Checking Sequence Construction Using Multiple Adaptive Distinguishing Sequences

by
Canan Güniçen

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

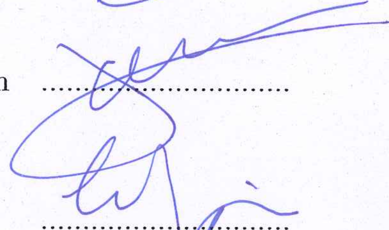
Sabancı University

January, 2015

Checking Sequence Construction Using Multiple Adaptive Distinguishing Sequences

Approved by:

Assoc. Prof. Hüsnü Yenigün ....
(Thesis Supervisor)

Assoc. Prof. Guy Vincent Jordan .... 
(Thesis Co-Supervisor)

Assoc. Prof. Cem Güneri ....

Prof. Dr. Kemal İnan ....

Assist. Prof. Kamer Kaya ....

Date of Approval: Jan 5, 2015

© Canan Güniçen 2015

All Rights Reserved

Acknowledgments

I would like to state my gratitude to my supervisor, Hüsnü Yenigün for everything he has done for me, especially for his invaluable guidance, limitless support and understanding.
I would like to thank Hasan Ural and Guy-Vincent Jourdan for supporting this work with precious ideas and comments.
I would like to thank my family for never leaving me alone.
The financial support of Sabanci University is gratefully acknowledged.
I would like to thank TUBITAK for the financial support provided.

Çoklu Durum Belirleme Dizileriyle Kontrol Dizisi Üretimi

Canan Güniçen

EECS, Yüksek Lisans Tezi, 2015

Tez Danışmanı: Hüsnü Yenigün
Tez Eşdanışmanı: Guy Vincent Jourdan

Anahtar Kelimeler: Biçimsel Sınama Yöntemleri, Kontrol Dizisi, Çoklu Durum Belirleme Dizileri

Özet

Bu çalışmada Sonlu Durum Makinaları (SDM) bazlı sınamada yeni bir kontrol dizisi üretim yöntemi verilmektedir. Tek bir durum tanıma dizisi kullanmakta olan literatürdeki mevcut yöntemlerin aksine, birden fazla durum tanıma dizisinin kullanılması önerilmektedir. Birden fazla durum tanıma dizisinin kullanımı ile kontrol dizisi üretimi sırasında daha kısa durum belirleme dizileriyle kontrol dizisinin uzunluğunun azaltılacağı öngörülmektedir. Önerilen yöntem iki safhadan oluşmaktadır. Ilk safhada, birden fazla durum tanıma dizisi kullanılarak bir sınama dizisi ω oluşturulmaktadır. Ikinci safhada ise ω tekrar ele alınıp yapılan eklentilerle bir kontrol dizisi haline getirilmektedir. Bu çalışmada yeni yöntemin mevcut yöntemlere göre daha kısa kontrol dizileri ürettiğini gösteren deneysel çalışmalar da sunulmaktadır.

Checking Sequence Construction Using Multiple Adaptive Distinguishing Sequences

Canan Güniçen

EECS, Master's Thesis, 2015

Thesis Supervisor: Hüsnü Yenigün
Thesis Co–Supervisor: Guy Vincent Jourdan

Keywords: Formal Testing Methods, Checking Sequences, Adaptive Distinguishing Sequences

Abstract

A new method for constructing a checking sequence for finite state machine (FSM) based testing is introduced. Unlike its predecessors, which are based on state recognition using a single state identification sequence, our approach makes use of multiple state identification sequences. Using multiple state identification sequences provides an opportunity to construct shorter checking sequences, based on a greedy approach of choosing a state identification sequence that best suits our goal at different points during the construction of the checking sequence. Our approach has two phases. In the first phase, a test sequence ω is constructed using multiple state identification sequences. The sequence ω is not guaranteed to be a checking sequence, however it is further extended to a checking a sequence by the second phase of our method. We present the results of an experimental study showing that our two phase approach produces shorter checking sequences than the previously published methods.

Contents

1 Introduction **1**

2 Preliminaries **6**

2.1 FSM Fundamentals 6

2.1.1 Extending Next State and Output Functions 7

2.1.2 Some Properties of FSMs 7

2.2 Representing an FSM by a Directed Graph 8

2.2.1 Paths of Input Sequences 9

2.3 Distinguishing Sequences 9

2.3.1 Preset Distinguishing Sequence 9

2.3.2 Adaptive Distinguishing Sequence 10

2.3.3 Multiple Adaptive Distinguishing Sequence 11

2.4 Checking Sequences based on Distinguishing Sequences 12

3 State Recognition using Multiple ADS Trees **14**

3.1 Cross Verification 15

3.2 Extended State Recognition Definition 19

3.3 Checking Sequences: Sufficient Condition 21

3.4 Generation of Recognition Automaton 23

3.5 State Recognition on Recognition Automaton 24

3.6 Merging Nodes on Recognition Automaton 27

4 Checking Sequence Generation Algorithm **33**

4.1 Mutual Dependency Between ADS Trees 34

4.2 Phase 1: Sequence Generation 36

4.2.1 Sequence Extension Options 36

4.3 Phase 2: Checking if a sequence is a checking sequence 48

5 Construction and Selection of ADS Trees **54**

5.1 ASP Formulation of ADS 55

5.1.1 Optimization 58

5.2 ADS Tree Generation Using an ADS 58

5.3 ADS Tree Selection Algorithm 62

6 Experimental Results **64**

6.1 Comparison with Simao et al.s Method 64

6.2 Contribution of Pair ADS Tree Selection Algorithm 68

6.3 The Negative Effect Of Cross Verification 70

6.4 Contributions of Phase 1 and Phase 2 71

7 Conclusion and Future Work **72**

List of Figures

2.1 The FSM M_0 6

2.2 A Path $P_{M,\tilde{x}}$, constructed by using input sequence $\tilde{x}$ and the FSM M_0 9

2.3 An ADS Tree $\mathcal{A}^1$ 11

2.4 An ADS Tree $\mathcal{A}^2$ 11

2.5 An ADS Tree $\mathcal{A}^3$ 11

3.1 The FSM M_1 with two ADS trees: a and b. 16

3.2 A subgraph of the FSM M_0: aasaaa is a CS 17

3.3 A subgraph of the FSM M_0: bbtbbb is a CS 17

3.4 Spanning tree of the FSM M_1 18

3.5 This FSM is not isomorphic to the M_1 but produces the same output response to aasaaabbtbbb 18

3.6 Two subpaths of $P_{N,\omega}$ 20

3.7 A Path $P_{N,\omega}$ 24

3.8 A Path $P_{M,\omega}$ 24

3.9 Showing ADS tree $\mathcal{A}^2$ is legal 26

3.10 Valid observations based on $\mathcal{A}^2$ 26

3.11 An Application of Rule 1 and 2 on R 27

3.12 An Application of Rule 3 and 4 on R 28

3.13 Merging nodes n_4, n_5 and n_6 on R 28

3.14 Merging nodes n_4 and n_{12} on R 29

3.15 Merging nodes n_4 and n_{13} on R . . . 29
3.16 Showing ADS tree $\mathcal{A}^3$ is legal . . . 30
3.17 Valid observations on R based on the ADS Tree $\mathcal{A}^3$ . . . 30
3.18 Merging nodes n_7, n_8 and n_{14} on R . . . 31
3.19 Showing ADS Tree $\mathcal{A}^1$ is legal . . . 31
3.20 Merging nodes n_1, n_2 and n_{10} on R . . . 32
3.21 A Collapsed Recognition Automaton R . . . 32

4.1 Two subpaths of $P_{N,\omega}$ . . . 35
4.2 Backtracking Example . . . 37
4.3 The Tree T constructed for backtracking . . . 38
4.4 Updated Recognition Automaton R after sequence extension . . . 39
4.5 Recognition Automaton R after merging operations . . . 39
4.6 Transition verification example . . . 41
4.7 Sequence extension on R . . . 42
4.8 Updated Recognition Automaton R after transition verification . . . 43
4.9 Missing Transition Verification Example . . . 44
4.10 Recognition Automaton R after sequence extension . . . 45
4.11 Recognition Automaton R after merging operations . . . 45
4.12 A Recognition Automaton R . . . 46
4.13 Valid Observation of ADSs on the Path $P_{N,\omega}$ . . . 49
4.14 Sequence Extension on the Path $P_{N,\omega'}$ . . . 51
4.15 A Path $P_{N,\omega'}$ after merging nodes n_7, n_8, n_9, n_{14} and n_{15} . . . 51
4.16 A Path $P_{N,\omega'}$ after merging nodes n_1, n_2, n_3, n_{10}, n_{11}, n_{16}, n_{17} . . . 52
4.17 Resulting Recognition Automaton R . . . 53

5.1 A Partial ADS Tree . . . 60
5.2 Initial Partial ADS Tree . . . 61

5.3 A Partial ADS Tree Step 1 61
5.4 A Complete ADS Tree 62

List of Tables

3.1 ADS Table 25

6.1 Improvement in CS Lengths 65

6.2 Improvement in CS Lengths with 4 additional input symbols 66

6.3 Improvement in CS Lengths with 8 additional input symbols 66

6.4 Experimental results for FSMs without an improvement 67

6.5 Improvement in CS Lengths (Single ADS Tree Selection Algorithm) . 68

6.6 Improvement in CS Lengths with 4 additional input symbols (Single ADS Tree Selection Algorithm) 69

6.7 Improvement in CS Lengths with 8 additional input symbols (Single ADS Tree Selection Algorithm) 69

6.8 Percentage of single ADS tree included in multiple ADS trees 70

6.9 Improvement in CS Lengths without Cross Verification 70

6.10 Contribution of Phase 2 to CS Length 71

Chapter 1

Introduction

A Finite State Machine (FSM) is an abstract structure with a finite set of states where the application of an input symbol results in a state transition along with the production of a respective output symbol. FSMs have been used to model many types of systems in diverse areas including sequential circuits [14], software design [10], communication protocols [7, 10, 11, 22, 25], object-oriented systems [5], and web services [4, 18]. Many systems are implemented using FSM-based models. FSM-based modelling techniques are also often employed in defining the control structure of a system specified by using languages such as SDL [3], Estelle [8], and the State Charts [17]. With the advanced computer technology, as the systems constructed using FSM-based modelling became more complex, distributed and large to fulfill complicated tasks, it becomes harder to create systems, get the functionality of systems right and test them since they are becoming less reliable. As an inevitable result, testing becomes an indispensable part of the system design and implementation. Considerable amount of the cost of software development is spent on software testing. Thus, the research that perceives testing FSM-based models as an optimization problem and ensures the reliability of these models gained importance. This motivates the study of FSM-based testing to ensure the correct functioning of systems and to discover the aspects of their behaviours. Therefore, FSM-based testing is a research area that is motivated to answer these reliability demands.

Testing is a fundamental step in any software development process. It consists in applying a set of experiments to a system, with multiple aims, for obtaining

some piece of unknown information to check the correctness or to measure the performance of the system. These different aims give rise to the different classes of testing problems. Some classes of the testing problems, pioneered in the paper of Moore [13]. Here we will consider two types of testing problem. In the first type, we are given a finite state machine with a known state-transition diagram but with an unknown current state. We are asked to perform an experiment in order to find the unknown current state. In other words, the specification of the finite state machine is available, but we do lack information about in which state it is currently. The information about its current state is found by applying an input sequence to the finite state machine so that information desired about its current state can be deduced from its input/output (I/O) behavior. State identification problem can be given as a specific example for this type of testing problem. In the state identification problem, the initial state of the machine is identified by applying a test sequence which is a *UIO(Unique Input/Output)* sequence [29] and by observing the output response of the machine, we are able to tell which state the machine was because each different state of an FSM gives a different output response to the $UIOs$.

The second type of testing problem we consider is conformance testing. In conformance testing, we have an implementation which we want to test whether it conforms to its specification or not. In other words, conformance testing tries to determine if an implementation, that is intended to implement some specification, is a correct implementation of its specification or not. In general, we lack information about the implementation and we would like to deduce this information by providing a sequence of input symbols to the implementation and observing the sequence of output symbols produced. Let FSM M be the specification of a system and N be an implementation. Conformance testing tries to answer the question whether N is equivalent to M. The notion of equivalence of $FSMs$ is that for any I/O pair that is defined for the specification M, the implementation N should produce the same sequence of output symbols O like M as a response to the input sequence I. An *Implementation Under Test (IUT)* is considered to be an FSM N given as a black box. IUT is an FSM N which we lack information about its transitions we assume that it has at most as many states as M and to have the same input alphabet as M. Thus the approach that is used to test an FSM-based system is to apply an input sequence and observe the output symbols produced by the IUT. Conformance testing uses this sequence of output symbols obtained by observing the response of the input sequence and tries to deduce the correct functioning of IUT by comparing

the output symbols produced by the IUT against the expected output symbols produced by the specification FSM M. This is called the conformance testing or the fault detection problem. An input sequence that can determine if IUT is a correct or a faulty implementation of the specification is called a *checking sequence*.

State verification is a crucial part of the conformance testing since the main aim is to find a correspondence between the states of the implementation N and the specification M. Also, for an input sequence to be a checking sequence, it has to verify every transition of the specification M [31]. State verification experiment gives the information at which state the IUT currently is. A transition verification can only be performed by recognizing the initial and the final states of the transition. Recognition of states is required to determine whether the IUT is in the correct state before the input symbol of the transition is applied, also to check if it reaches to the correct state after the application of the input symbol of the transition, while giving the expected output symbol. There are special sequences that solve the state verification problem. Three techniques are proposed for state verification: Distinguishing Sequence (DS) [29], Unique Input Output (UIO) sequences [29] and Characterizing Sets [23]. Checking sequence generation methods using the above special sequences are called the *D-Method* [29], *U-Method* [29], and *W-Method* [10, 27] respectively.

According to the survey in [25], the earliest published work on conformance testing dates back to the 50's and activities mainly focused on automata theory and sequential circuit testing. Machine identification problem was published in 1956 Moore's paper [13]. In this paper, he studied the problem of machine identification where there is an FSM with a known number of states and problem is to determine the state diagram of the unknown FSM by observing its I/O behaviour. As well as the machine identification problem, he also raised the topic of conformance testing problem.

In 1964, Hennie [19] proposed a method called *D–method* that is if the FSM has a preset distinguishing sequence of length L then one can construct a checking sequence of length polynomial in L and the number of states of the FSM. However, not every FSM has a PDS so that Hennie also proposed a method that generates exponentially long checking sequences for the case where PDS cannot be found for an FSM. Following this work, it has been widely assumed that fault detection is easy if the FSM has a PDS and hard otherwise. Later other checking sequence

generation methods that are based on UIO sequences [1, 29], characterizing sets [10] and transition tours [11, 27] were proposed.

Although there were some studies in late 60's and early 70's, conformance testing became a more active research area at the beginning of 90's and is studied due to its applications in testing communication protocols. Because protocols are set of rules and behaviours that describe how a computer system in a network should behave and their implementations should be tested to decide whether they conform the defined behaviours or not. Therefore, it made the conformance testing one of the central problems in protocols so that they are modelled by FSMs with a small number of states, but a large number of input and output symbols. The methods proposed so far was only used in some special cases, since conformance testing of large machines with many states, input and output symbols cannot be handled by using a brute force approach requiring an exponential length test sequence.

Later studies focused on the cases where the checking sequence length stays polynomial. Those were the methods which use PDS for state verification where a reliable reset in the implementation may or may not be available. In general, some improvements are introduced using global optimization techniques. In [1], UIO is used instead of the PDS and the checking sequence generation problem is modelled as a rural Chinese postman tour problem and a checking sequence generation problem is solved by computing the minimum-cost tour of the transition graph of a finite state machine. This optimization problem is further improved in [20, 21]. In [21], the method that uses a predefined set of sequences for state verification and an acyclic set of transitions is improved by stating how these sets should be chosen. In [20], method proposed in [21] is further improved by making the state verification sequences to verify set of states at once.

This optimization problem is further improved by eliminating redundant transition verification subsequences [9]. In [32], the checking sequence generation problem is further optimized by eliminating the overlapping parts of the PDS. These were all methods trying to provide global optimization. In [2], Simao et al. proposed a method that optimizes the sequence locally, instead of global optimization. The basic idea in [2] is to exploit overlapping between sequences used in state verification. They obtained better results in comparison to global optimization methods in most cases with this approach. In this thesis, we also try to optimize checking sequence locally, but we reduce the length of the checking sequence by using multiple

ADSs(*Adaptive Distinguishing Sequences*) for state verification.

There are many methods to build checking sequences based on ADSs. One point that is common to many of these methods is the requirement to select one ADS among the possible ones, when there are many. Some results are known regarding the selection of such a sequence: in [33], it is shown that some of the possible state identification sequences may lead to shorter checking sequences because they would facilitate the overlap. It is generally believed that choosing an overall shorter distinguishing sequence should yield shorter checking sequences, but [30] have shown that finding the shortest ADS is NP-complete. By and large, most published papers on the topic of checking sequence generation are essentially mute on the topic of the choice of the ADS and focus mainly on generating as good a checking sequence as possible, given the selected ADS. Nonetheless, none of the published papers considers using multiple state identification sequences in checking sequence construction. Therefore, this thesis pioneers usage of multiple state identification sequences.

The contributions of this thesis to the conformance testing are threefold. First, we present an Answer Set Programming (ASP) formulation to generate an ADS. Therefore, we utilize the construction of ADSs. Secondly, we present a method that determines whether a given input sequence is an adaptive distinguishing sequence based checking sequence or not. Lastly, we present a method that generates a checking sequence. Our main contribution is using multiple ADSs to generate a checking sequence. We redefine the concept of state recognition in the context of multiple ADSs. In addition, we investigate the advantages and disadvantages of recognizing a state with multiple ADS. We introduce the concept of *Cross Verification* which is essential for state recognition by multiple state identification sequences. Experiments show that our method achieves a reduction in the length of checking sequence compared to the method in [28] that uses a single ADS to generate a checking sequence.

The rest of this thesis is organized as follows. In Chapter 2, the basic information on FSMs and conformance testing is provided. In Chapter 3, concept of state recognition in the context of the multiple ADS is presented. In Chapter 4, our checking sequence generation algorithm is provided in detail. In Chapter 5, we present the ASP formulation of the shortest ADS. In Chapter 6, we present the experimental results. Finally Chapter 7 contains the concluding remarks.

Chapter 2

Preliminaries

2.1 FSM Fundamentals

An *finite state machine* (FSM) is specified by a quintuple $M = (S, X, Y, \delta, \lambda)$, where

- S is a *finite set of states* with $n = |S|$.
- X is a *finite set of input symbols* with $p = |X|$.
- Y is a *finite set of output symbols* with $q = |Y|$.
- δ is a *state transition function* that maps $S \times X$ to S.
- λ is an *output function* that maps $S \times X$ to Y.

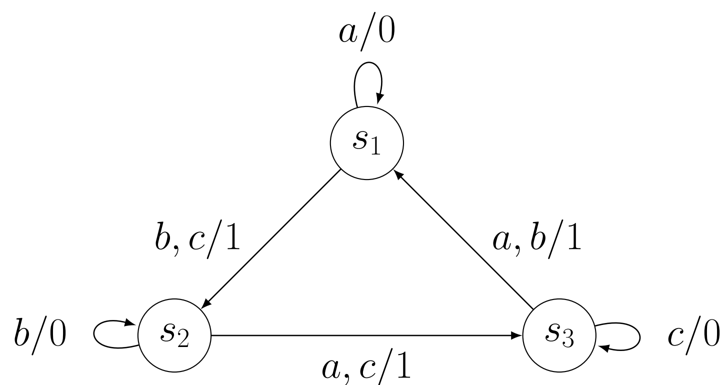


Figure 2.1: The FSM M_0

An FSM M_0 used as a running example throughout the thesis is depicted in Figure 2.1. Here, $S = \{s_1, s_2, s_3\}$, $X = \{a, b, c\}$, and $Y = \{0, 1\}$. From the arc $s_1 \rightarrow s_2$ with label $b/1$, it is possible to deduce that, if M_0 receives input symbol b when in state s_1, then it produces the output symbol 1 and moves to state s_2. A *transition* τ is defined by a tuple $(s_i, s_j; x/y)$ in which s_i is the *starting state*, x is the *input symbol*, $s_j = \delta(s_i, x)$ is the *ending state*, and $y = \lambda(s_i, x)$ is the *output symbol*.

2.1.1 Extending Next State and Output Functions

The functions δ and λ can be extended to take input sequences as follows. For a state $s \in S$, an input sequence $\tilde{x} \in X^\star$, and input symbol $x \in X$ and let $x\tilde{x} \in X^\star$ denote the input sequence obtained by concatenation of $\tilde{x}$ and x (that is a juxtaposition of input (output) sequences and input (output) symbols mean concatenation) then the transition and output functions are extended to sequence of inputs as

- $\tilde{\delta}(s, x.\tilde{x}) = \tilde{\delta}(\delta(s, x), \tilde{x})$ where $\tilde{\delta}(s, \varepsilon) = s$.
- $\tilde{\lambda}(s, x.\tilde{x}) = \lambda(s, x).\tilde{\lambda}(\delta(s, x), \tilde{x})$ where $\tilde{\lambda}(s, \varepsilon) = \varepsilon$.

Note that, for the empty sequence ε we define $\delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$. Throughout the thesis, we will denote functions $\tilde{\delta}$ and $\tilde{\lambda}$ as δ and λ, respectively.

2.1.2 Some Properties of FSMs

Two states s_i and s_j of M are *equivalent* if, for every input sequence $\tilde{x} \in X^\star$, $\lambda(s_i, \tilde{x}) = \lambda(s_j, \tilde{x})$. If $\lambda(s_i, \tilde{x}) \neq \lambda(s_j, \tilde{x})$, then $\tilde{x}$ *distinguishes* between s_i and s_j. For example, the input sequence a distinguishes states s_1 and s_2 of M_0.

Now we will define some properties regarding FSMs with the help of the definition of equivalent states.

- FSM M is *completely specified* if the transition function $\delta(s_i, x)$ and $\lambda(s, x)$ is defined for each $s \in S$ and for each input symbol $x \in X$. In other words, FSM M is completely specified if δ and λ functions are total functions, otherwise it is partially specified.

- Two FSMs, M_1 and M_2 are *equivalent* if and only if, for every state of M_1, there is an equivalent state of M_2 and vice versa.
- FSM M is *minimal* if there is no FSM with fewer states than M is equivalent to M. For an FSM M to be minimal, no two states of M are equivalent. There are algorithms that computes an equivalent minimal FSM when an FSM is given as an input [13].
- FSM M is *strongly connected* if for each pair of states (s_i, s_j) there exists an input sequence $\tilde{x}$ such that $\delta(s_i, \tilde{x}) = s_j$.
- FSM M *deterministic* if for each state $s \in S$ and for each input symbol $x \in X$, M has at most one transition with start state s and input symbol x.

Note that, the way we define an FSM by using functions for δ & λ (instead of relations) only allows us to denote deterministic machines. For nondeterministic machines, relations are used instead of functions. In this thesis, we consider only deterministic and completely specified FSMs. Therefore, λ and δ are total functions.

2.2 Representing an FSM by a Directed Graph

An FSM M can be represented by a digraph $G = (V, E)$ where a set of vertices V represents the set of states S of M, and a set of directed edges E represents the transitions of M. Each edge $e = (v_i, v_j; x/y) \in E$, is a state transition $\tau = (s_i, s_j; x/y)$ from the state s_i to state s_j with an input symbol $x \in X$ and an output symbol $y \in Y$, where v_i and v_j are the starting and ending vertices of e (states of τ), and input/output (i.e., I/O) pair x/y is the label of e, denoted by $label(e)$. Two edges e_i and e_j are called *adjacent* if the ending vertex of e_i and the starting vertex of e_j are same. We will also use (v_i, v_j) to denote an edge when the edge label is not important.

A path $P = (n_1, n_2; x_1/y_1)(n_2, n_3; x_2/y_2) \ldots (n_{r-1}, n_r; x_{r-1}/y_{r-1})$, $r \geq 1$, of $G = (V, E)$ is a finite sequence of adjacent (not necessarily distinct) edges in E. The I/O sequence $x_1 x_2 \ldots x_{r-1}/y_1 y_2 \ldots y_{r-1}$ is called the label of P. P is also represented by $(n_1, n_r; \tilde{x}/\tilde{y})$, where $\tilde{x}/\tilde{y}$ is the label of P.

For two paths $P_1 \& P_2$, $P_1 P_2$ denotes the concatenation of $P_1 \& P_2$, provided that the ending vertex of the last edge in P_2 is the same as the starting vertex of the first

edge in P_2. A path P' is a subpath of P, if there exist paths P_1 and P_2 such that $P = P_1 P' P_2$.

2.2.1 Paths of Input Sequences

Let $P_{M,\tilde{x}}$ be the path that starts from a designated state of M and follows the transition function along the application of the input sequence $\tilde{x}$. For example, if we assume that we start from the state s_1 of M_0 and $\tilde{x} = aabb$, then $P_{M,\tilde{x}}$ is shown in Figure 2.2.

Since the starting state is known, the state corresponding to each node in $P_{M,\tilde{x}}$ is known. These states are given as s_i's in Figure 2.2. With each node of $P_{M,\tilde{x}}$, we also associate an identifier m_i in order to be able to refer to the individual nodes in $P_{M,\tilde{x}}$.

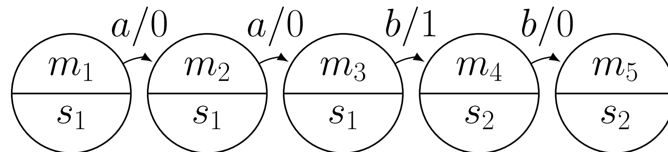


Figure 2.2: A Path $P_{M,\tilde{x}}$, constructed by using input sequence $\tilde{x}$ and the FSM M_0

This kind of path representation will be used to define *Recognition Automaton* which is one of the key concepts of this thesis.

2.3 Distinguishing Sequences

As stated in 1, we will consider the use of distinguishing sequences for checking sequence generation. Distinguishing sequences are special sequences used for state identification. There are two types of distinguishing sequences: preset and adaptive distinguishing sequences.

2.3.1 Preset Distinguishing Sequence

A *Preset Distinguishing Sequence* (PDS) of an FSM M is an input sequence D in response to which every state of M gives a distinct output sequence. For instance,

ba is a PDS for FSM M_0 shown in Figure 2.1.

- $\lambda(s1, ba) = 11$
- $\lambda(s2, ba) = 01$
- $\lambda(s3, ba) = 10$

If the specification FSM has a PDS, then the state verification problem is solved by applying the PDS to the state that is to be verified. However not every minimal FSM has a PDS [23]. To determine if an FSM has a PDS is a PSPACE-complete problem [24].

2.3.2 Adaptive Distinguishing Sequence

An *Adaptive Distinguishing Sequence* (ADS) of M is a decision tree rather than a sequence. Different from the conventional terminology in the literature, we call an ADS of an FSM M as *ADS tree* of M. We use the term ADS to refer to an ADS of a state. ADS of the state s corresponds to a root-to-leaf path of an ADS tree related to the states. Below we make formal definitions of ADS and ADS tree.

An ADS tree $\mathcal{A}$ for an FSM with n states, is a rooted decision tree with n leaves, where the leaves are labeled by distinct states of M, internal nodes are labeled with input symbols, the edges emanating from a node are labeled with distinct output symbols. The concatenation of the labels of the internal nodes on a path from root to leaf labeled by a state s_i represents an ADS $\mathcal{A}_i$ of the state s_i and the concatenation of edge labels on the same path corresponds to the output sequence $\mathcal{Y}_i$ that is produced in response to $\mathcal{A}_i$ by s_i. In other words, $\lambda(s_i, \mathcal{A}_i) = \mathcal{Y}_i$.

Note that, since the output symbols on the edges originating from the same node are distinct, for any other state s_j, we have $\lambda(s_i, \mathcal{A}_i) \neq \lambda(s_j, \mathcal{A}_i)$.

An ADS tree $\mathcal{A}$, specifies an ADS for each state s_i in M. We also use the notation $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ to denote the ADS tree $\mathcal{A}$ as a set of ADSs for the states of M.

Note that PDS is a special case of ADS where for all states s_i, $\mathcal{A}_i = D$. Therefore every FSM which has a PDS also has an ADS. However the inverse is not true. That is there exist FSMs with an ADS but no PDS. Compared to PDS, ADS has

some advantages. Determining the existence of an ADS and finding one if exist is polynomial in number of states and number of inputs [24].

2.3.3 Multiple Adaptive Distinguishing Sequence

Throughout this paper, we use multiple ADS trees such that $\boldsymbol{A} = \{\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^k\}$ where $k \geq 1$ is the number of ADS trees and $\mathcal{A}^i$ is the ith ADS tree in the set. Since there are multiple ADS trees, then every state s_i has multiple ADSs. The set of all ADSs for a state s_i represented by $\boldsymbol{A}_i = \{\mathcal{A}_i^1, \mathcal{A}_i^2, \ldots \mathcal{A}_i^l\}$ where $1 \leq l \leq k$. Note that $k \geq 1$. Since more than one ADS tree in $\boldsymbol{A}$ can have the same ADS for s_i.

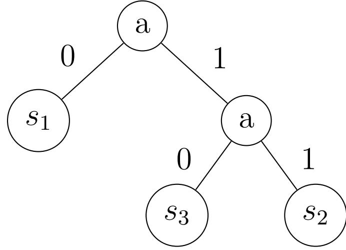


Figure 2.3: An ADS Tree $\mathcal{A}^1$

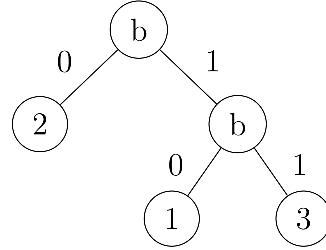


Figure 2.4: An ADS Tree $\mathcal{A}^2$

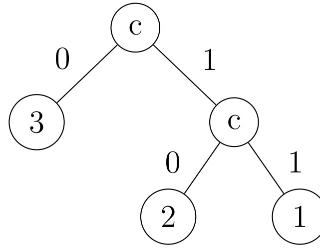


Figure 2.5: An ADS Tree $\mathcal{A}^3$

For FSM M_0 given in Figure 2.1, a set of ADS trees $\boldsymbol{A} = \{\mathcal{A}^1, \mathcal{A}^2, \mathcal{A}^3\}$ is given in Figures 2.3, 2.4 and 2.5. For this set of ADS trees, the set of ADSs of the states of M_0 are:

- $\boldsymbol{A}_1 = \{a, bb, cc\}$
- $\boldsymbol{A}_2 = \{aa, b, cc\}$
- $\boldsymbol{A}_3 = \{aa, bb, c\}$

This thesis considers the problem of generating an efficient checking sequence from a deterministic and completely specified FSM M by using multiple adaptive distinguishing sequences.

2.4 Checking Sequences based on Distinguishing Sequences

A checking sequence generated from an FSM is used in testing to demonstrate correctness of an implementation under test. In a checking sequence, distinguishing sequences are applied in order to recognize the state of the implementation.

Let $M = (S, X, Y, \delta, \lambda)$ denote an FSM that models a black box implementation N. A checking sequence is a test sequence that verifies the implementation is correct. We have the following usual assumptions on N. It has the same I/O alphabet as M, and the number of states of N is at most the same as that of M. The faults in the implementation are caused by the faulty implementation of output and/or next state functions. Let $\phi(M)$ be the set of FSMs that have at most n states and the same input and output alphabets as M and $N \in \phi(M)$. N is *isomorphic* to M if there is a one-to-one and onto function f on the state sets of M and N such that for any state transition $(s_i, s_j; x/y)$ of M, $(f(s_i), f(s_j); x/y)$ is a transition of N. A checking sequence for M is an input sequence starting at a designated state of M and distinguishes M from any $N \in \phi(M)$ that is not *isomorphic* to M. In other words, when a checking sequence is applied to any faulty implementation of the specification M, the output produced by the black box implementation will be different than the output produced by the specification M as an indication of one or more faults.

The main aspect of a checking sequence is that it defines a one to one and onto function f between state set of specification M and state set of implementation N. This is accomplished by the concepts of state recognition and transition verification. We will define these concepts using distinguishing sequences of FSM M as follows.

Let $P = (n_1, n_{r+1}; \tilde{x}/\tilde{y})$ be a path in G from n_1 to n_{r+1} with the label $\tilde{x}/\tilde{y} = x_1x_2 \ldots x_r/\ y_1y_2 \ldots y_r$. Also let $\mathcal{A}$ be an ADS of M. There are two types of recognitions, namely *d-recognition* and *t-recognition* [31]. A node in P is said to be recognized as some state of M if it is either d-recognized or t-recognized where

d-recognition and t-recognition are defined as follows:

- A node n of P is d-recognized as the state s_i of M if n is the starting node of a subpath of P with label $\mathcal{A}_i/\lambda(s_i, \mathcal{A}_i)$
- A node n_i of P is t-recognized as state s of M if there are two subpaths $(n_q, n_i; \tilde{x}/\tilde{y})$ and $(n_j, n_k; \tilde{x}/\tilde{y})$ of P such that n_q and n_j are recognized as the same state s' of M, n_k is recognized as state s of M.

In addition, a transition verification is defined as follows. A transition $\tau = (s_i, s_j; x/y)$ of M is verified if there is an edge $(n_k, n_{k+1}; x/y)$ of P such that nodes n_k and n_{k+1} are recognized as states s_i and s_j of M respectively.

The following theorem from [31] (rephrased in our notation) states a sufficient condition for a checking sequence.

Theorem 1 *Let ω be the input portion of the label of a path P of directed graph G (for FSM M) such that every transition is verified in P. Then ω forms a checking sequence for M.*

This thesis considers the problem of generating a checking sequence by using multiple ADS trees. The state recognitions and transition verifications is performed by using multiple ADSs. This requires some modifications on the definition of state recognition and transition verification concepts. These modifications are presented in the next chapter.

Chapter 3

State Recognition using Multiple ADS Trees

In the literature state recognition is performed using a distinguishing sequence or a characterizing set or a set of UIO sequences but none of them use multiple of these sequences to recognize a state.

Let M be a minimal, completely specified, deterministic and strongly connected FSM with n states. $\phi(M)$ denotes the set of FSMs such that each FSM $N \in \phi(M)$ has at most n states and has the same input and output alphabets as M. An input sequence ω is a checking sequence for M if and only if ω distinguishes between M and all elements of $\phi(M)$ that are not isomorphic to M. A checking sequence ω is designed to be applied at a particular state s_1 of M. Before the application of a checking sequence, the implementation is *initialized* to bring N to its state (node n_1) which is supposed to correspond to s_1 of M (e.g. by using a homing sequence followed by a transfer sequence). Then ω is applied to node n_1 of N. ω is a checking sequence for M if and only if $\lambda(s_1, \omega) \neq \lambda(n_1, \omega)$ for any faulty implementation N. Hence when checking sequence ω is applied on any faulty implementation N, the output produced by N will be different than the output produced by specification M.

A checking sequence defines a one to one and onto function f between the states of the specification M and the nodes of the implementation N and tries to show that there is a correspondence between specification M and implementation N in terms of transition and output functions. Thus, our job is to find this correspondence by using state recognitions. But in the context of multiple ADS trees, state recognition

differs from the conventional definition. We explain the state recognition in the context of multiple ADS trees in this chapter.

Let $P_{M,\omega}$ be the path that is produced by the application of checking sequence ω on s_1 of M where nodes are labeled as m_i's. Let $P_{N,\omega}$ be the path that is produced by the application of ω on N after the initialization of N as explained above where nodes are labeled as n_i's. For any $m_i \in P_{M,\omega}$, we know the corresponding state s_j of M by tracing the application of ω on M starting from s_1. For N to be a correct implementation of M, for any n_i, we should be able to understand that n_i corresponds to the same state as m_i. Therefore, we already have an idea for each n_i to which state it should turn out to correspond in the end. However, we need to derive sufficient evidence for the state corresponding to n_i's based on the response of n_i to the part of the checking sequence applied to it. The evidence we gather along the application of ω is defined as the *state recognition in the context of multiple ADS tree* and the rules we use for state recognition are given below. As stated before, there are similar state recognition rules considering the case where only one ADS tree is used in the literature. In our work, we use multiple ADS trees and extend the definition of state recognition that uses multiple ADS trees. The extended definition of state recognition using multiple ADS trees is recursive. We first give an intuitive explanation to provide a better understanding.

3.1 Cross Verification

Using multiple ADS trees has a particular disadvantage, which we call *Cross Verification*. In order to explain the problem, let us suppose that $\mathcal{A}_i^j$ and $\mathcal{A}_i^k$ are two ADSs for a state s_i, and they are applied to the implementation at nodes n and n', and the expected outputs are observed. When one considers the application of $\mathcal{A}_i^j$ and $\mathcal{A}_i^k$ independently, both n and n' are recognized as the state s_i. However, we cannot directly infer from the application of $\mathcal{A}_i^j$ and $\mathcal{A}_i^k$ that n and n' are actually the same implementation states. A faulty implementation may have two different states, and we might be applying $\mathcal{A}_i^j$ and $\mathcal{A}_i^k$ at those states. Therefore, one needs to make sure that n and n' are actually the same implementation states as well. This requires some additional information to be extracted based on the observations from the implementation.

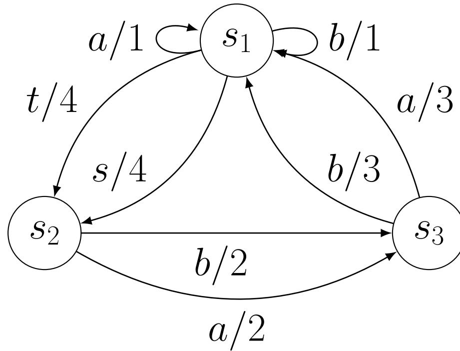


Figure 3.1: The FSM M_1 with two ADS trees: a and b.

To explain the need for cross verification, suppose that we are given the FSM M_1 in Figure 3.1. We can split the original FSM M_1 into two subgraphs such that each subgraph has all the states of the original FSM and a subset of the edges. The union of the subgraph is the original graph.

We split the M_1 into two subgraphs as shown in Figure 3.2 and Figure 3.3. Then we generate checking sequences for each subgraph, using a different ADS tree each time. We use two simple ADS trees a and b for subgraphs shown in Figure 3.2 and Figure 3.3, respectively. Then, we generate the checking sequences for each graph as $CS_1 = aasaaa$ and $CS_2 = bbtbbb$. Since both sequences start and end in state 1, we can simply concatenate them to attempt to create a checking sequence for original FSM M_1, e.g. $CS_3 = aasaaabbtbbb$. Unfortunately, the resulting sequence is not a checking sequence: the FSM shown in Figure 3.5 produces the same output sequence as the response to CS_3 with the FSM of Figure 3.1, although it is not isomorphic to the FSM shown in Figure 3.1.

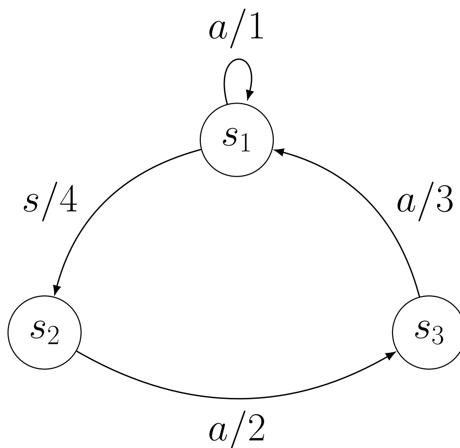


Figure 3.2: A subgraph of the FSM M_0: aasaaa is a CS

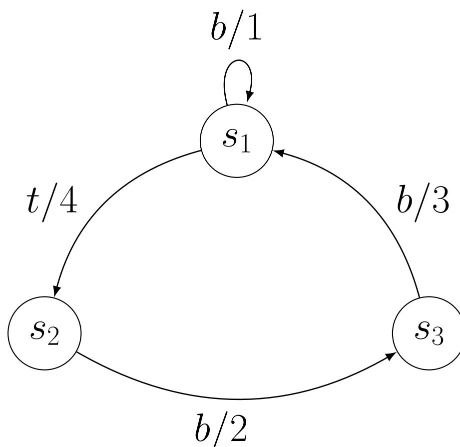


Figure 3.3: A subgraph of the FSM M_0: bbtbbb is a CS

The problem is that although each subgraph is independently correctly verified by its own checking sequence, the states that are identified in each subgraph do not correspond to each other (in some sense, states s_2 and s_3 are swapped between the two subgraphs in this example). What we need to do, in addition to the above, is to force the fact that the node recognized by each application of the ADS in different subgraphs correspond to one another. One simple solution is to create a spanning tree on top of the original graph, and add the recognition of the spanning tree in each of the subgraph, This way, we know that the nodes in different subgraphs correspond to the same implementation states as well.

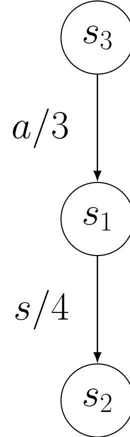


Figure 3.4: Spanning tree of the FSM M_1

For example, if we add the spanning tree shown in Figure 3.4, the checking sequence for subgraph in Figure 3.2 doesn't change since the tree is included in it, while the checking sequence for the second subgraph in Figure 3.3 becomes $CS_2 = bbtbbbsbab$, and the combined checking sequence is $aasaaabbtbbbsbab$, which does not produce the expected output sequence on the FSM of Figure 3.5.

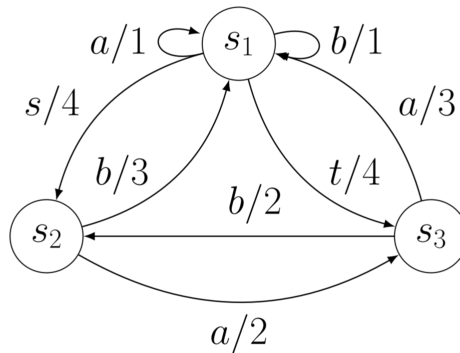


Figure 3.5: This FSM is not isomorphic to the M_1 but produces the same output response to aasaaabbtbbb

In our algorithm, we overcome this problem by differentiating between the concepts of "d-recognition" and "d-recognition by an ADS $\mathcal{A}_i^j$". We declare a node d-recognized if it is d-recognized by $\mathcal{A}_i^j$ for all j's. This requirement forces an observation of the application of each ADS $\mathcal{A}_i^j$ on the same implementation state. Such a set of observations provides information that the states recognized by different ADSs are the same implementation states. Therefore, we cross verify the node by all ADSs.

3.2 Extended State Recognition Definition

Consider a checking sequence ω and ADS tree A^j and the path $P_{M,\omega}$. The checking sequence generation methods that use single ADS tree in the literature guarantee that there is a subpath $(n_p, n_q/\mathcal{A}_i^j; \lambda(s_i, \mathcal{A}_i^j))$ for every state s_i of M where n_p corresponds to s_i. In other words, for an ADS tree $\mathcal{A}^j$, a node n_p in $P_{N,\omega}$ is recognized as the state s_i if $\mathcal{A}_i^j$ is applied at n_p and the response is $\lambda(s_i, \mathcal{A}_i^j)$. This provides an evidence of the existence of a state in N that is similar to s_i, which is actually the state of N corresponding to the node n_i. If we have such an evidence for every state s_i in $P_{N,\omega}$, then we call $\mathcal{A}^j$ as a *legal ADS tree* for N.

In multiple ADS tree case, we also need to check whether all ADSs $\mathcal{A}_i^j$ of ADS trees $\mathcal{A}^j$ are applied to the nodes corresponding to the state s_i. All ADS trees $\mathcal{A}^j$ has to be a legal ADS tree for a sequence to be a checking sequence. However, applying each ADS $\mathcal{A}_i^j$ to the node n_p corresponding to the state s_i is costly. Therefore, we gather information through the nodes that are already recognized by an ADS that belongs to a legal ADS tree as the same implementation states. In other words in $P_{N,\omega}$, we can combine the information belonging to the different nodes that are d-recognized as same implementation states. Acquiring this indirect information relies on the nodes that are d-recognized by an ADS belongs to a legal ADS tree. That's why we call this notion recursive.

Since the legality of an ADS tree is a recursive notion, we explain it inductively. Therefore the base case for the notion of a legal ADS tree is following:

An ADS tree $\mathcal{A}^j$ is called a *legal ADS tree* if for all $\mathcal{A}_i^j \in \mathcal{A}^j$, $\mathcal{A}_i^j$ is observed as a subpath $(n, \tilde{n}; \mathcal{A}_i^j/\lambda(s_i, \mathcal{A}_i^j))$ on $P_{N,\omega}$ where node n is assumed to be the state s_i.

Inductive definition of a legal ADS tree will be done after the definition of valid observation.

In literature, the methods use a single ADS tree to generate a checking sequence do not consider the notion of the legal ADS tree, since ADS tree is legal by nature of checking sequence. Therefore, definition of d-recognition does not consider the legality of an ADS tree. But when we use multiple ADS trees, a node n is *d-recognized by* $\mathcal{A}_i^j$ if there is a subpath $(n, \tilde{n}; \mathcal{A}_i^j/\lambda(s_i, \mathcal{A}_i^j))$ of path $P_{N,\omega}$ where $\mathcal{A}^j$ is a legal ADS tree.

Intuitively, two nodes are *i-equivalent* if they correspond to same implementation

state. Formally, two nodes n_p and n_q are *i-equivalent* if and only if they are both d-recognized by $\mathcal{A}_i^j$ where $\mathcal{A}^j$ is a legal ADS tree. Note that nodes are i-equivalent to themselves. When we identify two different nodes to be i-equivalent, this information can provide some additional evidence which do not exist in the linear view of the path $P_{N,\omega}$. For example, consider the path given in Figure 3.6 , where the node n_i is recognized as state s_1. If the nodes n_2 and n_5 are understood to be the same implementation state (i.e. i-equivalent), then we also obtain an additional observation for n_1 which applying aa of n_1 would produce 12 as a response.

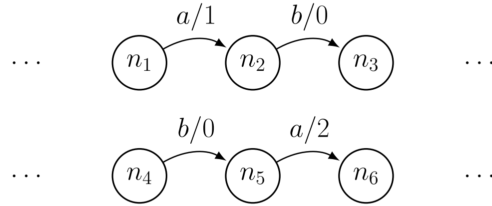


Figure 3.6: Two subpaths of $P_{N,\omega}$

It is stated before that we can also gather additional evidence regarding ADSs through the nodes that are recognized as the same implementation state. That's why i-equivalence changes the definition of the legal ADS tree as a result of the recursion. To explain this, we first have to define *valid observation of* $\mathcal{A}_i^j$ *for* s_i.

Definition 1 *There exists subpaths:*

- $(n_{p_0}, n_{p_1}; \alpha_1/\beta_1)$
- $(n'_{p_1}, n_{p_2}; \alpha_2/\beta_2)$

  ...

  ...
- $(n'_{p_k}, n_{p_{k+1}}; \alpha_{k+1}/\beta_{k+1})$

Such that n_{p_l} *and* n'_{p_l} *are i-equivalent for* $1 \leq l \leq k$*, then there is a* valid observation of α for s *from* n_{p_0}*, where* $\alpha = \alpha_1, \ldots, \alpha_{k+1}$ *and* s *is the state corresponding node* n_{p_0}.

Valid observation definition let us define the notion of a legal ADS tree and d-recognition since, we no longer need to have an evidence of an application of $\mathcal{A}_i^j$ as

a subpath, but we can obtain such an evidence as a valid observation. Now we can make inductive definition for a legal ADS tree. We call an ADS tree $\mathcal{A}^j$ a legal ADS tree for N, if $\forall s_i \in S, \exists$ a node $n \in P_{N,\omega}$ such that there is a valid observation of $\mathcal{A}_i^j$ from n. Obviously, a node n is d-recognized by $\mathcal{A}_i^j$ if there is a valid observation of $\mathcal{A}_i^j$ from n. Also the node n is *d-recognized* as s_i if it is d-recognized by $\mathcal{A}_i^j$ for all j as the state s_i.

Definition 2 *We now formally define this mutually recursive notion as follows:*

- d-recognition by $\mathcal{A}_i^j$*: A node n_p is* d-recognized by $\mathcal{A}_i^j$ *as a state s_i of specification if*
  - *There is a valid observation of $\mathcal{A}_i^j$ from n_p.*
  - *There exist a node n_q which is d-recognized by $\mathcal{A}_i^j$ as s_i and nodes n_p and n_q are i-equivalent.*
- i-equivalence*: Two nodes n_p and n_q (not necessarily distinct nodes) are recognized as equivalent implementation states (shortly* i-equivalent*) if*
  - *For some $\mathcal{A}_i^j$, both n_p and n_q are d-recognized by $\mathcal{A}_i^j$ as s_i.*
  - *There exist nodes n'_p and n'_q that are i-equivalent and we have valid observation of α both from n'_p and n'_q that ends with n_p and n_q, respectively.*
- d-recognition*: A node n of $P_{N,\omega}$ is said to be $d-recognized$ as state s_i of specification M if for all j, node n is d-recognized by $\mathcal{A}_i^j$ as a state s_i of M.*

For transition verification, a transition $t = (s, s'; x/y)$ of specification M is verified if there is a subpath $(n_i, n_{i+1}; x/y)$ of $P_{N,\omega}$ and nodes n_i and n_{i+1} are d-recognized as states s and s' of M.

As [31] suggested ω is a checking sequence if every transition of M is verified.

3.3 Checking Sequences: Sufficient Condition

This section gives a sufficient condition for a sequence to be a checking sequence. This result is a consequence of Theorem 1. Normally, the main condition for a sequence to be a checking sequence is to verify every transition of M [31]. In

this section we will introduce the notion of *Recognition Automaton* and redefine the sufficient condition for a sequence to be a checking sequence in the context of Recognition Automaton.

Recognition Automaton is a graph $R = (V_R, E_R)$ such that each $v_i \in V_R$ corresponds to the partitioning of the nodes of $P_{N,\omega}$ as $\prod = \pi_1, \pi_2, \ldots, \pi_k$ where π_i is the set of all nodes in $P_{N,\omega}$ that are i-equivalent to each other. $R = (V_R, E_R)$ is defined as, $\exists v_i$ for each $\pi_i \in \prod$ and $(v_i, v_j) \in E_R$ if and only if $\exists n_l \in \pi_i$ and $n_{l+1} \in \pi_j$ such that $(n_l, n_{l+1}; -/-)$ is a subpath of $P_{N,\omega}$.

Definition 3 *A node v_i of R is d-recognized if $\exists n \in \pi_i$ such that n is d-recognized.*

Since π_i is set of nodes that are i-equivalent to each other, once at least one of them is d-recognized, then $\forall v \in \pi_i$ are d-recognized by definition.

Definition 4 *If $\mathcal{A}_i^j$ can be traced on R starting from node v_i, then v_i is d-recognized by $\mathcal{A}_i^j$ as the state s_i.*

Since the nodes in π_i are i-equivalent to each other, any path traced on R is a valid observation for the nodes in π_i. Thus, any path $(v_q, v_p; \mathcal{A}_i^j/\lambda(s_i, \mathcal{A}_i^j))$ on R, is a valid observation starting from the node v_q and v_q is d-recognized by $\mathcal{A}_i^j$ as the state s_i.

Lemma 2 *If R is isomorphic to M, then all nodes of R are d-recognized.*

Proof. *Consider a node $v_i \in R$ and an ADS $\mathcal{A}_i^j$. Since R is isomorphic to M, $\mathcal{A}_i^j$ for all i and j can be traced on R and the evidence of all the nodes in π_i producing the expected output to $\mathcal{A}_i^j$ can be obtained and all nodes of R are d-recognized.*

■

Lemma 3 *If R is isomorphic to M, then ω is a checking sequence.*

Proof. *If R is isomorphic to M, then all nodes of R are d-recognized by Lemma 1. By isomorphism between R and M, all transitions of M exists in R. Therefore, all transitions of M are verified. Using Theorem 1, ω is a checking sequence.* ■

3.4 Generation of Recognition Automaton

As stated earlier, recognition automaton R is a representation of i-equivalence between nodes of $P_{N,\omega}$ since each $v_i \in V_R$ corresponds to the partitioning of the nodes of $P_{N,\omega}$ as $\prod = \pi_1, \pi_2, \ldots, \pi_k$ where π_i is the set of all nodes in $P_{N,\omega}$ that are i-equivalent to each other. If i-equivalence between nodes are ignored, then recognition automaton R is simply a path $P_{N,\omega}$ where nodes n_i represents states visited in N when ω is applied.

If we can find a one to one correspondence between the nodes of $P_{N,\omega}$ and $P_{M,\omega}$, and observe that every transition of $P_{N,\omega}$ is verified then we can say that ω is a checking sequence of M. We consider $P_{N,\omega}$ as a graph R to find this correspondence between nodes of $P_{N,\omega}$ and $P_{M,\omega}$ and call this graph R as the initial recognition automaton. It is called that way, since initially we just assume that nodes n_i of R corresponds to the states that should be visited along the application of ω on M starting from state s_1 but we are not sure about this assumption and try to gather recognition information to recognize nodes n_i correctly. While we process the recognition automaton, we reduce the number of nodes in R, as we merge the nodes that correspond to the same implementation states. We call reduction of nodes in R as *collapse* of R. Therefore, if R eventually collapses to M where R is initially a path $P_{N,\omega}$ then ω is a checking sequence by Lemma 3.

Formally, given a I/O sequence ω/y we consider a path $P_{N,\omega}$. Then we represent $P_{N,\omega}$ as a graph. We call this graph recognition automaton and represent it as $R = (V_R, E_R)$ where initially $V_r = \{n_1, n_2, \ldots, n_{k+1}\}$ and $E_R = \{(n_i, n_{i+1}; x/y) | (n_i, n_{i+1}; x/y) \in P_{N,\omega}\}$ and $|\omega| = k$.

For example, consider the I/O sequence $\omega/y = aabbbcccaacbacbbb/001001001010101 10$ and $P_{N,\omega}$ given in Figure 3.7. Since $P_{N,\omega}$ represents the nodes visited and outputs gathered by the application of the input sequence ω on the implementation N starting from the node that is assumed to correspond to s_1, $P_{N,\omega}$ can be considered as initial the recognition automaton R where we did not observe the recognition of nodes yet.

The main aim is to recognize each node in the recognition automaton R so that if R becomes isomorphic to M, we can conclude that ω is a checking sequence. Beginning with the initial R, we try to find the correspondence between the nodes of $P_{M,\omega}$ and

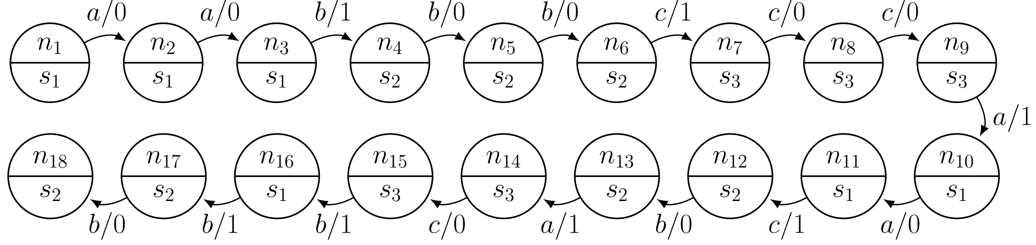


Figure 3.7: A Path $P_{N,\omega}$

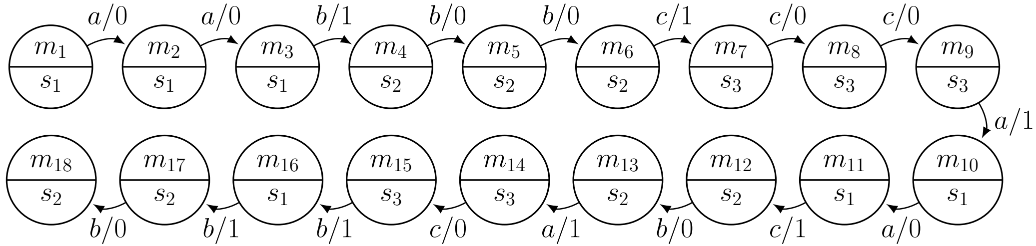


Figure 3.8: A Path $P_{M,\omega}$

$P_{N,\omega}$ shown in Figure 3.7 and Figure 3.8. We assume that the node n_1 corresponds to the state s_1 of M and all n_i's are assumed to be the states that should be visited if we apply the same input sequence to the specification FSM M. These respective states of n_i's are shown in the lower parts of the nodes. By recognizing the nodes, we verify the assumption that the node corresponds to the state shown in the lower part of the node. This recognition process is explained in the next section.

3.5 State Recognition on Recognition Automaton

Given an I/O sequence ω/y, considering the path $P_{N,\omega}$ we form the initial recognition automaton R as explained above. Once the recognition automaton R is generated without considering the state recognitions the partitioning of the nodes of R is $\prod = \pi_1, \pi_2, \ldots, \pi_{k+1}$ where $k = |\omega|$ and $\prod_i = \pi_i$ because $\forall n_i \in P_{N,\omega}$, n_i is i-equivalent only to itself. In other words, $\pi_i = \{n_i\}$. So that, the initial recognition automaton R is the same as $P_{N,\omega}$.

In Figure 3.7 and Figure 3.8, we see that both implementation N and specification M give the same output sequence in response to ω. But to conclude that N is a correct implementation of M, we need to show ω is a checking sequence. We know that a sufficient condition for ω to be a checking sequence is that R can be reduced

into a form that is isomorphic to M.

We collapse R into M by considering state recognitions. To do that, whenever an evidence regarding to a node correspondence between $P_{N,\omega}$ and $P_{M,\omega}$ is found, partitioning $\prod$ is updated. One way of recognizing a node is to look for an occurrence of a valid observation corresponding to the application of an ADS $\mathcal{A}_i^j$ in R. That is if R has a subpath $(n, n'; \mathcal{A}_i^j, \lambda(s_i, \mathcal{A}_i^j))$ then the node n cannot be any state other than s_i when $\mathcal{A}^j$ is a legal ADS tree. Therefore, such nodes can easily be recognized as the corresponding states and the recognition automaton R can be collapsed by merging the nodes that are recognized as the same states.

Since state recognition using multiple ADS trees is a mutually recursive notion, we need to consider intuitive definitions of this notion first. Intuitively, to recognize a node n_p in R as the state s_i, we need to observe the output of $\lambda(s_i, \mathcal{A}_i^j)$ as a response to $\mathcal{A}_i^j$ starting from node n_p, where $\mathcal{A}^j$ is a legal ADS tree. Therefore, the first task to start state recognition within R is to find which ADS trees that are legal.

Since we check whether N is a correct implementation of M or not, we need to consider the ADS trees that belong to M shown in Figures 2.3, 2.4 and 2.5. First thing is to find which ADS trees are legal. It can be done by simply traversing R and by observing whether there is a valid observation of all the ADSs belong to an ADS tree generating correct responses according to the Table 3.1. The output responses of states s_i's to the $\mathcal{A}_i^j$'s of $\mathcal{A}^j$ shown in Table 3.1, where $\mathcal{A}^j$'s are ADS trees shown in Figures 2.3, 2.4 and 2.5.

		$\mathcal{A}_i^j$	s_1	s_2	s_3
$\mathcal{A}^1$	$\mathcal{A}_1^1$	a	0	1	1
	$\mathcal{A}_2^1$	aa	00	11	10
	$\mathcal{A}_3^1$	aa	00	11	10
$\mathcal{A}^2$	$\mathcal{A}_1^2$	bb	10	00	11
	$\mathcal{A}_2^2$	b	1	0	1
	$\mathcal{A}_3^2$	bb	10	00	11
$\mathcal{A}^3$	$\mathcal{A}_1^3$	cc	11	10	00
	$\mathcal{A}_2^3$	cc	11	10	00
	$\mathcal{A}_3^3$	c	1	1	

Table 3.1: ADS Table

Consider the ADS tree $\mathcal{A}^2$. From nodes n_3, n_4 and n_{15} which correspond to states s_1, s_2 and s_3 respectively, we observe the corresponding output responses $\lambda(s_i, \mathcal{A}_i^2)$

as shown in Figure 3.9 labeled as red nodes and edges. Hence $\mathcal{A}^2$ is a legal ADS tree and by using it, valid observations and recognitions based on $\mathcal{A}^2$ can be performed and i-equivalence between nodes can be stated.

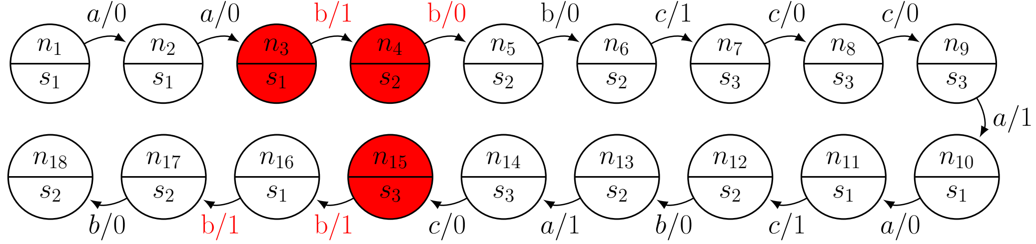


Figure 3.9: Showing ADS tree $\mathcal{A}^2$ is legal

We show the valid observations based on $\mathcal{A}^2$ in Figure 3.10 with blue nodes and edges. According to these valid observations following evidences are gathered:

- Nodes n_3 and n_{16} are d-recognized by $\mathcal{A}^2_1$ as state s_1.
- Nodes n_4, n_5, n_{12} and n_{17} are d-recognized by $\mathcal{A}^2_2$ as state s_2.
- Node n_{15} is d-recognized by $\mathcal{A}^2_3$ as state s_3.

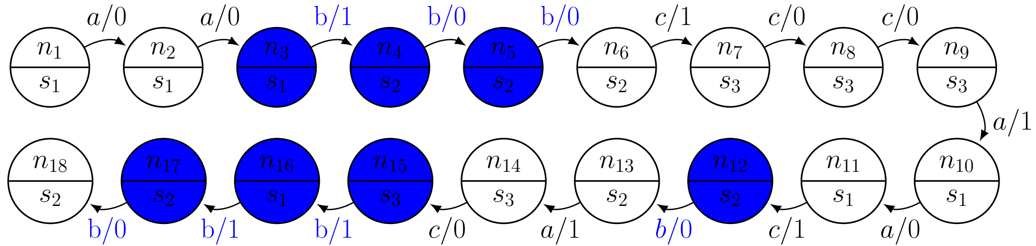


Figure 3.10: Valid observations based on $\mathcal{A}^2$

We know that the nodes d-recognized by the same ADS as the same specification state are i-equivalent. Since each $v_i \in V_R$ corresponds to the partitioning of nodes $\prod = \pi_1, \pi_2, \ldots, \pi_k$ where π_i is the set of all nodes in $P_{N,\omega}$ that are i-equivalent to each other, we can treat i-equivalent nodes as a single node of R. We call this operation "merging nodes" and it is defined in following section.

3.6 Merging Nodes on Recognition Automaton

In the previous section we encounter nodes that are i-equivalent. For example, nodes n_3 and n_{16} are i-equivalent since they are both d-recognized by $\mathcal{A}_1^2$ where $\mathcal{A}^2$ is a legal ADS. Therefore, we know that they belong to the same $\pi_i \in R$. For representation purposes, we show sets π_i's as a single node in R and call this operation merging.

Whenever we understand that two nodes n and n' of R are i-equivalent, we merge those nodes in R by the following rules:

Rule 1 Updating the start node of each edge emanating from node n' as n

Rule 2 Updating the end node of each edge ending at node n' as n

Rule 3 Eliminate the edges emanating from node n with same input label and merge the ending nodes of corresponding edges.

Rule 4 If node n' is *d-recognized by* some $\mathcal{A}_i^j$ where node n is not *d-recognized by* $\mathcal{A}_i^j$, then we treat node n as it is *d-recognized by* $\mathcal{A}_i^j$.

Consider the first two rules of merging operation applied on nodes n_3 and n_{16} of R. Then R will be evaluated to the graph shown in Figure 3.11. Note that we update the starting node of the edges emanating from n_{16} as n_3. Also, we updated the ending node of the edges ending at n_{16} as n_3. Now, the node n_3 has emanating edges with the same input label b. Using Rule 3, we eliminate the emanating edges that have the same input label by merging their ending nodes n_4 and n_{17}. Merging the nodes n_4 and n_{17} leads to merging operation on nodes n_5 and n_{18} too. As a result of consecutive merging operations, R becomes as shown in Figure 3.12.

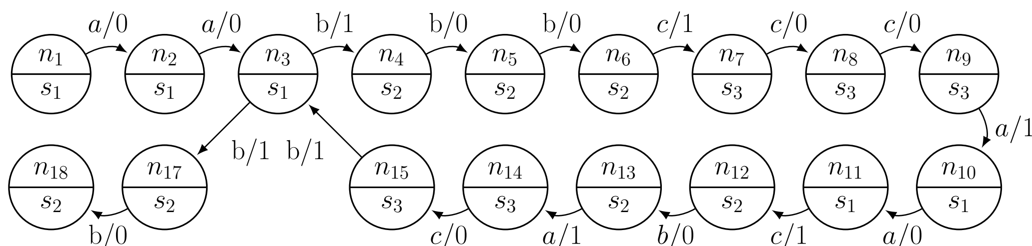


Figure 3.11: An Application of Rule 1 and 2 on R

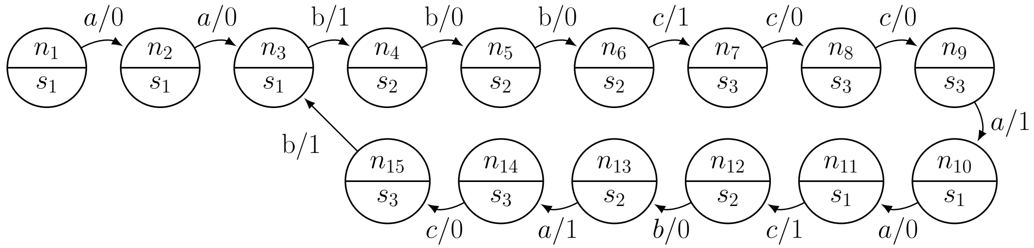


Figure 3.12: An Application of Rule 3 and 4 on R

Now we will proceed to merge nodes n_4 and n_5 since they are i-equivalent. When we eliminate n_5 and update the ending node of the edge emanating from n_4 with input label b as n_4 and update the start node of the edge emanating from n_5 as n_4, node n_4 will have two edges emanating from n_4 with input label b. Therefore, we understand that n_6 is also equivalent to n_4, since they are both ending nodes of the edges emanating from n_4 with input label b. The Figure 3.13 shows R when we merge n_4, n_5 and n_6.

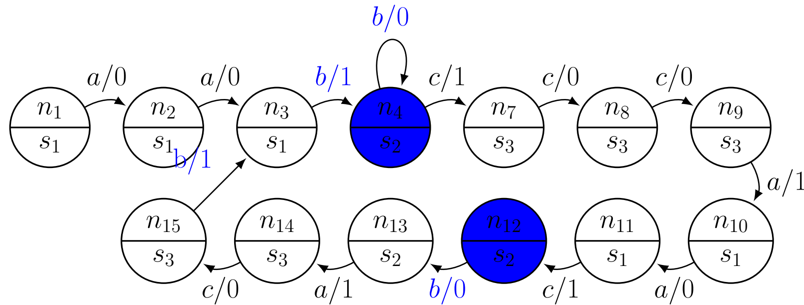


Figure 3.13: Merging nodes n_4, n_5 and n_6 on R

In addition, we know that node n_4 and n_{12} should be merged since they are also i-equivalent. We should update the edges emanating from and ending at node n_{12} shown in Figure 3.13 as blue edges. After this update, the node n_4 has two different edges emanating from itself with the same input label b as shown in Figure 3.14 which also has to be handled by merging two ending nodes n_4 and n_{13} of those edges. After completing every merging operation that can be done based on the i-equivalent nodes stated before, the resulting recognition automaton R shown in Figure 3.15.

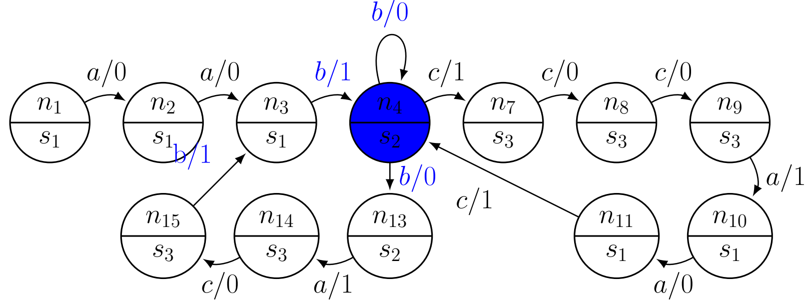


Figure 3.14: Merging nodes n_4 and n_{12} on R

Note that in Figure 3.15, the node n_4 is d-recognized by an ADS $\mathcal{A}_2^2$. Blue edges show the valid observations based on the ADS $\mathcal{A}^2$.

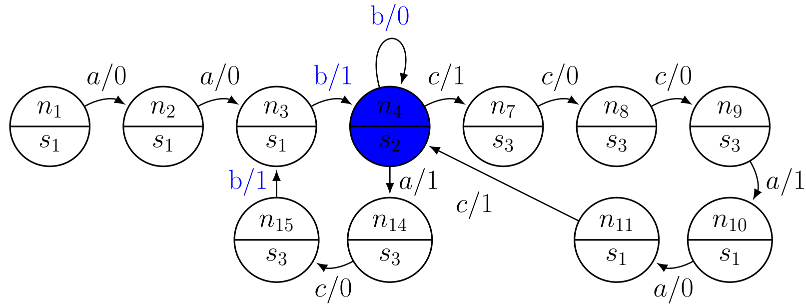


Figure 3.15: Merging nodes n_4 and n_{13} on R

Now the next task is to find out if we can observe evidences on R that make any other ADS tree legal. Consider the ADS tree $\mathcal{A}^3$ and blue edges on Figure 3.16. Note that we observe $\mathcal{A}_1^3$ starting from node n_{11} which is assumed to be state s_1, $\mathcal{A}_2^3$ starting from node n_4 which is assumed to be state s_2 and $\mathcal{A}_3^3$ starting from node n_7 which is assumed to be state s_3. Therefore, ADS tree $\mathcal{A}^3$ is now legal and can be used to find valid observations and state recognitions.

We show the valid observations based on $\mathcal{A}^3$ in Figure 3.17 with blue nodes and edges. According to these valid observations following evidences are gathered:

- Node n_{11} is d-recognized by $\mathcal{A}_1^3$ as state s_1.
- Node n_4 is d-recognized by $\mathcal{A}_2^3$ as state s_2.
- Nodes n_7, n_8 and n_{14} are d-recognized by $\mathcal{A}_3^3$ as state s_3.

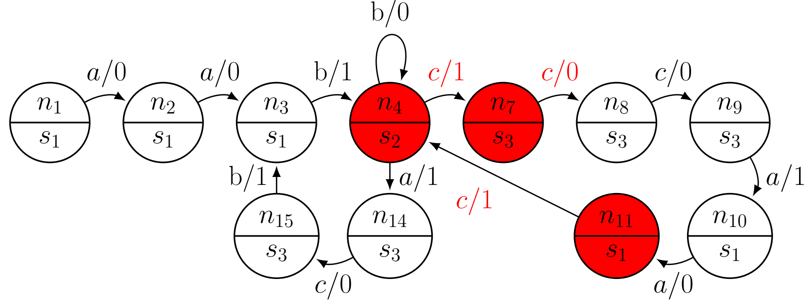


Figure 3.16: Showing ADS tree $\mathcal{A}^3$ is legal

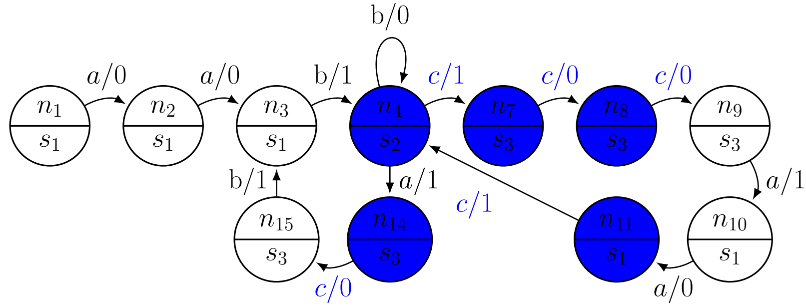


Figure 3.17: Valid observations on R based on the ADS Tree $\mathcal{A}^3$

We can now merge nodes n_7, n_8 and n_{14} since they are i-equivalent to each other. When we merge them, R becomes as shown in Figure 3.18. Note that node n_7 has three edges emanating from itself with input label c as shown in Figure 3.18, so that we have to merge the ending nodes n_7, n_9 and n_{15} of those edges. Figure 3.19 shows the final R, when we finish the merging operations based on $\mathcal{A}^3$. Now we should find out if $\mathcal{A}^1$ is legal or not. Figure 3.19 shows the evidences of $\mathcal{A}^1$ being legal. Note that we observe $\mathcal{A}^1_1$ starting from node n_{10} which is assumed to be state s_1, $\mathcal{A}^1_2$ starting from node n_4 which is assumed to be state s_2 and $\mathcal{A}^1_3$ starting from node n_7 which is assumed to be state s_3. Therefore, ADS tree $\mathcal{A}^1$ is legal and can be used to find valid observations and state recognitions.

By using $\mathcal{A}^1$, following d-recognitions can be done:

- Nodes n_1, n_2 and n_{10} is d-recognized by $\mathcal{A}^1_1$ as state s_1.
- Node n_4 is d-recognized by $\mathcal{A}^1_2$ as state s_2.
- Node n_7 is d-recognized by $\mathcal{A}^1_3$ as state s_3.

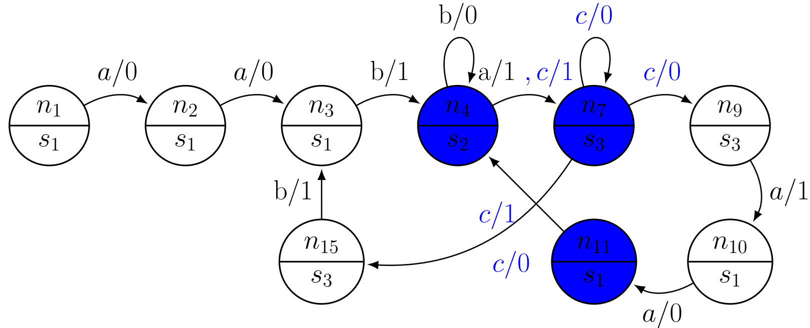


Figure 3.18: Merging nodes n_7, n_8 and n_{14} on R

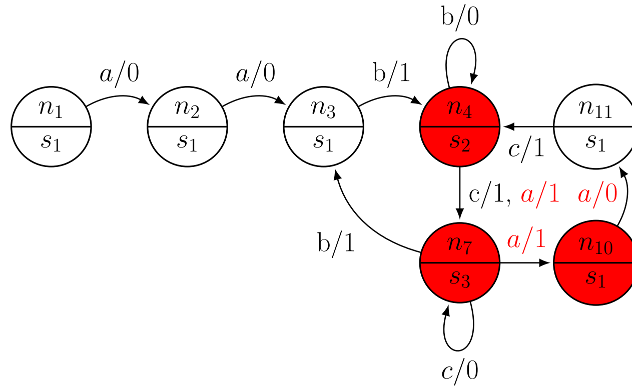


Figure 3.19: Showing ADS Tree $\mathcal{A}^1$ is legal

Therefore, we should merge the nodes n_1, n_2 and n_{10}. When we merge them, there will be three different edges that emanate from the same node n_{10} with the same input label a, so that the ending nodes of these edges should also be merged which correspond to n_{10}, n_3 and n_{11}. As a result, the final form of R will be equal to M as shown in Figure 3.21 which proves that ω is a checking sequence since R becomes isomorphic to M.

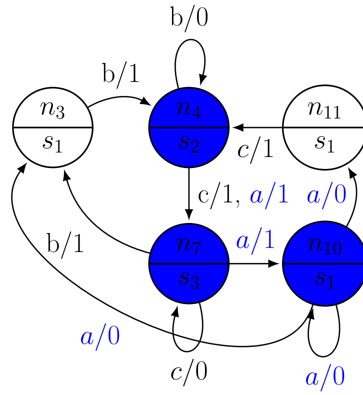


Figure 3.20: Merging nodes n_1, n_2 and n_{10} on R

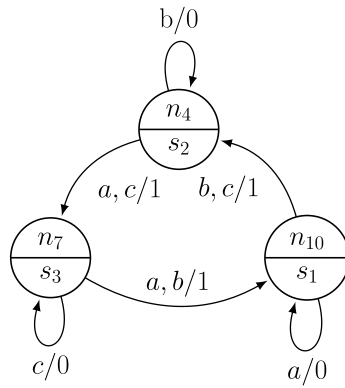


Figure 3.21: A Collapsed Recognition Automaton R

Chapter 4

Checking Sequence Generation Algorithm

In this chapter, a method to generate a checking sequence using multiple ADS trees will be presented. This method generates a checking sequence by extending the current sequence at each iteration similar to the methods given in [2, 28]. However the proposed method differs from the methods in [2, 28] since it considers multiple ADS trees. In addition, our method consists of two phases like the method in [2]. In the first phase an input sequence ω is generated but ω is not guaranteed to be a checking sequence. If it is not a checking sequence, then the method moves on to the second phase and performs a post-processing. In the post-processing phase, ω is further extended until it becomes a checking sequence. Our method uses the concept of recognition automaton and constructs a recognition automaton at each iteration while generating the checking sequence. Since the sufficient condition for a sequence to be a checking sequence is to collapse recognition automaton into the specification machine M as told in Chapter 3, the main task the method tries to accomplish is extending the sequence in a way it collapses the recognition automaton to the specification M. Therefore, the method will be explained through the concept of recognition automaton.

In the following sections, the two phases of the method are explained in detail. The reason why the sequence generated in the first phase may not be a checking sequence is presented. Lastly, we reveal how we extend ω so that it becomes a checking sequence in the post-processing phase.

4.1 Mutual Dependency Between ADS Trees

We stated that the input sequence ω generated in the first phase of the algorithm is not guaranteed to be a checking sequence. The reason for that is we do not apply the rules of state recognition as presented in Chapter 3. We ignore the legal ADS concept to provide local optimization. Therefore, the application of the state recognition definition differs in the algorithm.

For a sequence $\tilde{\omega}$ to be a checking sequence, R which is equal to the path $P_{N,\tilde{\omega}}$ initially, should collapse into the specification M. To collapse R into the M, we should merge the i-equivalent nodes. Therefore, d-recognition of the nodes is a must. For a node to be d-recognized by $\mathcal{A}_i^j$ as the state s_i, the ADS tree $\mathcal{A}^j$ has to be a legal ADS tree. Therefore, waiting for an ADS tree to become a legal one for state recognitions is a costly action. That's why we decide to ignore the legal ADS tree concept. In other words, the node is d-recognized by $\mathcal{A}_i^j$ as the state s_i even though the ADS tree $\mathcal{A}^j$ is not legal. We call this kind of recognition *Conditional State Recognition.* Conditional state recognition does not require a legal ADS tree. Therefore, just an observation of a subpath of recognition automaton R with label $\mathcal{A}_i^j$ is enough. However, conditional state recognition becomes an actual state recognition when the ADS tree the recognition is done with becomes a legal one.

Definition 5 *According to this, we define conditional state recognition as follows:*

- conditional d-recognition by $\mathcal{A}_i^j$*: A node* n_p *is* d-recognized conditionally by $\mathcal{A}_i^j$ *as a state* s_i *of specification if*
  - *There is a subpath* $(n_p, n_q; \mathcal{A}_i^j/\lambda(n_p, \mathcal{A}_i^j))$ *of* R*, or*
  - *There exist a node* n_q *which is d-recognized conditionally by* $\mathcal{A}_i^j$ *as* s_i *and nodes* n_p *and* n_q *have i-equivalence candidacy.*
- conditional i-equivalence*: Two nodes* n_p *and* n_q *(not necessarily distinct nodes)* are i-equivalent conditionally *and they are recognized as implementation states conditionally if*
  - *For some* $\mathcal{A}_i^j$*, both* n_p *and* n_q *are d-recognized conditionally by* $\mathcal{A}_i^j$ *as* s_i*.*

– *There exist nodes n'_p and n'_q that are i-equivalent conditionally and there exist two subpaths $(n'_p, n_p; \alpha/\lambda(n'_p, \alpha))$ and $(n'_q, n_q; \alpha/\lambda(n'_q, \alpha))$ of R.*

- conditional d-recognition*: A node n of R is* d-recognized conditionally *as the state s_i of specification M if for all j, node n is d-recognized conditionally by $\mathcal{A}_i^j$ as a state s_i of M.*

Checking sequence generation methods existing in the literature use a single ADS tree. When a checking sequence ω is constructed by using a single ADS tree, then it is guaranteed that the ADS tree is a legal ADS tree since all ADSs belong to it will be applied to its respective nodes at least one time explicitly. But in our case, we use multiple ADS trees, and the application of all ADSs belonging to ADS trees is a very costly action. Hence we try to avoid explicit application of all ADSs at all states. To this end, we gather state recognition information using valid observations obtained through the use of i-equivalent states. In this way we can reduce the length of a checking sequence. So that while preventing the costly part of using multiple ADS trees, we can take the advantage of the ADSs with relatively short lengths for transition verification.

As mentioned above we try to use i-equivalent nodes to gather information about other ADSs that are not applied explicitly. To explain this idea, suppose that there are two subpaths $(n_p, n_q; \mathcal{A}_i^j/\lambda(s_i, \mathcal{A}_i^j))$ and $(n_r, n_s; \alpha/\lambda(s_k, \alpha))$ of $P_{N,\omega}$ as in Figure 4.1. Also, suppose that n_p and n_s are i-equivalent, n_r is claimed to be state s_k, and $\alpha\mathcal{A}_i^j$ equals to $\mathcal{A}_k^l$. Therefore, we can say that $\mathcal{A}_k^l$ is applied on node n_r. In this way, we obtain a valid observation for the application of $\mathcal{A}_k^l$ at node n_r, even though there is no such subpath of $P_{N,\omega}$ corresponding to an explicit application.

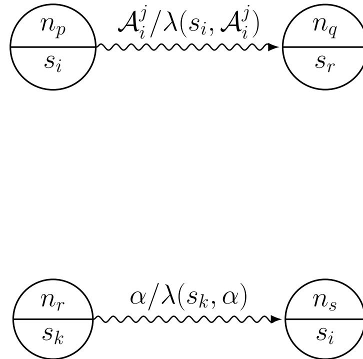


Figure 4.1: Two subpaths of $P_{N,\omega}$

In our algorithm we change the definition of recognitions, therefore we do not check if ADS tree is legal or not, explicitly. We use partial information about the application of ADSs that is not guaranteed to be correct. This creates a dependency between the ADS trees in terms of legality. To explain this, let us take two subpaths a $(n_p, n_q; \mathcal{A}_i^j/\lambda(s_i, \mathcal{A}_i^j))$ and $(n_r, n_s; \alpha/\lambda(s_k, \alpha))$ of $P_{N,\omega}$ just like in Figure 4.1, and assume that n_p and n_s are both d-recognized by $\mathcal{A}_i^j$ conditionally. Therefore, n_p and n_s are i-equivalent nodes conditionally. Also suppose that ADS tree $\mathcal{A}^j$ is not a legal ADS and the node n_r corresponds to the state s_k and $\alpha\mathcal{A}_i^j$ equals to $\mathcal{A}_k^l$. Since the ADS tree $\mathcal{A}^j$ is not a legal ADS tree, the information that the node n_r is d-recognized by $\mathcal{A}_k^l$ we gather over node n_p is not guaranteed to be correct. Therefore, the node n_r is d-recognized conditionally by $\mathcal{A}_k^l$. The ADS tree $\mathcal{A}^l$ is dependent on the ADS tree $\mathcal{A}^j$ because of the information we gathered about $\mathcal{A}_k^l$ over the nodes that are d-recognized by $\mathcal{A}_i^j$ conditionally. To get rid of this dependency, it is enough to have valid observations that make the ADS tree $\mathcal{A}^j$ legal.

4.2 Phase 1: Sequence Generation

In the first phase of the method, an input sequence ω, which may not be a checking sequence, is constructed iteratively. In this method, the recognition of a node in R is achieved by using the recognition types declared in Section 3.5. Since the sequence is extended iteratively similar to the methods in [12, 2], we have plenty of options about how to extend the sequence. In this section, after presenting the sequence extension options, we will explain the decision mechanisms to make a choice between these options.

Note that current node n_c is the node of R corresponding to the last node of the path $P_{N,\tilde{\omega}}$. The node n_c is updated within the each extension of $\tilde{\omega}$.

4.2.1 Sequence Extension Options

In this section we will present the ways we use to extend the sequence. There are four ways of extending the sequence and they are explained below:

- *State recognition by backtracking*:

As we find i-equivalent states, R collapses into a graph with a smaller number of nodes and the nodes become more connected to each other since edges, with different input labels, of i-equivalent nodes, originate from the same node in the collapsed form of R. Therefore, we know that when we follow the edges in the reverse direction starting from the current node n_c, we can find different paths from nodes of R to n_c. The main purpose of finding these paths is that there is a possibility that these paths could be extended to provide a state recognition.

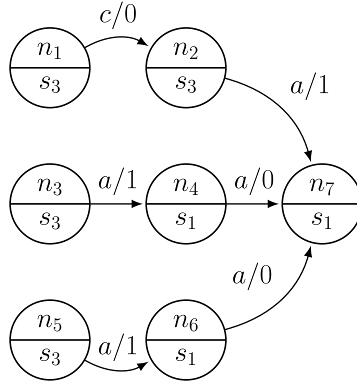


Figure 4.2: Backtracking Example

Consider the Figure 4.2. For backtracking purposes, the tree T in Figure 4.3 is constructed based with a root n_c which is the node n_7. T is constructed by selecting n_c as a root and appending the edges in the reverse direction to the nodes. T is allowed to have a depth of length of the longest ADS we use. Therefore, by doing a breadth-first search on T, we can find any path from n_c to any node which is no longer than the longest ADS.

The purpose of the reverse breadth-first search is to find the node n_p where its d-recognition can be completed by extending the path from n_p to n_c. In other words, the path from the node n_p that is found by a reverse breadth-first search to the current node n_c is a prefix of an ADS $\mathcal{A}_i^j$ where n_p is not d-recognized by $\mathcal{A}_i^j$ as the state s_i yet. Formally, n_c is the last node of the path $P_{N,\tilde{\omega}}$, there is a node n_p and a subpath $(n_p, n_c; \alpha/\lambda(n_p, \alpha))$ in R where node n_p is assumed to be the state s_i and it is not d-recognized by $\mathcal{A}_i^j$. Also α is a prefix of $\mathcal{A}_i^j$. Therefore, the node n_p can be d-recognized by $\mathcal{A}_i^j$ by simply adding β where $\mathcal{A}_i^j = \alpha\beta$.

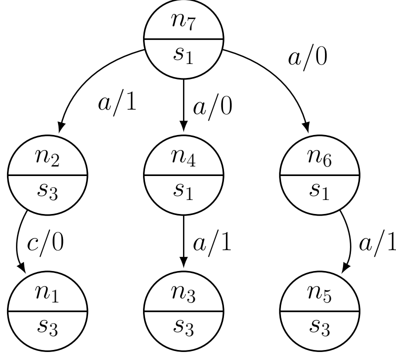


Figure 4.3: The Tree T constructed for backtracking

Below we list the backtracking candidates:

- Extend the path $(n_2, n_7; a/1)$ in accordance with the $\mathcal{A}_2^1 = aa$ by appending an edge with input label a to the node n_7. Therefore, the node n_2 is d-recognized by $\mathcal{A}_3^1$ as state s_3.
- Extend the R in accordance with the $\mathcal{A}_1^1 = a$ by appending edge with input label a to the node n_7. Therefore, the node n_7 is d-recognized by $\mathcal{A}_1^1$ as state s_1.

Note that backtracking includes the appending ADS to the current state for state recognition.

After finding candidates, the algorithm should decide which candidate to use. It makes this choice greedily as follows:

Rule 1 If there is a unique shortest extension sequence, then this shortest sequence is used.

Rule 2 If there are multiple candidates for Rule 1, one of them is chosen randomly.

For this case, we cannot distinguish the candidates. Therefore, we extend the sequence by appending the edge with input label a.

For example consider node n_2 in Figure 4.2, and assume that we try to d-recognize the node n_2 by $\mathcal{A}_3^1$ as state s_3. Since current node is n_c, to d-recognize n_2 by $\mathcal{A}_3^1$ we need to extend the current sequence by $\beta = a$. After extending the sequence by a, we append a new edge $(n_7, n_8; a/0)$ to R as shown

in Figure 4.4. If we assume that $\mathcal{A}^1$ is a legal ADS tree then the node n_2 is now d-recognized by $\mathcal{A}_3^1$.

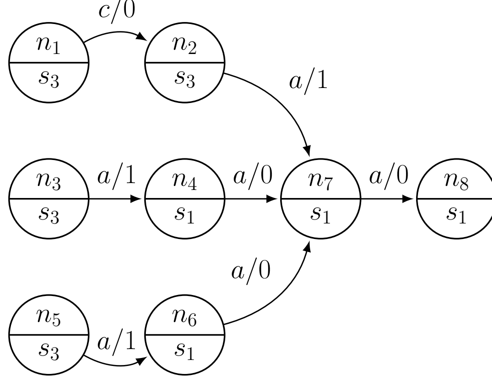


Figure 4.4: Updated Recognition Automaton R after sequence extension

We understand that both nodes n_3 and n_5 are already d-recognized by $\mathcal{A}_3^1$ in Figure 4.2 where we assume $\mathcal{A}^1$ is a legal ADS tree. With this information, we know that nodes n_2, n_3 and n_5 are i-equivalent since they are all d-recognized by $\mathcal{A}_3^1$. Without considering this i-equivalence relation, the recognition automaton R is the graph shown in Figure 4.4. Below we list the i-equivalent nodes of R of Figure 4.4:

- Nodes n_2, n_3, n_5 are i-equivalent since they are all d-recognized by $\mathcal{A}_3^1$.
- Nodes n_4, n_6, n_7 are i-equivalent since they are all d-recognized by $\mathcal{A}_1^1$.

When we consider i-equivalence relation between these nodes and merge them, R becomes the automaton given Figure 4.5.

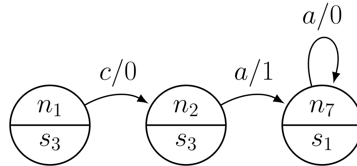


Figure 4.5: Recognition Automaton R after merging operations

- *Extension for transition verification*:

  In some cases, the current node n_c is d-recognized conditionally so that our algorithm does not choose to extend the sequence to do a state recognition.

Our algorithm aims to verify its unverified transitions since appending an ADS would be useless while n_c is d-recognized conditionally. The unverified transitions of n_c correspond to the edges whose ending node is not d-recognized or d-recognized conditionally. Therefore, algorithm checks the emanating edges of n_c to find edges with ending node without a d-recognition or conditional d-recognition. After finding such edges, the algorithm calculates the required extension sequences. The extension sequences are calculated as follows. Suppose that the algorithm finds an edge $(n_c, n_{c,p}; x_p/y_p)$ where the node $n_{c,p}$ corresponds to the state s_i and $n_{c,p}$ is not d-recognized or d-recognized conditionally by $\mathcal{A}_i^j$ as the state s_i. Therefore, one extension possibility is to use $x_p\mathcal{A}_i^j$, in order to obtain d-recognition or conditional d-recognition of $n_{c,p}$ by $\mathcal{A}_i^j$.

Formally, let $\{n_c, n_{c,i}; x_i/y_i$ be the set of outgoing edges of n_c, where $n_{c,i}$ corresponds to the state s_i. Then the set of possible extension sequences are $x_i\mathcal{A}_i^j$, where $\mathcal{A}_i^j$ is an ADS such that $n_{c,i}$ is not d-recognized by $\mathcal{A}_i^j$ yet.

The current node n_c could have more than one edges $(n_c, n_{c,1}; x_1/y_1), \ldots, (n_c, n_{c,k}; x_k/y_k)$. Therefore, when algorithm operates to find an extension sequence to do transition verification, we would also have more than one extension candidates. Formally, there are edges $(n_c, n_{c,1}; x_1/y_1)$ $, \ldots (n_c, n_{c,k}; x_k/y_k)$ and some of the nodes $n_{c,p}$ is not d-recognized or d-recognized conditionally by $\mathcal{A}_i^j$, $\forall j$ as the state s_i. Therefore, we would have multiple extension sequences like $x_p\mathcal{A}_i^j$ and we call the set of sequences $x_p\mathcal{A}_i^j$ as a candidate set for transition verification extension.

After finding candidates, the algorithm should decide which candidate to use. It makes this choice greedily as follows:

Rule 1 If there is a unique shortest extension sequence, then this shortest sequence is used.

Rule 2 If there are multiple shortest possible extension sequences, then the extension sequence for $n_{c,i}$ with the fewest number of remaining d-recognition by ADSs is used.

Rule 3 If there are multiple candidates for Rule 2, one of them is chosen randomly.

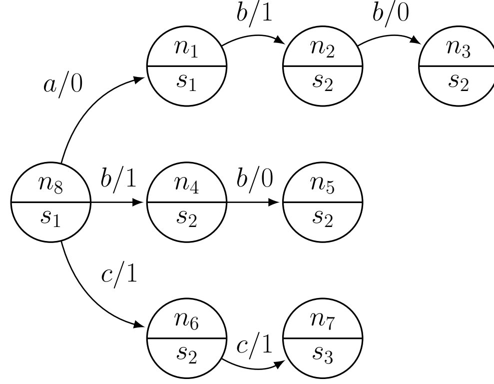


Figure 4.6: Transition verification example

Assume that the Figure 4.6 is a part of R. According to this:

- n_c is d-recognized conditionally since it is d-recognized conditionally by $\mathcal{A}_1^j$ for all j.
- A as a d-recognition candidacy by $\mathcal{A}_1^2$.
- B has a d-recognition candidacy by $\mathcal{A}_2^2$.
- D does not have any d-recognition candidacy.

Therefore sequence extension candidates will be as follows:

- For node A:
  * $x_1 = a$ and A could have a d-recognition candidacy by $\mathcal{A}_1^1$. Therefore overall extension sequence would be $x_1.\mathcal{A}_1^1 = aa$.
  * $x_1 = a$ and A could have a d-recognition candidacy by $\mathcal{A}_1^3$. Therefore overall extension sequence would be $x_1.\mathcal{A}_1^3 = acc$.
- For node B:
  * $x_2 = b$ and B could have a d-recognition candidacy by $\mathcal{A}_2^1$. Therefore overall extension sequence would be $x_1.\mathcal{A}_2^1 = baa$.
  * $x_2 = b$ and B could have a d-recognition candidacy by $\mathcal{A}_2^3$. Therefore overall extension sequence would be $x_1.\mathcal{A}_2^3 = bcc$.
- For node D:
  * $x_3 = c$ and D could have a d-recognition candidacy by $\mathcal{A}_3^1$. Therefore overall extension sequence would be $x_1.\mathcal{A}_3^1 = caa$.

* $x_3 = c$ and D could have a d-recognition candidacy by $\mathcal{A}_3^2$. Therefore overall extension sequence would be $x_1.\mathcal{A}_3^2 = cb$.
* $x_3 = c$ and D could have a d-recognition candidacy by $\mathcal{A}_3^3$. Therefore overall extension sequence would be $x_1.\mathcal{A}_3^3 = ccc$.

In this case the shortest extension sequences are $x_1\mathcal{A}_1^1 = aa$ and $x_3\mathcal{A}_3^2 = cb$. Therefore the algorithm finds two same length shortest extension sequences and cannot make a decision based on Rule 1. Now it tries to make a choice based on Rule 2 and explores the number of conditional d-recognition of the nodes n_1 and n_6. Since we know that n_1 is d-recognized conditionally by $\mathcal{A}_1^2$ and n_6 is not d-recognized conditionally, it chooses sequence $x_1\mathcal{A}_1^1 = aa$ to extend the sequence $\tilde{\omega}$. If the algorithm hits the case that could not differentiate these two nodes based on Rule 2 then it would make a choice randomly as stated in Rule 3.

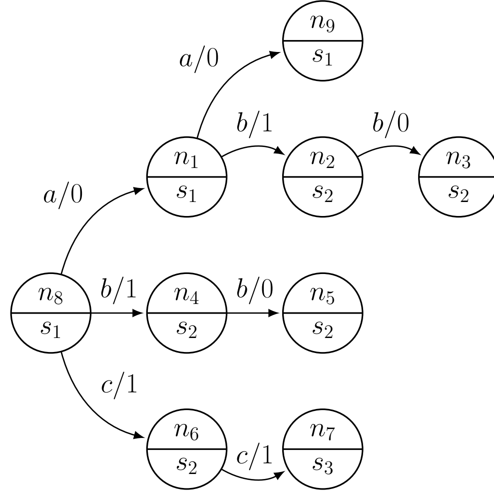


Figure 4.7: Sequence extension on R

After appending the extension sequence aa to $\tilde{\omega}$, the resulting R is shown in Figure 4.7. Now we know that nodes n_1 and n_8 are conditionally i-equivalent because they are both d-recognized conditionally by $\mathcal{A}_1^1$ Hence we can merge them. According to the merging rules given in 3, they lead us to merge nodes n_4 and n_5 to the nodes n_2 and n_3, respectively. The resulting R is shown in Figure 4.8.

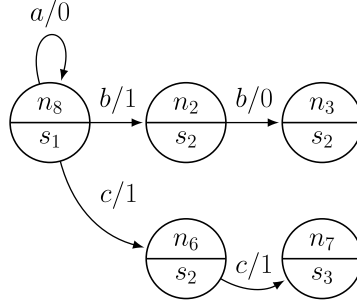


Figure 4.8: Updated Recognition Automaton R after transition verification

- *Extension for missing transition verification*:

  In some cases the current node n_c is d-recognized by $\mathcal{A}_i^j$ conditionally, $\forall j$ as the state s_i and the outgoing edges n_c has that correspond to transitions of M are verified. Hence the algorithm does not choose to extend sequence to do a state recognition or a transition verification. In this case, the algorithm searches for the edges of n_c that corresponds to transitions of M which are missing in R.

  Therefore, the algorithm aims to find these absent transitions. After finding such transitions, the algorithm calculates the required extension sequences. The extension sequences are calculated as follows. Suppose that the algorithm finds a transition $t = (s_i, s_k; x/y)$ where the current node n_c does not have an edge with input label x and n_c corresponds to the state s_i. Then it finds an ADS $\mathcal{A}_k^j$ which can be used to perform conditional d-recognition for the node to be created as an ending node of the edge which corresponds to the transition t on the recognition automaton R.

  Formally, let $\{n_c, n_{c,i}, x_i/y_i\}$ be the set of outgoing edges of n_c. Suppose that X_{n_c} is the set of input symbols where the node n_c does not have an edge with input symbol $x \in X_{n_c}$. Also, let $\{s_i, s_k; x/y\}$ be the set of transitions, where the node n_c corresponds to the state s_i and $x \in X_{n_c}$. Then the set of possible extension sequences are $x\mathcal{A}_k^j$, $\forall j$.

  After finding possible extension sequences, the algorithm should decide which one to use. It makes this choice greedily as follows:

Rule 1 If there is a unique shortest extension sequence, then this shortest sequence is used.

Rule 2 If there are multiple candidates for Rule 1, one of them is chosen randomly.

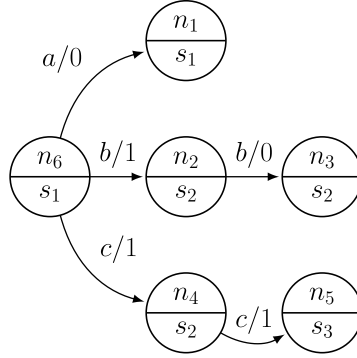


Figure 4.9: Missing Transition Verification Example

Consider the recognition automaton R in Figure 4.9 and assume that the nodes n_1,n_2 and n_4 are d-recognized conditionally and the input alphabet of the specification $\tilde{M}$ is $\{a, b, c, d, e, f\}$. Note that the input alphabet of $\tilde{M}$ is different from M. According to this:

- n_6 is d-recognized conditionally since it is d-recognized conditionally by $\mathcal{A}_1^j$ for all j.
- The transitions with input label a, b and c are verified.

Suppose that n'_c is the ending node of transition that is going to be added as an edge to the recognition automaton R. Therefore sequence extension candidates will be as follows:

- For transition $t = (s_1, s_1; d/0)$:
  * $d\mathcal{A}_1^1 = da$ to d-recognize n'_c conditionally by $\mathcal{A}_1^1$.
  * $d\mathcal{A}_1^2 = dbb$ to d-recognize n'_c conditionally by $\mathcal{A}_1^2$.
  * $d\mathcal{A}_1^3 = dcc$ to d-recognize n'_c conditionally by $\mathcal{A}_1^3$.
- For transition $t = (s_1, s_1; e/0)$:
  * $e\mathcal{A}_1^1 = ea$ to d-recognize n'_c conditionally by $\mathcal{A}_1^1$.
  * $e\mathcal{A}_1^2 = ebb$ to d-recognize n'_c conditionally by $\mathcal{A}_1^2$.
  * $e\mathcal{A}_1^3 = ecc$ to d-recognize n'_c conditionally by $\mathcal{A}_1^3$.

– For transition $t = (s_1, s_2; f/0)$:

  * $f\mathcal{A}_2^1 = faa$ to d-recognize n'_c conditionally by $\mathcal{A}_2^1$.
  * $f\mathcal{A}_2^2 = fb$ to d-recognize n'_c conditionally by $\mathcal{A}_2^2$.
  * $f\mathcal{A}_2^3 = fcc$ to d-recognize n'_c conditionally by $\mathcal{A}_2^3$.

For this case the shortest extension sequences are $d\mathcal{A}_1^1 = da$, $e\mathcal{A}_1^1 = ea$ and $f\mathcal{A}_2^2 = fb$. Therefore the algorithm cannot decide over the same length extension sequences differentiate based on Rule 1. Therefore it makes a choice randomly as stated in Rule 2 and extend the sequence with $d\mathcal{A}_1^1 = da$.

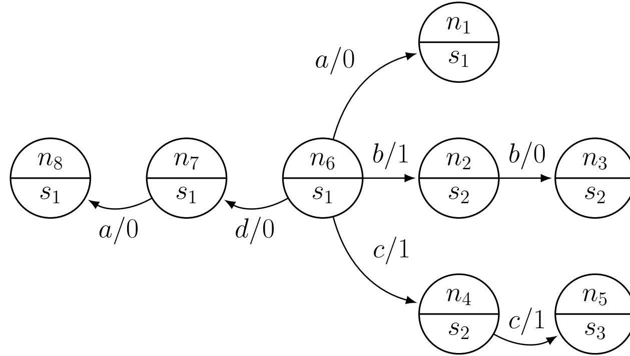


Figure 4.10: Recognition Automaton R after sequence extension

We append da to $\tilde{\omega}$ and update recognition automaton R like in Figure 4.10. Now we know that nodes n_6 and n_7 are i-equivalent since they are both d-recognized by $\mathcal{A}_1^1$ as state s_1 conditionally. Hence we merge them and R becomes as shown in Figure 4.11.

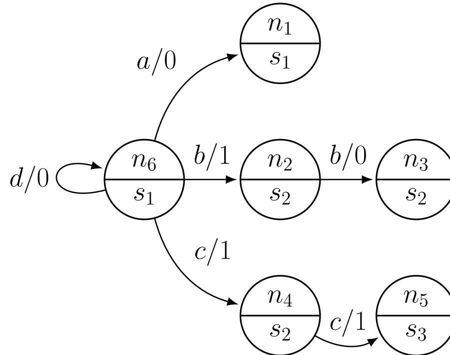


Figure 4.11: Recognition Automaton R after merging operations

- *Shortest path to the unrecognized node*:

In some cases current node n_c is d-recognized by $\mathcal{A}_i^j$ conditionally, $\forall j$ as the state s_i and all of its transitions are verified. Therefore, d-recognizing the node n_c conditionally or verifying its transitions will be pointless. If we encounter such a case, we make a breadth first search in R from the current node n_c to find the shortest path to a node n'_c such that:

1 n'_c is not d-recognized conditionally, or

2 n'_c is d-recognized conditionally but has at least one unverified transition, or

3 n'_c is d-recognized conditionally but has at least one missing transition

Formally, there are subpaths $(n_c, n'_c; w_s/w_y)$ of R. The node n_c is d-recognized conditionally and does not have any missing transition and all transitions of n_c are verified. But the node n'_c is not d-recognized by an ADS conditionally or has unverified transitions or has missing transitions. Therefore we call set of sequences ω_s as a candidate set for shortest path extensions. After finding candidates, the algorithm should decide which candidate to use. It makes this choice greedily as follows:

Rule 1 If there is a unique shortest extension sequence, then this shortest sequence is used.

Rule 2 If there are multiple candidates for Rule 1, one of them is chosen randomly.

After appending ω_s to $\tilde{\omega}$, and updating the recognition automaton R accordingly, it follows the procedures explained in this section previously based on the new current node.

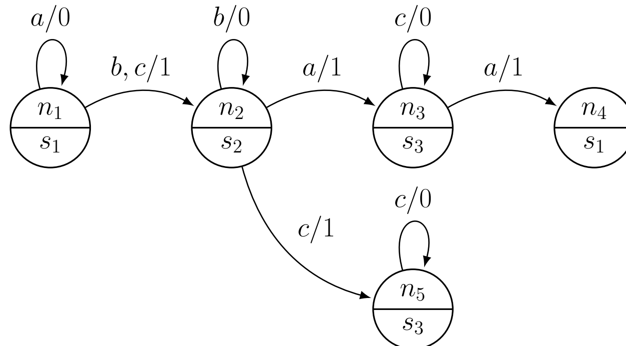


Figure 4.12: A Recognition Automaton R

Consider the Figure 4.12, according to this where $n_c = n_1$:

- The nodes n_3, n_4 and n_5 are not d-recognized conditionally.
- The nodes n_3, n_4 and n_5 have missing and unverified transitions.

Therefore, the algorithm tries to find a shortest path from the node n_1 to the nodes n_3, n_4 and n_5. According to this, sequence extension candidates will be as follows:

- For node n_3:
    * The path $(n_1, n_3; \omega_s/\omega_y)$ where w_s could be *ba*.
- For node n_4:
    * The path $(n_1, n_4; \omega_s/\omega_y)$ where w_s could be *baa*.
- For node n_5:
    * The path $(n_1, n_5; \omega_s/\omega_y)$ where w_s could be *bc*.

For this case the shortest extension sequences are *ba* and *bc*. Therefore the algorithm cannot decide over the same length extension sequences based on Rule 1. Therefore, it makes a choice randomly as stated in Rule 2 and extend the sequence with *ba*.

We append *ba* to $\tilde{\omega}$ and update the recognition automaton R. This update does not change the structure of R but makes the node n_3 the new current node n_c.

Now we know that the current node is n_3 and we can continue to the next iteration of the algorithm by taking the node n_3 into consideration.

The sequence extension options are presented above in this section. Now we will present the algorithm that reveals how we use those options. As it is seen in Algorithm 1, we start to construct the recognition automaton R with a single node n_1 that corresponds to state s_1 of M. We then iteratively extend it by considering the extension options we have. First we check if we could extend the sequence by backtrack extension. If we could not find a situation to do backtrack extension, then we look for a transition verification. If we encounter a node with all transitions verified, then we look for a missing transition. If we could not find any missing transition,

which means the current node is d-recognized and all of its transitions are verified, then we find a path that takes the current node to the nearest node which is not d-recognized or has a missing transition or has an unverified transition.

Algorithm 1 Sequence generation algorithm

Input: Deterministic and completely specified FSM M
Input: Set of ADSs $\mathcal{A} = \{\mathcal{A}^1, \ldots, \mathcal{A}^k\}$ where $k > 0$
Output: $\tilde{\omega}$ an input sequence for M

```
ω̃ ← ϵ
R is the recognition automaton with a single node n_1
n_c ← n_1 where n_c is the current node
while R ≠ M do
  γ ← backtrack(R, n_c, 𝒜 ) // try to extend using backtracking
  if γ = ϵ then
    γ ← transitionverification(R, M, n_c, 𝒜 ) // try to do transition verification
    if γ = ϵ then
      γ ← missingtransition(R, M, n_c, 𝒜 ) // try to extend by verifying a missing
      transition
      if γ = ϵ then
        γ ← shortestpath(R, n_c, 𝒜 )
      end if
    end if
  end if
  ω̃ ← ω̃γ
  n_c ← update(R, γ)
end while
```

Within an iteration of the algorithm, we update the recognition automaton R as we append new extension sequence to $\tilde{\omega}$ and do the conditional recognitions. In the update part, we merge the nodes as we find out that they are i-equivalent conditionally. The Algorithm 1 stops when R is isomorphic to M.

4.3 Phase 2: Checking if a sequence is a checking sequence

Previously we noted that, the sequence generated by Phase 1 may not be a checking sequence due to dependencies between ADS trees in terms of legality. Therefore, in Phase 2 we operate on $P_{N,\omega}$ to detect dependencies between ADS trees. If it

finds dependency then it breaks it by extending the sequence $\tilde{\omega}$ to provide more observations. In the end, all ADS trees become legal and $\tilde{\omega}$ is turned out to be a checking sequence.

In Phase 2, we initialize the recognition automaton R equals to $P_{N,\omega}$. Phase 2 stops when R collapses into a form that is isomorphic to the specification M. At each iteration, it first tries to find a legal ADS tree in R, for which some i-equivalent nodes are not merged. If it finds such an ADS tree, then it updates the recognition automaton R and merge the i-equivalent nodes. If it cannot find a legal ADS tree, then it extends the sequence based on the set of rules to complete missing valid observations of ADSs of an ADS tree. The algorithm ends when all ADS trees are legal and R is isomorphic to M.

Consider the input sequence generated by Phase 1 is $\omega/y = aabbbcccaacbacbbb/0010$ 0100101010110 and $P_{N,\omega}$ given in Figure 4.13. In Phase 2, we try to find a legal ADS tree by considering the legal observations of ADSs. It is stated that the purpose of Phase 2 is to check whether ω is a checking sequence. If it is not, the sequence is converted into a checking sequence by making every ADS tree legal by extending the sequence. The sequence is extended to provide valid observation of ADSs.

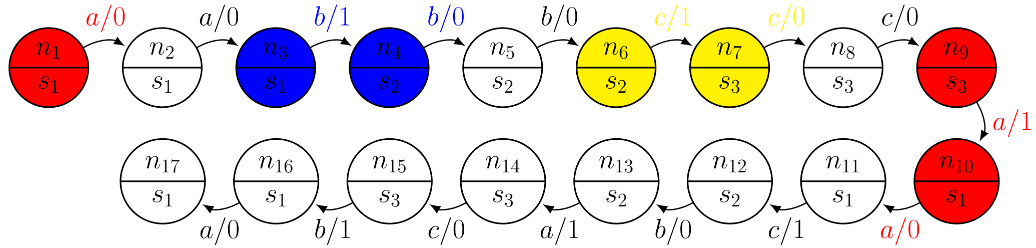


Figure 4.13: Valid Observation of ADSs on the Path $P_{N,\omega}$

Consider Figure 4.13, the algorithm investigates the valid observations of ADSs. The legal observations present on $P_{N,\omega}$ are as follows:

- The subpath $(n_1, n_2; a/0)$ corresponds to legal observation of $\mathcal{A}_1^1$.
- The subpath $(n_3, n_5; bb/10)$ corresponds to legal observation of $\mathcal{A}_1^2$.
- The subpath $(n_4, n_5; b/0)$ corresponds to legal observation of $\mathcal{A}_2^2$.
- The subpath $(n_6, n_8; cc/10)$ corresponds to legal observation of $\mathcal{A}_2^3$.

- The subpath $(n_7, n_8; c/0)$ corresponds to legal observation of $\mathcal{A}^3_3$.
- The subpath $(n_9, n_{11}; aa/10)$ corresponds to legal observation of $\mathcal{A}^1_3$.

Now we know that none of the ADS trees we use is a legal ADS tree since we do not see valid observations for $\mathcal{A}^j_i$, $\forall i$ for any j. The algorithm tries to convert at least one of the ADS trees into a legal one by extending the sequence. The algorithm calculates extension candidates as follows:

- To legalize the ADS tree $\mathcal{A}^1$, we need to append $\mathcal{A}^1_2$ to the node corresponds to state s_2. Since the current node is n_{17} and it does not have any outgoing edge, the sequence should be extended to transfer the node n_{17} to some node corresponding to state s_2. This transfer could be provided by an extension sequence ω_t which is b or c. Therefore, the total extension sequence could be either $b\mathcal{A}^1_2 = baa$ or $c\mathcal{A}^1_2 = caa$.
- To legalize the ADS tree $\mathcal{A}^2$, we need to append $\mathcal{A}^2_3$ to the node corresponding to state s_3. Since current node is n_{17} and it does not have any outgoing edge, the sequence should be extended to transfer the node n_{17} to some node corresponding to state s_3. This transfer could be provided by an extension sequences ω_t which is ba or bc or ca or cc. Therefore, the total extension sequence is $\omega_t\mathcal{A}^2_3$.
- To legalize the ADS tree $\mathcal{A}^3$, we need to append $\mathcal{A}^3_1$ to the node corresponding to state s_1. Since current node n_{17} corresponds to state s_1, we can directly extend the sequence by $\mathcal{A}^3_1 = cc$.

After determining the extension candidates, the algorithm makes this choice among the candidates greedily as follows:

1 If there is a unique shortest extension sequence, then this shortest sequence is used.

2 If there are multiple candidates for Rule 1, one of them is chosen randomly.

According to the rules above, the algorithm chooses the shortest extension sequence and it is cc in this case. Hence the sequence is extended by cc and recognition automaton is updated as shown in Figure 4.14.

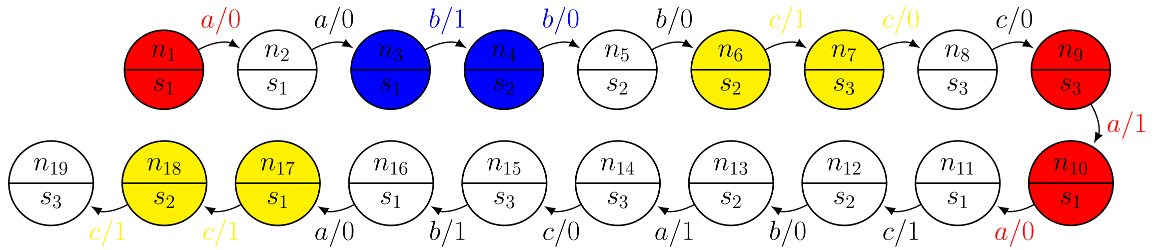


Figure 4.14: Sequence Extension on the Path $P_{N,\omega'}$

This extension made ADS tree $\mathcal{A}^3$ legal, therefore the next task the algorithm performs is to find the nodes d-recognized by $\mathcal{A}^3$. Nodes n_7, n_8 and n_{14} are all d-recognized by $\mathcal{A}^3$ as state s_3, hence they are i-equivalent. The information we gathered from the nodes n_7 and n_8 indicates that the nodes n_9 and n_{15} are also i-equivalent to n_7. Therefore nodes n_7, n_8, n_9, n_{14} and n_{15} can be merged in n_7. The resulting recognition automaton R is shown in Figure 4.15.

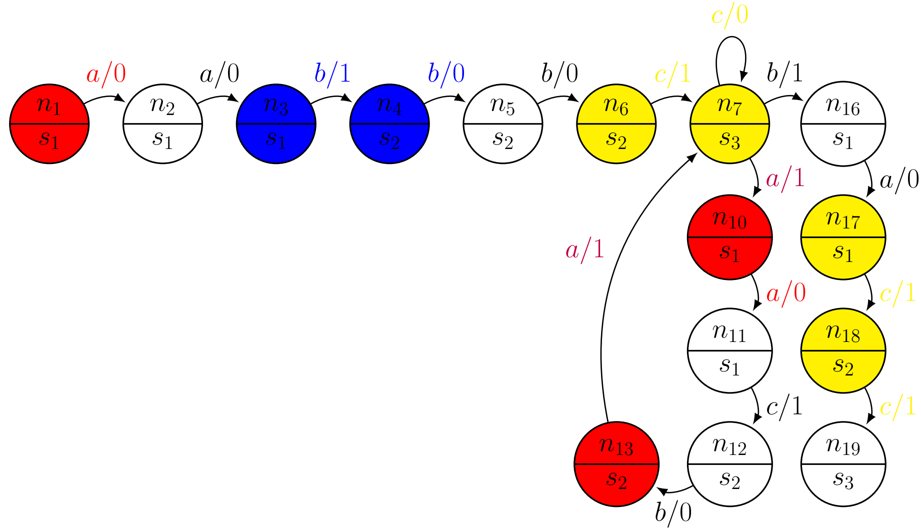


Figure 4.15: A Path $P_{N,\omega'}$ after merging nodes n_7, n_8, n_9, n_{14} and n_{15}

For the next iteration of the algorithm, it will again search for a legal ADS tree on R with i-equivalent nodes that are not merged yet. Note that, the i-equivalence relation based on the ADS trees $\mathcal{A}^1$ and $\mathcal{A}^2$ is not considered in the previous iteration. Therefore, the algorithm investigates the valid observations of ADSs that belong to ADS trees $\mathcal{A}^1$ and $\mathcal{A}^2$. The legal observations regarding ADS trees $\mathcal{A}^1$ and $\mathcal{A}^2$ present on R are as follows:

- The subpath $(n_1, n_2; a/0)$ corresponds to legal observation of $\mathcal{A}_1^1$.
- The subpath $(n_{13}, n_{10}; aa/11)$ corresponds to legal observation of $\mathcal{A}_2^1$.
- The subpath $(n_7, n_{11}; aa/10)$ corresponds to legal observation of $\mathcal{A}_3^1$.
- The subpath $(n_3, n_5; bb/10)$ corresponds to legal observation of $\mathcal{A}_1^2$.
- The subpath $(n_4, n_5; b/0)$ corresponds to legal observation of $\mathcal{A}_2^2$.

Now we know that the ADS tree $\mathcal{A}^1$ is legal since there are valid observations for $\mathcal{A}_i^1$, $\forall i$ on R. The algorithm identifies the i-equivalent nodes based on the ADS tree $\mathcal{A}^1$. The nodes n_1, n_2, n_{10}, n_{16} are i-equivalent nodes since they are all d-recognized by $\mathcal{A}_1^1$ as s_1. Merging these nodes also gives the information that the nodes n_3, n_{11} and n_{17} are also i-equivalent to the node n_1. Therefore the algorithm merges the nodes n_1, n_2, n_3, n_{10}, n_{11}, n_{16}, n_{17} into the node n_1. The resulting recognition automaton R is shown in Figure 4.16.

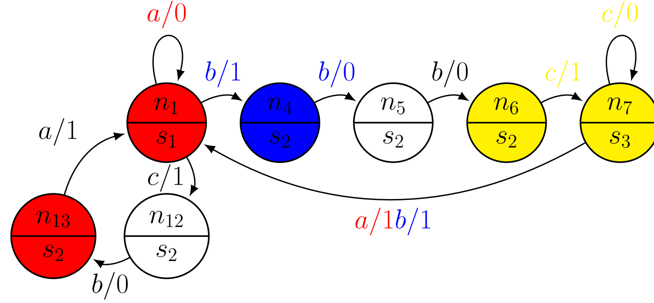


Figure 4.16: A Path $P_{N,\omega'}$ after merging nodes n_1, n_2, n_3, n_{10}, n_{11}, n_{16}, n_{17}

In the next iteration of the algorithm, it will again search for a legal ADS tree with i-equivalent nodes that are not merged yet. Therefore, the algorithm investigates the valid observations of ADSs that belong to ADS tree $\mathcal{A}^2$. The legal observations of the ADS tree $\mathcal{A}^2$ present on R are as follows:

- The subpath $(n_1, n_5; bb/10)$ corresponds to legal observation of $\mathcal{A}_1^2$.
- The subpath $(n_4, n_5; b/0)$ corresponds to legal observation of $\mathcal{A}_2^2$.
- The subpath $(n_7, n_4; bb/11)$ corresponds to legal observation of $\mathcal{A}_3^2$.

Now we know that the ADS tree $\mathcal{A}^2$ is legal since there are valid observations for $\mathcal{A}_i^2$, $\forall i$ on R. The algorithm identifies the i-equivalent nodes based on the ADS tree $\mathcal{A}^2$. The nodes n_4, n_5, n_{12} are i-equivalent nodes since they are all d-recognized by $\mathcal{A}_2^2$ as s_2. Merging these nodes also gives the information that the nodes n_6 and n_{13} are also i-equivalent to the node n_4. Therefore the algorithm merges the nodes n_4, n_5, n_6, n_{12} and n_{13} into the node n_4. The resulting recognition automaton R is shown in Figure 4.17.

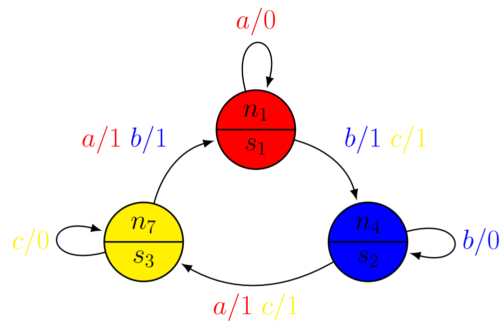


Figure 4.17: Resulting Recognition Automaton R

Since the resulting recognition automaton is isomorphic to the specification M and all of the ADS trees we use are legal, sequence ω becomes a checking sequence by only extending the sequence generated by Phase 1 with cc.

Chapter 5

Construction and Selection of ADS Trees

In this thesis, we used multiple ADS trees. Therefore, we need to generate ADS trees for each FSM M we used in the experiments. We think that the ADS trees generated for M should include shorter ADSs for each state of M, so that we could use these shorter ADSs for transition verification to reduce the length of the checking sequence.

An ADS is a sequence for a state q where it distinguishes the state q from any other state s of M in terms of their output responses to the ADS while preventing other states to be merged into the same state during the application of the ADS. In other words, when an ADS is applied to all states of M, the state q produces a different output sequence as a response to the ADS from any other state. Although the output responses of the states other than q could be the same with each other, application of the ADS on other states does not cause states to be indifferentiable.

Formally, an ADS for a state q is an input sequence α such that:
If $\forall s \in S$, $s \neq q$ then $\lambda(s, \alpha) \neq \lambda(q, \alpha)$ and for any prefix α' of α, $\forall s, s' \in Q$ if $\lambda(s, \alpha') = \lambda(s', \alpha')$ then $\delta(s, \alpha') \neq \delta(s', \alpha')$.

In other words, they never merge into the same state, when their output responses are the same. To generate ADS trees that have shorter ADS for a state q, we use the shortest ADS for the state q. Since an ADS is one branch of the ADS tree, and we know that an ADS prevents other states to be merged into the same state, we can complete the other branches of the ADS tree after we construct the branch corresponding to the state q.

In this section, we explain how we use the idea stated above to generate ADS trees. As stated above, an ADS is needed to generate an ADS tree. Therefore, we present how we generate an ADS using Answer Set Programming (ASP). In addition, we propose a method to choose a set of ADS trees from the ADS tree pool.

5.1 ASP Formulation of ADS

Lee and Yannakakis have reported that checking the existence of an ADS can be decided in polynomial time [24]. However, computing a shortest ADS for a given state q is an NP-hard problem [30]. Therefore, it is a hard problem to solve. In this thesis, we take an advantage of the usefulness of Answer Set Programming [6, 26] to solve this optimization problem.

In this thesis, we formulate the problem of computing a shortest ADS for a state q in Answer Set Programming [26] - a knowledge representation and reasoning paradigm with an expressive formalism and efficient solvers for NP-Hard problems. The idea of ASP is to formalize a given problem as "program" and to solve the problem by computing models called "answer sets" [16] of the program using "ASP solvers", such as CLASP [15].

To formulate the problem of generating a shortest ADS, we should consider the decision version of the problem. Therefore, let us first consider the decision version of the shortest ADS problem:
For an FSM $M = (S, X, Y, \delta, \lambda)$, a state q of M, and a positive integer constant c, decide whether q has an ADS α of length c.

To do that, we need a set of atoms that will represent the transitions, states, input and output symbols of FSM M and other atoms and set of rules to formulate the decision version of the problem of generating an ADS, so that we can find an answer set that represents the ADS. Without loss of generality, we represent states and input and output symbols of a FSM $M = (S, X, Y, \delta, \lambda)$, by the range of numbers $1 \dots n$ and $1 \dots j$ and $1 \dots k$ ($n = |Q|$, $j = |X|$, $k = |Y|$), respectively. Then an FSM $M = (S, X, Y, \delta, \lambda)$ can be described in ASP by three forms of atoms given below:

- $state(s)$ ($1 \leq s \leq n$) describing the states in Q,
- $i\text{-}symbol(x)$ ($1 \leq x \leq j$) describing the input symbols in X,

- *o-symbol*(y) $(1 \leq y \leq k)$ describing the output symbols in Y,
- *transition*(s, x, y, s') $(1 \leq s, s' \leq n,\ 1 \leq x \leq j,\ 1 \leq y \leq k)$ describing the transitions where $\delta(s, j) = s'$, $\lambda(s, j) = y$

We represent possible lengths i of sequences by atoms of the form *step*(i) $(1 \leq i \leq c)$. An ADS α of length c is characterized by atoms of the form *ads*(i, x) $(1 \leq i \leq c,$ $1 \leq x \leq j)$ describing that the ith symbol of the sequence α is x.

Using these atoms, we can represent the decision version of the shortest ADS problem with a "generate-and-test" methodology used in various ASP formulations. In the following, we represent ASP formulations based on this approach.

In this ASP formulation, we use an auxiliary concept of a *path characterized by a sequence* $\alpha_1, \alpha_2, \ldots, \alpha_c$ *of symbols in* X, which is defined as a sequence $q_1, q_2, \ldots, q_{c+1}$ of states in Q and a sequence of output symbols $y_1, y_2, \ldots, y_c$ such that $\delta(q_i, \alpha_i) = q_{i+1}$ and $\lambda(q_i, \alpha_i) = y_i$ for every i $(1 \leq i \leq c)$. The existence of such a path of length i in M from a state s to a state q (i.e., the reachability of a state q from a state s by a path of length i in M) characterized by the first i symbols of a word α is represented by atoms of the form *path*$(s, i+1, y, q)$ defined as follows:

$$\begin{aligned}
&\mathit{path}(s, 1, s) \leftarrow \mathit{state}(s) \\
&\mathit{path}(s, i+1, q) \leftarrow \mathit{path}(s, i, r), \mathit{ads}(i, x), \mathit{transition}(r, x, y, q) \\
&\mathit{active}(s, i+1), \mathit{state}(s), \mathit{state}(r), \mathit{state}(q), \textit{i-symbol}(x) \\
&\textit{o-symbol}(y), \mathit{step}(i)
\end{aligned} \tag{5.1}$$

In this formulation, first we "generate" a sequence α of c symbols by the following choice rule:

$$1\{\mathit{ads}(i, j) : \textit{i-symbol}(j)\}1 \leftarrow \mathit{step}(i) \tag{5.2}$$

where *step*(i) is defined by a set of facts:

$$\mathit{step}(i) \leftarrow \qquad (1 \leq i \leq c) \tag{5.3}$$

To specify the state where we want to find an ADS for, we use the atom *adsState*(q).

We also "generate" the output sequence that is defined by *output*(i, y) atom where

it is the sequence generated when the ADS is applied to the state q that we are trying to find an ADS for, by the following rule:

$$\begin{aligned}&\textit{o-symbol}(i,y) \leftarrow \textit{path}(q,i,r), \textit{ads}(i,x), \textit{transition}(r,x,y,s),\\&\textit{state}(s), \textit{state}(r), \textit{state}(q), \textit{adsState}(q)\\&\textit{i-symbol}(x), \textit{o-symbol}(y), \textit{step}(i)\end{aligned} \tag{5.4}$$

We try to distinguish the output of a state s from the output generated by the state q when we apply the ADS. Therefore, we check the states whether they are still needed to be differentiated or not and if they need to, then we label them as active at step i with the atom $\textit{active}(s,i)$ by using the following rules:

$$\begin{aligned}&\textit{active}(s,0) \leftarrow \textit{state}(s)\\&\textit{active}(s,i+1) \leftarrow \textit{path}(s,i,r), \textit{ads}(i,x), \textit{transition}(r,x,y,q)\\&\textit{active}(s,i), \textit{output}(i,y), \textit{state}(s), \textit{state}(r), \textit{state}(q)\\&\textit{i-symbol}(x), \textit{o-symbol}(y), \textit{step}(i)\end{aligned} \tag{5.5}$$

We also specify the states that are already distinguished from the state q. To do that we use the atom $\textit{active}(s,i)$ and when we find a state that is active at step i but not active at step $i+1$, then we know that the state s is differentiated and we label it as $finished(s,i)$ to specify that it is differentiated from state q at step i by the following rule:

$$\begin{aligned}&\textit{finished}(s,i+1,y,z) \leftarrow \textit{path}(s,i,r), \textit{transition}(r,x,y,z), \textit{ads}(i,x)\\&\textit{active}(s,i), \textit{not}\,\textit{active}(s,i+1), \textit{state}(s), \textit{state}(r), \textit{state}(z)\\&\textit{i-symbol}(x), \textit{o-symbol}(y), \textit{step}(i)\end{aligned} \tag{5.6}$$

After specifying the states that are distinguished, we also check that whether they are merged into the same state in any step since it is a necessary condition for an ADS. While differentiating the state q from any other state, it does not let the other states to be merged into the same state. This condition is guaranteed by the following rule:

$$\begin{aligned}&\leftarrow \textit{finished}(s,i,y,z), \textit{finished}(q,i,y,z), \textit{state}(s), \textit{state}(q), \textit{state}(z)\\&\textit{o-symbol}(y), \textit{step}(i)\end{aligned} \tag{5.7}$$

The union of the program ASP that consists of the rules (5.1), (5.2), (5.3), (5.4), (5.5), (5.6), (5.7), with a set of facts describing an FSM M has an answer set iff there exists an ADS of length c for state q.

5.1.1 Optimization

The ASP formulation given in Section 5.1 with a set of facts describing an FSM M, have answer sets if the given FSM M has an ADS of length c for a state q. In order to find the shortest length ADS, one can perform a binary search on possible values of c.

In this section, we present another ASP formulation where we let the ASP solver decide the length l of a shortest ADS, where $l \leq c$:

$$1\{shortest(l) : 1 \leq l \leq c\}1 \leftarrow \tag{5.8}$$

and declare possible lengths of sequences:

$$step(j) \leftarrow shortest(i) \qquad (1 \leq j \leq i \leq c). \tag{5.9}$$

Next, we ensure that l is indeed the optimal value, by the following optimization statement

$$\#minimize[shortest(l) = l] \leftarrow \tag{5.10}$$

We denote by ASP^{opt} the ASP formulation obtained from ASP by adding the rules (5.8) and (5.10), and replacing the rules (5.3) by the rules (5.9). If ASP^{opt} with a set of facts describing an FSM M has an answer set X then X characterizes a shortest ADS of state q for M.

5.2 ADS Tree Generation Using an ADS

In Section 5.1, we describe the ASP formulation to generate shortest ADS for a particular state q. Now we will describe how we generate an ADS tree using an ADS. It is stated that we use multiple ADS and main purpose is to get advantage of

relatively shorter ADSs of states while doing the transition verification. Therefore, we need ADS trees that have shorter ADSs for different sets of states. In this way, we have relatively shorter ADSs for each state and once we are done with the cross verification, we can use the shorter ADSs to verify transitions. This is the main idea for reducing the length of the checking sequence even though the cross verification is costly.

The ADS trees that have relatively shorter ADSs for the set of states is called *unbalanced ADS trees*. We generate unbalanced trees using an ADS of a particular state q. Given a branch of an unknown ADS tree as an ADS, our job is to find out the other branches of the ADS tree. To do that, we apply the ADS to all the states step by step. We keep the set of states that respond with same output sequence and when a set of states is distinguished from the state q, we stop applying the ADS to that set of states. In the end we acquire sets of states where the states in the same set are not distinguished from each other. Therefore, if we find an ADS (sub)tree that distinguishes the states within the same set, we can complete all the branches of the ADS tree and generate an ADS tree with shorter ADS for state q.

Formally, let the sequence $\alpha = \alpha_1 \dots \alpha_c$ be an ADS for state q where FSM M has n states in total and the length of α is c. Let α^i be a prefix $\alpha_1 \dots \alpha_i$ of the sequence α. Let $Q_i = \{s \in Q | \lambda(s, \alpha^i)) = \lambda(q, \alpha^i)\}$. In other words Q_i is the set of states whose output response is same as the output response of q up to and including the ith step. Let $\overline{Q}_i = Q_{i-1} \backslash Q_i$ be the set of states whose output responses are distinguished from q at the ith step. Let $B_{x/y} = \{s \in B | \lambda(s, x) = y\}$ be the set of states in B which produce the output sequence y for the input sequence x. Let $B = \overline{Q}_i$, then the partition $\prod_i = \{B_{x/y} | y \in \lambda(B, x)\}$. The partition $\prod_i$ stands for the set of states whose output responses to α^i are the same with each other while their output responses are distinguished from the state q.

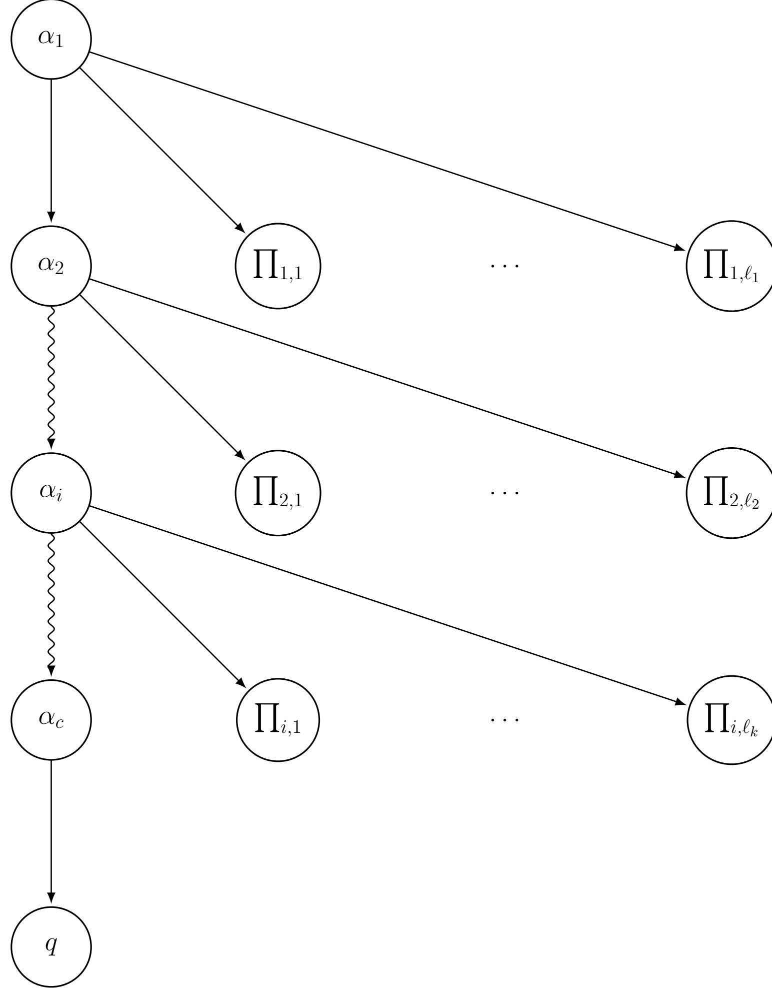


Figure 5.1: A Partial ADS Tree

Consider the partial ADS tree shown in Figure 5.1. Let $\prod_{i,j}$ be the jth set of partition $\prod_i$. Therefore each $\prod_{i,j}$ stands for the set of states that respond to α^i with the same output sequence. To complete the ADS tree, the states in $\prod_{i,j}$ are needed to be distinguished from each other. Note that $\prod_{i,j}$ corresponds to the initial states. However, for the subtree to be rooted at the node corresponding to $\prod_{i,j}$, we need to consider the current states reached by the application of α^i. The current states are $\delta(\prod_{i,j}, \alpha^i)$. Hence, in order to complete the ADS tree, we need to find an ADS (sub)tree, that can distinguish the set of states $\delta(\prod_{i,j}, \alpha^i)$.

To distinguish the states within $\prod_{i,j}$, we use an ADS tree generated by the LY Algorithm [24]. We denote this ADS tree by A^{LY}. We know that each $s \in \delta(\prod_{i,j}, \alpha^i)$ has an ADS within A^{LY} but we do not simply append these ADSs to the states

$s \in \delta(\prod_{i,j}, \alpha^i)$. We apply them step by step until we distinguish all the states in $\delta(\prod_{i,j}, \alpha^i)$ from each other. Step by step application of an ADS corresponds to the application of an ADS symbol by symbol. Therefore, we can detect the symbol which we can distinguish the states and prevent from any unnecessary sequence extension.

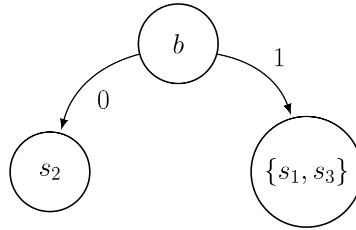


Figure 5.2: Initial Partial ADS Tree

Now, we will explain the whole process with an example. According to the FSM M in Figure 2.2, an ADS for state s_2 is $\alpha = b$. Suppose that the ADS tree generated by LY Algorithm A^{LY} is shown in Figure 2.5. The partial ADS tree after the application of ADS $\alpha = b$ to all states is given in Figure 5.2. There is one set $\prod_{1,1}$ in the partition $\prod_1$ and it includes the states s_1 and s_3. Therefore the states s_1 and s_3 are needed to be distinguished. First we need to find where those states go after the application of $\alpha = b$. We know that $\delta(s_1, b) = s_2$ and $\delta(s_3, b) = s_1$. As a result, the set $\delta(\prod_{1,1}, b)$ becomes $\{s_1, s_2\}$. The ADSs of states s_1 and s_2 from the ADS tree A^{LY} are cc and cc respectively. Since we applied the ADSs step by step to the set $\delta(\prod_{1,1}, b)$, we first apply c to the set $\delta(\prod_{1,1}, b)$. The resulting partial ADS tree is given in Figure 5.3. Since both states respond to c with the output symbol 1, they are not distinguished from each other yet.

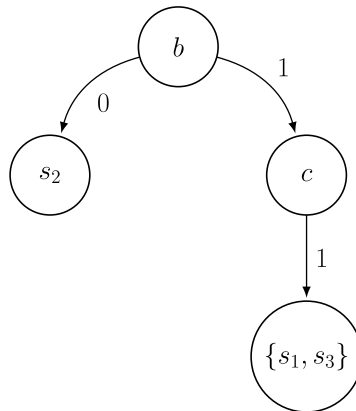


Figure 5.3: A Partial ADS Tree Step 1

Then we apply the next symbol of the ADSs which is also c in this case. Now we have the set $\delta(\prod_{1,1}, bc) = \{s_2, s_3\}$. We apply c to the set $\{s_2, s_3\}$. The output responses of states s_2 and s_3 are $\lambda(s_2, c) = 0$ and $\lambda(s_3, c) = 1$. Therefore, the initial states s_1 and s_3 are managed to be distinguished from each other by the application of bcc. The complete ADS tree is shown in Figure 5.4.

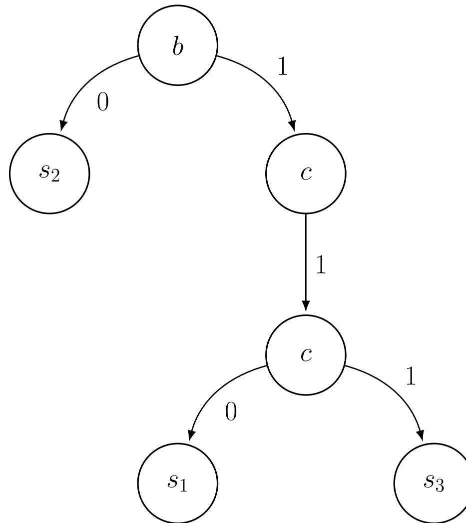


Figure 5.4: A Complete ADS Tree

For the experiments we conduct, we generate ADS trees based on the shortest ADS for each state of the FSM under test. Therefore an FSM M with state number n, has n ADS trees in the beginning where each ADS tree favoring a particular state s by having a shortest ADS for the state s. Nevertheless, it is obvious that it cannot be advantageous to use all of the ADS trees because the cost of cross verification could be quite big. We need to develop an ADS selection algorithm to pick the ADS trees that could reduce the length of the checking sequence by providing shorter ADSs for transition verification and keeping the cost of cross verification at an acceptable level. In the next section, we explain a method to reduce the length of checking sequences by choosing the ADS trees from a given ADS tree pool.

5.3 ADS Tree Selection Algorithm

As stated in Section 5.2, we generate an ADS tree for each state. In this section, we present an algorithm to choose a subset of ADS trees that we can use to generate a

checking sequence shorter than the one generated when a single ADS tree is used.

Having an ADS tree for each state s is advantageous, since it reduces the cost of verification of incoming transitions of s. On the other hand, having a large number of ADS trees increases the cost of cross verification. Therefore there is a trade-off that we need to consider.

One idea to select a subset of ADS trees could be the following. Given a set of ADS trees A, it is possible to design a heuristic that is based on a cost function that estimates the length of a checking sequence if a certain subset $A' \subset A$ is used. The cost function could be based on the sum of the lengths of the required transition and state verification sequences by making use of all ADS trees in A'. However, our initial experiments showed that, such a cost function does not realistically give an estimate on the length of checking sequences. Luckily, checking sequence generation algorithms are quite fast. Therefore, rather than using a cost function to estimate the length of a checking sequence, one can simply generate a checking sequence using all ADSs in A'. We proposed a greedy algorithm that creates subsets of ADSs greedily and generate checking sequences based on this idea.

Let $CS(M, A')$ be an algorithm that constructs a checking sequence for M using all ADS trees in A'. The algorithm to select ADS trees that generates shorter checking sequences is an iterative algorithm and is composed of two phases. Phase 1 finds a pair $A^i, A^j \in A$, such that $CS(M, \{A^i, A^j\})$ is the shortest checking sequence for M among all possible pairs of ADS trees. Phase 2 starts with the best pair $A' = \{A^i, A^j\}$ found in Phase 1, and greedily extends the subset A' by adding a new ADS tree into A'. Formally, as long as there exists an ADS tree $A^k \in A \setminus A'$ such that $CS(M, A' \cup \{A^k\})$ is shorter than $CS(M, A')$, the algorithm extends A'. Among all possible ADS trees $A^k \in A \setminus A'$, the one giving the largest reduction in the length of the checking sequence is chosen, where a tie is broken randomly. After we identify such an ADS tree A^k, the subset A' is updated as $A' = A' \cup \{A^k\}$.

This algorithm does not optimize the set of ADS trees globally, but it makes the best choice at each iteration and it optimizes the length of checking sequence locally. For the rest of the thesis we call this algorithm *Pair ADS Tree Selection Algorithm*. In addition, we use *Single ADS Tree Selection Algorithm* to refer the same algorithm that starts with a single ADS Tree that generates the shortest checking sequence instead of a pair of ADS trees.

Chapter 6

Experimental Results

In this section the experimental results for the checking sequence generation method will be discussed. The methods have been implemented in Java and the experiments have been executed on a machine with 2,5 GHz Intel Core i5 and 4 GB DDR3 RAM.

The FSMs that are used in experiments are generated by using the random FSM generation tool reported in [12]. For the experiments, 10 sets of FSMs are used. Each set of FSMs contains 100 FSMs having number of states n, where n is ranging from 10 to 100 (increasing with a step size of 10). Each FSM has 5 input symbols and 5 output symbols. Also each FSM has an ADS tree. The tool we are using implements LY algorithm [24] to construct an ADS tree and is biased toward finding ADS trees that generally contain repetitions of the same input symbol. But since we need multiple ADS trees, we use the method to construct ADS trees explained in Chapter 5. In this section, we explain how we compare the performance of our method with the method in [2] that uses a single ADS tree. The comparisons will be in terms of the checking sequence length. In addition, we present the experimental results regarding the effect of cross verification and dependencies between ADS trees. We also compare the ADS tree selection algorithms.

6.1 Comparison with Simao et al.s Method

For the experimental results that will be presented in this section, the method presented in Chapter 3 is used to generate a checking sequence. We used the ADS

tree selection algorithms presented in Chapter 5 to find a set of ADS trees to generate the shortest checking sequence. These results are compared with the method in [2]. The experiments show that our method can outperform the method in [2] in most cases.

Table 6.1 shows for each set of 100 FSMs with a number of states ranging from 10 to 100, the average checking sequence length improvement of our method compared to the method in [2]. "Number of FSMs" column stands for the number of FSMs we achieve the improvement. Therefore, the improvements listed in column "% Improvement" is calculated based on the cases where we achieve an improvement, and it shows the average percentage improvement on the length of checking sequences.

Number of States	% Improvement	Number of FSMs
10	12,80	61
20	11,00	61
30	9,27	66
40	8,11	74
50	7,79	78
60	7,01	95
70	6,83	95
80	6,61	100
90	6,32	100
100	5,98	100

Table 6.1: Improvement in CS Lengths

Note that with the increasing number of states the average improvement in the length decreases. It is because the number of states increases the cost of cross verification. To increase the improvement in checking sequence length, we try to increase the effect of transition verification in checking sequence construction. It is previously stated that we have a trade-off between cross verification and transition verification in multiple ADS trees case. In other words, number of ADS trees used increases the cost of cross verification, but the shorter ADSs in ADS trees favoring different states decreases the cost of transition verification. Therefore, to show the effect of shorter ADSs on transition verification cost, we randomly append the transitions to the states of the FSMs we used to gather the results in Table 6.1.

We append 4 transitions to each state of FSMs with different input labels. In other words, we expand the input alphabet of the FSMs by 4. Table 6.2 shows the effect

of additional transitions. With the additional transitions, both the number of FSMs that we observe the improvement and the improvement on checking sequence lengths increase. This is because the cross verification cost is constant while we decrease the transition verification cost.

Number of States	% Improvement	Number of FSMs
10	13,40	69
20	11,50	78
30	10,50	84
40	10,33	83
50	9,42	83
60	8,65	87
70	7,95	96
80	7,49	100
90	6,94	100
100	6,52	100

Table 6.2: Improvement in CS Lengths with 4 additional input symbols

To support this result that backs up our idea about the effect of shorter ADSs on transition verification cost, we again append 4 more transitions to each state of FSMs with different input labels and run the same tests on them. The result is shown in Table 6.3. After appending more transitions we observe further improvements in our results. Therefore, we conclude that our method using multiple ADS trees performs better with the increasing number of transitions, because shorter ADSs are advantageous for transition verification.

Number of States	% Improvement	Number of FSMs
10	14,60	78
20	12,50	82
30	11,30	92
40	10,90	93
50	10,40	93
60	9,08	94
70	8,16	97
80	8,08	100
90	7,52	100
100	6,98	100

Table 6.3: Improvement in CS Lengths with 8 additional input symbols

Number of States	% Improvement	% Improvement (+4)	% Improvement (+8)
10	-6,18	-4,6	-4,45
20	-5,65	-3,92	-3,72
30	-3,55	-2,72	-2,41
40	-3,21	-2,39	-2,04
50	-4,17	-3,25	-2,74
60	-4,55	-3,22	-2,87
70	-5,09	-4,26	-3,42
80	-	-	-
90	-	-	-
100	-	-	-

Table 6.4: Experimental results for FSMs without an improvement

In Table 6.4, we present the results regarding the cases where we do not observe a reduction in the length of checking sequences. These results show the average percentage increase in the length of checking sequences for these cases. We know that as the number of input symbols of an FSM increases, the improvement percentage on the length of checking sequences increases from the previous experimental results. This statement can also be supported by the results in Table 6.4, where we see that as the number of input symbols increases, the average increase on the length of checking sequence decreases.

We also know that as the number of states increases, the improvement percentage decreases from the previous results. For the cases we fail to observe an improvement, this statement still holds. As the number of states increases, the average percentage increase on the length of checking sequences decreases for most of the cases. For FSMs with 50 or more states, this statement does not hold. This is because for such FSMs, the average is calculated based on a small number of FSMs since as the number of states increases, the number of cases we fail to observe an improvement decreases.

6.2 Contribution of Pair ADS Tree Selection Algorithm

One can ask the question that "Why the ADS tree selection algorithm starts with the pair of ADS trees?". First, we conduct the same sequence of tests with the single ADS tree selection algorithm that is same with pair ADS tree selection algorithm but it starts with a single ADS tree not a pair of ADS trees. The experiments show that when we start the ADS selection algorithm with a single ADS tree, we obtain lesser improvement.

Table 6.5, 6.6 and 6.7 correspond to the same set of experiments above. The improvement we obtain with a single ADS tree selection algorithm shows the same characteristics with the pair ADS tree selection algorithm with respect to increasing number of states and increasing number of transitions. The improvement decreases with the increasing number of states and increases with the additional transitions. However, overall improvement is less than the pair ADS tree selection algorithm. Therefore, the idea is that to approach to the globally optimal solution, it is better to start the ADS tree selection algorithm with a combination of ADS trees with cardinality bigger than 1. In other words, our algorithm could perform better if we start the ADS tree selection algorithm with a set of 3 ADS trees.

Number of States	% Improvement	Number of FSMs
10	8,82	49
20	7,51	57
30	6,61	65
40	5,48	63
50	4,97	51
60	4,20	48
70	4,41	42
80	2,56	89
90	2,45	98
100	2,02	98

Table 6.5: Improvement in CS Lengths (Single ADS Tree Selection Algorithm)

This result can also be supported by the statistics shown in Table 6.8. Table 6.8 shows the percentage of the cases where the single ADS tree that generates a shortest checking sequence is included in the set of multiple ADS trees that is found by

pair ADS tree selection algorithm. According to this, starting ADS tree selection algorithm with a pair of ADS trees makes a drastic difference.

Number of States	% Improvement	Number of FSMs
10	11,83	63
20	8,62	80
30	8,62	86
40	7,36	92
50	6,36	85
60	5,92	85
70	5,52	88
80	4,66	100
90	4,25	100
100	4,22	98

Table 6.6: Improvement in CS Lengths with 4 additional input symbols (Single ADS Tree Selection Algorithm)

Number of States	% Improvement	Number of FSMs
10	12,24	74
20	8,97	89
30	9,71	95
40	8,60	96
50	7,54	94
60	6,64	95
70	6,56	95
80	6,12	100
90	5,37	100
100	5,28	100

Table 6.7: Improvement in CS Lengths with 8 additional input symbols (Single ADS Tree Selection Algorithm)

Table 6.8 shows the percentage of the ADS trees used to generate shortest checking sequence by the method in [2] included in the set of ADS trees used to generate shortest checking sequence with our method. The percentage decreases with the increasing number of states. With the increasing number of states, we generate ADS trees as many as state number. Therefore, the possibility of the single ADS tree being included in the set of multiple ADS trees is decreasing.

Number of States	Percentage of Single ADS Inclusion
10	33,75
20	15,62
30	14,45
40	11,0
50	10,75
60	9,80
70	10,25
80	8,70
90	7,00
100	5,75

Table 6.8: Percentage of single ADS tree included in multiple ADS trees

6.3 The Negative Effect Of Cross Verification

We stated that the improvement decreases with the increasing number of states of FSMs because of the increasing cost of cross verification. We want to explore the effect of cross verification more. Therefore, we conduct the same set of experiments by ignoring the cross verification. In other words, we altered the algorithm such that for the d-recognition of the node, it is enough to d-recognize the node by a single ADS A_i^j. Without cross verification the improvement of our method beats the algorithm in [2] with marked difference. Results are shown in Table 6.9.

Number of States	% Improvement with no additional transitions	% Improvement with 4 additional transitions	%Improvement with 8 additional transitions
10	34,48	60,99	72,63
20	28,12	54,88	69,11
30	25,85	54,32	68,54
40	22,23	53,57	63,49
50	20,45	53,90	57,85
60	19,42	51,00	55,98
70	19,33	49,80	53,81
80	21,16	46,18	50,89
90	18,83	43,12	49,34
100	15,12	40,65	46,67

Table 6.9: Improvement in CS Lengths without Cross Verification

6.4 Contributions of Phase 1 and Phase 2

In this section, we will analyze experimental results of our method only and present the contributions of Phase 1 and Phase 2 to the average checking sequence length. Table 6.10 shows the contribution Phase 2 to the average checking sequence length. The percentage contribution of Phase 2 to the checking sequence length increases with the size of the FSM. The average percentage contribution of Phase 2 to the length of the checking sequence seems to be around 25%.

Number of States	Percentage Contribution of Phase 2	Average Number of ADS trees used
10	2,93	3,02
20	6,96	3,79
30	13,48	5,3
40	19,50	6,17
50	23,03	6,15
60	28,08	6,55
70	26,63	5,02
80	35,57	7,14
90	38,06	8,48
100	46,69	9,0

Table 6.10: Contribution of Phase 2 to CS Length

The Table 6.10 also shows the average number of ADS trees used. The number of ADS trees used increases with the size of the FSM. Therefore, it increases the dependency between ADS trees. This might be the reason for the percentage contribution of Phase 2 to the checking sequence length increases with the size of the FSM because Phase 2 is responsible for breaking the dependencies between ADS trees.

Chapter 7

Conclusion and Future Work

In this thesis, three aspects of FSM based testing is addressed.

First contribution of this thesis is the formulation of the problem of finding a shortest adaptive distinguishing sequence for a state q of an FSM M using Answer Set Programming. It is an NP-Hard problem and the Answer Set Programming is used to solve this optimization problem. Two different ASP formulations are given. In this way, we utilize the construction of ADS trees and create the ADSs suitable for our needs.

Another contribution of the thesis is a method that can answer the following question: Given an input output sequence X/Y and a set of ADS trees for an FSM M, is X/Y a checking sequence for M which is generated by using the proposed method that uses state recognition techniques already existing in the literature, such as d- and t-recognition. However we also introduce some novel state recognition methods for multiple state identification sequences. Although using multiple state identification sequences increases the cost of cross verification, we showed that for most of the cases we decrease the length of checking sequence by taking advantage of using shorter ADSs for transition verification.

The major contribution of the thesis is a new adaptive distinguishing sequence based checking sequence generation algorithm. Our method is based on a local optimization. It provides locally optimal decisions based on the concept of the recognition automaton. By using recognition automaton, we can detect the best extensions that shorten the checking sequence. Our method consists of two phases, in the first

phase a sequence is generated with little consideration in state recognition. If the sequence generated in the first phase is not a checking sequence then it is extended to a checking sequence in Phase 2.

We stated that there are many methods to build checking sequences based on ADS trees. These methods all require a single ADS tree to be given, and they are mute to the potential ADS trees that can be used for generating a checking sequence. They generally focus on generating as good a checking sequence as possible, given the selected ADS. It is not exceptional that an FSM that has one ADS actually has more than one. Thus, it is interesting to question the choice of a particular ADS for the entire checking sequence. Indeed, the different ADS trees of a given FSM may have different properties that would be interesting to exploit for the construction of the checking sequence. The most obvious one is that a given ADS tree might be quite short for some of the states and longer for others, while another ADS tree might be the opposite. Major contribution of our method is to use ADS trees that best suit our goal of a shorter checking sequence at different points in the checking sequence construction.

The experimental results have shown that our method achieves a reduction in the length of the checking sequence over the method presented in [2]. We think that, there is still a room for further improvement using our method. The experiments show that approximately 25% of the checking sequence length stem from the extensions in Phase 2 and this extension length can be reduced. In Phase 2 of the algorithm, we implemented a very simple idea to extend the sequence to a checking sequence and it does not investigate the dependency between ADS trees globally. However a closer analysis of the final form of the recognition automaton may actually yield shorter extensions required. As a future work, we want to find some good heuristics that makes these extensions more cleverly. It may also be worthwhile to reconsider our eager and careless conditional state recognition approach in Phase 1. In addition, the experiments showed that without considering cross verification, our method achieves much better results. Therefore, another research direction could be investigating a clever way to cross verify the nodes.

We also stated that we estimate a cost function to calculate the optimal ADS tree set. The experiments showed that the cost function does not reveal realistic results. Therefore, we used ADS tree selection algorithms that generate the checking sequence while selecting the optimal set of ADS trees. Obviously, this is time

consuming. However, one of the research directions could be the detection of the characteristics of the optimal set of ADS trees by working on the FSMs to estimate a better cost function.

Another promising research direction seems to be the generation of ADS trees that optimizes the checking sequence length. In our work, we generate ADS trees that favor a state of the FSM. This is feasible for the FSMs with a number of states less than 100. But for an FSM with extremely large number of states, this is not practical. Therefore, ADS tree generation should be different for larger FSMs. For example, the ASP formulation of ADS can be adjusted to generate shortest ADSs for a group of states and selection of the group of states can be optimized.

Bibliography

[1] Alfred V. Aho, Anton T. Dahbura, David Lee, and M. mit Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991.

[2] Adenilso Sim ao and Alexandre Petrenko. Generating checking sequences for partial reduced finite state machines. pages 153–168. 2008.

[3] F. Belina and D. Hogrefe. The ccitt-specification and description language sdl. *Comput. Netw. ISDN Syst.*, 16(4):311–341, March 1989.

[4] Aysu Betin-Can and Tevfik Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 248–257, Washington, DC, USA, 2004. IEEE Computer Society.

[5] R. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley, 2000.

[6] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, December 2011.

[7] E. Brinksma. *A Theory for the Derivation of Tests*. University of Twente, Department of Computer Science, 1988.

[8] S. Budkowski and P. Dembinski. An introduction to estelle: A specification language for distributed systems. *Comput. Netw. ISDN Syst.*, 14(1):3–23, March 1987.

[9] Jessica Chen, Robert M. Hierons, Hasan Ural, and Hsn Yenign. Eliminating redundant tests in a checking sequence. In Ferhat Khendek and Rachida Dssouli,

editors, *TestCom*, volume 3502 of *Lecture Notes in Computer Science*, pages 146–158. Springer, 2005.

[10] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978.

[11] A.T. Dahbura, K.K. Sabnani, and M.U. Uyar. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE*, 78(8):1317–1326, Aug 1990.

[12] Emre Dincturk. A two phase approach for checking sequence generation, 2009.

[13] E.F. Moore. Gedanken-experiments on sequential machines. In C.E. Shannon and J. MacCarthy, editors, *Automata Studies*, pages 129–153, Princeton, New Jersey, 1956. Princeton University Press.

[14] A.D. Friedman and P.R. Menon. *Fault detection in digital circuits.* Computer applications in electrical engineering series. Prentice-Hall, 1971.

[15] Martin Gebser, Benjamin Kaufmann, Andr Neumann, and Torsten Schaub. A conflict-driven answer set solver. In *LPNMR*, pages 260–265, 2007.

[16] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. pages 365–386, 1991.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[18] May Haydar, Alexandre Petrenko, and Houari Sahraoui. Formal verification of web applications modeled by communicating automata. In David de Frutos-Escrig and Manuel Nez, editors, *Formal Techniques for Networked and Distributed Systems FORTE 2004*, volume 3235 of *Lecture Notes in Computer Science*, pages 115–132. Springer Berlin Heidelberg, 2004.

[19] F. C. Hennine. Fault detecting experiments for sequential circuits. In *Proceedings of the 1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '64, pages 95–110, Washington, DC, USA, 1964. IEEE Computer Society.

[20] R. M. Hierons and H. Ural. Correction to: Reduced length checking sequences. *IEEE Transactions on Computers*, 58(2):287–287, 2009.

[21] Rob M. Hierons and Hasan Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629, 2006.

[22] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[23] Zvi Kohavi. *Switching and Finite Automata Theory: Computer Science Series.* McGraw-Hill Higher Education, 2nd edition, 1990.

[24] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320, March 1994.

[25] D. Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996.

[26] Vladimir Lifschitz. What is answer set programming? In *AAAI*, pages 1594–1597, 2008.

[27] S. Naito and M. Tsunoyama. *Ilth IEEE Fault Tolerant Comput. Symp.*

[28] Alexandre Petrenko, Adenilso da Silva Simão, and Nina Yevtushenko. Generating checking sequences for nondeterministic finite state machines. In *ICST*, pages 310–319, 2012.

[29] Krishan K. Sabnani and Anton T. Dahbura. A protocol test generation procedure. *Computer Networks*, 15:285–297, 1988.

[30] Uraz Cengiz Türker and Hüsnü Yenigün. Hardness and inapproximability of minimizing adaptive distinguishing sequences. pages 264–294, 2014.

[31] Hasan Ural, Xiaolin Wu, and Zhang Fan. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93, 1997.

[32] Hasan Ural and Fan Zhang. Reducing the lengths of checking sequences by overlapping. In M. mit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2006.

[33] B. Yang and H. Ural. Protocol conformance test generation using multiple uio sequences with overlapping. *SIGCOMM Comput. Commun. Rev.*, 20(4):118–125, August 1990.