A TWO PHASE APPROACH FOR CHECKING SEQUENCE GENERATION

by

MUSTAFA EMRE DİNÇTÜRK

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

August 2009

A TWO PHASE APPROACH FOR CHECKING SEQUENCE GENERATION

APPROVED BY:

Assist. Prof. Dr. Hüsnü Yenigün, (Thesis Supervisor)

. . . .

Prof. Dr. Kemal İnan

. . . .

Assoc. Prof. Dr. Albert Levi

. . . .

Assoc. Prof. Dr. Tonguç Ünlüyurt

. . . .

Assoc. Prof. Dr. Berrin Yanıkoğlu

. . . .

DATE OF APPROVAL: . . . .

© Mustafa Emre Dinçtürk 2009
All Rights Reserved

A TWO PHASE APPROACH FOR CHECKING SEQUENCE GENERATION

Mustafa Emre Dinçtürk
Computer Science and Engineering, Master's Thesis, 2009
Thesis Supervisor: Hüsnü Yenigün

Keywords: FSM based testing, Checking Sequence, Random FSM Generation

Abstract

A new method for constructing a checking sequence for finite state machine (FSM) based testing is introduced. It is based on a recently suggested method which uses quite a different approach than almost all the methods developed since the introduction of the checking sequence generation problem around half a century ago. Unlike its predecessor which aggressively tries to recognize the states by applying identification sequences, our approach relies on yet to be generated parts of the sequence for this. The method may terminate without producing a checking sequence. We also suggest a method to check if a sequence is a checking sequence for this purpose. If it turns out not be a checking a sequence, a post processing phase extends the sequence further. We present the results of an experimental study showing that our two phase approach produces shorter checking sequences than the previously published methods. This experimental study is performed on FSMs that are randomly generated by using a tool implemented within this work to support this and other FSM based testing studies.

KONTROL DİZİSİ ÜRETİMİ İÇİN İKİ AŞAMALI BİR YAKLAŞIM

Mustafa Emre Dinçtürk
Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2009
Tez Danışmanı: Hüsnü Yenigün

Anahtar Kelimeler: SDM Bazlı Sınama, Kontrol Dizileri, Rastlantısal SDM Üretimi

Özet

Bu çalışmada Sonlu Durum Makinaları (SDM) bazlı sınamada yeni bir kontrol dizisi üretim yöntemi verilmektedir. Bu yöntem, yakın geçmişte öne sürülen ve problemin yaklaşık yarım asır önce ortaya konuluşundan beri kullanılan tüm yöntemlerden farklı bir yaklaşıma sahip yeni bir yöntemi temel almaktadır. Yenilik olarak, agresif bir şekilde durum belirleme dizileriyle durumların tanınması yerine, kontrol dizisine daha sonra yapılacak eklentilerin bu sorunu çözeceği öngörülmektedir. Ancak bu yöntemin kontrol dizisi üretememe ihtimali bulunmaktadır. Bu nedenle yine bu çalışma içerisinde verilen bir dizinin kontrol dizisi olup olmadığını kontrol eden bir yöntem de geliştirilmiştir. Eğer üretilen dizinin bir kontrol dizisi olmadığı anlaşılırsa, dizi ikinci bir aşamada tekrar ele alınıp yapılan eklentilerle bir kontrol dizisı haline getirilmektedir. Bu çalışmada yeni yöntemin mevcut yöntemlere göre daha kısa kontrol dizileri ürettiğini gösteren deneysel çalışmalar da sunulmaktadır. Bu deneysel çalışmalarda kullanılan Sonlu Durum Makinaları yine bu çalışma süresinde gerçekleştirilmiş bir rastlantısal SDM üretme aracı kullanılarak üretilmiştir.

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.

I would like to thank Gülden Sarıcalı and Birol Yüceoğlu for giving me encouragement and motivation.

I would like to thank TUBİTAK for the financial support provided.

Table of Contents

1 Introduction **1**

2 Preliminaries **4**

2.1 FSM Fundamentals 4

2.1.1 Extending Next State and Output Functions 4

2.1.2 Some Properties of FSMs 5

2.2 Representing an FSM by a Directed Graph 5

2.3 Distinguishing Sequences 6

2.3.1 Preset Distinguishing Sequence 7

2.3.2 Distinguishing Set (Adaptive Distinguishing Sequence) 7

2.4 Checking Sequences based on Distinguishing Sequences 7

3 Random FSM Generation **10**

3.1 Component Graph 11

3.2 Free Edge and Set of Free Edges 12

3.2.1 Existence of a Free Edge in a Strongly Connected Graph . . . 12

3.2.2 Existence of a Free Edge in a not Strongly Connected Graph . 14

3.3 Forcing Strongly Connectedness 15

3.3.1 Finding a Set of Free Edges in a Component 15

3.3.2 Making a Graph Strongly Connected 17

3.4 Forcing Initial Reachability 20

3.4.1 Method 1: Using a Backbone Component Graph 21

3.4.2 Method 2: Generate an Initial Reachable Graph with Random Components 24

3.5 Shuffling 25

3.6 Providing Input/Output Probabilities 26

4 Checking if a Sequence is a Checking Sequence 27

4.1 Introduction 27

4.2 Uncertainty Automaton 28

4.3 State Recognition Using Uncertainty Automaton 29

4.3.1 Candidate Elimination Using Incompatible Sets 32

4.3.2 Candidate Elimination Using Candidate Trial 38

4.3.3 Using Candidate Elimination Methods Together 43

4.4 Thoughts on Uncertainty Automaton 43

5 Overview of Simão *et al.*'s Method 45

6 Our Checking Sequence Generation Method 47

6.1 Phase 1: Sequence Generation 47

6.2 Phase 2: Extending Sequence Q to a Checking Sequence 52

6.3 Experimental Results 54

6.3.1 Comparison with *Simão et al.*'s Method 55

6.3.2 Contributions of Phase 1 and Phase 2 59

6.3.3 Effect of Candidate Elimination Using a Set of Incompatible Nodes 60

7 Conclusion 65

List of Figures

2.1 FSM M_1 6

4.1 Initial Uncertainty Automaton 29
4.2 Uncertainty Automaton after nodes merged 31
4.3 Copy Uncertainty Automaton 40
4.4 Uncertainty Automaton 41
4.5 Uncertainty Automaton 42
4.6 Final Uncertainty Automaton 42

6.1 FSM M_2 53
6.2 Final Uncertainty Automaton for Q generated in Phase 1 54
6.3 Final Uncertainty Automaton for $Q' = Qbab$ 55
6.4 Average CS Lengths 56
6.5 Our Method's CS Lenghts as a Box Plot 58
6.6 Average Improvements Over *Simão et al.*'s Method 60
6.7 Improvements Over *Simão et al.*'s Method as a Box Plot 61
6.8 Average Method Execution Times 62
6.9 Contributions of Phase 1 and Phase 2 to CS Length 62
6.10 Percentage Contribution of Phase 2 CS Length 63
6.11 Distribution of Execution Time between Phase 1 and Phase 2 63
6.12 Effect of Candidate Elimination Using a Set of Incompatible Nodes on Length 64
6.13 Effect of Candidate Elimination Using a Set of Incompatible Nodes on Time 64

List of Tables

4.1 Candidate Sets For the Uncertainty Automaton in Figure 4.1 after d-recognition 30
4.2 Candidate Sets For the Uncertainty Automaton in Figure 4.2 31
4.3 Incompatible Sets For the Uncertainty Automaton in Figure 4.2 . . . 33
4.4 Candidate Sets for the Uncertainty Automaton in Figure 4.2 36
4.5 Candidate Sets For the Uncertainty Automaton in Figure 4.4 41
4.6 Incompatible Sets For the Uncertainty Automaton in Figure 4.4 . . . 41
4.7 Candidate Sets For the Uncertainty Automaton in Figure 4.5 42
4.8 Candidate Sets For the Uncertainty Automaton in Figure 4.6 43

6.1 Iteration 1 50
6.2 Iteration 2 51
6.3 Iteration 3 51
6.4 Iteration 4 51
6.5 Iteration 5 52
6.6 Candidate Sets For the Uncertainty Automaton in Figure 6.2 53
6.7 Candidate Sets For the Uncertainty Automaton in Figure 6.3 54
6.8 Average CS Lengths 56
6.9 Average Improvements Over *Simão et al.*'s Method 59

Chapter 1

Introduction

A Finite State Machine (FSM) is an abstract structure with a finite set of states where application of an input causes a state transition along with the production of an output. FSMs are widely used to model systems in diverse areas such as sequential circuits, communication and software protocols[4, 1, 7, 2, 21, 23, 18]. Many systems are implemented using FSM based models. As these systems became more complicated and large, the research for techniques to ensure the reliability of these systems gained importance. FSM based testing is a research area that is motivated to answer these reliability demands.

In conformance testing, the aim is to ensure that an implementation conforms to its specification. In other words, conformance testing tries to answer the question if an implementation, that is intended to implement some specification, is a correct implementation of its specification or not. When the specification of a system is modeled as an FSM M then the implementation can also be considered as an FSM N and the question becomes whether N is equivalent to M. By equivalence of FSMs it is meant that if for any sequence of inputs that is defined in M, N produces the same sequence of outputs as M. An Implementation Under Test (IUT) is considered to be a black box. That is IUT is an FSM N with unknown transitions but it is generally assumed to have 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 some inputs and observe the outputs produced by the IUT. Using only this output observation the correct functioning of IUT is tried to be deduced by comparing the outputs produced by the IUT against the expected outputs produced by the

specification FSM M. An input sequence that can determine if IUT is a correct or faulty implementation of specification M is called a *checking sequence*.

An important problem in conformance testing is state verification. That is, a mechanism is needed to know in which state the IUT is. This is necessary since a checking sequence has to verify every transition of the specification FSM and verification of a transition requires verification of the initial and the final states of a transition. That is we need to know that IUT is in the correct state before an input is applied (so that the we can know which output to expect) and reaches to the correct state after the input is applied. State verification problem can be solved using *Preset Distinguishing Sequence (PDS)* [9], *Unique Input Output (UIO) sequence* [22] and *Characterizing Set* [9]. A PDS is an input sequence that produces different outputs for different states. Therefore if the specification FSM has a PDS, then the state verification problem is solved easily by applying the PDS at the state to be verified. However not every minimal FSM has a PDS [15] and to determine if an FSM has a PDS is a PSPACE-complete problem [16].

According to the survey in [17], the literature of conformance testing begins in 1950's. In 1956 Moore's paper on machine identification problem was published [19]. In his paper, he studied the problem of obtaining the state diagram of an unknown FSM with given number of states by only observing its input output behavior. He also stated the conformance testing problem. In 1964, Hennie proposed a method using PDS for generating a checking sequence with length polynomial in length of PDS and machine size [10]. Hennie's method that uses PDS to generate checking sequences is called D-method. He also gave an algorithm that generates exponentially long checking sequences for the case when a distinguishing sequence cannot be found. Later several other checking sequence generation methods that are based on UIO sequences, characterizing sets and transition tours were proposed. These methods are called U-Method [22], W-Method [4] and T-Method [20] respectively.

Although there were some studies in 70's and 80's, conformance testing became a more active research area in the beginning of 90's thanks to applications in testing communication protocols. Especially distinguishing sequence based methods became popular. The studies were focused on the improvement of previous methods using global optimization techniques. In [2], using a graph theoretical approach, the

checking sequence generation problem modeled as a Rural Chinese Postman Problem. In [14, 11] this optimization model was further improved. In addition to that, in [3] it is shown that some transition verification sequences could be eliminated from the optimization model and in [26] the model is improved to produce shorter checking sequences by making use of overlapping of distinguishing sequences. In [24], Simão *et al.* proposed an approach that is different than previous work. Instead of trying global optimization, they designed an algorithm that makes local optimization. With this approach, they achieved better results than global optimization methods in most cases.

The contributions of this thesis to the conformance testing are threefold. First we present the details of a tool that generates random FSMs that we require to measure and compare the performances of checking sequence generation methods. Second we present a method that attempts to determine if a given input sequence is a distinguishing sequence based checking sequence or not. Lastly we present a method that generates distinguishing sequence based checking sequences. Our method is basically a modification of Simão *et al.*'s method. Experiments show that our method achieves an average reduction of at least 7% in checking sequence length compared to Simão *et al.*'s method.

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, the details of our random FSM generation tool is provided. In Chapter 4, our method to check if a given sequence is a DS based checking sequence is presented in detail. In Chapter 5, an overview of the Simão *et al.*'s checking sequence generation method from [24] is provided. In Chapter 6, we present details of our checking sequence generation method together with experimental results. Finally Chapter 7 contains the concluding remarks.

Chapter 2

Preliminaries

2.1 FSM Fundamentals

An FSM (*finite state machine*) is specified by a tuple $M = (S, s_1, I, O, \delta, \lambda)$ where

- $S = \{s_1, s_2, \ldots, s_n\}$ is the *finite set of states* and n is the number of states
- $s_1 \in S$ is the *initial state*
- I is the *finite set of inputs*
- O is the *finite set of outputs*
- $\delta : S \times I \rightarrow S$ is the *next state function*
- $\lambda : S \times I \rightarrow O$ is the *output function*

For two states s_i and s_j, an input x and an output y if $\delta(s_i, x) = s_j$ and $\lambda(s_i, x) = y$ then intuitively this means the machine M performs a transition from state s_i to state s_j when input x is applied and it produces output y as a response to this input. We will also denote such a transition by $(s_i, s_j; x/y)$.

An input symbol $x \in I$ is *defined* at state s if $\delta(s, x)$ and $\lambda(s, x)$ are defined.

2.1.1 Extending Next State and Output Functions

The next state function δ and the output function λ can be extended to sequences as follows. Let $x \in I$ be an input symbol and $X \in I^*$ be an input sequence and let $xX \in I^*$ denote the input sequence obtained by concatenation of x and X

(that is juxtaposition of input (output) sequences and input (output) symbols mean concatenation) then

- $\delta(s, xX) = \delta(\delta(s, x), X)$ and

- $\lambda(s, xX) = \lambda(s, x)\lambda(\delta(s, x), X)$

For the empty sequence ε we define $\delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$. An input sequence $X = x_1x_2 \ldots x_r \in I^*$ is *defined* at state s if $\forall 1 \leq i \leq r, x_i$ is defined at $\delta(s, x_1x_2 \ldots x_{i-1})$

2.1.2 Some Properties of FSMs

An FSM M is

- *deterministic* if for each state $s \in S$ and for each input symbol $x \in I$, M has *at most one* transition with start state s and input symbol x. Since the transitions of an FSM are defined by a function, in our setting an FSM is always deterministic. For nondeterministic machines, relations are used instead of functions.

- *completely specified* if for each state $s \in S$ and for each input symbol $x \in I$, $\delta(s, x)$ and $\lambda(s, x)$ are defined, that is when δ and λ are total functions.

- *minimal* if for any two different states $s_i, s_j \in S$, there is an input sequence $X \in I^*$ such that $\lambda(s_i, X) \neq \lambda(s_j, X)$.

- *initially reachable* if for each $s_i \in S$ there exists some input sequence $X \in I^*$ such that $\delta(s_1, X) = s_i$ (i.e. each state $s_i \in S$ is reachable from the initial state s_1)

2.2 Representing an FSM by a Directed Graph

An FSM M can be represented by a directed graph $G = (V, E)$ with set of vertices V and a set of directed edges E. In such a graph, each edge $e = (v_j, v_k; x/y) \in E$

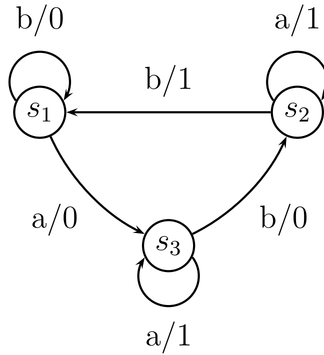

Figure 2.1: FSM M_1

with *label* x/y represents a transition $t = (s_j, s_k; x/y)$ from s_j to s_k with input x and output y. We will also use (v_j, v_k) to denote an edge when the edge label is not important. The vertices v_j and v_k of e are called *start* and *end* of e respectively and it is said that e *leaves* v_j and *enter*s v_k. Two edges e_j and e_k are called *adjacent* if end of e_j and start of e_k are same.

Any sequence of adjacent edges (not necessarily distinct) is called a *path*. We will denote a path $(n_1, n_2; x_1/y_1)(n_2, n_3; x_2/y_2) \ldots (n_r, n_{r+1}; x_r/y_r)$ as $P = (n_1, n_{r+1}; X/Y)$ where $X = x_1 x_2 \ldots x_r$ and $Y = y_1 y_2 \ldots y_r$. The nodes n_i correspond to vertices of G. Node n_1 is the *start* of P and n_{r+1} is the *end* of P. Input output sequence X/Y is called the *label* of P and X/Y is a *transfer sequence* from v_1 to v_r. X is the input portion and Y is output portion of X/Y respectively.

In graph G, a vertex v_k is *reachable* from vertex v_j, represented as $v_j \rightsquigarrow v_k$, if there exists a path P such that start of P is v_j and end of P is v_k. G is *strongly connected* if $\forall v_j, v_k \in V$, $v_j \rightsquigarrow v_k$ is satisfied. An FSM is *strongly connected*, if the digraph representing it is strongly connected.

2.3 Distinguishing Sequences

The checking sequence generation methods that will be discussed in this thesis require existence of a distinguishing sequence. Distinguishing sequences are special sequences used for state identification. Throughout thesis the phrase *identification sequence* always refers to distinguishing sequence. There are two types of distinguishing sequences that are explained next.

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_1 shown in Figure 2.1.

- $\lambda(s_1, ab) = 00$
- $\lambda(s_2, ab) = 11$
- $\lambda(s_3, ab) = 10$

2.3.2 Distinguishing Set (Adaptive Distinguishing Sequence)

A *Distinguishing Set* (or *Adaptive Distinguishing Sequence* – ADS) is multi-set of input sequences $\bar{D} = \{D_{s_1}, D_{s_2}, \ldots, D_{s_n}\}$ such that for any pair $D_{s_i}, D_{s_j} \in \bar{D}$ there exists a common prefix α of D_{s_i} and D_{s_j} such that $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$. The sequence D_{s_i} is called the ADS of state s_i.

For example, $\bar{D} = \{D_{s_1}, D_{s_2}, D_{s_3}\}$, where $D_{s_1} = a$ and $D_{s_2} = D_{s_3} = ab$, is a distinguishing set for FSM M_1 in Figure 2.1.

Note that PDS is a special case of ADS where for all states $D_{s_i} = D$. Therefore every FSM which has a PDS also has a distinguishing set. However the inverse is not true. That is there exist FSMs with a distinguishing set but no PDS. Compared to PDS, distinguishing sets have some advantages. Determining the existence of a distinguishing set and finding one if exist is polynomial in number states and number of inputs [16].

2.4 Checking Sequences based on Distinguishing Sequences

Let M be a completely specified, minimal, deterministic and strongly connected FSM that is represented by directed graph $G = (V, E)$. Also let $\Phi(M)$ be the set of FSMs such that each FSM $N \in \Phi(M)$ has at most as many states as M and has the same input and output sets as M. FSMs M and N are said to be *equivalent* if there does not exist an input sequence X such that $\lambda(s_1^M, X) \neq \lambda(s_1^N, X)$ where s_1^M

and s_1^N are the initial states of M and N respectively. If such an input sequence X exists then X is said to *distinguish* M and N. A *checking sequence* of M is an input sequence such that it distinguishes M from every FSM $N \in \Phi(M)$ that is not equal to M. Hence in the context of conformance testing, when checking sequence is applied on any faulty implementation N in $\Phi(M)$ the output produced by N will be different than the output produced by specification M.

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 and tries to show that if $(s_j, s_k; x/y)$ is a transition in M then N has a corresponding transition $(f(s_j), f(s_k); x/y)$. Thus testing using a checking sequence requires the concepts of state recognition and transition verification defined. We will define these concepts using distinguishing sequence of FSM M as follows.

Let $P = (n_1, n_2; x_1/y_1)\ (n_2, n_3; x_2/y_2) \ldots (n_r, n_{r+1}; x_r/y_r)$ be a path in G from n_1 to n_{r+1} with the label $X/Y = x_1x_2 \ldots x_r/y_1y_2 \ldots y_r$. Also let $\bar{D}$ be a distinguishing set of M. There are two types of recognition that we will define here, namely *d-recognition* and *t-recognition* [25]. A vertex 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_i of P is *d-recognized* as state s of M if n_i is start of a subpath of P with label $D_s/\lambda(s, D_s)$
- a node n_i of P is *t-recognized* as state s of M if there are two subpaths $(n_q, n_i; X'/Y')$ and $(n_j, n_k; X'/Y')$ of P such that n_q and n_j are *recognized* as s' of M, n_k is *recognized* as state s of M

In addition to that a transition verification is defined as follows. A transition $t = (s, s'; x/y)$ of M is *verified* (in P) if there is an edge $(n_i, n_{i+1}; x'/y')$ of P such that nodes n_i and n_{i+1} are recognized as states s and s' of M respectively and $x'/y' = x/y$.

The following theorem from [25] (rephrased in our notation) states a sufficient condition for a checking sequence.

Theorem 1. *Let X/Y be the label of a path P of directed graph G (for FSM M) such that every transition is verified in P. Then X (i.e. the input portion of label*

of P) forms a checking sequence *of M.*

Chapter 3

Random FSM Generation

Measuring and comparing the performances of a checking sequence generation algorithms generally require experimentation of the method on a set of FSMs. All checking sequence generation methods, including the methods discussed in this thesis, require these FSMs to have some properties. For example a method may require an FSM to be deterministic, completely specified, strongly connected, minimal and having a preset distinguishing sequence. Since these FSMs will be used for experimental purposes, it is also very important for the FSMs to have the element of randomness in their structure as much as possible so that they are still able to represent all possible FSMs with desired properties in a just manner. For this reason, we developed a tool that can generate deterministic and completely specified random FSMs with given number of states, number of input symbols and number of output symbols and having any of the following properties listed below

- Being strongly connected (or not)
- Being initially reachable (or not)
- Being minimal (or not)
- Having a preset distinguishing sequence (or not)
- Having an adaptive distinguishing sequence (or not)

Among these properties, strongly connectedness, initial reachability and having preset distinguishing sequence turned out to be very difficult to satisfy when it is left to pure chance. In other words, assigning transitions randomly between states was

not very efficient to generate FSMs with mentioned properties. Thus for these properties, after initial assignments of the transitions, the tool allows a post processing step to be applied on the generated random FSM to force the FSM to have the desired property. In the following sections the details of this post processing steps are explained for each property. However before examining post processing, below is the process of initial assignment of transitions explained as pseudo code.

Algorithm 1: Random Assignment of Transitions

Input: S finite set of states

Input: I finite set of input symbols

Input: O finite set of output symbols

Output: T list of transitions of a completely specified, deterministic FSM with randomly assigned transitions

```
1 T = ∅;
2 foreach state s ∈ S do
3     foreach input x ∈ I do
4         choose a random output symbol y from O;
5         choose a random destination state s' from S;
6         T = T ∪ {(s, s'; x/y)};
```

Since a new transition is created for each state and input symbol pair, the complexity of random assignment of transitions is $O(np)$ where $|S| = n$ and $|I| = p$.

3.1 Component Graph

The *component graph*, sometimes called *condensation*, of a digraph G is directed acyclic graph that have a vertex for each strongly connected component of G and the edges in component graph represents the connectivity between these components. A more formal definition is given below.

Definition 1. *Assuming that there are m strongly connected components of $G = (V, E)$ then the component graph of G is defined as $\bar{G} = (\bar{V}, \bar{E})$ where $\bar{V} = \{c_1, c_2, ..., c_m\}$ denotes the set of strongly connected components such that $\bar{V}$ is a partition of V and $\bar{E}$ is defined as $\bar{E} = \{(c_i, c_j) | c_i \neq c_j, \exists v_i \in c_i, v_j \in c_j \text{ s.t. } (v_i, v_j) \in E\}$.*

In other words, each vertex in $\bar{G}$ corresponds to a subset of vertices in G and there is an edge in $\bar{G}$ from a vertex c_i to another vertex c_j, if in G there is an edge from one of the vertices in c_i to one of the vertices in c_j.

3.2 Free Edge and Set of Free Edges

Let's define an edge $e = (v_i, v_j)$ of G where $v_i \in c_i$, as a free edge if the component c_i remains strongly connected when e is removed from G. Formally

Definition 2. *e is a* free edge *in G if $\bar{G} = (\bar{V}, \bar{E})$ and $\bar{G'} = (\bar{V'}, \bar{E'})$ satisfy $\bar{V} = \bar{V'}$ where $G' = (V, E')$ and $E' = E \setminus \{e\}$.*

In the following sections set of free edges for graph G will be denoted as F.

3.2.1 Existence of a Free Edge in a Strongly Connected Graph

Below we present a proof for existence of at least one free edge in a strongly connected graph $G = (V, E)$ where $|E| \geq 2/times|V|$.

Definition 3. *Let $G = (V, E)$ be a digraph. For a subset of the nodes Γ, Γ* contraction of G *is defined as $G(\Gamma) = (V', E')$ where $V' = (V \setminus \Gamma) \cup \{\gamma\}$ and*

$$\begin{aligned} E' = \ & \{(u,v) | u,v \notin \Gamma, (u,v) \in E\} \cup \\ & \{(u,\gamma) | u \notin \Gamma, v \in \Gamma, (u,v) \in E\} \cup \\ & \{(\gamma,v) | u \in \Gamma, v \notin \Gamma, (u,v) \in E\} \end{aligned}$$

Intuitively, in $G(\Gamma)$ all the nodes in Γ are removed and they are represented by a new fresh node γ. Those edges in G that are not from or to a node in Γ are preserved in $G(\Gamma)$. The edges between two nodes in Γ are removed in $G(\Gamma)$. An edge between a node in Γ and a node not in Γ is replaced by an edge using the node γ instead of the node in Γ.

Lemma 2. *Let $G = (V, E)$ be a digraph and $\Gamma \subseteq V$ be a subset of V. For two nodes $u, u' \in V \setminus \Gamma$, if there exists a path $u \rightsquigarrow u'$ in G, then there also exists a path $u \rightsquigarrow u'$ in $G(\Gamma)$.*

Proof. If the path $u \rightsquigarrow u'$ does not go through a node in Γ, then all the edges in $u \rightsquigarrow u'$ also exist in $G(\Gamma)$. Otherwise let $u \rightsquigarrow v$ ($v' \rightsquigarrow u'$, resp.) be the shortest prefix (the shortest suffix of, resp.) $u \rightsquigarrow u'$ such that $u, v' \in \Gamma$. By using Lemma 3 (Lemma 4, resp.), there exist a path $u \rightsquigarrow \gamma$ ($\gamma \rightsquigarrow u'$, resp.) in $G(\Gamma)$. Hence we have the path $u \rightsquigarrow \gamma \rightsquigarrow u'$ in $G(\Gamma)$. □

Lemma 3. *Let $G = (V, E)$ be a digraph and $\Gamma \subseteq V$ be a subset of V. For a node $u \in V \setminus \Gamma$, if there exists a path $u \rightsquigarrow u'$ to a node $u' \in \Gamma$ in G, then there also exists a path $u \rightsquigarrow \gamma$ in $G(\Gamma)$.*

Proof. Consider the shortest prefix $u \rightsquigarrow v$ of the path $u \rightsquigarrow u'$ such that $v \in \Gamma$. Let $u \rightsquigarrow v'$ be the path $u \rightsquigarrow v$ where the last edge (v', v) is removed. Since none of the nodes along the path $u \rightsquigarrow v'$ are in Γ, the edges on this path also exist in $G(\Gamma)$. Therefore we have the path $u \rightsquigarrow v'$ also in $G(\Gamma)$. Since $v' \notin \Gamma, v \in \Gamma, (v', v) \in E$, we have the edge (v', γ) in $G(\Gamma)$. Thus by combining the path $u \rightsquigarrow v'$ and the edge (v', γ) in $G(\Gamma)$, the desired result is obtained. □

Lemma 4. *Let $G = (V, E)$ be a digraph and $\Gamma \subseteq V$ be a subset of V. For a node $u \in V \setminus \Gamma$, if there exists a path $u' \rightsquigarrow u$ from a node $u' \in \Gamma$ in G, then there also exists a path $\gamma \rightsquigarrow u$ in $G(\Gamma)$.*

Proof. Consider the shortest suffix $v \rightsquigarrow u$ of the path $u' \rightsquigarrow u$ such that $v \in \Gamma$. Let $v' \rightsquigarrow u$ be the path $v \rightsquigarrow u$ where the first edge (v, v') is removed. Since none of the nodes along the path $v' \rightsquigarrow u$ are in Γ, the edges on this path also exist in $G(\Gamma)$. Therefore we have the path $v' \rightsquigarrow u$ also in $G(\Gamma)$. Since $v' \notin \Gamma, v \in \Gamma, (v, v') \in E$, we have the edge (γ, v') in $G(\Gamma)$. Thus by combining the edge (γ, v') and the path $v' \rightsquigarrow u$, the desired result is obtained. □

Lemma 5. *Let $G = (V, E)$ be a digraph and $\Gamma \subset V$ be a subset of V. If G is strongly connected then so is $G(\Gamma)$.*

Proof. Consider two nodes $u, v \notin \Gamma$. Since G is strongly connected, we have a path $u \rightsquigarrow v$ existing in G. By using Lemma 2, we also have such a path in $G(\Gamma)$. Consider now a node $u \notin \Gamma$. There must exist a path $u \rightsquigarrow \gamma$ in $G(\Gamma)$. To see this consider a node $v \in \Gamma$. Since G is strongly connected, there is a path $u \rightsquigarrow v$ in G. By using Lemma 3, the desired result is obtained. Finally, the existence of a path $\gamma \rightsquigarrow u$ can be shown by using a similar reasoning and Lemma 4. □

Lemma 6. *Let $G = (V, E)$ be a strongly connected digraph with $|E| \geq 2 \times |V|$. Then there exists at least one free edge in G.*

Proof. The proof is by induction on $|V|$. For $|V| = 1$ it is trivial to see that the claim holds. Let us consider the case $|V| > 1$. If G has a loop (that is if $(v, v) \in E$ for some $v \in V$) or if G has parallel edges (that is if there are multiple edges between the same pair of nodes), then we can remove the loop or one of the parallel edges and the graph will still be strongly connected. Suppose G has no loops and it has no parallel edges. Let $\Gamma = \{v_1, v_2, \dots, v_m\} \subseteq V$ be the nodes of a smallest cycle (i.e. a cycle with the smallest number of vertices) in G. As G has no loops, $m \geq 2$. Without loss of generality assume that, $\forall 1 \leq i < m, (v_i, v_{i+1}) \in E$ and $(v_m, v_1) \in E$. Note that these edges must be the only edges between the nodes of Γ. In other words, for three different nodes $v_i, v_j, v_k \in \Gamma$ it is not possible to have $(v_i, v_j), (v_i, v_k) \in E$ since Γ wouldn't be a smallest cycle otherwise. Therefore there are exactly m edges between the vertices in Γ.

Let us now consider $G(\Gamma)$. Since there are exactly m edges between the vertices in Γ, there are $|E| - m$ edges in $G(\Gamma)$. The number of vertices in $G(\Gamma)$ is $|V| - m + 1$.

First of all, the number of edges in $G(\Gamma)$ is more than two times the number of nodes in $G(\Gamma)$, i.e. $|E| - m \geq 2 \times (|V| - m + 1)$ since $|E| \geq 2 \times |V|$ and $m \geq 2$.

Furthermore by using Lemma 5, it is known that $G(\Gamma)$ is strongly connected as well.

Finally, $(|V| - m + 1) < |V|$ since $m \geq 2$ and therefore by using the induction hypothesis the proof is completed. □

3.2.2 Existence of a Free Edge in a not Strongly Connected Graph

Below we show that there exists at least one edge in a not strongly connected graph if the graph has nodes with outdegree greater than 1.

Theorem 7. *Let $G = (V, E)$ be a digraph where each node has the same outdegree $k \geq 2$ and let $G' = (V', E')$ be a strongly connected component of G. If G is not strongly connected, then there exists at least one free edge (u, v) in G where $u \in V'$.*

Proof. If there exists an edge $(u, v) \in E$ where $u \in V'$ and $v \in V \setminus V'$, then (u, v) is a free edge. If there is no such edge, then $|E'| = k \times |V'| \geq 2 \times |V'|$. In this case by using Lemma 6, there is a free edge (u, v) in G' where $u, v \in V'$. $\square$

3.3 Forcing Strongly Connectedness

If the user wants the generated FSM to be strongly connected, tool gives user an option of forcing strongly connectedness of the generated FSM by a post processing step rather than waiting for a strongly connected FSM to be generated by random assignment of transitions only. If this option is enabled, tool generates a random FSM by randomly assigning transitions and checks whether it is strongly connected. If it is not then the post processing to make the FSM strongly connected begins. Details of this process are explained in this section. Note that since an FSM can be represented as a directed graph, the process will be explained as a graph algorithm considering the underlying graph representation of the FSM.

3.3.1 Finding a Set of Free Edges in a Component

The problem of finding a set of free edges for a strongly connected component as large as possible is directly related to Minimum Equivalent Graph (MEG) problem. MEG problem is defined as follows. Given a directed graph $G(V, E)$ find the smallest subset E' of E such that E' still keeps the same reachability relations between vertices in V. When MEG problem is restricted to strongly connected graphs then it is called the minimum Strongly Connected Spanning Subgraph (SCSS) problem which is NP-HARD [8]. As you may notice if we can find a solution to the minimum SCSS problem for a component c_i then we can find a set of free edges with maximum cardinality for c_i and vice versa. That is because if E' is the solution to the minimum SCSS problem for a strongly connected component c_i of $G(V, E)$ and if $E_i \subset E$ is defined as $E_i = \{(v_i, v_j) | v_i \in c_i\}$ then $(E_i \setminus E')$ is a set of free edges with maximum cardinality for c_i.

Although finding a set of free edges with maximum cardinality for a strongly connected component is NP-HARD, we still want to find as many free edges as possible. For this reason we use a very simple heuristic. When finding F we iterate on each

edge $e = (v_i, v_j) \in E$. If $v_i, v_j \in c_i$ we remove e and check if v_j is still reachable from v_i. If it is reachable then e is a free edge and included in F, otherwise we put e back. However there are cases where the reachability check can be skipped and an edge can be included in F directly. One such case is when $v_i = v_j$, that is e is a self-loop and it is guaranteed to be a free edge. Also any edge e satisfying $v_i \in c_i, v_j \notin c_i$ directly included in F since in that case e is an edge going to a vertex outside c_i and does not affect the strongly connectedness of c_i. Algorithm 2 describes this process formally. Note that except these two cases, if an edge e happens to be a free edge and thus is included in F and removed from E, an edge $e' \neq e$ which has not been considered yet and was previously a free edge before removal of e, might not be a free edge anymore. For that reason, the order in which the free edges are considered and included in F becomes important. In our implementation, since we want to affect randomness of the generated FSM as little as possible, we consider edges in a random order for inclusion in F.

Algorithm 2: Find Set of Free Edges

Input: $G = (V, E)$ graph

Output: F set of free edges for G

```
1 F = ∅;
2 E' = E;
3 foreach edge e = (v_i, v_j) ∈ E in some random order do
4     Let c_i and c_j be the components in G s.t. v_i ∈ c_i and v_j ∈ c_j;
5     if v_i = v_j OR c_i ≠ c_j OR v_i ⇝ v_j in G' = (V, E' \ {e}) then
6         F = F ∪ {e};
7         E' = E' \ {e};
```

The complexity of finding a set of free edges is analyzed as follows. Finding a set of free edges in a graph is performed by removing an edge and checking the reachability condition. After an edge $e = (v, v')$ is removed, checking if v' is still reachable from v takes $O(V + E)$ time using breadth first search. In the worst case the algorithm may try to remove all edges and check for reachability. Hence the complexity is $O((V + E)E)$. Since in our case the graph represents a completely specified FSM with n states and p inputs, that is $|V| = n$ and $|E| = np$, the

complexity is $O((n + np)np) = O(n^2p^2)$.

3.3.2 Making a Graph Strongly Connected

Making a graph strongly connected is an iterative process such that after each iteration the number of strongly connected components of the graph either reduces or stays same. The process terminates when the number of strongly connected components reduces to 1 and thus graph becomes strongly connected. To achieve this, the aim in each iteration is to find a set of free edges of the current graph and assign new destinations for each of them hoping that these new assignments will create new connections between components and reduce the number of strongly connected components. Note that Theorem 7 guarantees that if $G = (V, E)$ is not strongly connected then Algorithm 2 will find at least one free edge in each and every strongly connected component of G. Notice that by definition a free edge has no effect on the strongly connectedness of any component. Thus changing the destinations of free edges never has the risk of increasing the number of components. To be more clear and give the main idea, a more formal description of the algorithm is presented in Algorithm 3.

Algorithm 3: Make Graph Strongly Connected

Input: $G = (V, E)$ not strongly connected graph

Output: $G^* = (V, E^*)$ strongly connected graph obtained by changing destination vertices of some edges in G

```
1  G* = G;
2  while G* is not strongly connected do
3  |   Ḡ*(V̄, Ē*) = component graph of G*;
4  |   find a set of free edges F of G*;
5  |   remove F from E*;
6  |   foreach edge (v_i, v_j) ∈ F do
7  |   |   pick a random component c ∈ V̄*;
8  |   |   pick a random vertex v ∈ c;
9  |   |   E* = E* ∪ {(v_i, v)};
```

One important thing to notice is that the new destination for a free edge is

determined by firstly choosing a random component and then a random destination vertex within that component rather than choosing a random vertex in the graph directly. Also notice that we have no restrictions on which component to choose, so it can be the case that new destination for the free edge might be in the same component as the source of the free edge. Although in such a case, no connection is created between components, nevertheless the effect of new assignments on the randomness of the graph is much less. In addition to that, choosing the component of destination vertex first increases algorithm's chances for increasing the number of connections between components over the chance of choosing a destination within the same component as the source of free edge. Let's see how this is so. Assume that a graph G with n vertices initially have m strongly connected components $\bar{V} = \{c_1, c_2, ..., c_m\}$ and some component c_i satisfies $\forall j, j \neq i, n > |c_i| >> |c_j|$. That is c_i is a very large component compared to all other components in terms of number of the vertices it contains. Also let's assume that the component with the smallest cardinality is c_m and consider the chances of assigning a free edge of c_i to c_m. If we had chosen a vertex in the graph directly as the new destination of a free edge, a free edge whose source is in c_i will be assigned to a new destination in component c_m with a probability of $|c_m|/n$. Since $n >> |c_m|$, probability of creating a connection from the large component c_i to the smallest component c_m will be very small. However in our method, by choosing the component for the destination first, the probability of connection from the c_i to the c_m becomes $1/m$ which is in practice much greater than $|c_m|/n$.

Algorithm 3, although gives the main idea of our implementation, does not reflect the details correctly. In each iteration of the algorithm, it seems strongly connected components and set of free edges are computed from scratch for graph $G^* = (V, E^*)$. Computing these in each iteration can be very time consuming if G^* is large. Because of this, our implementation follows a different way, while doing the same thing in essence. Instead of working each time on the original graph, starting from the original graph, in each iteration we always work on the component graph of the previous iteration. Thus we are trying to make the component graph strongly connected which is actually same thing as making the original graph strongly connected. Thus after an iteration, if some components form a new strongly connected

component, the size of the graph we are working on reduces. However working on a new component graph in each iteration, instead of the original graph, requires us to remember the vertices within the components so that the changes that are made on the graph used in current iteration could be mapped to the graph on the previous iteration. For this reason, we use a stack that stores the vertices within the components and the free edges used in an iteration. When the last iteration finishes and the graph reduces to a single component, using the information stored in the stack, we are able to change the edges of the all previous iterations and including the initial graph so that it is now strongly connected.

In order to analyze the running time of the Algorithm 3, we need to know that how many times while loop iterates. We already know the running time of each step within while loop. The most expensive step happens to be finding free edges of a graph which has running time $O(n^2p^2)$ and dominates other steps. However we do not know how many times while loop will iterate exactly since the algorithm is probabilistic. Although in theory while loop may iterate infinitely many times, it will iterate until the number of strongly connected components reduces to 1. In the worst case scenario, initially we may have all vertices as a separate component hence there can be at most $|V| = n$ components. Further in the worst case scenario we assume that each component has only one free edge. Then by assigning new destinations to free edges, algorithm tries to create a cycle in the component graph. When a cycle is formed the components in the cycle becomes connected and number of strongly connected components reduces. For the worst case scenario we can calculate the probability of creating a cycle in the component graph and denote it as P. A rough calculation shows that $P > (n-1)!(n-1)/2n^{n-1}$. Also the expected worst case running time of the algorithm E can be found using $E = T/P$ where T is the running time of a single iteration. Hence the expected running time is $O(n^2p^2/((n-1)!(n-1)/2n^{n-1}))$ which is $O(n^n)$. Although worst case expected running time of the algorithm is very large, note that this is a very loosely calculated bound which considers a very extreme case. In practice the algorithm terminates in feasible time (for instance it takes approximately 1 second to generate a strongly connected FSM with 10000 states 5 inputs and 5 outputs).

3.4 Forcing Initial Reachability

Some checking sequence generation methods assume a reliable reset feature in the implementation. This feature guarantees that no matter at which state the machine currently is, applying a special input, called *the reset input*, takes the machine to the initial state.

Such a reset transition is modeled in a specification by a transition from each state to the initial state. The existence of these reset transitions relaxes the conditions on the other transitions. More explicitly stated, the machine has to be strongly connected. However for being strongly connected, it is now sufficient to be *initially reachable* only, i.e. all states must be reachable from the initial state. This condition combined with the reset transitions from all the states back to the initial state guarantees that the machine is strongly connected. To support the research for checking sequence generation under the assumption of reliable reset transitions, our random FSM generation tool supports generation of initially reachable but not strongly connected FSMs as well.

If an initially reachable FSM is desired, tool has two different methods of making a graph initially reachable. Which method to use is selected by user. Notice that a strongly connected FSM is also initially reachable. Because of that making an FSM initially reachable is only necessary when a not strongly connected FSM is desired.

Before explaining methods in detail, we need to establish an important property of initially reachable graphs.

Theorem 8. *The component graph $\bar{G} = (\bar{V}, \bar{E})$ of an initially reachable graph $G = (V, E)$ have only one vertex with indegree 0 and it contains the initial vertex.*

Proof. Consider the component c_i that contains the initial vertex. That means all components in $\bar{V} \setminus \{c_i\}$ are reachable from c_i. Firstly notice that c_i cannot have an incoming edge so its indegree is 0. This can be shown by a simple contradiction. If there had been an incoming edge (c_j, c_i) then that edge would form a cycle in component graph since c_j is reachable from c_i. Since a component graph is an acyclic graph by definition, a contradiction is reached. Secondly for all components in $\bar{V}$ to be reachable from c_i each one must have at least one incoming edge because a component with no incoming edge cannot be reached from another component.

These two facts prove that all vertices in $\bar{V}$ except c_i have indegree greater than 0. □

3.4.1 Method 1: Using a Backbone Component Graph

In this method, the user is given some control on the structure of component graph of the random graph that will be generated. Besides other inputs (number of states, number of input symbols and number of output symbols), the user can give number of strongly connected components and the number of vertices (states) within each component as input. Then according to this component structure given by the user, edges between these components are decided in a manner that makes the component graph initially reachable. This component graph is called the backbone component graph since any graph which has the same connections between its components as the backbone component graph is guaranteed to be initially reachable.

Notice that there can be many different backbone component graphs for a given number of components. For this reason generation of a backbone component graph is a process that results in one of the possible backbones by some random selection of edges between components.

Backbone Generation Assume that user wants m strongly connected components denoted as $\bar{V} = \{c_1, c_2, ..c_m\}$. We first need to assign an order to each component. Since we represent a component c_i with an integer index i, let's use natural order of integers as the order of components. Then we assign edges of the backbone component graph $\bar{G} = (\bar{V}, \bar{E})$ such that they satisfy following conditions.

1. $\forall j > 1 \ \exists i$ s.t. $i < j$ and $(c_i, c_j) \in \bar{E}$
2. $\forall j \neg \exists i$ s.t. $i > j$ and $(c_i, c_j) \in \bar{E}$

Simply, what these conditions establish are as follows. In condition 1 it is established that all components, except c_1, have at least one incoming edge from another component which is smaller in the ordering of components. That is all components are reachable from c_1. Condition 2 states that there can be no edge from a component with some large order to a component with a smaller order. This guarantees that

there is no cycle in the graph as a component graph must be acyclic. The algorithm for generating backbone is given in Algorithm 4.

Algorithm 4: Generate Backbone Component Graph

Input: m number of strongly connected components

Output: $\bar{G} = (\bar{V}, \bar{E})$ backbone component graph

```
1 V̄ = {c_1, c_2, ..., c_m};
2 Ē = ∅;
3 for i = 2 to m do
4 |   choose some nonempty random subset s of {1, ..., i};
5 |   foreach c_j s.t. j ∈ s do
6 |   |   Ē = Ē ∪ (c_j, c_i);
```

The complexity of generating a backbone component graph is $O(m^2)$, since for each of the m components some edges are added from a subset of m components.

Generating an Initially Reachable Graph Now we can present the generation of an initially reachable graph using the generated backbone component graph. Algorithm 5 describes this process.

Here are some remarks about Algorithm 5.

- At line 1, generation of a random graph with strongly connected components $\{c_1, c_2, ..., c_m\}$ each having size as given in $N = \{n_1, n_2, ..., n_m\}$ is achieved as follows. Firstly for each c_i a separate strongly connected graph with n_i vertices are generated using the tool. Then these m graphs are combined into one graph that consists of these m individual graphs.

- In the for loop between lines 5-8, for each edge in the backbone graph, it is made sure that the resulting graph has an edge between the corresponding components. This is achieved by changing the destination vertex of a free edge according to the edge in backbone graph and putting it back to set of edges.

- At the last line, all remaining free edges inserted back into the graph after their destinations are changed. Destinations are changed in such a way that

Algorithm 5: Generate Initial Reachable Graph Using Backbone Component Graph

Input: $N = \{n_1, n_2, ..., n_m\}$ component sizes

Output: $G = (V, E)$ initially reachable graph with components $\bar{V}$

1 generate a random graph $G(V, E)$ with $|N| = m$ strongly connected components each containing n_k vertices where $1 \leq k \leq m$;

2 generate a backbone component graph $\bar{G}$;

3 find a set of free edges F for G;

4 remove F from E ;

5 **foreach** *edge* $(c_i, c_j) \in \bar{E}$ *of* $\bar{G}$ **do**

6   pick some random free edge $(v_i, v_k) \in F$ s.t. $v_i \in c_i$;

7   pick some random vertex $v_j \in c_j$;

8     $E = E \cup (v_i, v_j)$;

9 add all remaining free edges to E after changing their destinations in a way that does not violate condition 2;

condition 2 is not violated, that is no cycle is introduced in the component graph. Two approaches implemented to achieve this. In the first approach all free edges are assigned destinations according to backbone graph whose edges already satisfy condition 2 and in the second approach a free edge whose source is in component c_i is assigned to some random component c_j such that $i < j$. When the first approach is used, the component graph of the generated random graph is same as the backbone component graph. However in the second approach the component graph of the generated random graph may contain connections that does not exists in the backbone component graph.

The complexity of Algorithm 5 is dominated by generating m strongly connected graphs in the first statement. Hence Algorithm 5 have the same complexity as generating m strongly connected graphs.

3.4.2 Method 2: Generate an Initial Reachable Graph with Random Components

When user does not care about the number of strongly connected components and number of vertices in components, so he wants these parameters to be random as well, then he can use the second method for generating an initially reachable random graph. In this method firstly a not strongly connected random graph is obtained by random assignment of edges. Then this graph is forced into an initially reachable graph, if it is not initially reachable already.

Intuitively, the method works as follows. Let $\bar{V}_0 \subseteq \bar{V}$ be the set of vertices in the component graph with 0 indegree. Initially in the component graph there are always more than one vertex with 0 indegree, since otherwise graph would be already initially reachable. The main aim of the method is to reduce the cardinality of $\bar{V}_0$ to one and thus making the graph initially reachable. In each iteration some random vertex c_i from $\bar{V}_0$ is chosen and it is removed from $\bar{V}_0$ after increasing its indegree. Indegree of c_i is increased by using free edges of some randomly chosen subset of vertices which cannot be reached from c_i. That is new connections are made to c_i from vertices that are not reachable from c_i. It is important to make these new connections from vertices that are not reachable from c_i, since this guarantees that we do not create a cycle in the component graph. Although at the end of the iteration c_i is removed from $\bar{V}_0$, this does not necessarily reduce the cardinality of $\bar{V}_0$. This is because, the edges between vertices of two different components are free edges by definition and since destinations of free edges are changed in order to make new connections to c_i, a vertex in the component graph may lose its only incoming edge. Hence its indegree becomes 0 and it must be included in $\bar{V}_0$. For this reason, at the end of each iteration $\bar{V}_0$ is updated along with the component graph $\bar{G}$. Even though theoretically algorithm does not have guarantee of termination, in practice this does not seem to be a problem.

More formal description of the algorithm is presented in Algorithm 6.

Algorithm 6 is a probabilistic algorithm with large complexity. Although in theory it has no guarantee for termination, in practice it terminates quickly.

Algorithm 6: Make a graph initially reachable

Input: $G = (V, E)$ graph to make initially reachable

Result: G is initially reachable

1 $\bar{G} = (\bar{V}, \bar{E})$ = component graph of G;
2 find a set of free edges F for G;
3 $\bar{V}_0$ = vertices in $\bar{G}$ with 0 indegree;
4 **while** $\bar{V}_0$ *have more than one element* **do**
5 pick some random $c_i \in \bar{V}_0$;
6 $\bar{V}_i$ = set of components not reachable from c_i ;
7 pick some random subset S of $\bar{V}_i$;
8 **foreach** $c_s \in S$ **do**
9   pick a free edge $e = (v_i, v_j)$ such that $v_i \in c_s$;
10   set destination of e to some randomly chosen vertex in c_i;
11   update $\bar{G}$;
12   update $\bar{V}_0$;

3.5 Shuffling

To decrease the time spent to generate an FSM with a preset distinguishing sequence, tool contains an option called shuffle. When user wants to generate a random FSM with a preset distinguishing sequence, tool generates an initial FSM and checks if it has a preset distinguishing sequence. What this option provides is that if the FSM has not any distinguishing sequence then rather than creating a new FSM from scratch, tool randomly assigns new input and output symbols for each transition and checks again for the existence of a distinguishing sequence. This operation called shuffling and takes less time than creating a new FSM from scratch. Notice that during shuffling, sources and destinations of transitions are not changed. Thus properties such as strongly connectedness and initial reachability is not affected after shuffling. When this option is enabled user can also provide how many times shuffling takes place before a new FSM created from scratch or an FSM with preset distinguishing sequence is generated.

The running time of a single shuffle operation is $O(np)$ where n is number of

states and p is number of input symbols. That is because every transition is considered once and there are np transitions in a completely specified, deterministic FSM.

3.6 Providing Input/Output Probabilities

Recall that the assignment of output symbols to the transitions are performed randomly. For each input symbol x and output symbol y, the number of x/y transitions seen in the FSMs randomly generated in this way turns out be more or less the same.

To test a heuristic developed for generating UIO sequences based on the frequency (how rare or how frequent) of transitions' I/O labels [5], our tool has an option that allows user to specify the probability for each I/O pair to be seen in the FSM.

These probabilities are given in a regular text file that we call the i/o distribution file. Each line of the file should be in the form i o p where i is an input symbol, o is an output symbol and p is a probability as a percentage. Since tool generates completely specified FSMs, number of transitions that have input symbol i is always same and it is exactly number of states. That is because each state must have a transition with input i in a completely specified FSM. On the other hand, no restriction exists for output symbols. Then a line in the file means that among all transitions which have input symbol i, p percent of them should have output symbol o in the FSM.

Chapter 4

Checking if a Sequence is a Checking Sequence

4.1 Introduction

Given any input output sequence X/Y of a specification FSM M, it is desirable to know whether X is a checking sequence of M or not. Further if it is known that X is not a checking sequence of M, it seems beneficial to able to get some information about how close X is to a checking sequence of M. For example, during the operation of checking sequence generation algorithm, with this information algorithm will be knowledgeable about how close the current sequence to a checking sequence and will have the opportunity to use it as a guide to make decisions on how to extend the current sequence so that generating a checking sequence is possible. The checking sequence generation method that will be explained in Chapter 6 uses such an approach.

In this section, we propose a distinguishing sequence based method which checks if the input portion X of an input output sequence X/Y is a DS based checking sequence of specification FSM M. If it is not, the method is still able to provide some information about how close X is to a checking sequence.

4.2 Uncertainty Automaton

As explained in Section 2.4, a checking sequence for an FSM M distinguishes M from all FSMs in the set $\Phi(M)$ where $\Phi(M)$ is the set of FSMs with at most as many states as M and having the same input output sets. Hence to determine if the input portion X of a input output sequence X/Y of M is a checking sequence, initially we treat X/Y as an I/O sequence that is produced by an FSM in $\Phi(M)$. That is initially we only assume that X/Y is a sequence that is produced by some unknown machine $N \in \Phi(M)$ and what we want to know is that if N is equivalent to M or not. Since X/Y is an I/O sequence, this sequence corresponds to some sequence of transitions that visits a sequence of states of this unknown FSM N. Let's consider the path $P = (n_1, n_r; X/Y)$ where nodes n_i represents states visited in N when X is applied. If we can find a correspondence between the states of M and the nodes in P and see that P verifies every transition of M then we can say that X is a checking sequence of M. To find this correspondence between the states of M and nodes in P, we consider P as a graph and call this as the uncertainty automaton. It is called that way, since initially we do not know which node corresponds to which state of M and there is the possibility that a node n_i could be any of the states of M. Hence we associate each node n_i with a set of states that it may correspond to and call that set as the candidate set of node n_i. While we process the uncertainty automaton we try to reduce the number of states in candidate sets of each node.

Formally, given an input output sequence X/Y we consider a path $P = (n_1, n_{r+1}; X/Y)$. Then we represent P as a graph. We call this graph as uncertainty automaton of P and represent it as $G_P = (V_P, E_P)$ where initially $V_P = \{n_1, n_2, ..., n_{r+1}\}$ and $E_P = \{(n_i, n_{i+1}; x/y) | (n_i, n_{i+1}; x/y) \text{ in } P\}$.
Furthermore let's define $C : V_P \mapsto 2^S$ where S is the set of states of M. In other words C maps each node n_i to a set of states of FSM M such that $C(n_i)$ is called the candidate set of n_i and represents the set of states that n_i can be recognized as.

For example consider the I/O sequence $X/Y = aabababbba/0101100100$ and FSM M_1 given in Figure 2.1. Then initial uncertainty automaton is generated according to the sequence X/Y as shown in Figure 4.1. Each node in the initial uncertainty automaton have all states of M_1 in their candidate sets, i.e. $\forall 1 \geq i \geq 11, C(n_i) = \{s_1, s_2, s_3\}$. That is they can be recognized as either s_1 or s_2 or s_3.

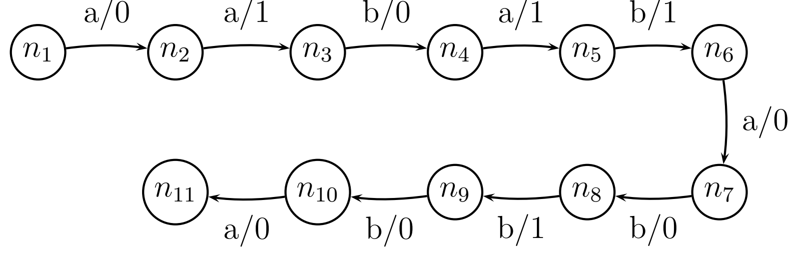

Figure 4.1: Initial Uncertainty Automaton

The main aim of the method is to recognize each node in the uncertainty automaton. Beginning with the initial uncertainty automaton, method tries to eliminate states from the candidate sets of nodes. We will propose several techniques to eliminate states from the candidate sets. Using these techniques if a candidate set of a node becomes singleton then that node is recognized. That is when the candidate set of a node n_i contains a single node, say s, that means n_i is recognized as state s of M, i.e. candidate set of n_i will be $C(n_i) = \{s\}$.

4.3 State Recognition Using Uncertainty Automaton

Given an input output sequence X/Y, considering the path with label X/Y $P = (n_1, n_{r+1}; X/Y)$ we form the initial uncertainty automaton G_P as explained above. G_P is initialized such that for each node $n_i \in V_P$, $C(n_i)$ contains all the states in FSM M. Later we try to recognize the nodes of G_P by reducing the candidate sets of the nodes. The uncertainty reduces as the candidate sets of the nodes get smaller.

One easy way of recognizing a node is to look for an occurrence of ADS of a state. That is if the path P has a subpath $(n_i, n_j; X'/Y')$ such that X' is ADS of a state s and $\lambda(s, X') = Y'$ then the node n_i cannot be any state other than s. Therefore such nodes can easily be recognized as the corresponding states and the candidate sets of those nodes can be updated accordingly.

For example, consider the distinguishing set $\bar{D} = \{D_{s_1}, D_{s_2}, D_{s_3}\}$ where $D_{s_1} = a/0, D_{s_2} = ab/11, D_{s_3} = ab/10$ for the FSM M_1 in Figure 2.1. Using $\bar{D}$ in the initial uncertainty automaton shown in Figure 4.1, we can *d-recognize*

- Nodes n_1, n_6 and n_{10} as state s_1

- Node n_4 as state s_2
- Node n_2 as state s_3

and update the candidate sets as shown in Table 4.1.

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3\}$	$C(n_3) = \{s_1, s_2, s_3\}$	$C(n_4) = \{s_2\}$
$C(n_5) = \{s_1, s_2, s_3\}$	$C(n_6) = \{s_1\}$	$C(n_7) = \{s_1, s_2, s_3\}$	$C(n_8) = \{s_1, s_2, s_3\}$
$C(n_9) = \{s_1, s_2, s_3\}$	$C(n_{10}) = \{s_1\}$	$C(n_{11}) = \{s_1, s_2, s_3\}$	

Table 4.1: Candidate Sets For the Uncertainty Automaton in Figure 4.1 after d-recognition

Whenever we understand that two nodes of an uncertainty automaton correspond to the same state of M, we merge those two nodes into one single node. We can understand that two nodes n_i and n_j correspond to the same state in two different ways.

- n_i and n_j are both recognized as the same state s of M, that is $C(n_i) = C(n_j) = \{s\}$.
- there exist two subpaths $(n_p, n_i; X'/Y')$ and $(n_q, n_j; X'/Y')$ with the same label in G_P where n_p and n_q are understood to correspond to the same state of M.

After we understand two nodes correspond to the same state, we merge them by using the following merge operation.

Merging Nodes A node n_j is merged into other node n_i by

1. setting the start of each edge leaving n_j as n_i
2. setting the end of each edge entering n_j as n_i.
3. updating the candidate set of n_i as $C(n_i) = C(n_i) \cap C(n_j)$

Intuitively, as a result of step 1 and 2 above each edge leaving and entering n_j now leaves and enters the node n_i. If step 1 creates a node n_i that has two leaving edges with the same label then the end nodes of these edges are also understood

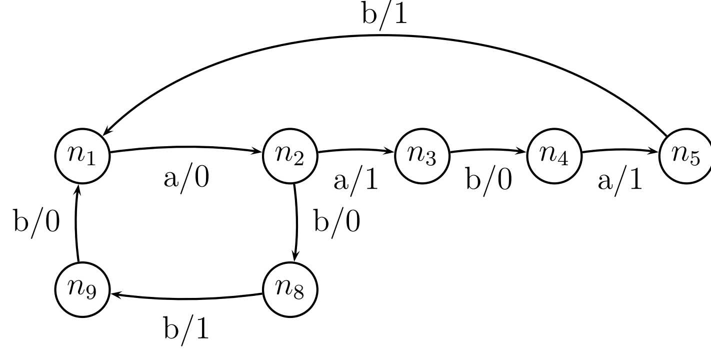

Figure 4.2: Uncertainty Automaton after nodes merged

to be corresponding to the same state, hence they will be merged as well. For this reason, the uncertainty automaton always stays deterministic at the end of merge operations. In step 3, the candidate sets of the merging nodes is intersected because two nodes are understood to be corresponding to the same state. Although we may not know which state they correspond to, it is obvious that it must be one of the states in the intersection of the candidate sets of two nodes.

Also notice while merging two nodes n_i and n_j if $C(n_i) \cap C(n_j)$ is a singleton, say $\{s\}$ then that means after merging resulting node is recognized as state s of M. In fact the t-recognition explained in Section 2.4 will be realized when merging two nodes n_i and n_j where $|C(n_i)| = 1$ and $|C(n_j)| > 1$.

After n_j is merged into n_i, n_j is removed from the uncertainty automaton, since all the information that is stored in n_j is now available in n_i.

For example, consider the uncertainty automaton in Figure 4.1 and candidate sets given in Table 4.1. Since nodes n_1, n_6 and n_{10} are recognized as state s_1, they must be merged into one node, let's merge them as n_1. In addition to that, since n_1, n_6 and n_{10} enters to n_2, n_7 and n_{11} with the label $a/0$ respectively, nodes n_2, n_7 and n_{11} must be merged into one node, let's merge them as n_2. At this point no more merging is possible and the resulting uncertainty automaton is shown in Figure 4.2 along with the candidate sets shown in Table 4.2.

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3\}$	$C(n_3) = \{s_1, s_2, s_3\}$	$C(n_4) = \{s_2\}$
$C(n_5) = \{s_1, s_2, s_3\}$	$C(n_8) = \{s_1, s_2, s_3\}$	$C(n_9) = \{s_1, s_2, s_3\}$	

Table 4.2: Candidate Sets For the Uncertainty Automaton in Figure 4.2

The running time for a single merge operation on an uncertainty automaton is the summation of time spent for setting the edges and intersecting candidate sets of the nodes that are merging. For a single node the maximum number of outgoing edges can be p (number of input symbols) and the maximum number incoming edges can be r (number of edges in the uncertainty automaton). Hence the total time spent for setting edges is $O(p + r)$. In addition to that, since in a candidate set there can be at most n elements (states), intersection of candidate sets takes $O(n^2)$ without using a special data structure for set operations. Hence the running time for a single merge operation is $O(p + r + n^2)$. The running time for all possible merge operations is simply the multiplication of number of possible merge operations and time spent for a single merge. In an uncertainty automaton the maximum number of possible merges is bounded by the number of nodes in the uncertainty automaton(i.e. $O(r) = r + 1$). This case happens when a single node is merged with all other nodes. Hence the running time for all merge operations is $O(r) \times O(p + r + n^2) = O(pr + r^2 + n^2 r)$.

4.3.1 Candidate Elimination Using Incompatible Sets

With the techniques explained so far, all the state recognitions on the uncertainty automaton can also be realized by using d- and t-recognition only. In this section a technique that eliminates states from the candidate set of a node will be explained which in turn allows the techniques explained above to recognize more states than d- and t-recognitions alone can achieve. Before explaining this technique, we need to define a compatibility relation between the nodes of an uncertainty automaton.

Compatibility of Nodes Two nodes n_i and n_j are defined to be compatible if

- $C(n_i) \cap C(n_j) \neq \emptyset$ and
- for any input sequence $X \in I^*$ that is defined for n_i, either X is not defined for n_j or if paths $(n_i, n_t, X/Y_i)$ and $(n_j, n_u, X/Y_j)$ exists in G_P then $Y_i = Y_j$ and n_t and n_u are compatible and vice versa.

An important property of compatibility relation is that it is symmetric, that is if n_i is compatible with n_j then n_j is compatible with n_i.

Incompatible Set of a Node In this candidate elimination technique, we need to know for each node n_i, the set of nodes that n_i is not compatible with. For this reason, let's define N as $N : V_P \mapsto 2^{V_P}$. $N(n_i)$ is called the *incompatible set* of n_i and contains the set of nodes that are not compatible with n_i.

For the uncertainty automaton given in Figure 4.2, the incompatible sets are shown in Table 4.3. If we consider $N(n_3)$, nodes n_5 and n_8 are in $N(n_3)$ since when

$N(n_1) = \{n_2, n_4\}$	$N(n_2) = \{n_1, n_4, n_5, n_8\}$	$N(n_3) = \{n_5, n_8, n_9\}$
$N(n_4) = \{n_1, n_2\}$	$N(n_5) = \{n_2, n_3, n_9\}$	$N(n_8) = \{n_2, n_3, n_9\}$
$N(n_9) = \{n_3, n_5, n_8\}$		

Table 4.3: Incompatible Sets For the Uncertainty Automaton in Figure 4.2

input b is applied node n_3 produces output 0 whereas n_5 and n_8 produce output 1. $N(n_3)$ also contains n_9 since with $b/0$ n_3 enters node n_4 and n_9 enters node n_1 but n_1 and n_4 incompatible. n_1 and n_4 are incompatible because the intersection of their candidate sets is empty.

Finding Incompatible Sets Algorithm 7 gives formal description of finding incompatible sets for nodes. Finding incompatible sets of each node in uncertainty automaton is a process with two phases. In the first phase (lines 1-6) each pair of nodes are considered separately. For any pair of nodes, say n_i and n_j, if $C(n_i) \cap C(n_j)$ is empty then these two nodes are incompatible. Otherwise taking only edges that leaves n_i and n_j into account, compatibility of n_i and n_j is checked. That is, if both nodes have an edge leaving with input symbol x, say $(n_i, n_u; x/y)$ and $(n_j, n_t; x/y')$, then these nodes are incompatible if the output they produce are different ($y \neq y'$).

Since in this first phase only single input symbols are considered for checking compatibility, this phase does not give a final result about the compatibility of two nodes. It only tells if incompatibility of two nodes can be deduced by any input sequence of length 1. However there may be cases that n_i and n_j might be incompatible but it can only be seen when all possible input sequences defined at n_i and n_j are considered. In the second phase of the algorithm (lines 7-12), this case is handled. In this phase, we iterate over a list of node pairs (L) that are incompatible to each other. Initially this list contains all pairs of nodes that are

found to be incompatible in the first phase. Algorithm iteratively removes a pair from the list say n_i and n_j and checks if there are two nodes n_u and n_t such that $(n_t, n_i, x/y)$ and $(n_u, n_j, x/y)$. If this is the case then we can deduce that n_t and n_u are not compatible as well. That is because, n_t and n_u enters to incompatible nodes with same label x/y. Then new pair n_t and n_u is added to list of incompatible nodes if they are not considered before. This is checked by keeping track of each pair considered in this second phase in a separate list (H). After all such nodes that reach to n_i and n_j with the same label are added to the list then the pair n_i, n_j is removed from the list. Algorithm terminates when the list becomes empty, thus all incompatible sets are found.

Algorithm 7: Find Incompatible Sets

Input: $G_P = (V_P, E_P)$ uncertainty automaton

Result: $N(n_i)$ is found for each node $n_i \in V_P$

```
1  L = ∅ ;                         // List of node pairs to be processed
2  foreach pair of nodes (n_i, n_j) in V_P × V_P do
3      if C(n_i) ∩ C(n_j) = ∅ OR (∃x (n_i, n_u, x/y_i), (n_j, n_t, x/y_j) ∈ E_P
       s.t. y_i ≠ y_j then
4          L = L ∪ {(n_i, n_j)};
5          N(n_i) = N(n_i) ∪ {n_j};
6          N(n_j) = N(n_j) ∪ {n_i};
7  H = ∅ ;                  // List of incompatible node pairs processed
8  while L is not empty do
9      pick a pair (n_i, n_j) from L ;
10     L = L \ {(n_i, n_j)};
11     H = H ∪ {(n_i, n_j)};
12     foreach n_t and n_u s.t. (n_t, n_i, x/y) and (n_u, n_j, x/y) do
13         if (n_t, n_u) ∉ H then
14             L = L ∪ {(n_t, n_u)} ;
15         N(n_t) = N(n_t) ∪ {n_u};
16         N(n_u) = N(n_u) ∪ {n_t};
```

The running time of the finding incompatible sets given in Algorithm 7 can be analyzed as follows. In first phase (first for loop) for each pair of nodes the intersection of candidate sets and outgoing edges are considered. Finding intersection of candidate sets takes $O(n^2)$ time and there are $O(p)$ outgoing edges for any node. Since there are $O(r^2)$ node pairs, in the first phase the total time spent is $O(n^2r^2 + pr^2)$. The second phase may also consider all pair of nodes and the most time consuming operation in this phase is to check if a pair of nodes is considered before. Although in the description given in the Algorithm 7 the list H is used to check this, in our implementation we check if a pair of nodes was considered before by looking whether one of the nodes is already in the incompatible set of the other. Since each incompatible set can have at most $O(r)$ nodes, this check is done in $O(r)$ time. Considering all pairs of nodes, the running time of the second phase is $O(r^3)$. Hence the total running time of the algorithm is $O(r^3 + n^2r^2 + pr^2)$.

Candidate Elimination Using a Recognized Node Before explaining the technique *candidate elimination using a set of incompatible nodes*, we will first consider a special case of the technique. We will call this special case as *candidate elimination using a recognized node.*

Consider a node n_i that is recognized as state s of M and suppose that there exists a node n_j which has not been recognized yet. Let's assume n_j is known to be incompatible with n_i (i.e. $n_j \in N(n_i)$), and $s \in C(n_j)$. In other words, s is still a candidate state for n_j but we also know that n_j correspond to a different state than n_i which is recognized as s. This actually proves that n_j cannot be recognized as s and hence s can be removed from $C(n_j)$.

This elimination process is applied to all node pairs (n_i, n_j) such that $C(n_i) = \{s\}$ and $n_j \in N(n_i)$ by setting $C(n_j) = C(n_j) \setminus \{s\}$.

For example, considering the uncertainty automaton in Figure 4.2, the following candidate eliminations using a recognized node is possible. Since node n_2 is recognized as state s_3 and is incompatible with nodes n_5 and n_8, state s_3 is removed from $C(n_5)$ and $C(n_8)$. Table 4.4 shows the updated candidate sets after these eliminations.

Assuming incompatible sets are given, the running time for a single application

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3\}$	$C(n_3) = \{s_1, s_2, s_3\}$	$C(n_4) = \{s_2\}$
$C(n_5) = \{s_1, s_2\}$	$C(n_8) = \{s_1, s_2\}$	$C(n_9) = \{s_1, s_2, s_3\}$	

Table 4.4: Candidate Sets for the Uncertainty Automaton in Figure 4.2

of candidate elimination using recognized nodes is $O(nr)$. That is because there can be at most n recognized nodes and for each of them we can do eliminations on all the nodes in its incompatible sets each of which can have at most $O(r)$ nodes. If the method achieves an elimination then it has to be applied again. The method can be applied until all nodes recognized or no more elimination possible. However before each application of this candidate elimination method incompatible sets have to be recomputed since as a result of candidate eliminations incompatibility relations may change. For this reason we have to add the running time of finding incompatible sets to the running time of the method. Hence the running time of a single application of the method is $O(nr) + O(r^3 + n^2r^2 + pr^2) = O(r^3 + n^2r^2 + pr^2)$. As we see, running time for finding incompatible sets dominates the running time for a single application of the method, hence the first term $O(nr)$ is dropped. When we analyze how many times the method can be applied, we can say that in the worst case the method have to be applied $O(nr)$ times. That is because in the worst case, each application eliminates a single element from the candidate set of a single node. Hence in the worst case the running time of applying candidate elimination using a recognized node is $O(nr) \times O(r^3 + n^2r^2 + pr^2) = O(nr^4 + n^3r^3 + pnr^3)$.

Candidate Elimination Using a Set of Incompatible Nodes Note that the technique explained above cannot be applied when $|C(n_i)| > 1$. Although for a node $n_j \in N(n_i)$ it is guaranteed that n_i and n_j will correspond to two different states since the state corresponding to n_i is not found yet we cannot simply remove the entire set of states in $C(n_i)$ from $C(n_j)$.

However there is still a further chance for candidate elimination using a similar idea. Let's start by considering such an elimination on a simple case. Assume that there is a set consisting of two nodes n_i and n_j both of which are not recognized yet (i.e. $|C(n_i)| > 1$ and $|C(n_j)| > 1$) and are known to be incompatible (i.e. $n_j \in N(n_i)$ and $n_i \in N(n_j)$). If n_i and n_j have candidate sets such that $|C(n_i) \cup C(n_j)| = 2$,

then that means there are 2 candidate states that n_i and n_j can be recognized as. Since we also know that n_i and n_j is incompatible, n_i and n_j will be recognized as different states in $C(n_i) \cup C(n_j)$. Further let's assume that there is a third node n_u that is also not recognized yet and is known to be incompatible with both n_i and n_j. This incompatibility tells us that n_u cannot be recognized as any of the 2 states in $C(n_i) \cup C(n_j)$, thus the elimination $C(n_u) = C(n_u) \setminus (C(n_i) \cup C(n_j))$ is a valid operation.

As we have seen in the example above, there are cases when we can definitely know that a node cannot be recognized as a set of states rather than a single state. Thus elimination of multiple states from the candidate set of a node at once is possible. When we generalize this idea, we come up with the following formulation. Assume that for a set of k nodes, say $K = \{n_1, n_2, ..., n_k\}$ where $|K| = k$, the following conditions hold

1. if $k > 1$ then $\forall n_i \in K, |C(n_i)| > 1$
2. if $k > 1$ then $\forall n_i, n_j \in K$ if $n_i \neq n_j$ then $n_j \in N(n_i)$
3. if $k > 1$ then $|\bigcup_{i=1}^{k} C(n_i)| = k$
4. if $k = 1$ then $|C(n_1)| = 1$

In simple words, condition 1 states each node in K has not been recognized yet. Condition 2 states each node in K is incompatible with every other node in K. That is no two nodes in K can be recognized as the same state. Condition 3 states that there are k possible states that nodes in K can be recognized as. Hence combining condition 2 and 3, it is obvious that each node in K will be recognized as one of the k possible states and no other node in K will be recognized as that state. Although we have no information about which node will be recognized as which state, this is not necessary for elimination.

If there is any node, say n_u, different than the nodes in K and is incompatible with all the nodes in K then we can do the following elimination $C(n_u) = C(n_u) \setminus \bigcup_{i=1}^{k} C(n_i)$. Notice that all conditions except the last one assumes the case $k > 1$. That is because the case $k = 1$ refers to the special case where the only node in K must be already recognized and that is same as the special case examined as candidate elimination

using a recognized node.

Although candidate elimination method presents an important opportunity, finding a set K satisfying the conditions becomes more expensive as k gets large. In fact, finding a set K is same as solving the famous Clique problem on undirected graphs. That is because, let's assume we have found a set of nodes U with $|U| = m \geq k$ that satisfies conditions 1 and 3. Then we need to find a subset $K \subseteq U$ such that K satisfies condition 2. If we consider each node in U as a node of an undirected graph and put an undirected edge between the nodes that are incompatible with each other, then finding a clique of size k in this graph solves our problem of finding a subset K. Since clique problem is NPC, then candidate elimination using a set of incompatible nodes leads to exponential running time.

Considering the performance of the method, the maximum cardinality of the set of incompatible nodes, (k) can be given as a parameter. This provides a tuning chance for the trade off between running time and finer analysis. Searching for a set of incompatible nodes K with large cardinality,(k), may yield better results but increases the running time of the analysis.

4.3.2 Candidate Elimination Using Candidate Trial

Another method that allows elimination of candidates from the candidate set of an unrecognized node is what we call as candidate trial. In this method, for an unrecognized node, say n_i, a candidate state $s \in C(n_i)$ is chosen. Then a what-if analysis is performed assuming that n_i is recognized as s. In other words, a copy of the current uncertainty automaton is created and n_i is recognized as s on the copy automaton. n_i is recognized as s by simply merging it with the node that is already recognized as s, if there is such node. If there is not, that is n_i is the first node that will be recognized as s, then setting $C(n_i) = \{s\}$ is enough. After n_i is recognized as s on the copy automaton, the analysis continues using the methods explained before and it is checked that whether recognizing n_i as s causes a contradiction at some point. A contradiction is reached while recognitions and candidate eliminations performed as usual, at some point two nodes, say n_t and n_u needs to be merged but either $C(n_t) \cap C(n_u)$ is empty or for some input symbol x, n_t and n_u produce different outputs. If such a contradiction is reached at some point, then it is sure

that n_i should not be recognized as s. Hence s can be eliminated from $C(n_i)$ in the current uncertainty automaton. If no contradiction is reached then we can only conclude that in the current state of the uncertainty automaton, there is still chance for n_i to be recognized as s. Thus at this point elimination of s from $C(n_i)$ is not possible.

Considering the performance of the method, we think that it is reasonable not to use candidate trial method in a nested fashion. Although it is possible to put a limit on the depth of the nested candidate trial method calls, in our implementation we do not call candidate trial method within another candidate trial method call. This is simply because, not limiting candidate trial calls mean trying every possibility for every unrecognized node and that may increase running time drastically.

Considering the uncertainty automaton in Figure 4.2, we can continue candidate eliminations using the techniques explained above. When we want to use candidate elimination using set of incompatible nodes, there are two such sets satisfying the conditions of the method. One is the set of nodes $\{n_3, n_5, n_9\}$ and the other is $\{n_3, n_8, n_9\}$. However for both sets there is not any other node that is incompatible with all of the nodes in one of the sets, thus no elimination is possible.

However we can continue candidate eliminations using candidate trial method. Since among unrecognized nodes, node n_5 have only two candidates remaining, we like to consider candidate trial on node n_5 first. So assume that node n_5 is recognized as state s_1 on a copy of the current uncertainty automaton and this assumption leads to following merges and candidate eliminations on the copy uncertainty automaton.

- Node n_5 is merged with node n_1 (since we assume they are recognized as same state)

- State s_1 is eliminated from $C(n_9)$ (since node n_1 becomes incompatible with node n_9)

- State s_1 is eliminated from $C(n_8)$ (since node n_1 becomes incompatible with node n_8, thus node n_8 recognized as state s_2)

- State s_1 is eliminated from $C(n_3)$ (since node n_1 becomes incompatible with node n_3)

- Node n_8 merged with node n_4 (since they both recognized as state s_2)
- State s_2 is eliminated from $C(n_9)$ (since node n_9 becomes incompatible with node n_4, thus node n_9 recognized as state s_3)
- State s_2 is eliminated from $C(n_3)$ (since node n_3 becomes incompatible with node n_4,thus node n_3 recognized as state s_3)

After all of these merges and eliminations, the resulting copy uncertainty automaton is shown in Figure 4.3.

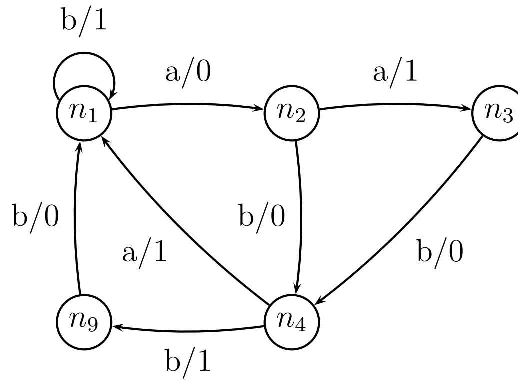

Figure 4.3: Copy Uncertainty Automaton

Since nodes n_3 and n_9 are now recognized as state s_3 and node n_2 has already been recognized as state s_3, these three nodes have to be merged. However node n_9 and node n_2 happens to be incompatible, since with $b/0$ they reach nodes that are recognized as different states (node n_2 reaches node n_4 that has been recognized as state s_2 and node n_9 reaches n_1 that has been recognized as state s_1). Thus there is a conflict and that means our assumption of recognizing node n_5 as state s_1 was false. So state s_1 can be eliminated from $C(n_5)$. That leaves only state s_2 in $C(n_5)$, hence node n_5 is now recognized as state s_2 and must be merged with node n_4.

Merging node n_5 with node n_4 makes the following candidate eliminations possible via candidate elimination using a recognized node.

- State s_2 is eliminated from $C(n_9)$ (since node n_9 is incompatible with node n_4)

- State s_2 is eliminated from $C(n_3)$ (since node n_9 is incompatible with node n_4)

The resulting uncertainty automaton is shown in Figure 4.4, the corresponding candidate sets in Table 4.5 and the incompatible sets in Table 4.6.

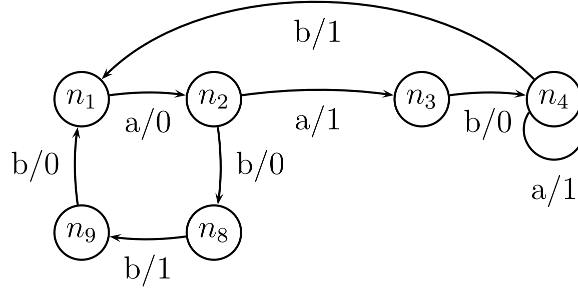

Figure 4.4: Uncertainty Automaton

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3\}$	$C(n_3) = \{s_1, s_3\}$
$C(n_4) = \{s_2\}$	$C(n_8) = \{s_1, s_2\}$	$C(n_9) = \{s_1, s_3\}$

Table 4.5: Candidate Sets For the Uncertainty Automaton in Figure 4.4

$N(n_1) = \{n_2, n_4\}$	$N(n_2) = \{n_1, n_4, n_8\}$	$N(n_3) = \{n_4, n_8, n_9\}$
$N(n_4) = \{n_1, n_2, n_3, n_9\}$	$N(n_8) = \{n_2, n_3, n_9\}$	$N(n_9) = \{n_3, n_4, n_8\}$

Table 4.6: Incompatible Sets For the Uncertainty Automaton in Figure 4.4

Now there is a possibility for candidate elimination using a set of incompatible nodes. Considering the set of nodes $K = \{n_3, n_9\}$, it can be seen that they form a set that can be used for elimination (since $C(n_3) \cup C(n_9) = \{s_1, s_3\}$ then $|K| = |C(n_3) \cup C(n_9)| = 2$ also node n_3 is incompatible with node n_9). If we consider node n_8 which is incompatible with both n_3 and n_9, then state s_1 can be eliminated from $C(n_8)$. Then node n_8 is recognized as state s_2 and must be merged with node n_4. This merge leads to merging of nodes n_1 and n_9. The resulting uncertainty automaton is shown in Figure 4.5 with candidate sets shown in Table 4.7.

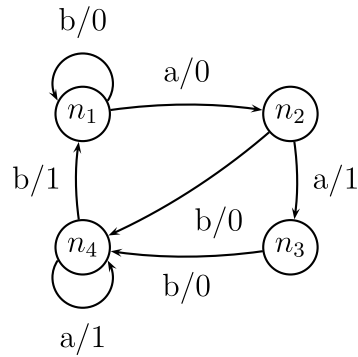

Figure 4.5: Uncertainty Automaton

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3\}$	$C(n_3) = \{s_1, s_3\}$	$C(n_4) = \{s_2\}$

Table 4.7: Candidate Sets For the Uncertainty Automaton in Figure 4.5

The only unrecognized node is node n_3 with candidate set $C(n_3) = \{s_1, s_3\}$. Making a candidate elimination using the recognized node n_1, it is possible to eliminate s_1 from $C(n_3)$ and recognize it as state s_3. Hence node n_3 must be merged with node n_2. Now all nodes are recognized as some state of FSM M_1. The final automaton is shown in Figure 4.6 with candidate sets shown in Table 4.8. Notice that the final automaton is now equivalent to FSM M_1 with all nodes recognized and all transitions of M_1 are verified. As a result it is concluded that X is a checking sequence for FSM M_1.

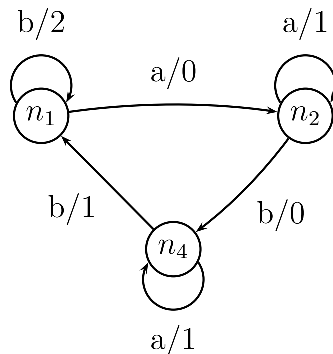

Figure 4.6: Final Uncertainty Automaton

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3\}$	$C(n_4) = \{s_2\}$

Table 4.8: Candidate Sets For the Uncertainty Automaton in Figure 4.6

4.3.3 Using Candidate Elimination Methods Together

After being seen the methods we use for candidate elimination, in this section we present how we use these methods together. The main consideration we have is to use the method that is cheaper as much as possible. Whenever a method fails to update the uncertainty automaton then we proceed to the next method which makes a more expensive analysis then the current method. What we mean by an update of the uncertainty automaton is an elimination of a candidate from the candidate set of any node. In addition to that whenever a method achieves an update, we brake the execution of the current method and continue with a cheaper method if possible. Hence we run the methods in the following order until none of the methods are able to update the uncertainty automaton or all nodes in the uncertainty automaton have been recognized.

1. Candidate Elimination Using a Recognized Node
2. Candidate Elimination Using a Set Of Incompatible Nodes
3. Candidate Elimination Using Candidate Trial

4.4 Thoughts on Uncertainty Automaton

As explained above, given an FSM M, an I/O sequence X/Y (which can be inferred from a given input sequence X by tracing it on M starting from s_1), together with an ADS $\bar{D}$, induces an uncertainty automaton N for an FSM M by using the method explained in this section. Although not proved formally, it is easy to see intuitively that if the uncertainty automaton N has as many states as M and is equivalent to M, then X is a checking sequence for M.

However, even if X is really a checking sequence for M, the method may not produce an uncertainty automaton equivalent to M. This may happen since X might actually be a checking sequence not using $\bar{D}$, or there might be other recognitions which cannot be performed by using our state recognition techniques. Therefore

when the final uncertainty automaton is not equivalent to M, we cannot be sure whether X is a checking sequence or not.

We believe that even when a given input sequence X does not produce an uncertainty automaton N equivalent to M, N can be used to decide how close X is being a checking sequence. The difference in the number of states and the sizes of the candidate states at the nodes of N can be used to produce such a metric.

Chapter 5

Overview of Simão *et al.*'s Method

In [24], Simão *et al.* presents a constructive method for generating checking sequences using distinguishing sets. In this section we provide a short description of the algorithm using our own notation.

Given an FSM M and a distinguishing set $\bar{D} = \{D_{s_1}, D_{s_2}, ...D_{s_n}\}$ of M, the algorithm generates a checking sequence $Q = x_1x_2...x_k$. For a sequence Q, let $P(Q)$ denote the path that starts from the initial state of M such that Q is the input portion of the label of $P(Q)$. That is $P(Q) = (n_1, n_2; x_1/y_1), (n_2, n_3; x_2/y_2), ..., (n_r, n_{r+1}; x_r/y_r)$. The algorithm iteratively constructs the sequence Q such that in the corresponding path $P(Q)$ all transitions of M are verified. Thus when all transitions are verified, Q becomes a checking sequence of M and algorithm terminates. Let Q_i denote the prefix of the checking sequence Q such that Q_i is obtained at the end of the ith iteration. Similarly let $P(Q_i)$ be the corresponding path for Q_i. At iteration i, algorithm produces the sequence Q_i by extending the sequence Q_{i-1}. Initially it is supposed that the implementation is at the initial state s_1 and in order to recognize it D_{s_1} has to be applied. Thus the sequence Q_1 is always D_{s_1}. In iteration i how to extend the sequence Q_{i-1} is decided based on whether the end vertex of $P(Q_{i-1})$ is recognized or not. In this method, a vertex in $P(Q)$ is recognized when it is *d-recognized* or *t-recognized* as explained in Section 2.4. If the end vertex of $P(Q_{i-1})$ is recognized then a transition verification sequence is appended to the current sequence, otherwise a state recognition is done by appending some identification sequence. Notice that in each case the sequence appended always ends with an identification sequence, hence when a state recognition is attempted the

longest possible overlapping between the identification sequences is considered. A more formal description of method is given in Algorithm 8.

Algorithm 8: Simão *et al.*'s Checking Sequence Generation Method as in [24]

Input: $\bar{D} = \{D_{s_1}, D_{s_2}, ..., D_{s_n}\}$ a distinguishing set for an FSM M

Output: Q a checking sequence for M

1. Q_0 is the empty sequence ;
2. $i = 1$;
3. **while** *there are unverified transitions* **do**
4. let $s_k = \delta(s_1, Q_{i-1})$;
5. **if** *end of* $P(Q_{i-1})$ *is recognized (as* s_k *of* M*)* **then**
6. Find a shortest verified transfer sequence β from s_k to some state s_j, such that s_j has some unverified transition $(s_j, s_u; x/y)$;
7. $Q_i = Q_{i-1}\beta x D_{s_u}$;

   else
8. Find the longest suffix χ of Q_{i-1} such that $Q_{i-1} = \alpha\chi$ and χ is also a prefix of D_{s_u}, where $s_u = \delta(s_1, \alpha)$, and the end vertex of $P(\alpha)$ is not recognized ;
9. $Q_i = Q_{i-1}\phi$ where $D_{s_u} = \chi\phi$;
10. Update recognized vertices in $P(Q_i)$;
11. Update verified transitions ;

Since the aim of the algorithm is to obtain a sequence that verifies every transition, after each extension to the current sequence the set of verified transitions must be updated. This is necessary for both cases. In case 1 when a transition verification sequence is appended to current sequence it is obvious that a transition will be verified, but note that a transition verification may lead to other transition verifications via *t-recognitions*. Hence more than one transition can be verified within an iteration. Likewise in case 2, that is when a suffix of identification sequence is appended to the current sequence, a previously unrecognized vertex will be *d-recognized* and that may also lead to transition verifications.

Chapter 6

Our Checking Sequence Generation Method

In this section a new method to generate checking sequences using distinguishing sets will be presented. Similar to Simão *et al.*'s [24] method, this method also constructs a checking sequence by extending the current sequence in each iteration. However unlike the method in [24], the method consists of two phases. In the first phase an input sequence Q is generated but Q is not guaranteed to be a checking sequence. If it is not, then method enters second phase and does some post-processing. In this post-processing phase, Q is further extended until it becomes a checking sequence.

6.1 Phase 1: Sequence Generation

In the first phase of the method, an input sequence Q, which may not be a checking sequence, is constructed iteratively. In this method, recognition of a vertex in $P(Q)$ can be achieved with *d and t-recognitions* as usual. However in our method a vertex can also be recognized conditionally. A conditional recognition of the start of an edge $(n_i, n_{i+1}; x/y)$ in $P(Q)$ is possible if this edge corresponds to an invertible transition $(s, s'; x/y)$ in FSM M. An invertible transition is defined as follows.

Definition 4. *A transition* $(s, s'; x/y)$ *is* invertible *if* $\forall s'' \in S$ *such that* $s'' \neq s$, *either* $\delta(s'', x) \neq s'$ *or* $\lambda(s'', x) \neq y$.

In simple words, a transition $(s, s'; x/y)$ is *invertible* if it is the only transition entering state s' with input x and output y.

Although in a different context, the idea of using invertible transition to reduce the checking sequence length has been suggested before in [12, 13, 6].

If $(s, s'; x/y)$ is an invertible transition of FSM M the recognition of the start vertex n_i of the edge $(n_i, n_{i+1}; x/y)$ as s in $P(Q)$ is possible when the following conditions are met.

- the end vertex n_{i+1} is recognized as state s' of M and
- For each state s'' of M such that $s'' \neq s$, $P(Q)$ contains an edge $(n_j, n_{j+1}; x/y')$ such that n_j is recognized as s'' and either $y' \neq y$ or n_{j+1} is recognized as some state different than s'

In simple words, recognized vertices of $P(Q)$ have to provide enough evidence that all states except s when input x is applied either produces an output different than y or enters a state different than s'. Thus, if an unrecognized vertex n_i in $P(Q)$ enters state s' with input x and output y, then the only remaining state that n_i can be recognized as is s. Invertibility of transition $(s, s'; x/y)$ is crucial in such conditional recognitions, since otherwise there would be at least one other state s'' different than s that also enters s' with input x and output y. In that case n_i cannot be recognized conditionally since there is not enough evidence to know whether n_i should be recognized as s or s''.

A conditional recognition is valid only when all conditions are satisfied. Hence it should be checked that if the conditions are satisfied. However in this method, we do not check if any of the conditions is satisfied in the first phase. Instead if there is an edge $(n_i, n_{i+1}; x/y)$ in $P(Q)$ and it corresponds to an invertible transition $(s, s'; x/y)$, then n_i directly assumed to be recognized as s without considering if there is enough evidence in $P(Q)$ to validate this recognition. That is why the sequence generated in the first phase of the algorithm is not guaranteed to be a checking sequence and some post processing may become necessary to obtain a checking sequence.

A description of the first phase of method is presented in Algorithm 9. At the end of the first phase, generated sequence Q is checked to see if it is a checking sequence using the method described in Chapter 4. If Q is a checking sequence the algorithm terminates. Otherwise Phase 2 of the algorithm is executed to extend Q to a checking sequence.

Algorithm 9: Phase 1

Input: $\bar{D} = \{D_{s_1}, D_{s_2}, ..., D_{s_n}\}$ a distinguishing set for an FSM M

Output: Q a possible checking sequence for M

```
1  Q_0 is the empty sequence ;
2  i = 1;
3  while there are unverified transitions do
4      let s_k = δ(s_1, Q_{i-1}) ;
5      if end vertex of P(Q_{i-1}) is recognized (as s_k of M) then
6          Find a shortest verified transfer sequence β from s_k to some state s_j,
           such that s_j has some unverified transition (s_j, s_u; x/y) ;
7          Q_i = Q_{i-1} β x D_{s_u} ;
       else
8          Q_i = Q_{i-1} D_{s_k} ;
9      Update recognized vertices in P(Q_i) ;
10     Update verified transitions ;
```

The length of the sequence generated by Phase 1 of the algorithm can be analyzed as follows. To be able to verify a transition, the last vertex of the sequence must be recognized. In worst case, the last vertex of the sequence is recognized after identification sequence is appended $(n + 1)$ times. Let $|\bar{D}|$ denote the length of the longest identification sequence in the distinguishing set $\bar{D}$. Then the last vertex of the sequence is recognized after at most $(n + 1) \times |\bar{D}|$ input symbols are appended. When the last vertex is recognized we may need to append a transfer sequence. The maximum length of a transfer sequence can be n in case all states must be visited. In addition to that since a transition verification sequence is concatenation of an input symbol and an identification sequence, its length can be at most $|\bar{D}|+1$. Summing all these gives the length of sequence required for a single transition verification. Since there are np transitions then upper bound for the length of generated sequence can be calculated by $np \times ((n + 1)|\bar{D}| + n + |\bar{D}| + 1)$ which is $O(n^2p|\bar{D}|)$.

The running time of Phase 1 is dominated by updating recognized vertices in the current sequence. Given a sequence of length l updating the recognized vertices in that sequence takes $O(l^2)$ time. Since the length of the sequence is generated in

Phase 1 is $O(n^2p|\bar{D}|)$, a single iteration of the while loop takes $O(n^4p^2|\bar{D}|^2)$. While loop iterates at most np times, hence the running time of Phase 1 is $O(n^5p^3|\bar{D}|^2)$.

Example

We will now illustrate the execution of Phase 1 will on the FSM M_1 shown in Figure 2.1 with the distinguishing set $\bar{D} = \{D_{s_1}, D_{s_2}, D_{s_3}\}$ where $D_{s_1} = a, D_{s_2} = ab, D_{s_3} = ab$. First note that all transitions in FSM M_1 is invertible so all nodes can be conditionally recognized if needed. Initially we have the empty sequence Q_0, hence we extend it by $D_{s_1} = a$. Thus $Q_1 = a$, $P(Q_1) = (n_1, n_2; a/0)$ and n_1 is *d-recognized* as state s_1. From now on, we will show iterations of algorithm using a table format. For the first iteration, Table 6.1 shows the current sequence Q_1 in the first row. Second row shows the initial vertex of the corresponding path $P(Q_1)$ for the input at the same column. Third row shows if the corresponding vertex in the second row is recognized, D is used for d-recognition, T is used for t-recognition and C is used for conditional recognition. Last row shows the corresponding states of M_1. That is, we can read Table 6.1 as, current input sequence Q_1 is a, in $P(Q_1)$ vertex n_1 enters vertex n_2 with input a and n_1 is *d-recognized* as state s_1, whereas n_2 is not recognized yet.

Sequence	a	
Vertex	n_1	n_2
Recognition	D	
State	s_1	s_3

Table 6.1: Iteration 1

In the second iteration the end vertex in $P(Q_1)$, n_2, is not yet recognized, thus we extend sequence by $D_{s_3} = ab$. With this extension n_2 is d-recognized and n_3 is conditionally recognized. Also the transitions $(s_1, s_3; a/0)$ and $(s_3, s_3; a/1)$ are verified. Iteration 2 is shown in Table 6.2.

In the third iteration the end vertex in $P(Q_2)$, n_4, is not yet recognized, thus we extend sequence by $D_{s_2} = ab$ as shown in Table 6.3. In this iteration, transitions $(s_3, s_2; b/0)$ and $(s_2, s_2; a/1)$ are verified.

In the fourth iteration the end vertex in $P(Q_3)$, n_6, is not yet recognized, thus

Sequence	a	a	b	
Vertex	n_1	n_2	n_3	n_4
Recognition	D	D	C	
State	s_1	s_3	s_3	s_2

Table 6.2: Iteration 2

Sequence	a	a	b	a	b	
Vertex	n_1	n_2	n_3	n_4	n_5	n_6
Recognition	D	D	C	D	C	
State	s_1	s_3	s_3	s_2	s_2	s_1

Table 6.3: Iteration 3

we extend sequence by $D_{s_1} = a$ as shown in Table 6.4. In this iteration, transition $(s_2, s_1; b/1)$ is verified.

Sequence	a	a	b	a	b	a	
Vertex	n_1	n_2	n_3	n_4	n_5	n_6	n_7
Recognition	D	D	C	D	C	D	T
State	s_1	s_3	s_3	s_2	s_2	s_1	s_3

Table 6.4: Iteration 4

In the fifth iteration, the end vertex in $P(Q_4)$, n_7, is recognized as s_3. Since all transitions of s_3 are verified, we need to transfer to a state with an unverified transition. Actually the only transition that remains unverified is $(s_1, s_1; b/0)$. Hence we transfer to s_1 with sequence *bb* first and then verify the transition with $bD_{s_1} = ba$. Hence in this iteration the sequence is extended by *bbba* as shown in Table 6.5. As every transition is now verified, sequence generation in Phase 1 is finished. If we check whether the generated sequence $Q = Q_5$ is a checking sequence using the method in Chapter 4, we see that Q is a checking sequence. Actually the example sequence used in Chapter 4 is the same sequence generated here and in that section it is shown that this sequence is checking sequence for M_1.

Sequence	a	a	b	a	b	a	b	b	b	a	
Vertex	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}
Recognition	D	D	C	D	C	D	T	T	T	D	T
State	s_1	s_3	s_3	s_2	s_2	s_1	s_3	s_2	s_1	s_1	s_1

Table 6.5: Iteration 5

6.2 Phase 2: Extending Sequence Q to a Checking Sequence

If the sequence generated in Phase 1, namely Q, is not a checking sequence, then Phase 2 of the algorithm is executed. In this post processing phase, Q is extended with some identification sequences. Just after Phase 1, using the method in Chapter 4 the sequence is checked whether it is a checking sequence or not. If Q cannot be understood to be a checking sequence, that means the final uncertainty automaton for Q has some unrecognized nodes. Hence in Phase 2, what we simply do is to find an unrecognized node in uncertainty automaton and extend the sequence with the corresponding identification sequence for that unrecognized node. The recognition of an unrecognized node may result in new recognitions via candidate eliminations and merges on the uncertainty automaton. If there are still unrecognized nodes remaining in the uncertainty automaton then the sequence is further extended until no unrecognized nodes remains in the uncertainty automaton, thus sequence becomes a checking sequence.

Among unrecognized nodes in the uncertainty automaton which one to recognize is chosen using a very simple approach. Assume that in the uncertainty automaton the current node is n_i. From n_i a shortest transfer sequence β (possibly empty) to an unrecognized node n_j is found in the uncertainty automaton. If n_j has to be recognized as s, then the sequence βD_s, where D_s is the identification sequence of s, is appended to the sequence.

However there might be some cases such that in uncertainty automaton a transfer sequence to some unrecognized node cannot be found from the current node n_i. That case happens when every node that is reachable from n_i is already recognized and at least one of these nodes has an undefined transition. Then we find the shortest

transfer sequence β to such a node. Assume that n_j is that node which is recognized as s and has some undefined transition, say x. Then the sequence is extended by the transition verification sequence appended to β, namely $\beta x D_{s'}$ where $s' = \delta(s, x)$. An application of this case can be seen in the example given in this section

Example

Now the execution of Phase 2 will be illustrated on an example. Consider the FSM M_2 given in Figure 6.1 with the distinguishing set $\bar{D} = \{D_{s_1}, D_{s_2}, D_{s_3}, D_{s_4}\}$ where $D_{s_1} = ab$, $D_{s_2} = ab$, $D_{s_3} = aa$ and $D_{s_4} = aa$.

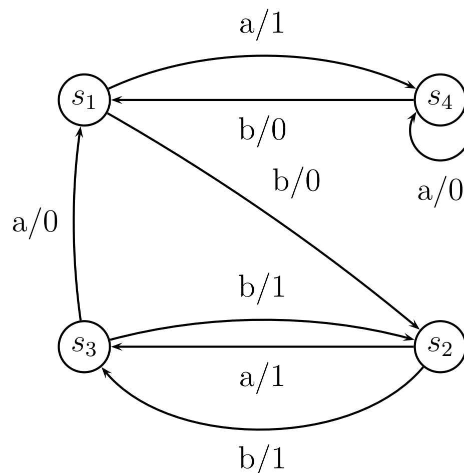

Figure 6.1: FSM M_2

Phase 1 generates the sequence $Q = ababbababbaaaaa$ for the FSM M_2 with the given distinguishing set $\bar{D}$. When it is checked whether Q is a checking sequence for M_2, the method described in Chapter 4 produces the final uncertainty automaton shown in Figure 6.2 with the candidate sets given in Table 6.6. Since there are still some unrecognized nodes in the uncertainty automaton, Q is not a checking sequence. Thus in Phase 2, Q needs to be extended to become a checking sequence for M_2.

$C(n_1) = \{s_1\}$	$C(n_2) = \{s_3, s_4\}$	$C(n_6) = \{s_2\}$	$C(n_7) = \{s_3, s_4\}$
$C(n_{11}) = \{s_3\}$	$C(n_{12}) = \{s_1, s_2\}$	$C(n_{13}) = \{s_4\}$	

Table 6.6: Candidate Sets For the Uncertainty Automaton in Figure 6.2

Notice that in the uncertainty automaton nodes n_2, n_7 and n_{12} are unrecognized, each having multiple states in their candidate sets. In order to extend sequence Q into a checking sequence, we need to recognize all nodes in the uncertainty automa-

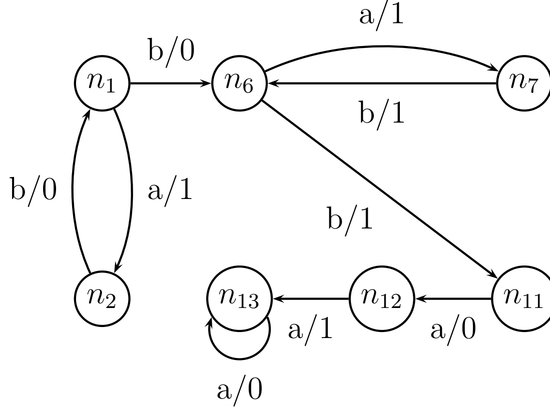

Figure 6.2: Final Uncertainty Automaton for Q generated in Phase 1

ton. Hence firstly we like to transfer to one of these unrecognized nodes using the shortest possible transfer sequence and recognize that node with the corresponding identification sequence from $\bar{D}$. However when sequence Q is traced (starting from the node that is recognized as the initial state s_1, i.e. node n_1) on the uncertainty automaton, the last node happens to be n_{13}. Note that node n_{13} is already recognized as s_4. As the only edge leaving node n_{13} is a self-loop, there is no way to transfer to an unrecognized node. The reason for that is node n_{13} have no defined edge with input b. In other words b transition of s_4 has not been verified yet, because possibly in Phase 1 some conditions that is required for a conditional recognition remained unsatisfied. To get rid of this situation, we extend Q to verify b transition of s_4. Hence the sequence $bD_{s_1} = bab$ is appended to Q (note that $\delta(s_4, b) = s_1$). When we check whether the extended sequence $Q' = Qbab$ is a checking sequence for M_2, the resulting uncertainty automaton happens to have all nodes recognized and all transitions verified as shown in Figure 6.3 and candidate sets in Table 6.7. Thus Q' is a checking sequence for M_2.

$C(n_1) = \{s_1\}$	$C(n_6) = \{s_2\}$	$C(n_{11}) = \{s_3\}$	$C(n_{13}) = \{s_4\}$

Table 6.7: Candidate Sets For the Uncertainty Automaton in Figure 6.3

6.3 Experimental Results

In this section the experimental results for the checking sequence generation method will be discussed. The methods have been implemented with Java and the exper-

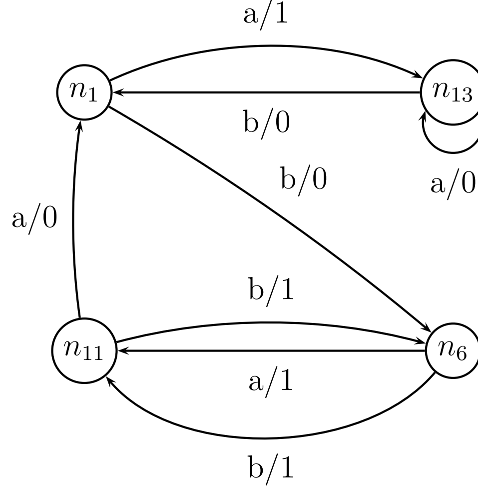

Figure 6.3: Final Uncertainty Automaton for $Q' = Qbab$

iments have been executed on a machine with Intel Xeon 2.33 GHz and 32 GB ram.

The FSMs that is used in experiments are generated using the random FSM generation tool whose details explained in Chapter 3. For the experiments 10 sets of FSMs are used. Each set of FSMs contain 200 FSMs having number of states n, where n is ranging from 10 to 100 (increasing with a step size 10). Each FSM has 5 input symbols and 5 output symbols. Also each FSM has a PDS. In this experimental setup PDS is used instead of distinguishing sets. That is because for a given FSM, the tool we are using is biased toward finding distinguishing sets that generally contain repetitions of the same input symbol. We call such sequences as uniform sequences. However for finding PDS there is no such bias. If an identification sequence is uniform, then this situation increases the chances of overlapping among identification sequences and causes biased results.

We are going to compare the performance of our method with *Simão et al.*'s method given in [24]. The comparisons will be in terms of checking sequence length and method execution time.

6.3.1 Comparison with *Simão et al.*'s Method

For the experimental results that will be presented in this section our method uses *candidate elimination using a recognized node* as the only candidate elimination method while checking if the sequence generated is a checking sequence. That is *candidate elimination using candidate trial* and *candidate elimination using a set of*

incompatible nodes are not used due to their high computational costs. However the experiments show that even with this reduced recognition ability our method can outperform *Simão et al.*'s method.

Number of States	Our Method	*Simão et al.*'s Method
10	179	207
20	451	528
30	788	893
40	1172	1320
50	1476	1665
60	1856	2043
70	2267	2492
80	2787	3046
90	3269	3559
100	3644	3944

Table 6.8: Average CS Lengths

Table 6.8 shows for each set of 200 FSMs with number of states ranging 10 to 100, the average checking sequence length of our method and *Simão et al.*'s method. Figure 6.4 shows the data in Table 6.8 as a chart.

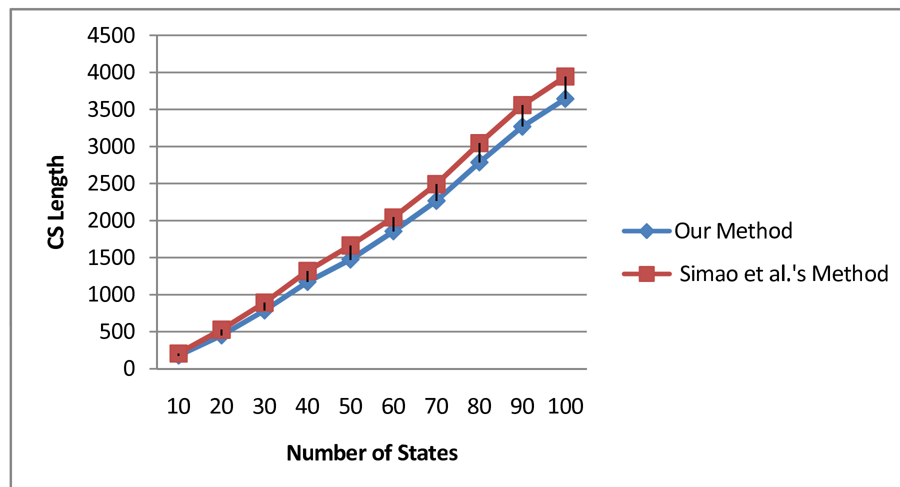

Figure 6.4: Average CS Lengths

Table 6.9 contains the average improvement over *Simão et al.*'s method in terms

of checking sequence length. First column shows the average improvement calculated for all FSMs. Second and third columns show the average improvement when FSMs with non-uniform distinguishing sequences and FSMs with uniform distinguishing sequences considered separately. The values are calculated by averaging the improvements of each FSM in a set. In Figure 6.6 average improvements are shown as a chart.

Considering the performance of our method only, Figure 6.5 shows the box plot for CS lengths. A box plot is interpreted as follows. For each set of FSMs on the horizontal axis the plot contains a box on a vertical line segment. The ends of a line segment marks the minimum and maximum values. The box contains the middle 50% of the values and its upper and lower edges are on the 75th percentile and 25th percentile respectively. The line inside the box shows the median value.

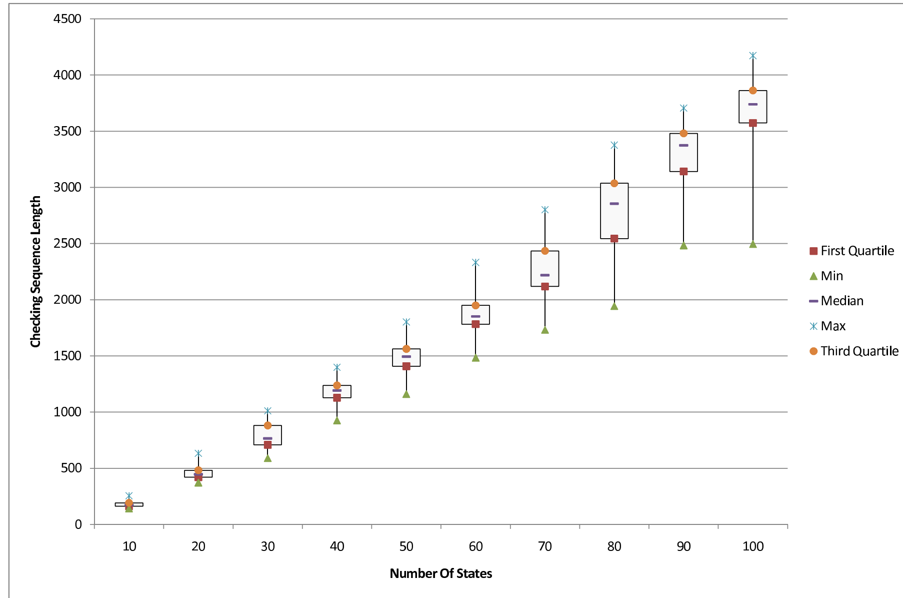

Figure 6.5: Our Method's CS Lenghts as a Box Plot

When FSMs with non-uniform distinguishing sequences are considered, our method on average produces significantly shorter checking sequences. The average improvement varies between $7,95\%$ and $15,23\%$. However when FSMs with uniform distinguishing sequences are considered our method's improvement is not significant and both methods have more or less the same performance. This performance difference between uniform and non-uniform distinguishing sequences stem from the fact that *Simão et al.*'s method uses overlapping between distinguishing sequences, but our method have no such consideration. Our method does not search for overlapping since using conditonal recognition we are able recognize nearly all nodes in the sequence (only if a transition is not invertible then conditional recognition does not work). However when distinguishing sequence is uniform then *Simão et al.*'s method can find many overlaps, thus it can also recognize many nodes like our method. For non-uniform distinguishing sequences chances of overlap greatly reduces, thus our method creates a significant difference in checking sequence length.

In Figure 6.7, the improvement of our method over *Simão et al.*'s method in checking sequence length shown as a box plot. It can be seen that although for some FSMs our method may perform worse than *Simão et al.*'s method, in majority of the cases it performs better and improvement can be as high as 30%.

When method execution times are considered, Figure 6.8 shows the average run-

Number of States	All FSMs (%)	FSMs with non-uniform DS (%)	FSMs with uniform DS (%)
10	12,61	12,99	2,14
20	14,22	15,23	2,64
30	11,50	12,78	1,63
40	11,08	11,65	2,08
50	11,32	11,55	2,39
60	9,06	9,34	1,17
70	8,88	9,20	2,09
80	8,36	8,80	1,37
90	8,07	8,34	-2,73
100	7,46	7,95	0,88

Table 6.9: Average Improvements Over *Simão et al.*'s Method

ning times of our method and *Simão et al.*'s method together as a chart. The results show that there is no significant difference in running time, with this experimental setup where we did not use expensive candidate elimination techniques. However the length of the sequence is reduced by at least 7% in our experiments on FSMs upto 100 states.

6.3.2 Contributions of Phase 1 and Phase 2

In this section, we will analyze experimental results for our method only and present the contributions of Phase 1 and Phase 2 to the average checking sequence length and average execution times.

Figure 6.9 shows the contributions of Phase 1 and Phase 2 to the average checking sequence lengths. In Figure 6.11 the average distribution of the execution time between Phase 1 and Phase 2 is shown. The percentage contribution of Phase 2 to the checking sequence length and time increases with the size of the FSM. Although the time taken by Phase 2 seems to contribute more and more as the size of the FSM grows, the percentage contribution of Phase 2 to the length of the checking sequence seems to be saturating around 30%. Percentage contribution of Phase 2 to CS length shown in Figure 6.10.

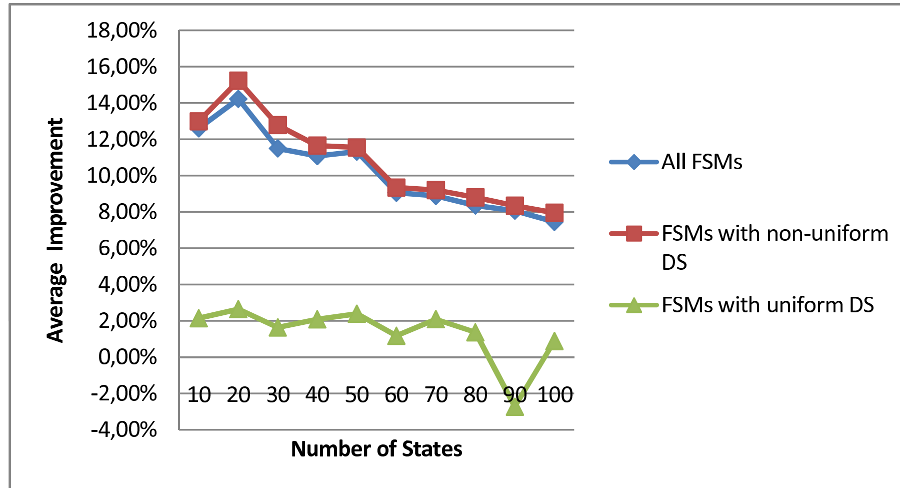

Figure 6.6: Average Improvements Over *Simão et al.*'s Method

6.3.3 Effect of Candidate Elimination Using a Set of Incompatible Nodes

When the candidate elimination method *candidate elimination using a set of incompatible nodes* is also used the average length of checking sequences generated by our method further reduces but the execution times increase as expected. Results will be given when the maximum cardinality, k, of the incompatible set is set to 5 and 10 ($k = 5$ and $k = 10$). Remember that $k = 1$ actually means using the method *candidate elimination using a recognized node* only and the results for this case were already given in the previous sections.

Note that the value of k has no effect on the length of the sequence generated at the end of the Phase 1 since during Phase 1 candidate elimination methods are not used. As the value of k increases a better analysis of the uncertainty automaton will be performed and this is expected to reduce the length of the extension introduced by Phase 2. Figure 6.12 shows the improvement on this extension length by giving the ratio of extension lengths when $k = 5$ and $k = 10$ to the extension length when $k = 1$. The experiments show that as the size of the FSM gets bigger although $k = 5$ and $k = 10$ cases perform a more complex analysis their extension length quickly approach to the extension length of case $k = 1$.

The value of k does not affect the time spent in Phase 1. As the value of k increases since a more complex analysis is performed the running time of Phase 2

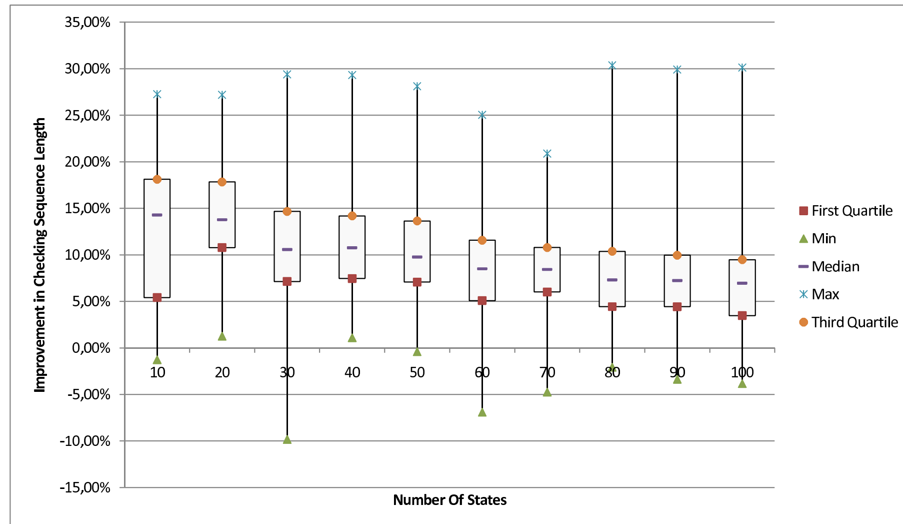

Figure 6.7: Improvements Over *Simão et al.*'s Method as a Box Plot

is expected to increase. Figure 6.13 shows the speed down factor when $k = 5$ and $k = 10$ compared to case $k = 1$. The experiments show that the analysis required by $k = 5$ and $k = 10$ cases slow down the execution at least more than 5 times.

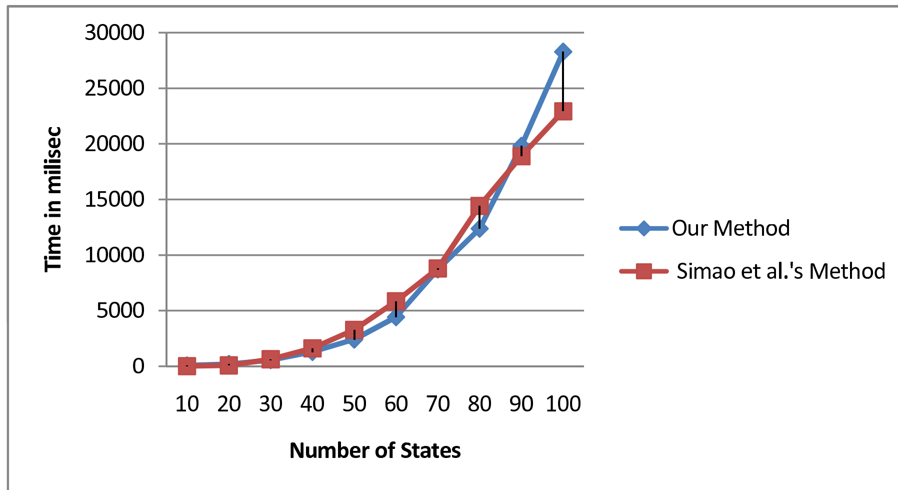

Figure 6.8: Average Method Execution Times

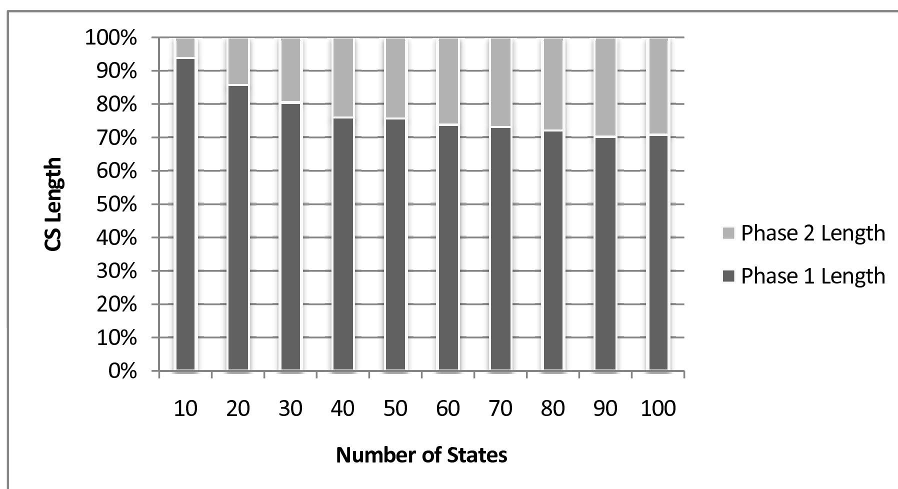

Figure 6.9: Contributions of Phase 1 and Phase 2 to CS Length

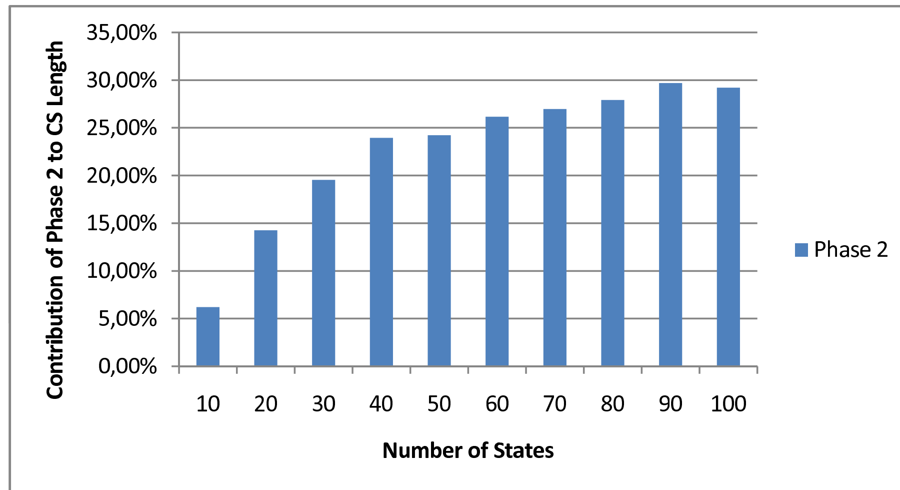

Figure 6.10: Percentage Contribution of Phase 2 CS Length

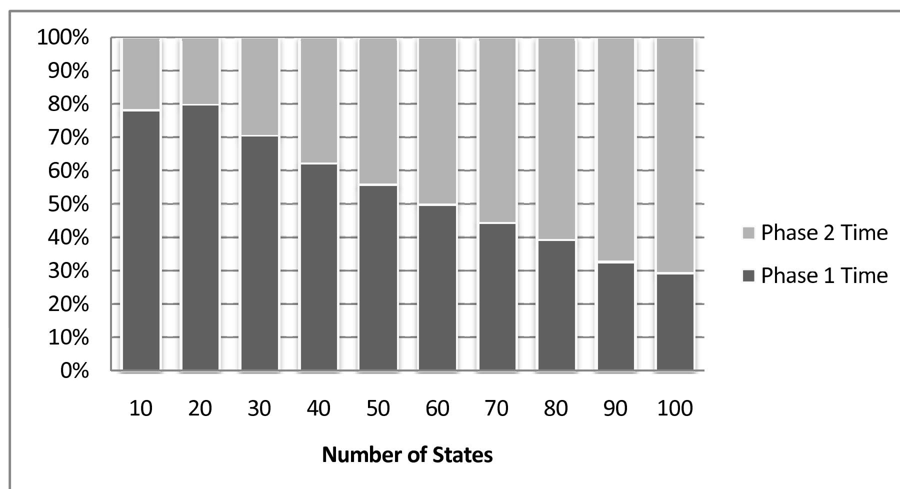

Figure 6.11: Distribution of Execution Time between Phase 1 and Phase 2

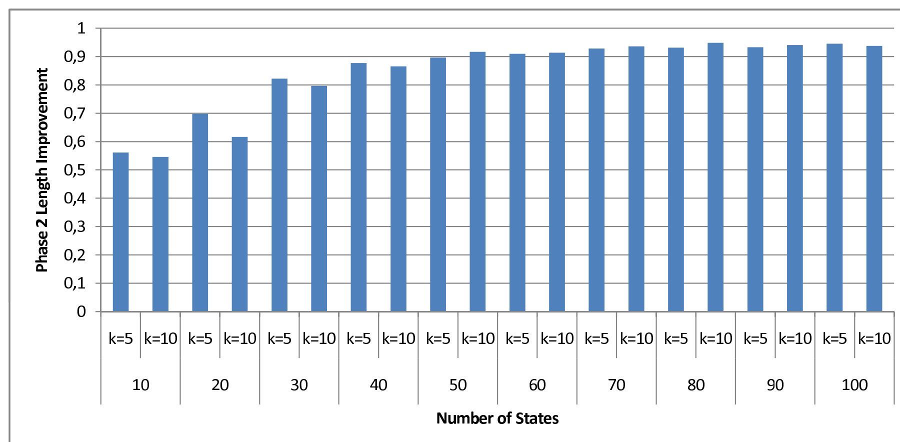

Figure 6.12: Effect of Candidate Elimination Using a Set of Incompatible Nodes on Length

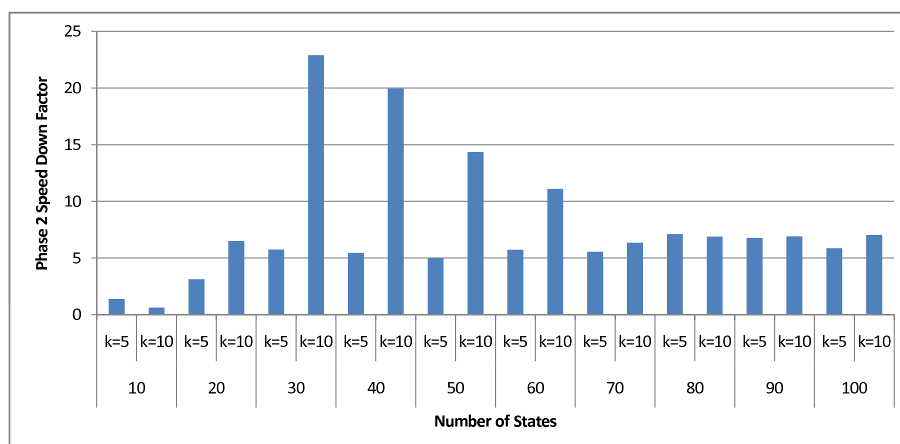

Figure 6.13: Effect of Candidate Elimination Using a Set of Incompatible Nodes on Time

Chapter 7

Conclusion

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

The absence of a benchmark set of FSMs makes it difficult to compare the performance of different checking sequence generation methods. Although it is not clear at all how well a randomly generated FSM will represent the features of a real FSM specification, it seems that there is no easy way of obtaining an extensive set of FSMs other than generating them randomly.

For this reason, we have implemented a random FSM generation tool in order to satisfy an important need in FSM based testing. For FSMs with different properties (such as being strongly connected or not, having a PDS/ADS or not, etc.) there are different and specialized methods for generating test and checking sequences. The tool does not only generate random FSMs of the certain type required by the checking sequence generation method developed within this thesis, but it can generate quite a wide range of types of FSMs with the required properties to support FSM based testing in general. Generation of a random FSM with some properties are difficult when it is left to chance. We developed some algorithms to obtain FSMs with these properties. Our main consideration while designing these algorithms was to affect the randomness of an FSM as little as possible. The thesis contains the details of the algorithms we developed to create random FSMs with such properties.

One especially hard to satisfy property turns out to be existence of preset distinguishing sequence (PDS). The existence check of PDS (which is known to be PSPACE–complete) is currently performed by an exponential analysis in the tool. Therefore generating large FSMs with PDS is quite difficult. Although we have

suggested the shuffling method to speed up regenerating candidate FSMs for this method, it seems that an approach that will generate a random FSM from a certain class of FSMs guaranteed to have PDS will be more effective.

Second contribution of the thesis is a method that can answer the following question: Given an input output sequence X/Y and a distinguishing sequence $\bar{D}$ for an FSM M, is X/Y a checking sequence for M which is generated by using $\bar{D}$ The method uses state recognition techniques already existing in the literature, such as d- and t-recognition. However we also introduce some novel state recognition methods. Although this increases the capability of the method to recognize a checking sequence, it is still possible that a sequence being found not to be a checking sequence even if it is really a checking sequence, i.e. we may have false negatives. However, it is not possible for us to have false positives. In other words, whenever a sequence is found to be a checking sequence, it is really a checking sequence for M.

Although this check is used as a termination condition in our checking sequence generation method, we believe the information provided by the uncertainty automaton can be used in the context of passive testing as well. In passive testing, the tester is nothing more than an observer of the IUT in its normal operation and cannot interfere with the operation of the system for testing purposes. Being a passive observer, the tester can only declare the implementation under test (IUT) to be faulty whenever an unexpected behavior is seen. However, if IUT keeps producing the expected responses, there is no verdict for the passive testing activity. We conjecture that the input output behavior X/Y of IUT as observed by the tester can be used to build an uncertainty automaton to see if the observation seen so far is a checking sequence or not. Even if it does not turn out to be a checking sequence, the size of the uncertainty automaton, probably together with the sizes of the candidate sets of the unrecognized nodes of it, may provide a metric to judge how close X/Y is to being a checking sequence.

The final and major contribution of the thesis is a new distinguishing sequence based checking sequence generation algorithm. Our method is based on a recent method that uses a local optimization. This local optimization based method in most cases yield better results than the existing global optimization based methods. 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 first phase is not a checking sequence then it is extended to a checking sequence in Phase 2. The experimental results have shown that our method achieves at least 7% reduction in the length of the checking sequence over the method that it is based on. We think that, there is still a room for further improvement using our method. The experiments show that approximately 30% 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. However a closer analysis of the final form of the uncertainty 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.

Another promising research direction seems to be the fusion of the new local optimization based methods and the old global optimization based methods. Although the new methods generate shorter sequences, they are based on very greedy ideas. Hence bringing in some kind of global knowledge into the algorithms may improve the performance even more.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A.V. Aho, A. T. Dahbura, D. Lee, and M.U. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, pages 1604–1615, 1991.

[3] Jessica Chen, Robert M. Hierons, Hasan Ural, and Hüsnü Yenigün. 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.

[4] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, pages 178–187, 1978.

[5] Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Automated unique input output sequence generation for conformance testing of fsms. *Comput. J.*, 49(3):331–344, 2006.

[6] Lihua Duan and Jessica Chen. Reducing test sequence length using invertible sequences. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 171–190. Springer, 2007.

[7] A.D. Friedman and P.R. Menon. *Fault Detection in Digital Circuits*. Englewood Cliffs, NJ: Prentice-Hall, 1971.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractabilty: A Guide to the Theory of NP-Completeness*. 1979.

[9] A. Gill. *Introduction to the Theory of Finite-State Machines.* New York: McGraw-Hill, 1962.

[10] F. C. Hennie. Fault detecting experiments for sequential circuits. *Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

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

[12] Robert M. Hierons. Extending test sequence overlap by invertibility. *Comput. J.*, 39(4):325–330, 1996.

[13] Robert M. Hierons. Testing from a finite-state machine: Extending invertibility to sequences. *Comput. J.*, 40(4):220–230, 1997.

[14] Robert M. Hierons and Hasan Ural. Reduced length checking sequences. *IEEE Trans. Comput.*, 51(9):1111–1117, 2002.

[15] Z. Kohavi. *Switching and Finite Automata Theory.* New York: McGraw-Hill, 1978.

[16] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.

[17] D. Lee and M. Yannakakis. Principles and methods of testing fsms - a survey. *Proceedings of the IEEE*, 84:1090–1123, 1996.

[18] R.E. Miller and S. Paul. On the generation of minimal-length conformance tests for communication protocols. *IEEE/ACM Trans. Netw.*, pages 116–129, 1993.

[19] Edward F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[20] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. *Proc. Ilth IEEE Fault Tolerant Comput. Symp.*, pages 238–243, 1981.

[21] I. Pomeranz and S.M. Reddy. Functional test generation for full scan circuits. *Proc. Conf. on Design, Automation and Test in Europe*, pages 396–403, 2000.

[22] K. K. Sabnani and A. T. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Syst.*, pages 285–297, 1988.

[23] D.P. Sidhu and T.K. Leung. Formal methods for protocol testing: a detailed study. *IEEE Trans. Softw. Eng.*, pages 413–426, 1989.

[24] Adenilso Simão and Alexandre Petrenko. Generating checking sequences for partial reduced finite state machines. In *TestCom '08 / FATES '08: Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems*, pages 153–168, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Hasan Ural, Xiaolin Wu, and Fan Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.

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