables to be accumulated.

In the second phase of the inconsistency detection and elimination, the interdependencies among the condition variables are investigated. The system of the linear constraints formed by the edge conditions of a path obtained in the DF graph traversal is used to detect the existence of condition inconsistencies.

The algorithms presented in this thesis eliminate inconsistencies by creating new nodes and edges. However, these nodes and edges are created only when needed to avoid unnecessary growth of the state space. For the cases where the state explosion is unavoidable, the size of the new graph is constantly monitored as the algorithms eliminate the inconsistencies.

Once inconsistencies are eliminated, realizable test sequences can be generated from the resulting consistent EFSM by using the test generation methods available for the FSM models.

The proposed methodology is currently being used to solve the *conflicting timers problem*, which

arises when a protocol has multiple timers running concurrently. Due to the conflicting timers, a test sequence of a protocol such as the Estelle specification of the MIL-STD-188-220 [32] may be interrupted by unexpected timeouts. Preliminary results show that generating test sequences for the MIL-STD 188-220 after eliminating the timer inconsistencies significantly improves the test coverage by including more transitions into the test sequences without timer interruptions.

Appendix A

In this appendix, the format used in the inconsistencies detection and elimination software package for EFSMs is presented. An EFSM graph used as input to the software package as well as the resultant EFSM graphs, after the software package eliminates each of action and condition inconsistencies, are also given.

In this format, the terms used to describe the EFSM graphs include:

- noof_variables: the number of the condition and action variables
- node_no: node number
- node_name: node name, which does not change with the number of times that the node is split
- node_type: indicates if the node is a loop entry/exit or a regular node
- outdeg is the number of outgoing edges of the node
- edge_type: indicates if the edge is a loop exit or a regular edge

- noof_cond: the number of condition of an edge
- <OP>: the operator associated with a condition
- noof_act: the number of actions of an edge (Note X_i can be any variable for a given action)
- const: the constant associated with the condition or action of an edge

The following format is used to describe an EFSM graph with n:

```
==========================================================================
noof_variables = k
variables: X1 X2 ... Xk
initial_node = initial node name

node_no = 1
node_name = node name
node_type = node type
outdeg = m
        edge =  1-st outdegree number
                to_node = ending node number
                edge_type = edge type
                noof_cond = c
                condition =
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                                      .
                                      .
                                      .
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                noof_act =  1
                action =
                          Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                      .
                                      .
                                      .
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const

        edge =  2-nd outdegree number
```

```
                to_node = ending node number
                edge_type = edge type
                noof_cond =  1
                condition =
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                                       .
                                       .
                                       .
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                noof_act =  1
                action =
                          Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                       .
                                       .
                                       .
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const

                      .
                      .
                      .
        edge =  m-th outdegree number
                to_node = ending node number
                edge_type = edge type
                noof_cond =  1
                condition =
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                                       .
                                       .
                                       .
                          a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                noof_act =  1
                action =
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                       .
                                       .
                                       .
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const


node_no = 2
```

```
node_name = node name
node_type = node type
outdeg = m
         edge =  1-st outdegree number
                  to_node = ending node number
                  edge_type = edge type
                  noof_cond =  1
                  condition =
                           a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                  noof_act =  1
                  action =
                           Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                           Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                        .
                                        .
                                        .
                           Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const

         edge =  2-nd outdegree number
                  to_node = ending node number
                  edge_type = edge type
                  noof_cond =  1
                  condition =
                           a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                  noof_act =  1
                  action =
                           Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                           Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                        .
                                        .
                                        .
                           Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                        .
                        .
                        .

         edge =  m-th outdegree number
                  to_node = ending node number
                  edge_type = edge type
                  noof_cond =  1
                  condition =
                           a1 * X1 + a2 * X2 + ... + an *Xn <OP>  const
                  noof_act =  1
                  action =
```

```
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                    .
                                    .
                                    .
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const

                    .
                    .
                    .
node_no = n
node_name = node name
node_type = node type
outdeg = m
        edge =  1-st outdegree number
                to_node = ending node number
                edge_type = edge type
                noof_cond =  1
                condition =
                        a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                noof_act =  1
                action =
                        Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                    .
                                    .
                                    .
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const

        edge =  2-nd outdegree number
                to_node = ending node number
                edge_type = edge type
                noof_cond =  1
                condition =
                        a1 * X1 + a2 * X2 + ... + ak *Xk <OP>  const
                noof_act =  1
                action =
                        Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                    .
                                    .
                                    .
                        Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
```

```
                    .
                    .
                    .

        edge =  m-th outdegree number
                to_node = ending node number
                edge_type = edge type
                noof_cond =  1
                condition =
                          a1 * X1 + a2 * X2 + ... + an *Xn <OP>  const
                noof_act =  1
                action =
                          Xi = a1 * X1 + a2 * X2 + ... + ak * Xk + const
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
                                       .
                                       .
                                       .
                          Xi = a1 * X1 + a2 * X2 + ... + an * Xn + const
=============================================================================
```

The following EFSM graph is used as an input to the inconsistencies detection and elimination software package.

```
================================================================================
noof_variables = 3
variables: X Y Z
initial_node = init

node_no = 0
node_name = init
node_type = REGULAR
outdeg = 2
         edge = 0
                  to_node = 1
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                            0 * X + 0 * Y + 1 * Z >  10
                  noof_act =  1
                  action =
                            Y = 0 * X + 0 * Y + 0 * Z + 0
         edge = 1
                  to_node = 1
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                              0 * X  + 0 * Y + 1 * Z <= 10
                  noof_act =  1
                  action =
                              Y = 0 * X + 0 * Y + 0 * Z + 10

node_no = 1
node_name = N1
node_type = REGULAR
outdeg = 1
         edge = 2
                  to_node = 2
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                           1 * X + 0 * Y + 0 * Z <= 5
                  noof_act =  1
                  action =
```

```
                        X = 0 * X + 0 * Y + 0 * Z + 10

node_no = 2
node_name = N2
node_type = REGULAR
outdeg = 2
         edge = 3
                  to_node = 3
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                           0 * X + 1 * Y + 0 * Z > 10
                  noof_act =  1
                  action =
                           X = 0 * X + 0 * Y + 0 * Z + 10
         edge = 4
                  to_node = 3
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                           0 * X + 1 * Y + 0 * Z <= 10
                  noof_act =  1
         action =
                  X = 1 * X + 0 * Y + 0 * Z + 1
node_no = 3
node_name = N3
node_type = REGULAR
outdeg = 2
         edge = 5
                  to_node = 4
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                           0 * X + 0 * Y + 1 * Z <= 10
                  noof_act =  1
                  action =
                           X = 0 * X + 0 * Y + 0 * Z + 10
         edge = 6
                  to_node = 4
                  edge_type = REGULAR
                  noof_cond =  1
                  condition =
                           0 * X + 0 * Y + 1 * Z > 10
                  noof_act =  1
```

```
                action =
                        Y = 1 * X + 0 * Y + 0 * Z + 10

node_no = 4
node_name = N5
node_type = REGULAR
outdeg = 1
        edge = 7
                to_node = 0
                edge_type = REGULAR
                noof_cond =  1
                condition =
                        1 * X + 0 * Y + 0 * Z < 10000
                noof_act =  1
                action =
                        Y = 1 * X + 0 * Y + 0 * Z + 10
================================================================================
```

The output graph after the software package eliminated the action inconsistencies from the above EFSM graph.

```
================================================================================
noof_variables = 3
variables: X Y Z
initial_node = init

node_no = 0
node_name = init
node_type  = REGULAR
outdeg = 2
         edge = 0
                  to_node = 1
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                           0.00 * X + 0.00 * Y + 1.00 * Z > 10.00
                  noof_act = 1
                  action =
                          Y = 0.00 * X + 0.00 * Y + 0.00 * Z + 0.00
         edge = 1
                  to_node = 5
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                           0.00 * X + 0.00 * Y + 1.00 * Z <= 10.00
                  noof_act = 1
                  action =
                           Y = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 1
node_name = N1
node_type  = REGULAR
outdeg = 1
         edge = 2
                  to_node = 2
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                           1.00 * X + 0.00 * Y + 0.00 * Z <= 5.00
                  noof_act = 1
                  action =
                           X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
```

```
                node_no = 2
node_name = N2
node_type  = REGULAR
outdeg = 2
        edge = 3
                to_node = 3
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        0.00 * X + 1.00 * Y + 0.00 * Z > 10.00
                noof_act = 1
                action =
                 X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
        edge = 4
                to_node = 3
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        0.00 * X + 1.00 * Y + 0.00 * Z <= 10.00
                noof_act = 1
                action =
                        X = 1.00 * X + 0.00 * Y + 0.00 * Z + 1.00
node_no = 3
node_name = N3
node_type  = REGULAR
outdeg = 2
        edge = 5
                to_node = 4
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        0.00 * X + 0.00 * Y + 1.00 * Z <= 10.00
                noof_act = 1
                action =
                        X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
        edge = 6
                to_node = 4
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        0.00 * X + 0.00 * Y + 1.00 * Z > 10.00
                noof_act = 1
                action =
                        Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
```

```
node_no = 4
node_name = N5
node_type  = REGULAR
outdeg = 1
         edge = 7
                  to_node = 0
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                           1.00 * X + 0.00 * Y + 0.00 * Z < 10000.00
                  noof_act = 1
                  action =
                           Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 5
node_name = N1
node_type  = REGULAR
outdeg = 1
         edge = 8
                  to_node = 6
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                           1.00 * X + 0.00 * Y + 0.00 * Z <= 5.00
                  noof_act = 1
                  action =
                           X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 6
node_name = N2
node_type  = REGULAR
outdeg = 1
         edge = 9
                  to_node = 7
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                   0.00 * X + 1.00 * Y + 0.00 * Z <= 10.00
                  noof_act = 1
                  action =
                           X = 1.00 * X + 0.00 * Y + 0.00 * Z + 1.00
node_no = 7
node_name = N3
node_type  = REGULAR
outdeg = 2
         edge = 10
```

```
                to_node = 8
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        0.00 * X + 0.00 * Y + 1.00 * Z <= 10.00
                noof_act = 1
                action =
                        X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
        edge = 11
                to_node = 8
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        0.00 * X + 0.00 * Y + 1.00 * Z > 10.00
                noof_act = 1
                action =
                        Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 8
node_name = N5
node_type  = REGULAR
outdeg = 1
        edge = 12
                to_node = 0
                edge_type = REGULAR
                noof_cond = 1
                condition =
                        1.00 * X + 0.00 * Y + 0.00 * Z < 10000.00
                noof_act = 1
                action =
                        Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
================================================================================
```

The final consistent EFSM graph after the software package eliminated each one of action and condition inconsistencies appears below.

```
==========================================================================

noof_variables = 3
variables: X Y Z
initial_node = init

node_no = 0
node_name = init
node_type  = REGULAR
outdeg  = 2
         edge = 0
                  to_node = 9
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                            -0.00 * X + -0.00 * Y + -1.00 * Z <= -10.01
                  noof_act = 1
                  action =
                            Y = 0.00 * X + 0.00 * Y + 0.00 * Z + 0.00
         edge = 1
                  to_node = 13
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                            0.00 * X + 0.00 * Y + 1.00 * Z <= 10.00
                  noof_act = 1
                  action =
                            Y = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 9
node_name = N1
node_type  = REGULAR
outdeg  = 1
          edge = 12
                  to_node = 10
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                            1.00 * X + 0.00 * Y + 0.00 * Z <= 5.00
                  noof_act = 1
```

```
                action =
                          X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 10
node_name = N2
node_type  = REGULAR
outdeg  = 1
        edge = 13
                to_node = 11
                edge_type = REGULAR
                noof_cond = 1
                condition =
                          0.00 * X + 1.00 * Y + 0.00 * Z <= 10.00
                noof_act = 1
                action =
                          X = 1.00 * X + 0.00 * Y + 0.00 * Z + 1.00
node_no = 11
node_name = N3
node_type  = REGULAR
outdeg  = 1
        edge = 14
                to_node = 12
                edge_type = REGULAR
                noof_cond = 1
                condition =
                          -0.00 * X + -0.00 * Y + -1.00 * Z <= -10.01
                noof_act = 1
                action =
                          Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 12
node_name = N5
node_type  = REGULAR
outdeg  = 1
        edge = 15
                to_node = 0
                edge_type = REGULAR
                noof_cond = 1
                condition =
                          1.00 * X + 0.00 * Y + 0.00 * Z <= 9999.99
                noof_act = 1
                action =
                          Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 13
node_name = N1
node_type  = REGULAR
```

```
outdeg  = 1
         edge = 16
                  to_node = 14
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                            1.00 * X + 0.00 * Y + 0.00 * Z <= 5.00
                  noof_act = 1
                  action =
                            X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 14
node_name = N2
node_type  = REGULAR
outdeg  = 1
         edge = 17
                  to_node = 15
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                            0.00 * X + 1.00 * Y + 0.00 * Z <= 10.00
                  noof_act = 1
                  action =
                            X = 1.00 * X + 0.00 * Y + 0.00 * Z + 1.00
node_no = 15
node_name = N3
node_type  = REGULAR
outdeg  = 1
         edge = 18
                  to_node = 16
                  edge_type = REGULAR
                  noof_cond = 1
                  condition =
                            0.00 * X + 0.00 * Y + 1.00 * Z <= 10.00
                  noof_act = 1
                  action =
                            X = 0.00 * X + 0.00 * Y + 0.00 * Z + 10.00
node_no = 16
node_name = N5
node_type  = REGULAR
outdeg  = 1
         edge = 19
                  to_node = 0
                  edge_type = REGULAR
                  noof_cond = 1
```

```
condition =
          1.00 * X + 0.00 * Y + 0.00 * Z <= 9999.99
noof_act = 1
action =
          Y = 1.00 * X + 0.00 * Y + 0.00 * Z + 10.00
```

==========

Bibliography

[1] I. Adler and N. Megiddo, "A Simplex Algorithm Whose Average Number of Steps is Bound Between Two Quadratic Functions of the Smaller Dimension," *Journal of the ACM*, vol. 32, No. 4, Oct., 1985, pp. 871-895.

[2] S. K. Abd-Hafiz and R. Basili, "A knowledge-Based Approach to the Analysis of Loops," *IEEE Trans. on Software Eng.*, vol. 22, No. 5, May 1996, pp. 339-360.

[3] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours," *IEEE Trans. on Communications*, vol. 39, no. 11, Nov. 1991, pp. 1604-1615.

[4] P. J. Ashenden, The Designer's Guide to VHDL. Morgan Kaufmann, San Francisco, CA., 1996.

[5] B. Beizer, Software Testing Techniques, Thomson Ccmputer Press, Boston, MA., 1990.

[6] H. V. Bertine, W. B. Elsner, P. K. Verma, and K. T. Tewani, "Overview of Protocol Testing Programs, Methodologies and Standards," *AT&T Tech. Journal*, Jan./Feb. 1990, pp. 7-16.

[7] J. Bhasker, A VHDL Primer, Prentice Hall PTR, Englewood Cliffs, NJ., 1995.

[8] E. Brinksma, "A Theory for the Derivation of Tests," In *Proc. IFIP Protocol Specif. Test. Verif. (PSTV)*, Amsterdam: North Holland. 1988.

[9] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems," *Computer Networks and ISDN Systems*, 14 (1987), pp. 3-23.

[10] L. Brömstrup and D. Hogrefe, "TESDL: Experience with Generating Test Cases from SDL Specifications," SDL '89: The Language at Work. Proc. Fourth SDL Forum, pp. 267-279, 1989.

[11] R. Castanet and R. Sijelmassi, "Methods and Semi-automatic Tools for Preparing Distributed Testing," in Protocol Specification. Testing and Verification, ed. B. Sarikaya and G. V. Bochmann, vol. VI, North Holland, Amsterdam, 1986.

[12] A. Cimitile, A. De Lucia, and M. Munro. "Qualifying Reusable Functions Using Symbolic Execution," Proc. of 2nd IEEE Working Conference on Reverse Engineering, Toronto, Canada, IEEE Comp. Soc. Press, July 1995. pp. 178-187.

[13] S. Chanson and J. Zhu, "A Unified Approach to Protocol Test Sequence Generation," Proc. of *IEEE INFOCOM* 1993, pp. 1d.1.1-1d.1.9.

[14] 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, Jan. 1996, pp. 57-79.

[15] T. Cheatham, G. Holloway, and J. Townley. "Symbolic Evaluation and the Analysis of Programs," *IEEE Trans. on Software Eng.*, vol. SE-5, No. 4, July 1979, pp. 402-417.

[16] T. Chow, "Testing Software Design Modeled by Finite-State Machines" *IEEE Trans. on Software Eng.*, vol. SE-4, no.3, May. 1978, pp. 178-187.

[17] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", *IEEE Trans. on Software Eng.*, vol. SE-2, No. 3. Sep. 1976, pp. 215-221.

[18] L. A. Clarke, "Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Trans. on Software Eng.*, vol. 15, No. 11, Nov. 1989, pp. 1318-1332.

[19] A. T. Dahbura, K. K. Sabnani, and M. Umit Uyar, "Algorithmic Generation of Protocol Conformance Testing," *AT&T Tech. Journal.* Jan./Feb. 1990, pp. 101-108.

[20] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli, "Software Specialization Via Symbolic Execution," *IEEE Trans. On Software Eng.*, vol. 17, No. 9, Sept. 1991, pp. 884-899.

[21] T. Cormen, C. Leiserson and R. Rivest. Introduction To Algorithms, McGraw Hill, 1996.

[22] S. Crawley, J. Inulska, and B. McClure. "OPD-Based Adaptive Management of Network Resources in Heterogeneous Defence Networks." IEEE Workshop on Distributed Systems Operations and Management, Newark, DE.. Oct. 1998.

[23] J. C. Corbett, "Evaluating Deadlock Detection Methods for Concurrent Software," *IEEE Trans. on Software Eng.*, vol. 22. No. 3, March 1996, pp. 161-180.

[24] P. D. Coward, "Symbolic Execution Systems - a Review," *Software Eng. Jour.*, Nov. 1988, pp. 229-239.

[25] I. C. Davidson, "International Conformance Testing - Towards the Next Decade," Proc. Second int'l Workshop Protocol Testing Systems. 1989, pp. 3-15.

[26] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. on Software Eng.*, vol. 17, No. 9. Sep. 1991, pp. 900-910.

[27] A. Duale and U. Uyar, "Generation of Feasible Test Sequences for EFSM Models," In H. Ural, R. Probert, and G. v. Bochmann, eds. *Proc. IFIP Int'l Conf. Testing o Communicating Systems, TestCom*, Ottawa, Sept. 2000, pp. 91-109.

[28] A. En-Nauaary, R. Dssaouli, F. Khendek, and A. Elqortobi, "Timed Test Cases Generation Based on State Characterisation Technique." *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, pp. 220-229, Madrid, Spain, Dec. 1998.

[29] S. Evens, Graph Algorithms, Computer Science Press, 1979.

[30] M. Fecko, P. Amer, U. Uyar, and A. Duale. "Test Generation in the Presence of Conflicting Timers," In H. Ural, R. Probert, and G. v. Bochmann, eds, *Proc. IFIP Int'l Conf. Testing o Communicating Systems, TestCom*, Ottawa. Sept. 2000, pp. 301-320.

[31] M. Fecko, U. Uyar, P. Amer, A. Sethi, T. Dzik, R. Menell, and M. McMahon, "A Success Story of Formal Description Techniques: Estelle Specification and Test Generation for MIL-STD 188-220," In R. Lai, ed, *FDT in Practice*, vol. 23 of Comput. Communic. (special issue), Spring 2000 (in press).

[32] M. Fecko, U. Uyar, A. Duale and P. Amer, "Efficient Test Generation for Army Network Protocols with Conflicting Timers," *In Proc. IEEE Milit. Commun. Conf. (MILCOM)*, Los Angeles, CA., Oct. 2000 (to appear).

[33] M. Fecko, Timing and Controllability Issues in Conformance Testing of Communications Protocols, PhD Thesis, April 1999.

[34] S. Fujiwara, G. Bochmann, F. Khendek. M. Amolou, and A. Ghedamsi, "Test Selection Based on Finite State Machine Models," *IEEE Trans. on Software Engr.*, 17(6):591-603, June 1991.

[35] J. G. Gowen, B. Nguyen and W. Butler. "An Experimental Using VHDL to Model and Simulate the ISDN LAPD Data Link Layer." In Proc. US Army Research Lab. Fed. Lab Symp.-ATIRP, College Park, MD., Feb. 1998, pp. 163-167.

[36] W. Grassmann and J. Tremblay, Logic and Discrete Mathematics *a Computer Science Perspective*, Prentice Hall, 1996.

[37] W. Hengeveld and J. Kroon, " Using Checking Sequences for OSI Session Layer Conformance Testing," Proc. Seventh Int'l Conf. Protocol Specification, Testing, and Verification, pp. 435-449, 1987.

[38] T. Higashino and G. Bochmann, "Automatic Analysis and Test Case Derivation for a Restricted Class of LOTOS Expressions with Data Parameters," *IEEE Trans. on Software Eng.*, vol. 20, No. 1, Jan. 1994, pp. 29-42.

[39] T. Higashino, A. Nakata, K. Taniguchi and A. Cavalli, "Generating Test Cases for a Timed I/O Automaton Model," In *Proc. IFIP Int'l Workshop on Test Commun. Syst. (IWTCS)*, pp. 197-214, Budapest, Hungary, Sept. 1999.

[40] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Inc, San Meteo, CA., 1990.

[41] D. Hogrefe, "On the Development of a Standard for Conformance Testing Based on Formal Specifications," *Comput. Stand. Interf.*. 14(3):185-190, 1992. Test Cases from SDL Specifications," SDL '89: The Language at Work, Proc. Fourth SDL Forum, pp. 267-279, 1989.

[42] U. Kodres, "Analysis of Real-Time Systems by Data Flowgraphs" , *IEEE Trans. on Software Eng.*, vol. SE-4. No. 3, May 1978, pp. 169-177.

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

[44] M. Gallagher, V. Narasimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Trans. on Software Eng.*. vol. 23. No. 8, Aug. 1997, pp. 473-484.

[45] J. Gowen, B. Nguyen, and W. Butler, "An Experimental Using VHDL to Model and Simulate the ISDN LAPD Data Link Layer." In Proc. US Army Research Lab. Fed. Lab Symp.-ATIRP, pp. 163-167, College Park, MD., Feb. 1998.

[46] G. Holtzmann, Design and Validation of Computer Protocols, Prentice Hall, Englewood Cliffs, NJ., 1991.

[47] W. Howden, "Symbolic Testing and Dissect Evaluation System," *IEEE Trans. on Software Eng.*, vol. SE-2, No. 4, July 1977, pp. 266-278.

[48] W. Howden, "DISSECT - A Symbolic Evaluation and Path Testing System," *IEEE Trans. on Software Eng.*, vol. SE-4, No. 1, Jan. 1978. pp. 70-73.

[49] W. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Trans. on Software Eng.*, vol. SE3, No. 3, Sept. 1976. pp. 208-215.

[50] D. Lee and M. Yannakakis, "Testing Finite-State Machines: State Identification and Verification," *IEEE Trans. on Computers*, vol 43. no. 3, March 1994, pp. 306-320.

[51] X. Li, T. Higashino, M. Higuchi and K. Taniguchi. "Automatic Generation of Extended UIO Sequences for Communication Protocols in an EFSM Model," In Proc. of 7th Int'l Workshop on Protocol Test Systems, Tokyo. Japan. Nov. 1997, pp. 225-240.

[52] P. C. Jorgensen, Software Testing. A Craftsman's Approach, CRC Press, New York, 1995.

[53] R. J. Linn and M. U. Uyar, Conformance Testing Methodologies and Architecture for OSI Protocols, IEEE Computer Society, Los Alamitos, CA., 1994.

[54] G. J. Mayers, The Art of Software Testing. John Willey and Sons, New York, NY., 1978.

[55] D. Y. Lee and J. Y. Lee, "A Well-Defined Estelle Specification for the Automatic Test Generation," *IEEE Trans. on Software Engr.*, vol 40. No. 4, 1991, pp. 526-541.

[56] G. Luo, R. Probert, and H. Ural, "Approach to Constructing Software Unit testing," *Software Engr. Journal*, Nov. 1995, pp. 245-252.

[57] G. Luo, G. Bochmann, and A. F. Petrenko. "Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-Method," *IEEE Trans. on Software Engr.*, vol. 20, No. 2, 1994, pp. 149-162.

[58] R. Miller and Song, "A Characterization of the Generated Test Sequences Generation Problem for EFSMs," Computer Science Technical Report Series, University of Maryland, College Park, MD., June 1998, CS-TR 3913.

[59] R. Miller and S. Paul, "Generating Conformance Test Sequences for Combined Control Flow and Data Flow of Communication Protocols." In Proc. of 12th International Symposium of Protocol Specification, Testing, and Verification. 1992, pp. 12-27.

[60] J. Moonen, J. Romijn, O. Sies, J. Sprinfintveld and L. Feijs, "A Two-level Approach to Automated Conformance Testing of VHDL Designs," Technical Report SEN-R9707 May 1997.

[61] B. Nguyen, "Using VHDL as an SDL." In Proc. US Army Research Lab. Fed. Lab Symp.-ATIRP, College Park, MD., Jan. 1997, pp. 289-293.

[62] M. H. Sherif and M. U. Uyar, "Protocol Modeling for Conformance Testing: Case Study for the ISDN LAPD Protocol," *AT&T Tech. Journal*, Jan./Feb. 1990, pp. 60-83.

[63] R. L. Probert, H. Ural and B. Yang "Software Quality World-wide: What Are the Practices in a Changing Environment?" *In Proc. The Sixth Inern'l Conference on Software Quality* (6ICSQ), Ottawa, Canada, 1996, pp. 171-180.

[64] S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. on Software Eng.*, vol. SE-11. No. 4. Apr. 1985.

[65] K. K. Sabnani and A. T. Dahbura, "A Protocol Test Generation Procedure," *Computer Networks and ISDN Systems*, vol. 15, 1988. pp. 285-297.

[66] K. Salah, H. Ural, and A. Williams. "Test Generation Based on Control and Data Dependencies within Systems Specified in SDL," (to appear in Computer Communications).

[67] B. Sarikaya, G. Bochmann, and E. Cerny. "A Test Design Methodology for Protocol Testing," *IEEE Trans. on Software Eng.*, Vol. SE-13. No. 5, May 1987, pp. 518-531.

[68] H. Schoots and H. Ural, "Data Flow Oriented Test Selection for LOTOS," *Computer Networks and ISDN Systems*, No. 27, 1995, pp. 111-1136.

[69] D. Sidhu and T. Leung, "Formal Methods for Protocol Testing: A Detailed Study," *IEEE Trans. on Software Eng.*, Vol. 15, No. 4, Apr. 1989. pp. 413-426.

[70] D. Solow, "Linear Programming: An Introduction to Finite Improvement Algorithms," North-Holland, 1984.

[71] R. Thavasinadar, "Test Case Generation and Fault Diagnosis Methods for Communication Protocols on FSM and EFSM Models." PhD Thesis, 1994.

[72] P. Thripathy, "A Unified Model for Protocol Test Case Design," PhD Thesis, 1992.

[73] J. Tretmans, "Conformance Testing with Labelled Transitions Systems and Test Generation," *Comput. Networks ISDN Syst.*, 29(1):49-79, 1996.

[74] H. Ural, "Formal Methods for Test Sequence Generation," *IEEE Trans. on Commun.*, 39(4):514-523, 1992.

[75] H. Ural, "A Test Derivation Method for Protocol Conformance Testing," Proc. Sixth Int' Conf. Protocol Specification, Testing and Verification, 1989, pp. 347-358.

[76] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing," *IEEE Trans. on Communications*, vol., 39, No.. 4. Apr. 1991, pp. 514-523.

[77] M. U. Uyar, A. Lapone, and K. K. Sabnani. "Algorithmic Verification of ISDN Network Layer Protocol," *AT&T Tech. Journal*, Jan./Feb. 1990, pp. 17-31.

[78] M. U. Uyar and A. T. Dahbura, "Optimal Test Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931" in Proc. Global Communications Conf., 1986, pp 68-72.

[79] M. U. Uyar and A. Y. Duale, "Inconsistencies in VHDL Specifications," In Proc. US Army Research Lab. Fed. Lab Symp.-ATIRP, College Park, MD., Jan. 1997, pp. 135-139.

[80] M. U. Uyar and A. Y. Duale, "Resolving Inconsistencies in VHDL Specifications," *In Proc. IEEE Milit. Commun. Conf. (MILCOM)*, Atlantic City, NJ., Oct. 1999 , No. 5.1.3.

[81] M. U. Uyar and A. Y. Duale, "Modeling VHDL Specifications as Consistent EFSMs," *In Proc. IEEE Milit. Commun. Conf. (MILCOM)*. Monterey, CA., Oct. 1997, pp. 740-744.

[82] M. U. Uyar and A. Y. Duale, "Removal of Inconsistencies in VHDL Specifications", In Proc. US Army Research Lab. Fed. Lab Symp.-ATIRP, College Park, MD., Feb. 1998, pp. 225-229.

[83] M. U. Uyar, M. A. Fecko, A. S. Sethi, and P. D. Amer, "Testing Protocols Modeled as FSMs with Timing Parameters," *Computer Networks*, 31(1999), pp. 167-1988.

[84] R. Vemuri and R. Kalyanaraman, "Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constrain Programming," *IEEE Trans. on VLSI Systems*, vol. 3, No. 2, June 1995, pp. 201-214.

[85] U. Voges, L, Gmeiner, and Mayrhauser. "SADAT-An Automated Testing Tool," *IEEE Trans. on Software Eng.*, vol. SE-6, No. 3, May 1980.

[86] E. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Trans. on Software Eng.*, vol. 16, No. 2, Feb. 1990, pp. 121-128.