Semen Cirit

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

July, 2011

POST-PROCESSING FOR CHECKING SEQUENCES

SEMEN CİRİT

APPROVED BY:

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

Assist.Prof.Dr. Cemal Yılmaz ....

Assoc.Prof.Dr. Berrin Yanıkoğlu ....

Assoc.Prof.Dr. Albert Levi ....

Assist.Prof.Dr. Kerem Bülbül ....

DATE OF APPROVAL: 07.09.2011

Post-processing for Checking Sequences

Semen Cirit

EECS, Master's Thesis, 2011

Thesis Supervisor: Hüsnü Yenigün

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

Abstract

There are several methods to generate a checking sequence (CS) from a given Finite State Machine M. These methods generate a CS in such a way that when the CS is traced on M, every node visited during this trace is recognized as some state of M and every transition of M is traversed. When the recognitions of the nodes in this trace are analyzed, it is observed that some of the nodes are recognized multiple times redundantly. This observation raises the following question: Is it possible to reduce the length of a given CS by eliminating redundant recognitions? In this thesis we focus on this question. We formalize the recognitions, detect multiple redundant recognitions and suggest a way to eliminate them to reduce the length of a given CS. An experimental study of our approach is also presented.

Kontrol Dizilerinin Kısaltma Amaçlı Analizi

Semen Cirit

EECS, Yüksek Lisans Tezi, 2011

Tez Danışmanı: Hüsnü Yenigün

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

Özet

Verilen bir Sonlu Durumlu Makina (SDM) M için bir çok kontrol dizisi (KD) üretme metodu bulunmuştur. Bu metodlar KD üretimini, KD M üzerinden gezilirken yapılmaktadır. Bu gezme sırasında ziyaret edilen tüm düğümler, M'de bulunan bazı durumlar tarafından tanımlanır ve M'de bulunan her bir bağlantı üzerinden geçilmiş olur. Düğümlerin tanımlanması incelendiğinde, bazı düğümlerin birden fazla kez gereksiz yere tanımlanmış oldukları gözlemlenebilmektedir. Bu gözlem şu soruyu ortaya çıkarmaktadır: KD'nin uzunluğununu gereksiz tanımlamaları ortadan kaldırarak azaltmak mümkün müdür? Bu çalışmada bu soru üzerinde durulmaktadır. Verilen bir KD uzunluğunu kısaltabilmek için, tanımlamalar somutlaştırılmış, birden fazla kez tanımlanan gereksiz düğümler bulunmuş ve bu düğümleri ortadan kaldırmak için bir çözüm sunulmuştur. Ayrıca bu yaklaşımın deneysel bir çalışması da bu tez içerisinde sunulmaktadır.

Acknowledgments

I am especially thankful to my supervisor, Hüsnü Yenigün, whose encouragement, guidance and support from the initial to the final level.

I would like to thank my family for never leaving me alone and their motivation.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the thesis.

Contents

List of Figures

List of Tables

1 Introduction

Behaviour of communication protocols, control circuits, machine learning systems can be modeled as finite state machines (FSMs) [2, 4, 21, 24, 25, 26]. Unified Modeling Language (UML), Specification and Description Language (SDL), and state charts also incorporate stated based representation for behavioural specifications [8, 18].

Given an FSM M representing the behavioural specification of a system, and an implementation I claimed to implement M, I is needed to be tested to check if it correctly implements M or not. The correctness of I with respect to M can be proved by applying an input sequence to I, observing the actual output sequence produced by I in response to the application of the input sequence, and comparing the actual output sequence of I with the expected output sequence from M. The input sequence and the expected output sequence compose a *test sequence*. Not every test sequence would prove I to be correct. In fact, it is not possible to design a test sequence that will prove I to be correct, in general. However under certain assumptions on M and I, this is possible and a test sequence accomplishing such a proof is called a *checking sequence* [10, 9, 16, 17].

The line of work for constructing such test sequences starts in 60s [10]. There were some studies in 70's and 80's [9, 4, 7, 23, 22], but area was more active in 90's [2, 21, 20, 19, 13, 14, 16]. The area has been still active within the last decade [17, 1, 11, 15, 6, 3, 28, 12, 27, 5]. Each new method tries to improve on the previous methods by constructing shorter test sequences with the help of a better analysis or by applying a new approach.

A checking sequence basically tests if every state in the specification machine M also exists in the implementation I. Furthermore, it also considers every transition in M and makes sure that the starting and ending states of each transition, and the output produced by the transition can be the same as in the specification M. Existence of a state of M in I is performed by using a special test sequence called a *state identification sequence*. There are several flavors of state identification sequences such as *preset distinguishing sequence*, *adaptive distinguishing sequence*, or *unique input/out sequence*, etc [19]. Among these state identification sequences, distinguishing sequences allows one to construct a polynomial length checking sequence.

A (preset or adaptive) distinguishing sequence D has a unique response from each state of M. Therefore when a distinguishing sequence is applied to I, the state of I before the application of D can be correlated to the corresponding state of M based on the output produced by I to D. In this case we say, the state of I is *d-recognized* as the corresponding state of M. It is also possible to *t-recognize* a state of I as a state of M by using the following observation: If there are two states of I recognized as the same state of M and if the test sequence applies the same test sequence at both of these states, then the final state reached after this test sequence must also be recognized as the same state of M. In this thesis we also make use of another recognition technique which we call *e-recognition* (stands for elimination recognition). Intuitively a state n of I is recognized as a state s of M when there are evidences that for each state s' of M where $s' \neq s$, n cannot be s'. All of these recognition methods will be explained more formally in Section 2 and in Section 3.

When a checking sequence generated by some method is analyzed, it is observed that some of the states in I are recognized multiple times (of course all of these recognitions will be for the same state of M), whereas only one recognition is sufficient for the purpose.

After this observation, it is natural to think that it might be possible to reduce the length of a given checking sequence by removing the parts of a checking sequence causing multiple recognitions.

The rest of the thesis is structured as follows. Section 2 provides an overview of related material. Section 3 then describes an approach to detect multiple recognitions in a checking sequence and how to eliminate them. In Section 4, we report an experimental study of the proposed approach by trying to reduce checking sequence generated by several methods from randomly generated FSMs. Finally, some concluding remarks are given in Section 5.

2 Preliminaries

2.1 Finite State Machines

2.1.1 FSM elements

An FSM (*finite state machine*) M is defined as a tuple $M = (S, X, Y, \delta_M, \lambda_M, D_M, s_0)$ in which S is a finite set of *states*, s_0 is the *initial state*, X is a finite *input alphabet*, Y is a finite *output alphabet*, δ_M is a next state function: $\delta_M : D_M \to S, \lambda_M$ is an output function: $\lambda_M : D_M \to Y$ and D_M is the specification domain of these functions: $D_M \subseteq S \times X$ [1].

An FSM M is *deterministic* if for each state $s \in S$ and for each input $i \in X$, there is at most one transition defined in M.

An FSM is *completely specified* if the functions δ_M and λ_M are total. In other words if D_M is equal to $S \times X$, M is *completely specified*, this means for each state $s \in S$ and for each input $i \in X$ there is a transition defined in M.

A transition is defined by a tuple $(s_i - x/y \to s_j)$ in which s_i is the *starting state*, x is the *input*, $s_j = \delta_M(s_i, x)$ is the *ending state*, and $y = \lambda_M(s_i, x)$ is the *output*. The transiton is $s_i - x \to s_j$ when the output is not used.

Supposed that;

$M = (S, X, Y, \delta_M, \lambda_M, D_M, s_1)$ and

$I = (T, X, Y, \Delta_I, \Lambda_I, D_I, t_1)$ are two FSMs.

Further, we suppose that specifications will be represented with M and implementations will be represented with notation I which are *complete*.

Two states, s_j of M and t_i of I, said to be *compatible* if and only if for every input sequence $\alpha = x_1 x_2 ... x_k \in X^*$ the machines produce the same output sequence, i.e. $\delta_M(s_j, \alpha) = \Lambda_I(t_i, \alpha)$. Otherwise the states are *distinguishable*.

If I and M are *complete*, the compatible states have been equivalent states. A machine M is *minimal (reduced)* if and only if no FSM with fewer states than M is equivalent to M or if every pair of its states is *distinguishable*.

For an input sequence $\alpha.x$:

$\delta_M(s_i, \alpha.x) = \delta_M(\delta_M(s_i, \alpha), x)$ while

$\lambda_M(s_i, \alpha.x) = \lambda_M(s_i, \alpha).\lambda_M(\delta_M(s_i, \alpha), x)$.

An FSM is also *initially reachable* if for each $s_i \in S$ there exists some input sequence $\alpha \in X^*$ such that $\delta_M(s_0, \alpha) = s_i$ (i.e. each state $s_i \in S$ is reachable from the initial state s_0)

2.1.2 FSM as a Directed Graph

An FSM M can be represented by a digraph (directed graph) $G = (V, E)$, with a vertex set $V = \{v_1, v_2, \ldots, v_n\}$ which represents the set S of states of M where $n = |S|$ number of elements, and with the edges $e = (v_j, v_k; x/y) \in E$ that represents a transition from state s_j to state s_k with input $x \in X$ and output $y \in Y$ [16].

v_j and v_k are the *start* and *end* of e and input/output (I/O pair) x/y is the *label* of e, denoted *label*(e). Two edges e_j and e_k are called *adjacent* if *end* of e_j and *start* of e_k are same.

A *path* $P = (n_1, n_2; x_1/y_1)(n_2, n_3; x_2/y_2) \ldots (n_{r-1}, n_r; x_{r-1}/y_{r-1}), r > 1$, of G is a finite sequence of (*not necessarily distinct*) *adjacent edges* in E, where each node n_i represents a vertex from V; n_1 and n_r are the *start* and *end* of P and $(x_1/y_1)(x_2/y_2) \ldots (x_{r-1}/y_{r-1})$ is the *label* of P, denoted *label*(P).

P can also represented by $(n_1, n_r; I/O)$, where *label*$(P) = I/O$ is the IO-sequence (x_1/y_1) $(x_2/y_2) \ldots (x_{r-1}/y_{r-1})$, input sequence $I = (x_1x_2 \ldots x_{r_1})$ is the input portion of I/O, and output sequence $O = (y_1y_2 \ldots y_{r_1})$ is the output portion of I/O.

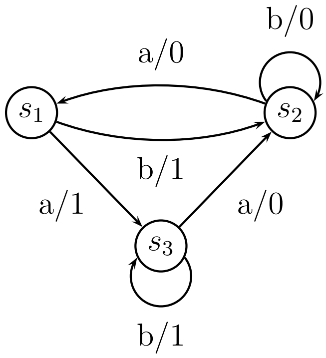

Figure 1: FSM M_1

G is *strongly connected*, if for all $v_i; v_j \in V$, there is a path from v_i to v_j.

The *cost* (or *length*) of an edge is the number of I/O pairs in the label of the edge.

The *cost* (or *length*) of path P is the sum of the costs of edges in P. The *concatenation* of two sequences (or paths) P and Q is denoted by PQ.

Consider the FSM M_1 given in Figure 1, according to definitions:

- For each state, there is at most one transition defined in FSM M_1, therefore FSM M_1 is *deterministic.*

- For each state, there is a transition defined in FSM M_1, therefore FSM M_1 is *completely specified.*

- Every pair of states in FSM M_1 are *distinguishable* so FSM M_1 is *minimal.*

- For every vertex in FSM M_1 there is a path between them, so FSM M_1 is *strongly connected.*

2.2 Transfer Sequence

The label of a path from s_i to s_j is the *transfer sequence* T of FSM M [11]. In other words, for each of two states $s_i, s_j \in S$, there exists an input sequence α, called a *transfer sequence* from state s_i to state s_j, such that $s_j = \delta(s_i, \alpha)$.

For example, there is a *transfer sequence* from state s_3 to s_1 with path $aa/00$ on FSM M_1 given in Figure 1.

If there exist a *transfer sequence* from all state s_i to state s_j, FSM M is said to be *strongly connected*. The FSM M is initially connected if there is a transfer sequence from the initial state s_0 to each state.

2.3 Distinguishing Sequence

There are two types distinguishing sequences; preset distinguishing sequence and adaptive distinguishing sequence. We actually use the adaptive distinguishing sequence but in order to give the concept and differencies between them. We need to metion about two of them.

2.3.1 Preset Distinguishing Sequence

An input sequence $x \in X$ is a preset distinguishing sequence (PDS) for an FSM M if the output sequence produced by M in response to x is different for each state[15].

For instance, for an input sequence D and for every pair of $s_i, s_j \in S$, $i \neq j$, $\lambda_M(s_i, D) \neq \lambda_M(s_j, D)$.

2.3.2 Adaptive Distinguishing Sequence

A PDS can be used as an input sequence and distinguish each state of the FSM. As for *adaptive distinguishing sequence* (ADS) is not really a sequence but a decision tree which exactly n leaves. The internal nodes of the tree are labeled with input symbols, its edges are labeled with output symbols, and its leaves are uniquely labeled with states [15]. That means while the tree is walked from the root through a leaf, we can find one of the state unique input and output sequences which is distinguished from the other leaves (states), such that for a common prefix α, $\lambda_M(s_i, \alpha) \neq \lambda_M(s_j, \alpha)$.

For every leaf of the tree, if D_i and y_i are the input and output strings respectively formed by the node and edge labels on the path from the root to the leaf labeled by state s_i of the FSM then $y_i = \lambda_M(s_i, D_i)$.

We call D_i as the ADS of state s_i.

For example, $D_1 = a/1$, $D_2 = aa/01$, $D_3 = aa/00$ are the distinguishing sequence of each state of M_1 on Figure 1.

ADS has more advantages than PDS on state identification. At first PDS is a kind of ADS, but the reverse is not true. Therefore, there can be FSMs with ADS but not PDS. If we can use adaptive distinguishing sequences instead of preset ones in a checking sequence generation method, then the method can be applied on a strictly larger set of FSMs. Because for a given FSM it is known that the shortest ADS is no longer than the shortest PDS [19].

2.4 Checking Sequence

The checking sequence concept is born in order to determine fault detection of an FSM. Assume that an $FSM_i \in \Phi(FSM_s)$ which is deterministic, strongly connected, completely specified and minimal and also assume that an $FSM_i \in \Phi(FSM_s)$ claimed to be an implementation of the FSM_s which is not change during execution and sets of inputs and outputs are identical to those FSM_s and also it has n distinct states.

The input sequence of FSM_s is called a *checking sequence*, when output sequence from FSM_i in response to the checking sequence is used to verify whether FSM_i is a correct implementation of FSM_s.

A *checking sequence* of FSM_s is an input sequence such that it distinguishes FSM_s from every FSM $FSM_i \in \Phi(FSM_s)$ that is not equal to FSM_s.

Checking sequence correctness is obtained by three steps:

1. FSM_i should be initialized.

2. FSM_i is checked whether it has at least n distinct states.

3. FSM_i is checked whether it implements all transitions of FSM_s.

For the first step of checking sequence correctness, there should be shown that if FSM_s has a $(s_j, s_k; x/y)$ transition, then FSM_i should have a corresponding transition $(f(s_j), f(s_k); x/y)$.

The state correctness are formally proved by the concepts *d-recognition* and *t-recognition*.

Consider a path P of G representing FSM_s and the nodes within it:

d-recognition:

A node n_i of P is *d-recognized* as state s of FSM_s if n_i is the start of a subpath of P with label $D/\lambda(s, D)$.

In Figure 2, a distinguishing sequence on a subpath with label $D/\lambda(a, D)$ is recognized for node n_i so node n_i is *d-recognized*.

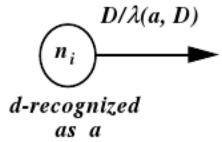

Figure 2: d-recognition

t-recognition:

A node n_i of P is *t-recognized* as state s' of FSM_s 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 FSM_s , n_k is recognized as state s' of FSM_s. The similar transfer sequences and recognitions are shown in Figure 3.

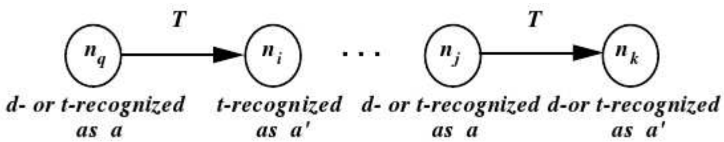

Figure 3: t-recognition

The last step of the checking sequence correctness is defined as follows. A transition $t = (s, s'; x/y)$ of FSM_s 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 FSM_s respectively and $x'/y' = x/y$.

The theorem below from [16] explains effectively checking sequence.

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

For the checking sequence in Figure 4, consider I/O sequence with a path $X/Y = aaaabaabaaabaa/10011001010001$ and FSM M_1 in Figure 1. In the

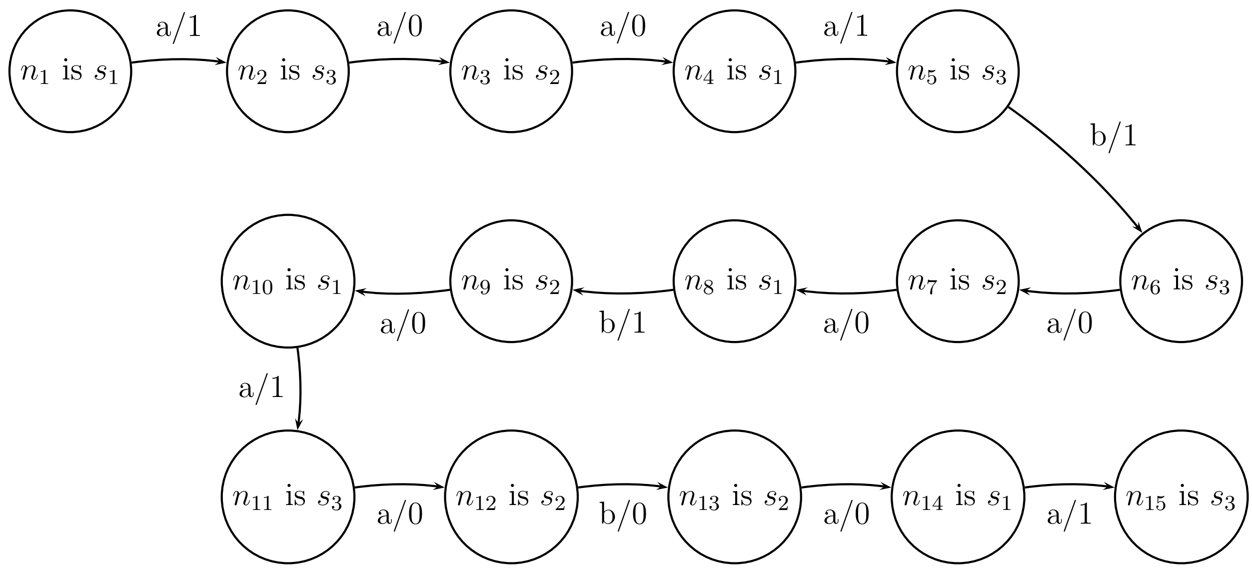

Figure 4: Checking Sequence of FSM M_1

given path, every transition is verified with *d-recognition* and *t-recognition*. Then, we can say that $X = aaaabaabaaabaa$ forms a checking sequence of FSM M_1 in Figure 1.

2.5 Random FSM Generation

In order to make checking sequence analysis and comparisons, a group of FSMs that have different number of states is needed. All checking sequence generation and optimization methods need special FSMs. Thanks to random FSM generation tool, the random FSM can be generated with special properties. This tool generate FSMs with 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)

In our thesis, the properties strongly connected, minimal and having adaptive distinguishing sequence are used.

3 Our Method

In this section a new checking sequence optimization method will be presented. In order to reduce the length of the checking sequence, we generate a new state recognition method is called *elimination recognition method*. We then describe the checking sequence as a *boolean formula* in order to enable the elimination and other state recognition methods to work on it and let them to find unnecessary transitions. We use an AND-OR graph in order to implement this checking sequence and we do an exhaustive search in order to find transitions that can be removed.

3.1 Elimination Recognition Method

A node n_i of P is recognized with this method as state s of FSM_s if there exists different nodes for each different state then s $(s' \neq s)$ and these nodes have also the same input as n_i. The output function of a node may or may not be the same as the output function of n_i. If the output function is the same, the next state functions should be different then n_i's output function. If the output functions are different, we do not need the next state function comparisons.

If we experience these type of nodes for each state different than s that mentioned above the *elimination recognition* method can be performed for n_i.

We can denote *elimination recognition* as *e-recognition*.

We can explain the recognition more formally:

Supposed that there exists a checking secuence with path P and FSM $FSMs$ and nodes n_i and n_{i+1} with a sequence $(n_i, n_{i+1}; x_i/y_i)$ can be recognized as states s and s'. Node n_i can be *e-recognized* as state s, if there exist sequences for all different states then s like:

1. $(n_k, n_{k+1}; x_k/y_k) \in P$, if node n_k is not recognized as state s, n_{k+1} is not recognized as s', $x_k = x_i$, $y_k = y_i$ or

2. $(n_l, n_{l+1}; x_k/y_k) \in P$, if node n_l is not recognized as state s, $x_k = x_i$, $y_k \neq y_i$ for all states of $FSMs$ different then state s.

For example, node n_3 in checking sequence in Figure 4 want to be recognized as state s_2 with a sequence $(n_3, n_4; a/0)$. We assume that n_4 is recognized as state s_1.

There exist two different states than s_2, therefore we trace the checking sequence if there exist transitions;

- that the incoming node is not recognized as state s_2 and outgoing node is not recognized as state s_1, while the input and output is $a/0$. These transitions are found from Appendix A and Figure 4:

  - $(n_6, n_7; a/0)$, We can see that from Appendix A.6 n_6 is recognized as state s_3 and from Appendix A.7 n_7 is recognized as state s_2

  - $(n_{11}, n_{12}; a/0)$, We can see that from Appendix A.11 n_{11} is recognized as state s_3 and from Appendix A.12 n_{12} is recognized as state s_2

- that the incoming node is not recognized as state s_2, while the input is a and output is different than 0: These transitions are found from Appendix A and Figure 4:

  - $(n_1, n_2; a/1)$, We can see that from Appendix A.1 n_1 is recognized as state s_1

– $(n_4, n_5; a/1)$, We can see that from Appendix A.4 n_4 is recognized as state s_1

– $(n_{10}, n_{11}; a/1)$, We can see that from Appendix A.10 n_{11} is recognized as state s_1

– $(n_{14}, n_{15}; a/1)$, We can see that from Appendix A.14 n_{14} is recognized as state s_1

3.2 Checking Sequence as Boolean Formula

In this thesis the checking sequence is formulated as boolean formula. The boolean formula is generated via *d*-, *t*-, *e*- recognition methods that mentioned on Section 2 and Section 3. All nodes on checking sequence path can be recognized by these methods. All recognition possibilities of node of a checking sequence are represented as boolean formula.

The boolean formula building structure has two different kind of values, these values represent *possible state and input functions* of a node on checking sequence path.

More formally, we can explain these two kind of values with:

1. node n_i can be recognized as state s can be shown as σ_i (*Possible state function* of node n_i)
2. node n_i transition can have x input function can be shown as ρ_i (*Possible input function* of node n_i)

The boolean formula generally is constructed with below instructions:

- All state recognition methods are formulated with σ and ρ.
- In order to represent *d*-, *t*-, *e*- recognitions of one node, the and ($\wedge$) operation is used between all σ and ρ.

- The and ($\wedge$) operation is also used between all *d*-, *t*-, *e*- recognition groups of two different checking sequence nodes.
- The or ($\vee$) operation exists between all *d*-, *t*-, *e*- recognitions of one checking sequence node.

3.2.1 Boolean Formula of d-recognition

The founded *d-recognitions* on the path are represented by only *possible input functions*, ρ.

Supposed that node n_i has a *d-recognition* with a distinguishing set $D = \{D_{s_i}, D_{s_{i+1}}\}$ where $D_i = x_i/y_i$, $D_{s_{i+1}} = x_{i+1}/y_{i+1}$, so this recognition equation is represented as:

$$\rho_i \wedge \rho_{i+1} \tag{1}$$

This means the node has a distinguishing sequence with two input functions and the occurrence of those input functions found for ρ_i, ρ_{i+1}.

For example, we know that node n_2 in checking sequence in Figure 4 has a *d-recognition* D_3 from Section 2.3.2. We represent this recognition in Appendix A.2.1 as:

$$\left| \; n_2 \rightarrow a \wedge n_3 \rightarrow a \; \right|$$

So

$$\left| \; \rho_2 \wedge \rho_3 \; \right|$$

3.2.2 Boolean Formula of t-recognition

The founded *t-recognitions* on the path are represented by *possible input functions* ρ and *possible state functions* σ.

There are two groups in the representation of *t-recognition* as a boolean formula; one part is for the *incoming transition* and the other part is for *similar transitions* of the node that has been *t-recognized.* The *incoming transition* part is defined by the previous node and its input function and it is represented by *possible state and input function* of one previous node of the node that has been *t-recognized. Similar transitions* are the transitions that includes the possible state of the node that has been *t-recognized* and possible state of its previous node. The *similar transition* part are represented by *possible state functions* of incoming and outgoing node and *possible input function* of incoming node of current *similar transition.*

Supposed that nodes n_{i+1} with a sequence $(n_i, n_{i+1}; x_i/y_i)$ can be *t-recognized* as state s'. And we know that n_i has already been recognized as s.

So the *incoming transition* part is represented as:

$$\sigma_i \wedge \rho_i \tag{2}$$

For example, consider node n_2 of the checking sequence in Figure 4. The incoming transition of node n_2 is $(n_1, n_2; a/1)$. It is represented in Appendix A.2.2 as:

$$\left| \; n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \; \right|$$

So

$$\left| \; \sigma_1 \wedge \rho_1 \; \right|$$

Supposed that there exists a checking secuence with path P and it has like; $(n_a, n_{a+1}; x_a/y_a)$, $(n_b, n_{b+1}; x_b/y_b)$ and $(n_c, n_{c+1}; x_c/y_c)$ are the *similar transitions*, that means n_a, n_b, n_c can be recognized as state s and n_{a+1}, n_{b+1}, n_{c+1} can be recognized as state s' and $\{x_a, x_b, x_c\} = x_i$, $\{y_a, y_b, y_c\} = y_i$.

So the *similar transition* part is represented as:

$$\begin{aligned}&\sigma_a \wedge \rho_a \wedge \sigma_{a+1}\\&\sigma_b \wedge \rho_b \wedge \sigma_{b+1}\\&\sigma_c \wedge \rho_c \wedge \sigma_{c+1}\end{aligned} \tag{3}$$

For example, node n_2 of the checking sequence in Figure 4 has a incoming transition $(n_1, n_2; a/1)$. The similar transtions is represented in Appendix A.2.2 as:

$$\left|\begin{array}{c} n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \wedge n_5 \text{ is } s_3 \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \wedge n_{11} \text{ is } s_3 \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \wedge n_{15} \text{ is } s_3 \end{array}\right|$$

So

$$\left|\begin{array}{c} \sigma_4 \wedge \rho_4 \wedge \sigma_5 \\ \vee \\ \sigma_{10} \wedge \rho_{10} \wedge \sigma_{11} \\ \vee \\ \sigma_{14} \wedge \rho_{14} \wedge \sigma_{15} \end{array}\right|$$

The incoming and similar transition parts are combined as:

$$\left|\begin{array}{c} \sigma_i \wedge \rho_i \end{array}\right| \wedge \left|\begin{array}{c} \sigma_a \wedge \rho_a \wedge \sigma_{a+1} \\ \vee \\ \sigma_b \wedge \rho_b \wedge \sigma_{b+1} \\ \vee \\ \sigma_c \wedge \rho_c \wedge \sigma_{c+1} \end{array}\right| \tag{4}$$

Node n_2 of the checking sequence in Figure 4 has a *t-recognition* in Appendix A.2.2 represented as:

$$\left| \; n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \; \right| \wedge \left| \begin{array}{c} n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \wedge n_5 \text{ is } s_3 \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \wedge n_{11} \text{ is } s_3 \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \wedge n_{15} \text{ is } s_3 \end{array} \right|$$

So

$$\left| \; \sigma_1 \wedge \rho_1 \; \right| \wedge \left| \begin{array}{c} \sigma_4 \wedge \rho_4 \wedge \sigma_5 \\ \vee \\ \sigma_{10} \wedge \rho_{10} \wedge \sigma_{11} \\ \vee \\ \sigma_{14} \wedge \rho_{14} \wedge \sigma_{15} \end{array} \right|$$

3.2.3 Boolean Formula of e-recognition

The founded *elimination recognitions* on the path are also represented by *possible input functions* ρ and *possible state functions* σ.

There are two groups in the representation of *e-recognition* as a boolean formula; one part is for the *outgoing node* of the node that has been *e-recognized*, and the other part, *different transitions* part, is for the nodes that can be recognized as a state different then the state of the node that has been *e-recognized.* The *outgoing node* is represented by *possible state function* of next node of the node that has been *e-recognized. Different transitions* are the transitions that has a incoming node that can be recognized as a state different then the *possible state function* of the node that has been *e-recognized* and these transitions have also the same input function as the node that has been *e-recognized*. The output function of a node may or not the same as the node that has been *e-recognized*. If the output function is same, the next state function should be different then the node that has been *e-recognized* has. So this type of *different transition part* can be represented by *possible state functions* of incoming and outgoing node and *possible input function* of incoming node of current *different transition*. If the output functions are different we do not need the next state function comparisons. So this type of *different transition part* can be represented by *possible state and input function* of incoming node of current *different transition*.

Supposed that nodes n_i with a sequence $(n_i, n_{i+1}; x_i/y_i)$ can be *e-recognized* as state s. And we know that n_{i+1} has already been recognized as s'.

So the *outgoing node* part is represented as:

$$\sigma_{i+1} \tag{5}$$

The *outgoing node* of the checking sequence node n_2 in Figure 4 is n_3, therefore the outgoing node part is represented in Appendix A.2.3 as:

$$\Big|\ n_3 \text{ is } s_2\ \Big|$$

So

$$\Big|\ \sigma_3\ \Big|$$

Supposed that there exists a checking secuence with path P and it has nodes n_i and n_{i+1} with a sequence $(n_i, n_{i+1}; x_i/y_i)$ can be recognized as states s and s'.

Supposed also that there exist a sequence $(n_k, n_{k+1}; x_k/y_k) \in P$ and node n_k are not recognized as state s, n_{k+1} are not recognized as s', $x_k = x_i$,

$y_k = y_i$:

So *different transition* part with same output function is represented as:

$$\sigma_k \wedge \rho_k \wedge \sigma_{k+1} \tag{6}$$

Node n_2 of the checking sequence in Figure 4 has a outgoing transition $(n_2, n_3; a/0)$ and node n_2 wants to be recognized as state s_3 and node n_3 is recognized as state s_2. Therefore the *different transition* part with same output is represented in Appendix A.2.3 as:

$$\left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \rightarrow a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \rightarrow a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \rightarrow a \wedge n_{10} \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \rightarrow a \wedge n_{14} \text{ is } s_1 \end{array}\right|$$

So

$$\left|\begin{array}{c} \sigma_3 \wedge \rho_3 \wedge \sigma_4 \\ \vee \\ \sigma_7 \wedge \rho_7 \wedge \sigma_8 \\ \vee \\ \sigma_9 \wedge \rho_9 \wedge \sigma_{10} \\ \vee \\ \sigma_{13} \wedge \rho_{13} \wedge \sigma_{14} \end{array}\right|$$

Supposed that there exist a sequence $(n_l, n_{l+1}; x_k/y_k) \in P$ and nodes n_l are not recognized as state s, $x_k = x_i$, $y_k \neq y_i$ for all states of $FSMs$ different then state s.

So *different transition* part with different output function is represented as:

$$\sigma_k \wedge \rho_k \tag{7}$$

Node n_2 of the checking sequence in Figure 4 has a outgoing transition $(n_2, n_3; a/0)$ and node n_2 wants to be recognized as state s_3 and node n_3 is recognized as state s_2. Therefore the *different transition* part with different output is represented in Appendix A.2.3 as:

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \end{array}\right|$$

So

$$\left|\begin{array}{c} \sigma_1 \wedge \rho_1 \\ \vee \\ \sigma_4 \wedge \rho_4 \\ \vee \\ \sigma_{10} \wedge \rho_{10} \\ \vee \\ \sigma_{14} \wedge \rho_{14} \end{array}\right|$$

Now supposed that FSM $FSMs$ has three different states s, s', s''. And node n_i is wanted to be *e-recognized* as s. We also supposed that there exist a sequence $(n_i, n_{i+1}; x_i/y_i)$ and n_{i+1} is recognized as s'.

If there exist a *different transition* part for all nodes other then s, it can be said that n_i can be *e-recognized* as s.

Supposed that there exist such below transitions for s', s'' respectively:

1. $(n_k, n_{k+1}; x_k/y_k) \in P$, if node n_k is recognized as state s', n_{k+1} is not recognized as s', $x_k = x_i$, $y_k = y_i$
2. $(n_l, n_{l+1}; x_k/y_k) \in P$, if node n_l is recognized as state s'', $x_k = x_i$, $y_k \neq y_i$
3. $(n_m, n_{m+1}; x_m/y_m) \in P$, if node n_m is recognized as state s'', $x_m = x_i$, $y_m \neq y_i$

So the *e-recognition* boolean equation will be:

$$\left| \sigma_k \wedge \rho_k \wedge \sigma_{k+1} \right| \wedge \left| \begin{array}{c} \sigma_l \wedge \rho_l \\ \vee \\ \sigma_m \wedge \rho_m \end{array} \right| \wedge \left| \sigma_{i+1} \right| \tag{8}$$

Node n_2 of the checking sequence in Figure 4 has a *e-recognition* that is represented in Appendix A.2.3 can be shown as:

$$\left|\begin{array}{c} \sigma_1 \wedge \rho_1 \\ \vee \\ \sigma_4 \wedge \rho_4 \\ \vee \\ \sigma_{10} \wedge \rho_{10} \\ \vee \\ \sigma_{14} \wedge \rho_{14} \end{array}\right| \wedge \left|\begin{array}{c} \sigma_3 \wedge \rho_3 \wedge \sigma_4 \\ \vee \\ \sigma_7 \wedge \rho_7 \wedge \sigma_8 \\ \vee \\ \sigma_9 \wedge \rho_9 \wedge \sigma_{10} \\ \vee \\ \sigma_{13} \wedge \rho_{13} \wedge \sigma_{14} \end{array}\right| \wedge \left| \sigma_3 \right|$$

3.2.4 Boolean Formula of a Checking Sequence Node

Between all possible state recognition methods of a node the or ($\vee$) operation exists.

So the node n_i boolean equation will be:

$$\left| \rho_i \wedge \rho_{i+1} \right| \vee \left| \left| \sigma_i \wedge \rho_i \right| \wedge \left|\begin{array}{c} \sigma_a \wedge \rho_a \wedge \sigma_{a+1} \\ \vee \\ \sigma_b \wedge \rho_b \wedge \sigma_{b+1} \\ \vee \\ \sigma_c \wedge \rho_c \wedge \sigma_{c+1} \end{array}\right| \right| \vee \left| \left| \sigma_k \wedge \rho_k \wedge \sigma_{k+1} \right| \wedge \left|\begin{array}{c} \sigma_m \wedge \rho_m \\ \vee \\ \sigma_l \wedge \rho_l \end{array}\right| \wedge \left| \sigma_{i+1} \right| \right| \tag{9}$$

Possible recognition methods boolean equation of checking sequence node n_2 in Figure 4 is represented in Appendix A.2.3:

$$\left| \rho_2 \wedge \rho_3 \right| \vee \left| \left| \sigma_1 \wedge \rho_1 \right| \wedge \left| \begin{array}{c} \sigma_4 \wedge \rho_4 \wedge \sigma_5 \\ \vee \\ \sigma_{10} \wedge \rho_{10} \wedge \sigma_{11} \\ \vee \\ \sigma_{14} \wedge \rho_{14} \wedge \sigma_{15} \end{array} \right| \right| \vee \left| \left| \begin{array}{c} \sigma_1 \wedge \rho_1 \\ \vee \\ \sigma_4 \wedge \rho_4 \\ \vee \\ \sigma_{10} \wedge \rho_{10} \\ \vee \\ \sigma_{14} \wedge \rho_{14} \end{array} \right| \wedge \left| \begin{array}{c} \sigma_3 \wedge \rho_3 \wedge \sigma_4 \\ \vee \\ \sigma_7 \wedge \rho_7 \wedge \sigma_8 \\ \vee \\ \sigma_9 \wedge \rho_9 \wedge \sigma_{10} \\ \vee \\ \sigma_{13} \wedge \rho_{13} \wedge \sigma_{14} \end{array} \right| \wedge \left| \sigma_3 \right| \right|$$

3.2.5 Boolean Formula of Checking Sequence

Between all nodes boolean equation, the and ($\wedge$) operation exists.

Supposed that all nodes on checking sequence path P has a boolean equation c. If $P = (n_1, n_2; x_1/y_1)(n_2, n_3; x_2/y_2) \ldots (n_{r-1}, n_r; x_{r-1}/y_{r-1}), r > 1$ and the boolean equations of the nodes are c_1, c_2, c_2, c_4, $\ldots$ c_{r-1}, c_r.

The checking sequence boolean equation will be:

$$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge \ldots c_{r-1} \wedge c_r \tag{10}$$

The boolean formula of checking sequence in Figure 4 represented in Appendix A is:

$$\begin{aligned} &c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \wedge c_7 \wedge c_8 \wedge c_9 \wedge c_{10} \wedge c_{11} \\ &\wedge c_{12} \wedge c_{13} \wedge c_{14} \wedge c_{15} \end{aligned} \tag{11}$$

3.3 AND OR Graph Construction

We represent the checking sequence boolean formula as an AND-OR graph. This graph has three group of nodes: *possible input functions* ρ, *possible state* *functions* σ, and boolean equation of nodes c. Each *possible state function* has a boolean equation. As we mentioned formerly, the boolean equation is a combination of different *possible input and state functions*.

3.3.1 Possible Input Function ρ Graph Construction

The *possible input function* node of the graph includes the current node and input of the checking sequence. Besides them, two kinds of boolean value also exist; one of them controls if the *possible input function* can be removed from the checking sequence, the other controls if the *possible input function* has already removed from the checking sequence.

3.3.2 Possible State Function σ Graph Construction

The *possible state function* node of the graph includes the current node and possible state of the checking sequence. Besides them, two kinds of boolean value also exist; one of them controls if the *possible state function* can be removed from the checking sequence, the other controls if the *possible state function* has already removed from the checking sequence. The boolean equation information, that proves the existence of *possible state function* of current checking sequence node, is also included.

3.3.3 Boolean Equation c Graph Construction

The boolean equation node, that proves the existence of the current node of the checking sequence, includes boolean equation of *d-recognition*, *t-recognition* and *e-recognition* separetly. Also it has a boolean value in order to return the boolean equation result. The boolean equation result is found by making or operations between *d-recognition*, *t-recognition* and *e-recognition* boolean values.

d-recognition Boolean Equation Graph Construction

As we mentioned on Section 3.2.1, the *d-recognition* is represented by only *possible input* functions. Therefore *d-recognition* has information of related *possible input function* nodes. Also a boolean value exists in order to control whether *d-recognition* is proved.

t-recognition Boolean Equation Graph Construction

As we mentioned on Section 3.2.2, the *t-recognition* node has *incoming transition* and *similar transitions* that consist of *possible state function* and *possible input function* nodes. Also a boolean value exists in order to control whether *t-recognition* is proved.

e-recognition Boolean Equation Graph Construction

As we mentioned on Section 3.2.3, the *e-recognition* node has *outgoing node* and *different transitions* of all states different than selected state, that consist of *possible state and input function* nodes. Also a boolean value exists in order to control whether *e-recognition* is proved.

3.4 Checking Sequence Transition Optimization

Our main problem is to find redundant nodes on checking sequence, that's why finding the biggest transition set, that can be removed, is crucial. As also we know that, trying all node combinations in order to find biggest node set is very expensive. Therefore a simple heuristic is used in order to find biggest consecutive node set and we try if this set is removable.

3.4.1 Finding a Possible Removable Input Function ρ

The algorithm is explained informally below and more precisely in Algorithm 2;

The algorithm start from the *boolean value* assignment of *possible input functions*, ρ. "True" value is gived for all *boolean value* of *possible input functions* by default. The other nodes boolean values are not known at that moment.

The *possible state function* σ and relevant *possible input function* ρ that wants to be removed get the "False" boolean values.

If a *d-recognition possible input functions* are all "True", the *boolean equation* c which has the relevant *d-recognition* get the "True" value,

When a *boolean equation* c gets the "True" boolean value, the *possible state function* σ which is proved by this *boolean equation* c have "True" boolean value. The newly "True" assigned σ boolean values effect also the other *boolean equations* that have the σ.

After each iteration at least one newly assigned *possible state function* σ should be found until all *possible state functions* get the "True" value.

If all of the σ boolean values get "True" excluding the node that is wanted to remove, all other nodes of the checking sequence can be proved. If all possible state functions σ are traced and there exist at least one that unassigned, there exist some nodes that can not be proved without the node that is wanted to remove.

The Algorithm 1 initializes the boolean values of of ρ_i and σ_i that will be removed.

The Algorithm 2 is the function that updates all the boolean values of c and σ in a loop and returns possible removable input function order on the checking sequence.

Algorithm 1: START-UPDATE

Input: $BI_i is \rho_i$ boolean value

Input: $BS_i is \sigma_i$ boolean value

$BI_i = False$;

$BS_i = False$;

Algorithm 2: UPDATE-ALL

Input: ρ_i possible input function

Output: i possible input function order in checking sequence

$BI = \rho$'s boolean value;

$BS = \sigma$'s boolean value;

$BC = c$'s boolean value;

$BCD = c$'s *d-recognition* boolean value;

set all BI to "True";

if *c has a* d-recognition *and all BI of* d-recognition *is "True"* **then**

    $BCD = True$;

    $BC = True$;

while *any BS has not get a "True" boolean value or all of BS, excluding the removing one, get "True"* **do**

    set the related BC values to "True";

    set the related BS values to "True";

if *all BS, excluding the removing one, get "True"* **then**

    return i;

Condider the tenth node of the checking sequence in Figure 4, n_{10}, if this node is removed, all and ($\wedge$) equations of a condition node that contains n_{10} are eliminated. Because BI_{10} and BS_{10} get "False" value, all and ($\wedge$) equations that contain these values also get "False" value. If we review equations on Appendix A we can see n_2, n_3, n_5, n_6, n_7 n_9, n_{11}, n_{13}, n_{15} have n_{10} in their boolean equations.

For example, consider n_9, it has an equation like as;

$$\left| \, n_9 \rightarrow a \wedge n_{10} \rightarrow a \, \right| \vee$$

$$\left| \, \left| \begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \end{array} \right| \wedge \left| \begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \rightarrow a \wedge n_3 \text{ is } s_2 \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \rightarrow a \wedge n_7 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \rightarrow a \wedge n_{12} \text{ is } s_2 \end{array} \right| \wedge \left| \, n_{10} \text{ is } s_1 \, \right| \, \right|$$

Each possibility of the equation has node n_{10}, therefore if n_{10} is removed n_9 can not be proved.

If we consider n_{15}, it has an equation like as:

$$\left| \, n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \, \right| \wedge \left| \begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \wedge n_2 \text{ is } s_3 \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \wedge n_5 \text{ is } s_3 \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \wedge n_{11} \text{ is } s_3 \end{array} \right|$$

When node n_{10} gets the false value, if one of the other possibilities returns true, node n_{15} can be proved.

3.4.2 Find Descending Consecutive List Set

The simple heuristic algorithm is explained informally below and more precisely in Algorithm 3;

At first all possible removable input functions ρ are found and added to a list. Then this list members are grouped consecutively and separeted to different lists and these lists are ordered by number of elements in descending order.

Algorithm 3: FIND-ALL-REMOVABLE

Output: l list of possible input functions

```
foreach ρ do
    i = UPDATE-ALL();
    l = l ∩ {i} ;
return l;
```

The Algorithm 3 finds all possible removable input functions and put them to a list. When nodes n_5, n_8, n_9, n_{11}, n_{12}, n_{13} in Figure 4 is tried to remove separetly from the checking sequence, all nodes, excluding selected possible removed node, is proved on Appendix A.

Algorithm 4: GROUP-AND-ORDER-CONSECUTIVES

Input: l possible input function index list

Output: cls consecutive list set

```
find consecutive groups;
order consecutive groups;
return cls;
```

The Algorithm 4 groups and orders consecutives and thus the nodes n_5, n_8, n_9, n_{11}, n_{12}, n_{13}, can be grouped as $\{\{n_{11}, n_{12}, n_{13}\}, \{n_8, n_9\}, \{n_5\}\}$.

3.4.3 Find All Removed Nodes

All removed node list can be found by trying to remove all possible consecutive lists. If the list is removable, we update the checking sequence by removing transitions related with consecutive list elements and by adding new binding nodes (Section 3.4.4) in lieu of removed consecutive list. Then, we remove this consecutive list from the set and we continue to find the other most larger consecutive list to remove.

If the list is not removable, we devide the list to two new list (if the list has for example $1, 2, 3, 4$ elements; the new lists will be $1, 2, 3$ and $2, 3, 4$) and add those new consecutive lists to the set and reorder the descending consecutive list set. We continue this operation until descending consecutive list set gets empty. At the end, all removed nodes are found.

The Algorithm 5 finds largest removed node set. When we look our example, largest consecutive node set is $\{n_{11}, n_{12}, n_{13}\}$. If the Algorithm 1 is run for all nodes of consecutive node set and we iterate Algorithm 2, we can realize that all nodes excluding consecutive node set is proved on Appendix A. But the transition $(s_2 - b/0 \rightarrow s_2)$ of FSM in Figure 1 can not be implemented from reduced checking sequence. Therefore the consecutive node list reordered and a new list created in order to reiterate the algorithm.

The new consecutive list is $\{\{n_{12}, n_{13}\}, \{n_{11}, n_{12}\}, \{n_8, n_9\}, \{n_5\}\}$. As we can also see the first two elements of the list can also be removed but the transition $(s_2 - b/0 \rightarrow s_2)$ of FSM in Figure 1 can not be implemented from reduced checking sequence again. So the consecutive node list reordered again, it is now $\{\{n_8, n_9\}, \{n_5\}, \{n_{11}\}, \{n_{12}\}, \{n_{13}\}\}$.

When $\{n_8, n_9\}$ are tried to remove, at this time the transition $(s_1 - b/1 \rightarrow s_2)$ of FSM in Figure 1 can not be implemented from reduced checking sequence.

Algorithm 5: FIND-LARGEST-REMOVED-NODE-SET

Input: *cls* consecutive list set

Output: *rnl* removed node set

$BIis\rho$'s boolean value;

$BSis\sigma$'s boolean value;

CS checking sequence BNS binding node set **while** *cls is not empty* **do**

  Try if the largest consecutive list *cl* is removable;

  foreach *ρ in cl* **do**

    START-UPDATE(ρ);

  UPDATE-ALL();

  if *cl is removable* **then**

    if *all states of FSM in Figure 1 can be initialized on reduced checking sequence* **then**

      $rnl = rnl \cap cl$;

      $BI = False$;

      $BS = True$;

      $CS = CS \cap BNS$;

    Remove *cl* from *cls*;

  else

    Remove *cl* from *cls*;

    Generate two new consecutive lists from *cl*;

    Add two new *cl* to *cls* and reorder *cls*;

return rnl ;

Now we have a consecutive node list as: $\{\{n_5\}, \{n_{11}\}, \{n_{12}\}, \{n_{13}\}, \{n_8\}, \{n_9\}\}$. All nodes in this list are tried to remove. Node $\{n_5\}$ can not be removed, because the transition $(s_3 - b/1 \rightarrow s_3)$ can not be implemented. Nodes $\{n_{12}\}$ or $\{n_{13}\}$ can not be removed, because again the transition $(s_2 - b/0 \rightarrow s_2)$ can not be implemented. Nodes $\{n_8\}$ or $\{n_9\}$ can not be removed, because again the transition $(s_1 - b/1 \rightarrow s_2)$ can not be implemented.

It is only left node n_{11}, when it is removed all other nodes can be proved on Appendix A and all transitions of FSM in Figure 1 can be implemented from reduced checking sequence.

3.4.4 Adding Binding Nodes

When a group of consecutive nodes are removed from checking sequence, there may be needed to add binding nodes in order to attach previous node of the first member of the consecutive list and next node of the last member of consecutive list. The shortest path from the previous node of the first member of the consecutive list to next node of the last member of consecutive list is found from the FSM of the checking sequence and if this binding node path is smaller than removed consecutive nodes set size, it is added to checking sequence. Otherwise the consecutive node set can not be removed and new consecutive node sets are tried to generate (see Algorithm 5).

In our example we know that, when node n_{11} is removed all other nodes are proved on Appendix A and all transitions of FSM in Figure 1 are implemented from reduced checking sequence. But we do not know now, the previous and next node of n_{11} can bind without adding a new node. From checking sequence in Figure 4, possible state of node n_{10} is s_1 and possible state of node n_{12} is s_2. It can be seen on FSM in Figure 1 that there is a direct transition between states s_1 and s_2. Therefore we can directly bind

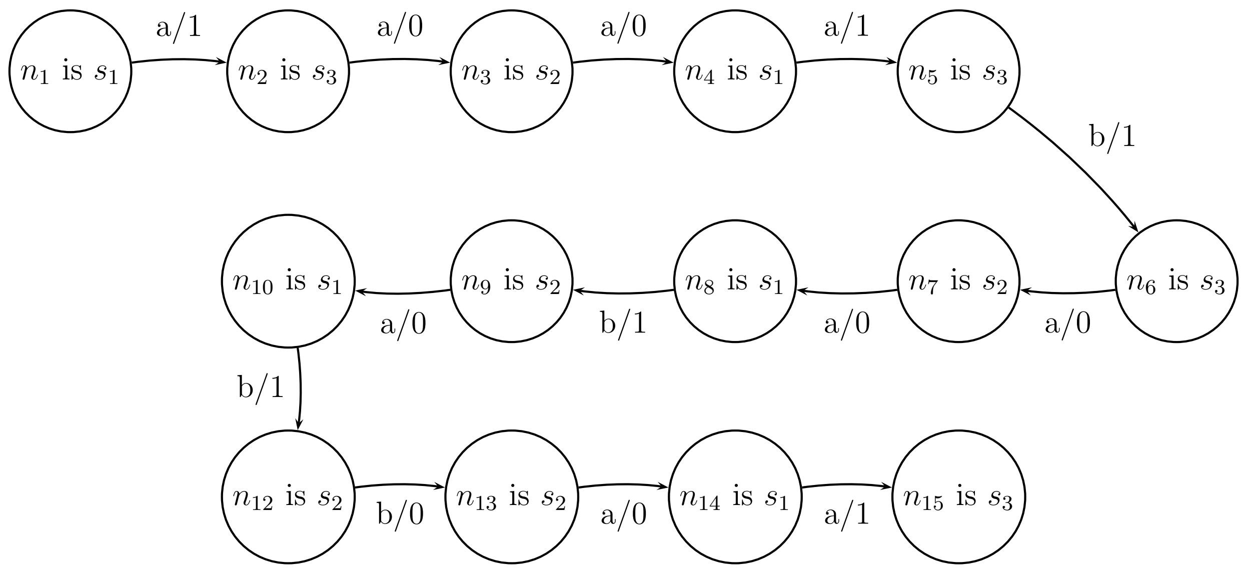

Figure 5: Reduced Checking Sequence of FSM M_1

node n_{10} and n_{12}. New checking sequence will be on Figure 5.

After a new checking sequence is found, the Algorithm 2 is reiterated in order to find new removable nodes. When nodes n_1, n_9 are tried to remove separetly from the checking sequence, all nodes excluding possible removed node, are all proved on Appendix A.

When node n_1 is tried to remove, the transition $(s_1 - a/1 \to s_3)$ of FSM in Figure 1 can not be implemented from reduced checking sequence.

When node n_9 is tried to remove, all states are implemented from the reduced checking sequence. But removing node n_9 is expensive, because the possible state of previous node n_8 and next node n_{10} are s_1 and FSM in Figure 1 does not have a direct transition between s_1 and s_1, so we can not directly bind previous and next nodes.

Therefore only node n_{11} can be removed from checking sequence in Figure 4.

4 Analysis

A variety of methods for the construction of checking sequences have been proposed in the literature [10, 17, 12, 3, 28, 27]. The newly found methods were focused on the improvement of previous methods.

In this section the checking sequence transition reduction analysis will be discussed and effectiveness of our approach will be compared according to different invented methods. The methods have been implemented with Java and the experiments have been executed on a machine with Intel Xeon 3.20 GHz and 64 GB ram.

The random FSM generation tool, that is mentioned in Section 2.5, is used in order to generate diffent FSMs . For the analysis, 4 different groups of FSM are used. Each group of FSM contains 30 FSMs and has 25, 50, 75, 100 number of states respectively. Each FSM has 5 input symbols and 5 output symbols.

We are going to compare our method according to 6 proposed different methods. The comparisons will be in terms of checking sequence length. The performance will be compared in terms of checking sequence length and method execution times.

We also examine the recognition possibility augmentation of example on Appendix A and how binding nodes and FSM implementation satisfaction reduce this possibility.

4.1 Comparisons with Other Methods

In this section, checking sequence optimization of our method is compared according to different methods. For the analysis, 6 different methods are line up in chronological order, so their checking sequence lengths are also line up in decreasing order.

The results show that even there exist low reductions at some points, our method can not be feasible on large FSMs and for all early invented methods.

States	M 1	M 2	M 3	M 4	M 5	M 6
100	25	17	0	10	4	0
75	25	16	0	12	3	0
50	22	23	1	19	6	0
25	24	23	5	15	5	0

Table 1: Number of Reduced Checking Sequences

Table 1 shows how many checking sequences were reduced for different methods form Method 1 to Method 6 depending on the number of states. As we can see from Table 1, as the methods get improved, numbers of checking sequences that are reduced decrease and the possibility that the checking sequences have not any reduction change increases.

States	M 1	M 2	M 3	M 4	M 5	M 6
100	159/7664	123/6852	0/3142	49/3059	6/2453	0/2329
75	112/5466	147/4964	0/2276	47/2237	11/1775	0/1689
50	111/3398	108/2759	44/1603	37/1469	13/1124	0/1067
25	111/1453	72/1072	33/623	25/619	15/501	0/486

Table 2: Ratio of Reduced Lengths over the Original Checking Sequence Lengths

Table 2 shows the ratio of average checking secuence lengths of different invented methods versus checking sequence length reductions with new invented method according to 4 different groups of 30 FSMs with different number of states ranging 25 to 100. While finding the averages, the checking sequences with 0 reductions are ignored for checking sequences groups that the reduction is possible. For instance, the 25 checking sequences of Method 1 for FSMs that has 75 states can be reduced, so the averages are found for checking sequence and reduction lengths on that 25 checking sequences. But any checking sequence of Method 6 can be reduced, therefore they directly get 0 for reduction average.

Even if a slight fluctuation exists on Method 3, while the methods are improved, the reduction averages are diminished and reach 0 for the last invented method.

States	M 1	M 2	M 3	M 4	M 5	M 6
100	2.07%	1.78%	0%	1.6%	0.24%	0%
75	2.05%	2.96%	0%	2.1%	0.61%	0%
50	3.27%	3.91%	2.74%	2.5%	1.16%	0%
25	7.64%	6.72%	5.3%	4.04%	2.99%	0%

Table 3: Gain Percentage for Different Invented Methods

For each different state group, the checking sequence reduction gain percentage comparisons are shown from Table 3. As the number of states increase, gain percentage is decreasing.

From Table 2 and 3, we can say that while the number of states increase and methods improved, the complexity of the checking sequences also increase and when try to remove a checking sequence node, binding the remaining nodes and satisfying FSM implementation is getting difficult.

4.2 Execution Time Analysis

During the analysis, it is noticed that most of the time is spent for *elimination recognition* boolean formula creation. Figure 6 shows *elimination recognition*

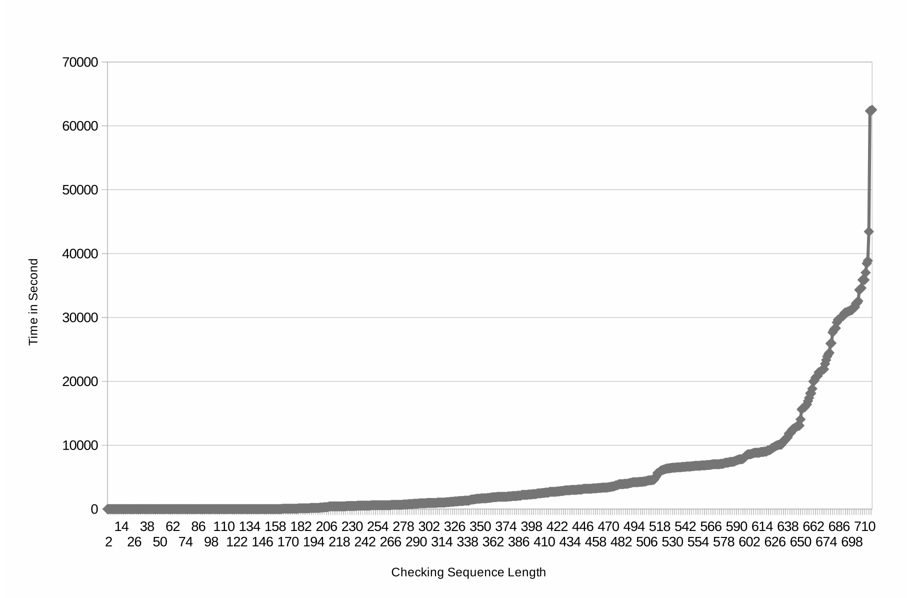

Figure 6: Method execution times by checking sequence length

boolean formula creation execution times in second according to checking sequence length, during reduction analysis. As we can see from Figure 6, as the length of checking sequence increase, the reduction analysis execution time increase exponentially.

4.3 Analysis of the Example

The example has 9 *d-recognition* boolean equations, 30 *t-recognition* boolean equations and 123 *e-recognition* boolean equations. The *e-recognition* boolean equations, that don't have the *possible input function* and *possible state function* of other recognitions have for a node of checking sequence, are 45. Therefore the recognition possibility of a state is augment $1,9\%$ on average.

Ignoring the binding nodes and FSM implementation satisfaction, $\{n_8, n_9, n_{10}, n_{11}, n_{12}, n_{13}\}$ nodes seem to be removed altogether. But if we consider these two conditions, only n_{11} can be removed. Therefore the reduction decrease 83%.

5 Conclusion

Finding the correctness of an FSM is the key point of FSM based testing. The *checking sequence* is used in order to determine this correctness. All invented checking sequence generation methods use distinguishing sequence and it is known that distinguishing sequence may not exist for all FSMs and the main problem of these methods is the reduction of the checking sequence length.

In this thesis, we addressed these problems and found a new state recognition method and tried to reduce checking sequence with a new approach, that is boolean formula based checking sequence optimization algorithm.

The new state recognition method is called *elimination method*. It finds new recognition conditions besides *d-* and *t- recognition* and augments the possibility of the state recognition. In this recognition method, the states conditions, other than the state that has been recognized, are examined and the states that has the same input function and different possible output function than the state that has been recognized has and states that has the same input and possible output function and different next state function than the state that has been recognized has, are taken to elimination recognition set. Therefore during the determination of a *elimination recognition*, the distinguishing sequence is not used.

The other contribution is a new checking sequence optimization algorithm. This algorithm uses boolean formula in order to represent *d-*, *t-* and *e-* recognition conditions of a state. All elements of a recognition condition are anded and all possible recognition conditions of a state are ored with each other. Indeed the checking sequence are represented as a global boolean formula which has ands between each group of state recognition conditions.

Even if, thanks to the new recognition method and new approach for optimizing checking sequence, the recognition possibility of a state is increased, trying to bind checking sequence nodes and to control whether the removable node is broken the FSM state implementation or not, diminish the checking sequence reduction and therefore the improvement can not be feasible with under all of these conditions.

For the improvements and future work, we use a simple heuristic in order to find largest removable node group of the checking sequence, it can be better to find more clever heuristics. The other improvement can be on generating *elimination recognition* conditions. Our algorithm now finds same *elimination recognition* conditions and prevents the regeneration of these same conditions. But a more powerful implementation can be found in order to find similar parts of the *elimination recognition* conditions, therefore we can also prevent regeneration of the same parts of *elimination recognition* conditions and may reduce the time cost.

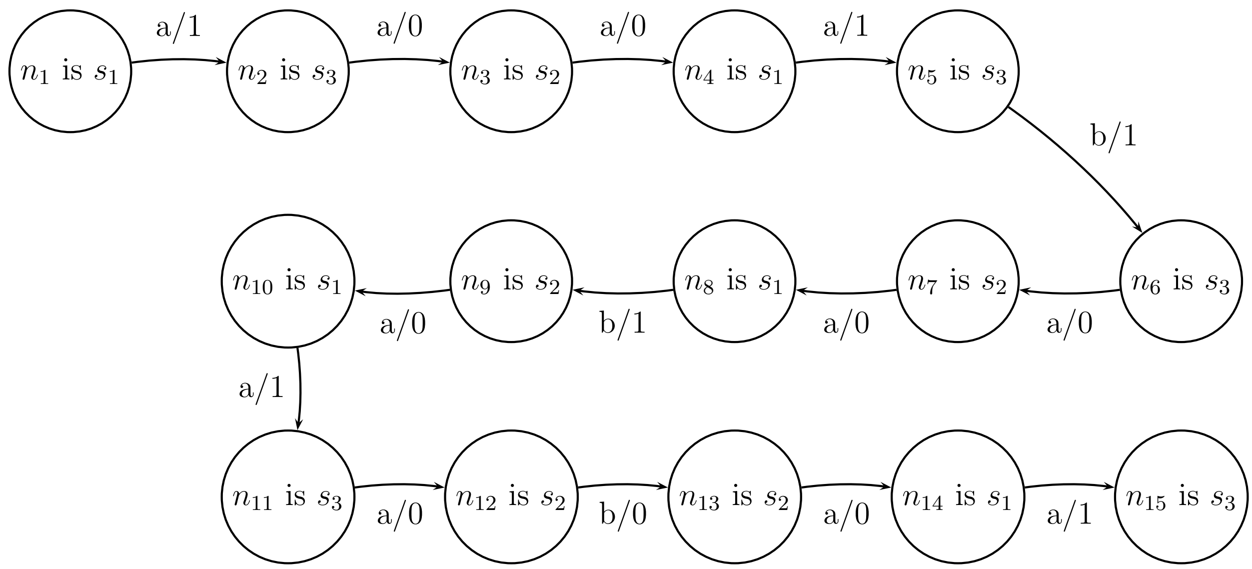

Figure 7: Checking Sequence of FSM M_1

A Boolean Equation of the Example

This appendix give the boolean equation of checking sequence of M_1 on Figure 7.

We use *possible state function* σ and *possible input function* ρ in order to explain boolean equations.

- Possibility of node k been state a, is denoted as n_k is s_a. It is the *possible state function* σ of k.
- Possibility of node k trasition input been x, is denoted as $n_k \to x$. It is the *possible input function* ρ of k.

A.1 Proposition of node n_1 as state s_1

A.1.1 d-recognition

$$\left|\; n_1 \to a \;\right|$$

A.1.2 e-recognition

$$\left|\begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \to a \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \end{array}\right| \wedge \left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \end{array}\right| \wedge \left|\; n_2 \text{ is } s_3 \;\right|$$

A.2 Proposition of node n_2 as state s_3

A.2.1 d-recognition

$$\left|\; n_2 \to a \wedge n_3 \to a \;\right|$$

A.2.2 t-recognition

$$\left|\; n_1 \text{ is } s_1 \wedge n_1 \to a \;\right| \wedge \left|\begin{array}{c} n_4 \text{ is } s_1 \wedge n_4 \to a \wedge n_5 \text{ is } s_3 \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \wedge n_{11} \text{ is } s_3 \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \wedge n_{15} \text{ is } s_3 \end{array}\right|$$

A.2.3 e-recognition

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \end{array}\right| \wedge \left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \wedge n_{10} \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \wedge n_{14} \text{ is } s_1 \end{array}\right| \wedge \left| \, n_3 \text{ is } s_2 \, \right|$$

A.3 Proposition of node n_3 as state s_2

A.3.1 d-recognition

$$\left| \, n_3 \to a \wedge n_4 \to a \, \right|$$

A.3.2 t-recognition

$$\left| \, n_2 \text{ is } s_3 \wedge n_2 \to a \, \right| \wedge \left|\begin{array}{c} n_6 \text{ is } s_3 \wedge n_6 \to a \wedge n_7 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \wedge n_{12} \text{ is } s_2 \end{array}\right|$$

A.3.3 e-recognition

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \end{array}\right| \wedge \left|\begin{array}{c} n_6 \text{ is } s_3 \wedge n_6 \to a \wedge n_7 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \wedge n_{12} \text{ is } s_2 \end{array}\right| \wedge \left| \, n_4 \text{ is } s_1 \, \right|$$

A.4 Proposition of node n_4 as state s_1

A.4.1 d-recognition

$$\left| \, n_4 \to a \, \right|$$

A.4.2 t-recognition

$$\left| \, n_3 \text{ is } s_2 \wedge n_3 \to a \, \right| \wedge \left| \begin{array}{c} n_7 \text{ is } s_2 \wedge n_7 \to a \wedge n_8 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \wedge n_{10} \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \wedge n_{14} \text{ is } s_1 \end{array} \right|$$

A.4.3 e-recognition

$$\left| \begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \to a \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \end{array} \right| \wedge \left| \begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \end{array} \right| \wedge \left| \, n_5 \text{ is } s_3 \, \right|$$

A.5 Proposition of node n_5 as state s_3

A.5.1 t-recognition

$$\left| \, n_4 \text{ is } s_1 \wedge n_4 \to a \, \right| \wedge \left| \begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \wedge n_2 \text{ is } s_3 \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \wedge n_{11} \text{ is } s_3 \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \wedge n_{15} \text{ is } s_3 \end{array} \right|$$

A.5.2 e-recognition

$$\left| \; n_8 \text{ is } s_1 \wedge n_8 \to b \wedge n_9 \text{ is } s_2 \; \right| \wedge \left| \; n_{12} \text{ is } s_2 \wedge n_{12} \to b \; \right| \wedge \left| \; n_6 \text{ is } s_3 \; \right|$$

A.6 Proposition of node n_6 as state s_3

A.6.1 d-recognition

$$\left| \; n_6 \to a \wedge n_7 \to a \; \right|$$

A.6.2 e-recognition

$$\left| \begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \end{array} \right| \wedge \left| \begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \wedge n_{10} \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \wedge n_{14} \text{ is } s_1 \end{array} \right| \wedge \left| \; n_7 \text{ is } s_2 \; \right|$$

A.7 Proposition of node n_7 as state s_2

A.7.1 t-recognition

$$\left| \; n_6 \text{ is } s_3 \wedge n_6 \to a \; \right| \wedge \left| \begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \wedge n_3 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \wedge n_{12} \text{ is } s_2 \end{array} \right|$$

A.7.2 e-recognition

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \end{array}\right| \wedge \left|\begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \wedge n_3 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \wedge n_{12} \text{ is } s_2 \end{array}\right| \wedge \left| n_8 \text{ is } s_1 \right|$$

A.8 Proposition of node n_8 as state s_1

A.8.1 t-recognition

$$\left| n_7 \text{ is } s_2 \wedge n_7 \to a \right| \wedge \left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \wedge n_{10} \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \wedge n_{14} \text{ is } s_1 \end{array}\right|$$

A.8.2 e-recognition

$$\left| n_5 \text{ is } s_3 \wedge n_5 \to b \wedge n_6 \text{ is } s_3 \right| \wedge \left| n_{12} \text{ is } s_2 \wedge n_{12} \to b \right| \wedge \left| n_9 \text{ is } s_2 \right|$$

A.9 Proposition of node n_9 as state s_1

A.9.1 d-recognition

$$\left| n_9 \to a \wedge n_{10} \to a \right|$$

A.9.2 e-recognition

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \end{array}\right| \wedge \left|\begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \wedge n_3 \text{ is } s_2 \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \to a \wedge n_7 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \wedge n_{12} \text{ is } s_2 \end{array}\right| \wedge \left| n_{10} \text{ is } s_1 \right|$$

A.10 Proposition of node n_{10} as state s_1

A.10.1 d-recognition

$$\left| n_{10} \to a \right|$$

A.10.2 t-recognition

$$\left| n_9 \text{ is } s_2 \wedge n_9 \to a \right| \wedge \left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \wedge n_8 \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \wedge n_{14} \text{ is } s_1 \end{array}\right|$$

A.10.3 e-recognition

$$\left|\begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \rightarrow a \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \rightarrow a \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \rightarrow a \end{array}\right| \wedge \left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \rightarrow a \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \rightarrow a \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \rightarrow a \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \rightarrow a \end{array}\right| \wedge \left| n_{11} \text{ is } s_3 \right|$$

A.11 Proposition of node n_{11} as state s_3

A.11.1 t-recognition

$$\left| n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \right| \wedge \left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \wedge n_2 \text{ is } s_3 \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \wedge n_5 \text{ is } s_3 \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \wedge n_{15} \text{ is } s_3 \end{array}\right|$$

A.11.2 e-recognition

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \rightarrow a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \rightarrow a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \rightarrow a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \rightarrow a \end{array}\right| \wedge \left|\begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \rightarrow a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \rightarrow a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \rightarrow a \wedge n_{10} \text{ is } s_1 \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \rightarrow a \wedge n_{14} \text{ is } s_1 \end{array}\right| \wedge \left| n_{12} \text{ is } s_2 \right|$$

A.12 Proposition of node n_{12} as state s_2

A.12.1 t-recognition

$$\left|\ n_{11} \text{ is } s_3 \wedge n_{11} \to a\ \right| \wedge \left|\begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \wedge n_3 \text{ is } s_2 \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \to a \wedge n_7 \text{ is } s_2 \end{array}\right|$$

A.12.2 e-recognition

$$\left|\ n_5 \text{ is } s_3 \wedge n_5 \to b\ \right| \wedge \left|\ n_8 \text{ is } s_1 \wedge n_8 \to b\ \right| \wedge \left|\ n_{13} \text{ is } s_2\ \right|$$

A.13 Proposition of node n_{13} as state s_2

A.13.1 d-recognition

$$\left|\ n_{13} \to a \wedge n_{14} \to a\ \right|$$

A.13.2 t-recognition

$$\left|\ n_{12} \text{ is } s_3 \wedge n_{12} \to a\ \right| \wedge \left|\begin{array}{c} n_5 \text{ is } s_3 \wedge n_5 \to a \wedge n_6 \text{ is } s_1 \\ \vee \\ n_8 \text{ is } s_3 \wedge n_8 \to a \wedge n_9 \text{ is } s_1 \end{array}\right|$$

A.13.3 e-recognition

$$\left|\begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \\ \vee \\ n_{14} \text{ is } s_1 \wedge n_{14} \to a \end{array}\right| \wedge \left|\begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \wedge n_3 \text{ is } s_2 \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \to a \wedge n_7 \text{ is } s_2 \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \wedge n_{12} \text{ is } s_2 \end{array}\right| \wedge \left|\ n_{14} \text{ is } s_1\ \right|$$

A.14 Proposition of node n_{14} as state s_1

A.14.1 d-recognition

$$\left| \, n_{14} \to a \, \right|$$

A.14.2 t-recognition

$$\left| \, n_{13} \text{ is } s_2 \wedge n_{13} \to a \, \right| \wedge \left| \begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \wedge n_4 \text{ is } s_1 \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \wedge n_8 \text{ is } s_1 \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \wedge n_{10} \text{ is } s_1 \end{array} \right|$$

A.14.3 e-recognition

$$\left| \begin{array}{c} n_2 \text{ is } s_3 \wedge n_2 \to a \\ \vee \\ n_6 \text{ is } s_3 \wedge n_6 \to a \\ \vee \\ n_{11} \text{ is } s_3 \wedge n_{11} \to a \end{array} \right| \wedge \left| \begin{array}{c} n_3 \text{ is } s_2 \wedge n_3 \to a \\ \vee \\ n_7 \text{ is } s_2 \wedge n_7 \to a \\ \vee \\ n_9 \text{ is } s_2 \wedge n_9 \to a \\ \vee \\ n_{13} \text{ is } s_2 \wedge n_{13} \to a \end{array} \right| \wedge \left| \, n_{15} \text{ is } s_3 \, \right|$$

A.15 Proposition of node n_{15} as state s_3

A.15.1 t-recognition

$$\left| \, n_{14} \text{ is } s_1 \wedge n_{14} \to a \, \right| \wedge \left| \begin{array}{c} n_1 \text{ is } s_1 \wedge n_1 \to a \wedge n_2 \text{ is } s_3 \\ \vee \\ n_4 \text{ is } s_1 \wedge n_4 \to a \wedge n_5 \text{ is } s_3 \\ \vee \\ n_{10} \text{ is } s_1 \wedge n_{10} \to a \wedge n_{11} \text{ is } s_3 \end{array} \right|$$

References

[1] Fsm-based incremental conformance testing methods. *IEEE Trans. Softw. Eng.*, 30:425–436, July 2004.

[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. Test design modeled by finite-state machines. *IEEE Trans. Software Eng.*, pages 178–187, 1978.

[5] M. Emre Dinçtürk. A two phase approach for checking sequence generation. Master's thesis, Sabanci University, 2009.

[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] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal test-case generation for uml statecharts. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:75–84, 2004.

[9] Guney Gonenc. A method for the design of fault detection experiments. *Computers, IEEE Transactions on*, C-19(6):551 – 558, june 1970.

[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] R. M. Hierons and H. Ural. Uio sequence based checking sequences for distributed test architectures. *Information and Software Technology*, 45:798–803, 2003.

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

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

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

[15] Robert M. Hierons, Guy-Vincent Jourdan, Hasan Ural, and Husnu Yenigun. Using adaptive distinguishing sequences in checking sequence constructions. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 682–687, New York, NY, USA, 2008. ACM.

[16] Robert M. Hierons and Hasan Ural. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(46):93–99, 1997.

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

[18] Florentin Ipate. Test selection for hierarchical and communicating finite state machines. *Comput. J.*, 52:334–347, May 2009.

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

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

[21] 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.

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

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

[24] B. Sarikaya, G. v. Bochmann, and E. Cerny. A test design methodology for protocol testing. *IEEE Trans. Softw. Eng.*, 13:518–531, May 1987.

[25] B. Settles. ABNER: An open source tool for automatically tagging genes, proteins, and other entity names in text. *Bioinformatics*, 21(14):3191–3192, 2005.

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

[27] 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.

[28] 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.