Department of Computer Science
San Marcos, TX 78666

Report Number TXSTATE-CS-TR-2011-26

THE INTRACTABILITY OF FINITE STATE MACHINE TEST SEQUENCES

Ngozi I. Ihemelandu
Carl J. Mueller

2011-01-03

TABLE OF CONTENTS

Page

LIST OF TABLES iii

LIST OF FIGURES iv

CHAPTER I INTRODUCTION 1

1.1 Motivation 1

1.2 Thesis Statement 2

CHAPTER II BACKGROUND 4

2.1 State Machines 4

2.1.1 Finite State Machines 6

2.1.2 Extended Finite State Machines 12

2.2 State Verification 13

2.2.1 Partial Specification and Completeness Assumption 14

2.2.2 Initial and Current State Uncertainty 16

2.2.3 The UIO Successor Tree 17

2.3 The State Verification Complexity 21

2.3.1 The State Verification time Complexity (Big O notation) 23

2.4 Cyclomatic Number 24

CHAPTER III RESEARCH HYPOTHESIS 25

3.1 Complexity of the State Verification Problem 26

3.2 Cyclomatic Number as a Predictive Metric 29

CHAPTER IV EXPERIMENTATION 31

4.1 Overview 31

4.2 Experiment Description 31

4.3 Experiment Execution 37

4.4 Experiment Result 38

CHAPTER V CONCLUSION AND FUTURE RESEARCH 46

APPENDIX A UIO SEQUENCE GENERATION ANALYSIS PROGRAM 51

APPENDIX B FSM RANDOM GENERATOR PROGRAM 90

APPENDIX C UNIQUE INPUT OUTPUT (UIO) GENERATOR PROGRAM 98

APPENDIX D TIMER ROUTINE 108

BIBLIOGRAPHY 110

LIST OF TABLES

Table Page

1. Randomly generated fully specified FSM with reflexive transitions 34
2. Set of UIO sequences generated for FSM in Table 1 35
3. A randomly generated fully specified FSM with no reflexive transitions........... 36
4. Set of UIO sequences generated for FSM in Table 3 36
5. A partially specified FSM.. 44
6. Transition Function for FSM in Table 5... 44
7. Set of UIO sequences generated for FSM in Table 5 45

LIST OF FIGURES

Figure | Page

1. The Chomsky Hierarchy ... 5
2. An FSM in which all states have UIO sequences, but there exist no preset distinguishing sequence. ... 10
3. An FSM in which states s1 and s2 have UIO sequences, but s4 does not. ... 10
4. An FSM in which no state has a UIO sequence ... 11
5. Transition diagram of a finite state machine M. ... 18
6. The successor tree T of the machine in Figure. 5 ... 20
7. The UIO successor tree T of the machine in Figure. 5 ... 20
8. State Chart diagram for Table 1 ... 35
9. State Chart diagram for Table 3 ... 36
10. Log-Log Graph of Predicted Execution Time for UIO sequence ... 39
11. Log-Log Graph for Observed Execution Time for UIO Sequence ... 40
12. Observed Execution Time for UIO sequence. ... 41
13. Observed Execution Time for UIO sequence ... 42
14. Transition Types on the performance of the UIO sequence generation ... 43
15. Observed Execution Time for UIO sequence generation. ... 45

THE INTRACTABILITY OF FINITE STATE MACHINE TEST SEQUENCES

CHAPTER I

INTRODUCTION

1.1 Motivation

Computers and software systems affect almost every facet of modern society. The use of software ranges from mundane tasks, such as information processing or air flight booking, to use in the medical field, such as in life saving devices like implanted-cardiac defibrillators. Because of the deep-seated role of software in almost every aspect of our daily lives, a small defect in a software application may result in severe financial losses and in some cases life loss.

The new Denver International Airport was to be a wonder of modern engineering. Its opening was delayed for almost a year, at a cost to the airport authority of over $1 million a day due to software-related failure of the automated baggage (Gibbs, Wayt, W. 1994). There are other documented real-life instances that illustrate the disastrous effect of software failures. (Gallagher 2007, Leveson 1995, Gleick 1996).

Most of the disastrous software failures in recent years are as a result of inadequate testing. Finite state machines are used to model external system behavior (black box view) or detailed execution of specific implementation (white box view) (Tian 2005).

1.2 Thesis Statement

Evaluating software with state behavior is a major research issue in software testing. State verification is one of the challenges in testing FSMs in which we know the state diagram of the system under test, and this machine is assumed to be in a particular state. The objective of the state verification experiment is to check that the assumption that a machine is in a particular state s is correct. An input sequence that solves this problem is known as the Unique Input Output Sequence (UIO) (Lee and Yannakakis 1994, 306-320; Lee and Yannakakis 1996, 1090-1123; Sabnani and Dahbura 1988, 285-297; Broy and others 2005). One of the most frequently cited papers on testing software with state-behavior is “Testing Finite-State Machines: State Identification and Verification”, in which Lee proved that it is PSPACE-complete to determine if a specific state ‘s’ of a given FSM ‘M’ has a UIO sequence (Lee and Yannakakis 1994, 306-320). PSPACE-complete problems are a set of problems in which there are presumably no efficient, polynomial time, algorithm for the general solution of the problem. Hence, a general solution would require an exponential amount of time to process as the number of the elements in the problem increases (Sipser 1997). It is interesting to note that this theorem advocated by Lee holds even when the FSM is restricted to binary input and output alphabets (Lee and Yannakakis 1994, 306-320).

The focus of this research is an investigation on the complexity of FSM with demonstrations that:

1. The State Verification testing challenge is PSPACE-complete as proved theoretically by Lee (Lee and Yannakakis 1994, 306-320).

2. The following factors contribute to the PSPACE intractability characteristic of the UIO generation algorithm
    I. Transitions types: loop and reflexive transition
    II. Number of states
    III. Input size
3. There exists a correlation between the cyclomatic number of a FSM and the time it takes to generate the UIO sequence.

Much work has been done on the generation of UIO sequence, but not much research has been done with respect to the causes of PSPACE-complete intractability in a classic solution to the state verification problem. Chapter II of this work introduces some concepts in graph theory and automata that lays the foundation for the intended research. Chapter III details the theoretical background and hypothesis of the research. Chapter IV describes the experimental analysis of the hypothesis developed in Chapter III. Chapter V lays out the findings of the research and the analysis of these findings.

CHAPTER II

BACKGROUND

In this Chapter, we shall focus on understanding some fundamental concepts of finite state machines, extended finite state machines, graph theory, successor tree, and McCabe's Cyclomatic number. We shall attempt to establish the relationship that exists among them. By establishing these relationships, concepts from graph theory will be applied in the analysis of FSM, Successor tree and McCabe's Cyclomatic number. To study the complexity of an FSM, we shall focus on the State Verification FSM testing problem.

2.1 State Machines

The Chomsky hierarchy is a nested hierarchy of classes of formal grammars. Each level of the Chomsky hierarchy specifies both the grammar formalism and the computational structure of the formal language class. The basis for the Chomsky hierarchy is the amount and organization of the memory required to process the languages at each level (Rich 2008). The Chomsky hierarchy consists of the following levels:

- Type 0 (semi-decidable): no memory constraint
- Type 1 (context-sensitive): memory limited by the length of the input string
- Type 2 (context-free): unlimited memory but accessible only in a stack (so only a finite amount is accessible at any point)
- Type 3 (regular): finite memory (Rich 2008)

The Chomsky hierarchy makes an obvious suggestion: Different grammar formalisms offer different descriptive power and may be appropriate for different tasks (Rich 2008). In other words, Chomsky hierarchy gives a platform that specifies the differences in state machines. The EFSM falls within the outermost level (type 0) of the Chomsky hierarchy because the set of variables V provides unlimited working memory for storing the results of intermediate steps of the computation. EFSM is a representation of the Turing machine.

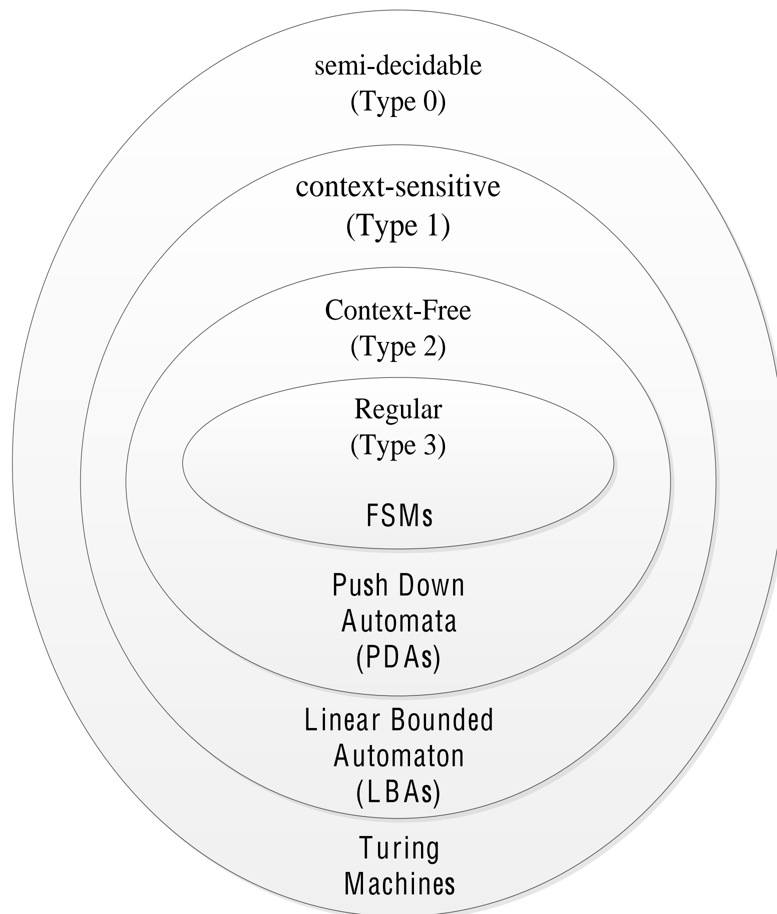

Figure 1. The Chomsky Hierarchy

2.1.1 Finite State Machines

A finite state machine (FSM) is a model of behavior using states and state transitions. It is a widely used model in just about every area of the software industry and is particularly popular with designers of telecommunication systems, communication protocols, embedded systems, and control systems. There are two types of finite state machines: Mealy machines and Moore machines (Lee and Yannakakis 1996, 1090-1123). FSMs are typically modeled as Mealy machines, which are deterministic machines that produce outputs on their state transitions after receiving inputs. An FSM M is a quintuple M = (I, O, S, δ, λ), where O, S, and I are finite and nonempty sets of input symbols, output symbols, and states, respectively.

δ: S x I → S is the state transition function;

λ: S x I → O is the output function (Lee and Yannakakis 1994, 306-320).

FSMs originate from Finite Automata which are defined as a quintuple (Q, Σ, δ, $q_{0,}$ F)

1. Q is a finite set called the states
2. Σ is a finite set called the alphabet
3. δ: Q x Σ → Q is the transition function
4. q_0 Є Q is the start state, and
5. F ⊆ Q

Finite Automata are primarily used in parsing for recognized languages. Input strings that are members of a given language should turn an Automaton to its final states but all other input strings turn the Automaton to states other than the final states (Sipser 1997). FSMs have output, while automata do not.

FSMs can be represented by state transition diagrams, which are directed multigraphs where the vertices correspond to the states of the FSM and the edges correspond to the state transitions. A directed graph that has multiple directed edges connecting the same vertices are called multigraphs. Formally, a directed graph (V, E) is defined as a nonempty set of vertices V and a set of directed edges E. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with an ordered pair (u, v) is said to start at u and end at v (Rosen 2007). A path in a directed graph is a sequence of edges where the terminal vertex of an edge is the same as the initial vertex in the next edge in the path (Rosen 2007). A path can pass through a vertex more than once and an edge can occur more than once in a path. In most graph-theory literature, loops are defined as edges that connect a vertex to itself and a circle or circuit is defined as a path that begins and ends at the same vertex (Rosen 2007). However, in this paper, circles or circuits will be referred to as loops and loops will be referred to as reflexive transitions. A directed graph G is strongly connected if there is a path from x to y and from y to x whenever x and y are vertices in the graph. Although this graphical representation of an FSM is intuitive and easy to interpret by human subjects, it becomes impractical when the number of states becomes large. State diagrams with more than 20 or 30 states are messy and hard to trace. Consequently, tabular representations are often used (Tian 2005).

FSM is used to study the intended system to be implemented and is used in the testing phase of product development to generate complete test suites to check conformance of the model and the actual implementation. To deduce lacking information from an FSM, a sequence of input symbols are provided to it and the resulting output symbols produced

are observed. The State Identification testing problem attempts to identify the current state of a given FSM with known states and transitions but an unknown current state. This is not always possible as there are FSMs for which no test exists that will allow us to identify their current states. The input sequence that solves this problem, if it exists, is called the distinguishing sequence (Lee and Yannakakis 1994, 306-320; Lee and Yannakakis 1996, 1090-1123).

Formally defined, a *preset distinguishing sequence* for a machine is an input sequence x such that the output sequence produced by the machine in response to x is different for each initial state, i.e., $\lambda(s_i , x) \neq \lambda(s_j , x)$ for every pair of states s_i, s_j, $i \neq j$ (Lee and Yannakakis 1994, 306-320; Lee and Yannakakis 1996, 1090-1123). Lee and Yannakakis showed that solving the preset distinguishing sequence challenge is PSPACE-complete (Lee and Yannakakis 1994, 306-320). They also proved that there are machines for which the shortest preset distinguishing sequence has exponential length. However, they presented a deterministic polynomial time algorithm and polynomial length for the adaptive distinguishing sequence problem. The state verification is a more restricted problem than the state identification problem because it narrows the problem to that of verifying that an FSM is in a given state. Hence, given an FSM M, it is assumed to be in a particular state $s \in S$. The objective is to check that this assumption is correct. This is possible if, and only if, that state has a Unique Input Output (UIO) sequence. Formally defined, a *UIO sequence* of a state s_0 is an input sequence x_0 such that the output sequence produced by the machine in response to x_0 from any state other than s_0 is different than that from s_0, i.e., $\lambda(s_i , x_0) \neq \lambda(s_0, x_0)$ for any $s_i \neq s_0$ (Lee and Yannakakis 1994, 306-320; Lee and Yannakakis 1996, 1090-1123).

If an FSM has a preset or an adaptive distinguishing sequence, then all states have UIO sequences. For a given machine, it is possible that no state has a UIO sequence, that some states have a UIO sequence and some do not, or all states have a UIO sequence; but there is no preset or adaptive distinguishing sequence (Lee and Yannakakis 1994, 306-320; Lee and Yannakakis 1996, 1090-1123). For instance, in the FSM shown in Figure 1, there exists no preset distinguishing sequence since states s2 and s3 give the same output for any input starting with an a; and s1 and s4 give the same output when the input starts with a b. However, all states have UIO sequences. Figure 2 shows an FSM in which some states have UIO sequences and others do not, and Figure 3 shows an FSM in which no state has a UIO sequence (Broy and others 2005).

UIO sequences can verify a larger class of machines than distinguishing sequence, this is one of the reasons for studying state verification. Hsieh introduced these sequences, and algorithms for finding them have been studied by Lee and Sabnani (Broy and others 2005, Lee and Yannakakis 1994, 306-320, Sabnani and Dahbura 1988, 285-297). Lee and Yannakakis proved that this problem is PSPACE-complete (Lee and Yannakakis 1994, 306-320). This thesis focuses on the results of these studies.

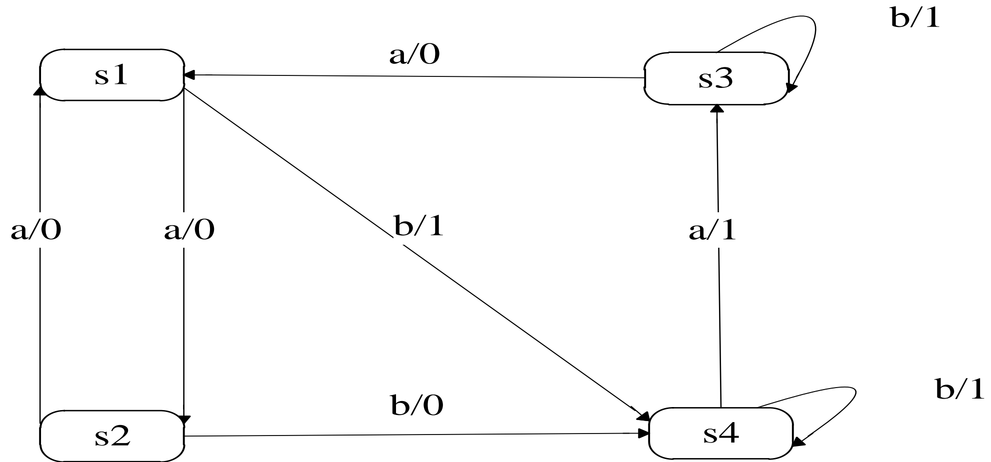

Figure 2 An FSM in which all states have UIO sequences, but there exist no preset distinguishing sequence.

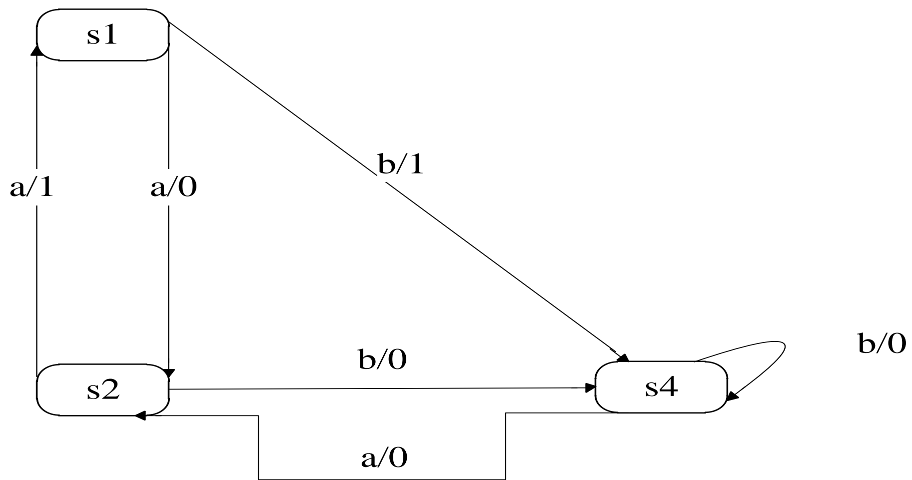

Figure 3 An FSM in which states s1 and s2 have UIO sequences, but s4 does not.

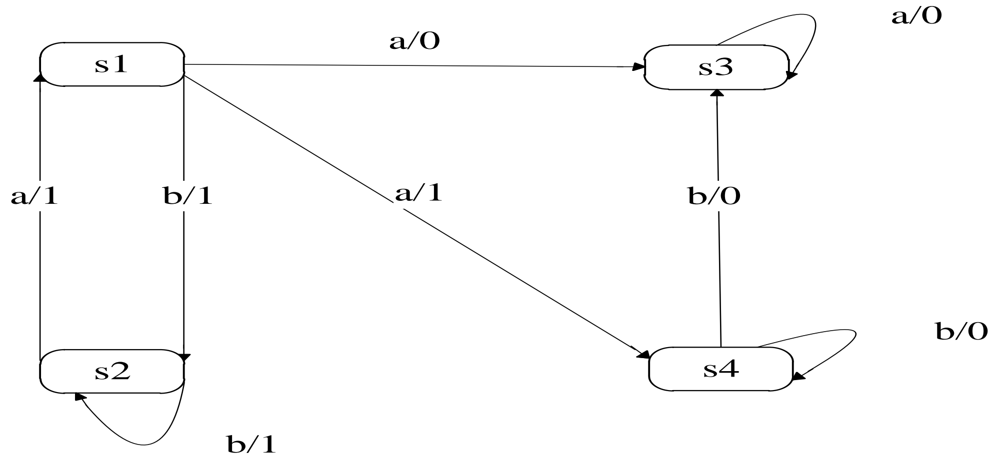

Figure 4 An FSM in which no state has a UIO sequence

However, the distinguishing sequences and UIO sequences provide solutions to the state identification and verification problems respectively, these sequences have been useful in the development of techniques to solve the conformance-testing problem, otherwise known as the fault detection problem. The main use for the state verification is as a part of the algorithms for conformance testing. Hence, UIO sequences are mainly used as part of conformance testing algorithms for the construction of checking sequences (Lee and Yannakakis 1994, 306-320; Broy and others 2005; Sabnani and Dahbura 1988, 285-297). In a conformance testing challenge, given an FSM *M* that models a specification and a black box implementation machine B, we want to check that B is a correct implementation of M. The sequence that solves this problem is known as the checking sequence. Formally defined, a checking sequence for an FSM M is an input sequence that distinguishes the class of machines equivalent to M from other machines (Broy and others 2005; Lee and Yannakakis 1996, 1090-1123). The checking sequence attempts to detect faulty transitions in the implementation machine by checking whether each state and edge in the FSM exists in the implementation and whether each edge has a correct label. Conformance is established by creating a isomorphism between the specification

and the implementation. The checking sequence consists of three parts: initial sequence, state recognition sequence and the transition checking sequence. The checking sequence brings M to a state and applies a sequence of inputs to recognize the state reached. Recognizing states can be based on distinguishing sequences or unique input-output (UIO) sequences. Many proposed checking sequence generation procedures are based on UIO sequences.

2.1.2 Extended Finite State Machines

A number of software applications require variables to provide a complete specification. Extended Finite State Machine Model (EFSM) is an enhanced state machine model based on the traditional finite state machine (FSM), which uses variables. This additional feature of an EFSM models the robust memory found in a software environment more closely. EFSMs are at the top of the Chomsky hierarchy because of the inclusion of variables in the model

A tuple (M) with six (6) elements formally defines an extended finite state machine (EFSM):

$$M = (\Sigma, V, Q, S_I, S_F, T)$$

Where:

Σ is a finite, nonempty set of events or operations,

V is a finite set of variables,

Q is a finite, nonempty set of states,

S_I is the initial state in the model where $sI \in Q$,

S_F is the finial state in the model where $sF \in Q$,

T is a finite nonempty set of transitions, where each transition t is represented by the tuple:

$t = \{S_O(t), S_T(t), e(t), C(t), A(t)\}$

where:

$S_O(t)$ is the transition's originating state,

$S_T(t)$ is the transition's terminating state,

e(t) is an event or operation causing the transition,

C(t) is the enabling conditional expression that enables the transition,

A(t) is a sequence of actions associated with a specific transition (MUELLER, 2003).

In an EFSM, the set of transitions T describes how the set of states Q, events Σ and variables V are used. In each transition element t, there is an event $e(t) \in \Sigma$ that provides the basic predictor for selecting one transition over another. The set of variables V provides storage for information necessary for the enabling condition C(t). The originating state $S_O(t) \in Q$ describes where the transition originates, and the terminating state $S_T(t) \in Q$ describes where the transition ends. The action sequence A(t) establishes values for variables found in V and other action statements necessary to describe the operation of the model (MUELLER, 2003).

2.2 State Verification

In the previous section, we introduced three finite state machine-testing problems. To study the complexity of a FSM we shall focus on the state verification-testing problem. In the state verification FSM testing problem, the state diagram of the system under test is

known, and it is assumed that the FSM is in a particular state. The objective of the test is to verify that the assumption made about the current state of the FSM is correct. To achieve this, an input sequence passes through the software under test, and the state verification experiment is used to verify that the sequence took the system to the expected state. The search process for finding the UIO sequence for a state of an FSM uses a tree known as the successor tree. By the definition of UIO sequences, different states of an FSM may have different UIO sequences; hence, many trees may be constructed for an FSM where each tree searches for the UIO sequence for a particular state. Before examining the structure of the UIO successor tree, it is imperative to define some concepts that will be used in the analysis of the UIO successor tree.

2.2.1 Partial Specification and Completeness Assumption

Finite State Machines are of two types: (i) deterministic and (ii) non- deterministic (Broy and others 2005). A deterministic FSM is one in which for every state, there can be one and only one transition for a particular input. A non-deterministic FSM is a FSM in which for any state, there can be more than one transition with the same input (Sun, Shen, and Feng 1997; Sipser 1997). In this paper, only the deterministic FSMs are considered. Deterministic FSMs are divided into two categories:

i. Completely (fully) specified
ii. Incompletely (partially) specified.

A FSM is completely specified if for every state and every valid input, the behavior of the FSM is specified. If on the other hand, the behavior for a particular input for a state in the FSM is not specified, the FSM is said to be incompletely or partially specified (Sun, Shen, and Feng 1997). Most FSMs are incompletely specified, and the specified

state-input behavior of an FSM is referred to as core behavior (Sabnani and Dahbura 1988, 285-297). For an input applied in a state of an FSM for which the output and the next state behavior are not specified in the core behavior, it is assumed that a null output is produced and it remains in its current state. This assumption is called the Completeness Assumption. (Sabnani and Dahbura 1988, 285-297; Broy and others 2005). An alternate completeness assumption of a specification FSM generates an error output and enters the error state whenever it receives an input in a state for which the behavior is not specified by the core behavior (Sabnani and Dahbura 1988, 285-297). In this paper, the completeness assumption concept adopted is strictly of the former type. An edge from each state corresponding to a non-core input is a self-loop with a null output in its label. These non-core edges are not shown in the directed graph representation of the randomly generated FSM. It is impossible to develop a technique for FSM state verification testing without any assumptions. Hence, for all machines considered in this paper, it is assumed that they are minimized, strongly connected and completely specified. A Mealy machine is minimized if it has no equivalent states (Broy and others 2005). If an FSM is not minimized, then there will always be states that do not have UIO sequences since there are at least two states that produce the same outputs for every input sequence. The classical algorithm for UIO sequence generation analyzed in this paper assumes that the FSMs are fully specified. For every state in the FSM, there must be a path to it from every other state. This ensures the absence of deadlock states in the FSM. By restricting, the FSM for consideration to deterministic machines where the next operation and next state depend solely on the current state and input, it is assumed that

there are no control variables or counters manipulated by the control operations which might influence transitions from a state as a response to input.

2.2.2 Initial and Current State Uncertainty

Given a machine M = (I, O, S, δ, λ) and an input sequence x, x induces a partition $\pi(x)$ on the set of states S of M, where two states si, sj are placed in the same block B of the partition if and only if they are not distinguished by x, i.e., λ(si, x) = λ (si, x). This partition $\pi(x)$ is called the *initial state uncertainty of x* (Lee and Yannakakis 1996, 1090-1123). The elements of the initial state uncertainty $\pi(x)$ are called blocks. Formally defined, blocks are nonempty subset of states (Broy and others 2005). $|\pi(x)|$ = number of blocks; $\pi(x)$ is called a partition because any two blocks in $\pi(x)$ are disjointed and the union of all blocks is the entire set of states S. Consider M with a set of states S = {a,b,c,d}, if M can initially be in any of this states then the initial state uncertainty π = (abcd) (Kohavi 1978). The aim of π is to identify or verify the initial or current state of M, by reducing the initial state uncertainty π until each block B in π is a singleton or the states in B of π cannot be further distinguished. A *singleton* is a component or block of a partition π that contains a single state.

The *current state uncertainty of x*: $\sigma(x) = \{ \delta(B,x) \mid B \in \pi(x) \}$ (Lee and Yannakakis 1996, 1090-1123). σ(x) is not necessarily a partition; i.e., the sets in σ(x) are not necessarily disjointed. The output produced by M in response to the input sequence x tells us to which member of σ(x) the current state belongs (Lee and Yannakakis 1996, 1090-1123).

For example, consider the machine M shown in Figure 3. Initially the initial state is unknown and could be any of the states.

initial state uncertainty $\pi(€) = \{\{s_1, s_2, s_3\}\}$ (*where € is the empty string*)

input symbol *b* induces the partition $\pi(b) = \{\{s_1, s_2\}_1, \{s_3\}_0\}$

current state uncertainty $\sigma(b) = \{\{s_2, s_3\}_1, \{s_1\}_0\}$

input symbol a induces the partition $\pi(ba) = \{\{s_1\}_{11}, \{s_2\}_{10}, \{s_3\}_{00}\}$

current state uncertainty $\sigma(ba) = \{\{s_2\}_{11}, \{s_3\}_{10}, \{s_1\}_{00}\}$

The subscript denotes the output for each block.

In terms of UIO sequences, this means that distinct states s_i, in the same block and partition with s_0 (where s_0 is the state for which a UIO sequence is considered) cannot be distinguished by the input sequence x. Hence, we say that our uncertainty about state s_0 of M has been reduced to a block of states. However, if the block is a singleton then our uncertainty is totally reduced.

2.2.3 The UIO Successor Tree

The concepts of trees in graph theory are applied in the techniques used for the derivation of UIO sequence. A tree is a connected undirected graph with no circuits, multiple edges or loops. Therefore, there exists a unique simple path between two of its vertices (Rosen 2007). A tree with a particular vertex designated as the root, every edge directed away from the root and a unique path from the root to each vertex of the graph is a rooted tree. Thus, a rooted tree is a directed graph. A rooted tree is a full m-ary tree if every internal vertex has exactly m children (Rosen 2007). In this paper, only rooted full m-ary trees will be considered which are referred to as successor trees in literatures on FSM model based testing. The successor tree of a specified machine M is a tree showing the behavior of the machine starting from all possible initial states (which is the set of states of the considered machine) under all possible input sequence (Lee and Yannakakis 1996, 1090-

1123). Each node of the successor tree has exactly as many outgoing edges as the number of inputs of the machine. Every internal node is annotated with a current state uncertainty which corresponds to the input sequence defined by the path starting from the root of the successor tree to the node under consideration. For example, consider the successor tree (shown in Figure 6), associated with the FSM M in Figure 5.

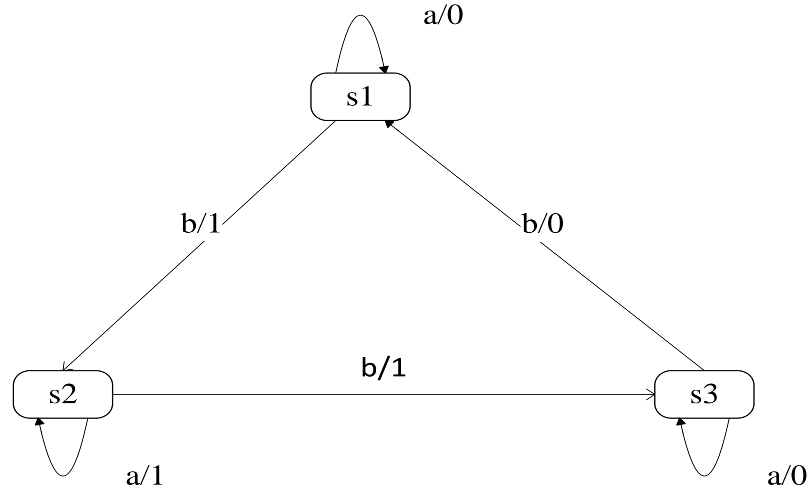

Figure 5 Transition diagram of a finite state machine M.

The root of the successor tree is annotated with the set of states S of M which is a one block partition.

$$\pi = \{\{s1, s2, s3\}\}$$

The input *a* refines the partition π to

$$\pi\ (a) = \{\{s1, s3\}, \{s2\}\}$$

Where the block or partition {s1, s3} corresponds to the output 0 and {s2} to 1. The left child node of the Root node in the successor tree T is annotated with the current state uncertainty {{s1, s3}, {s2}}. The input b refines the partition π to

$$\pi\ (b) = \{\{s1, s2\}, \{s3\}\}$$

Where the block {s1, s2} corresponds to the output 1 and {s3} to 0. Hence, the right child node of the Root node is annotated with the current state uncertainty {{s2, s3}, {s1}}. The input symbol a leaves the partition π (a) unchanged while b further refines π(a) to

$$\pi\ (ab) = \{\{s1\}, \{s3\}, \{s2\}\}.$$

The left and right children of the π(a) node are annotated by the current state uncertainty {{s1, s3}, {s2}} and {{s2}, {s3}, {s1}} respectively. Similarly, the input symbol a refines the partition π(b) to

$$\pi\ (ba) = \{\{s1\}, \{s2\}, \{s3\}\}.$$

and the input symbol b refines π(b) to

$$\pi\ (bb) = \{\{s1\}, \{s2\}, \{s3\}\}.$$

The left and right children of the π(b) node are annotated by the current state uncertainty {{s2}, {s3}, {s1}} and {{s3}, {s1}, {s2}} respectively. Since the blocks in the current state uncertainty of these nodes are singletons, these nodes are referred to as leaf nodes. The sequences ab, ba and bb are distinguishing or UIO sequences since they trace a path from the root of the successor tree to a leaf node which is a current state uncertainty with elements that are singletons.

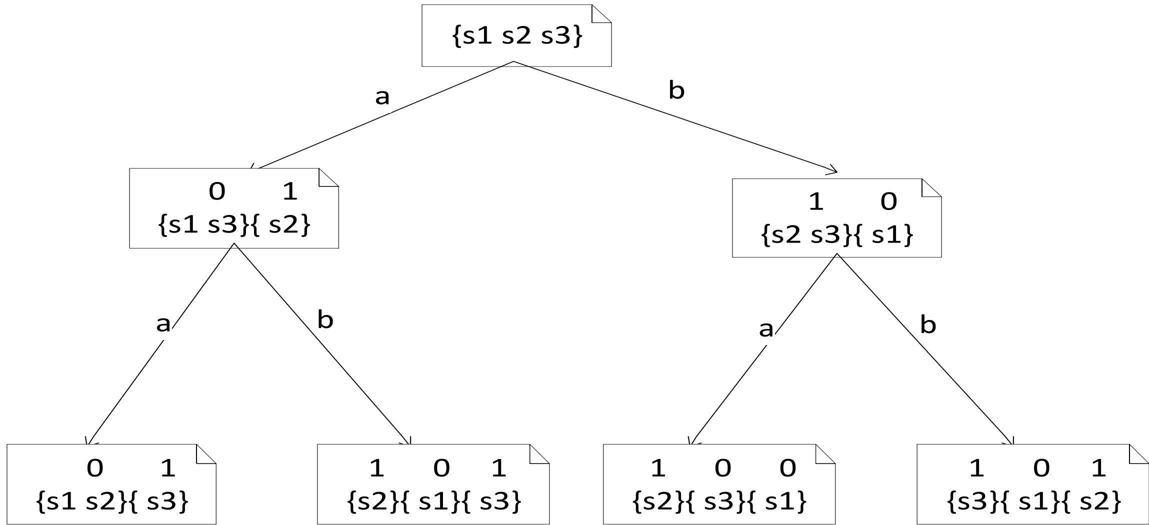

Figure 6 The successor tree T of the machine in Figure. 5

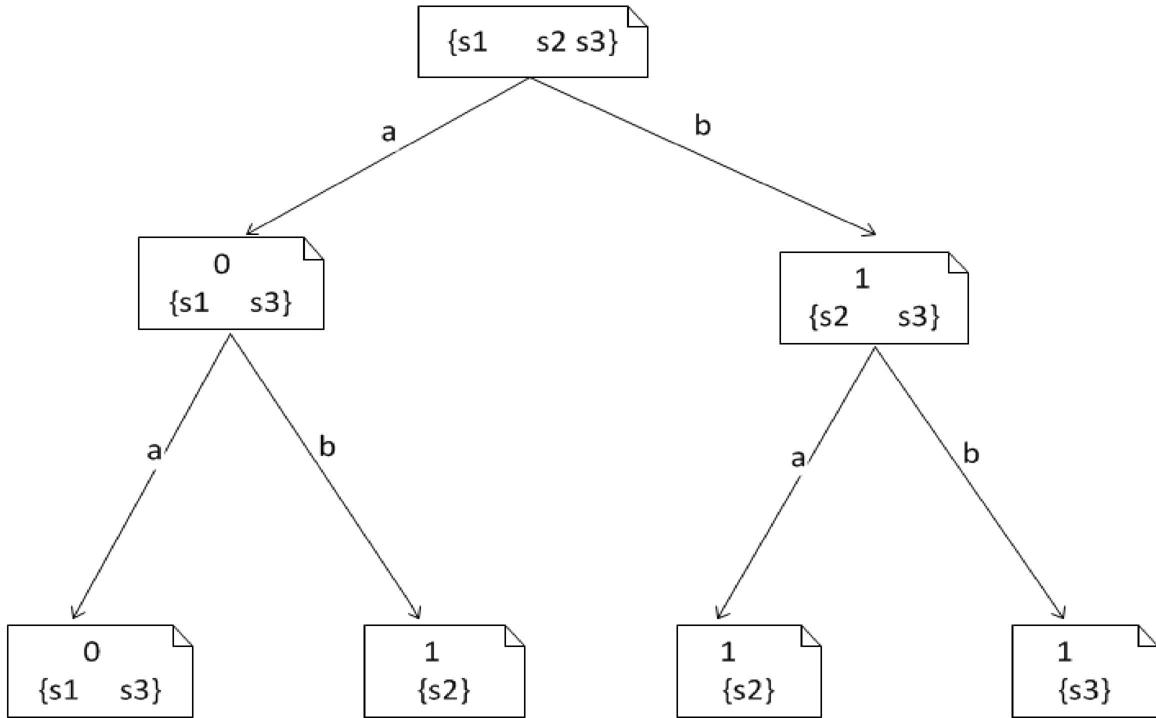

Figure 7 The UIO successor tree T of the machine in Figure. 5

A UIO tree is a successor tree for a particular state s of a FSM M. Only blocks in the current state uncertainty that contain the state for which a UIO sequence is sought are considered for further processing. States in other blocks imply that these states have different I/O behaviors; hence, these other blocks can be ignored.

Figure 7 shows a UIO successor tree for the state s1 of the FSM M in Figure 5. The blocks in the current state uncertainty which corresponds to the internal nodes of the UIO tree are limited to blocks that contain the state s1.

2.3 The State Verification Complexity

In this section, we would discuss the complexity class to which this decision problem belongs. Complexity classes play a very important role in characterizing the practical solvability of the problems that they contain.

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. P roughly corresponds to the class of problems that are realistically solvable on a computer (Sipser 1997). NP is the class of languages that can be solved in polynomial time on a nondeterministic Turing machine (Sipser 1997).
A decision problem B is NP-complete if it satisfies two conditions:

I. B is in NP

II. Every A in NP is polynomial time reducible to B (Sipser 1997).

If B is NP-complete and $B \in P$, then $P = NP$. However, it is widely assumed that $P \subseteq NP$.

PSPACE is the class of decision problems that are solvable in polynomial space on a deterministic Turing machine (Sipser 1997). A decision problem L belongs to the nondeterministic counterpart, NPSPACE if and only if there exists some nondeterministic Turing machine M that decides L in polynomial space (Sipser 1997; Rich 2008). However, Savitch's theorem shows that deterministic machines can simulate non-deterministic machines that use f (n) space by using a small amount of space $f^2(n)$

because the square of any polynomial is still a polynomial NPSPACE = PSPACE. For time complexity, such simulation seems to require an exponential increase in time (Sipser 1997).

A decision problem B is PSPACE-complete if it satisfies two conditions:

I. B is in PSPACE

II. Every A in PSPACE is polynomial time reducible to B (Sipser 1997).

If B merely satisfies condition 2, we say that it is PSPACE-hard (Sipser 1997). Complete problems are the most difficult problems in a complexity class because any other problem in the class is polynomial reducible to them. The completeness property of decision problems can be verified by showing the interrelation among various problems with respect to their difficulty. The major technique used for demonstrating that two problems are related is that of "reducing" one to the other by giving a constructive transformation that maps any instance of the first problem into an equivalent instance of the second in polynomial time (Garey and Johnson 1979). Such a transformation provides the means for converting any algorithm that solves the second problem into a corresponding algorithm for solving the first problem. Having a polynomial time reduction from one problem to the other ensures that any polynomial time algorithm for the second problem can be converted into a corresponding polynomial time algorithm for the first problem. If any PSPACE-complete language is in NP, then all of them are in NP and NP = PSPACE. Similarly, if any PSPACE-complete language is in P, then all of them are in P and P = NP = PSPACE. However, it is assumed that both subset relationships are proper (i.e., that $P \neq NP \neq PSPACE$ hence, $P \subseteq NP \subseteq PSPACE$) (Rich 2008).

2.3.1 The State Verification time Complexity (Big O notation)

The worst case time-complexity of the Algorithm (found in Appendix B) for the generation of UIO sequences is O $(n^2 (d_{max})^{2n2+2})$ where d_{max} is the largest out-degree of any state (or the largest number of outgoing edges from any state), and n is the number of states in the machine (Sabnani and Dahbura 1988, 285-297). Sabnani and Dahbura demonstrated the proof of this theorem (Sabnani and Dahbura 1988, 285-297).

As noted in Chapter 3, a rooted tree is called a full m-ary tree if every internal vertex has exactly m children. All FSMs considered in this study, are assumed to be completely specified; thus, the out-degree of every internal vertex of the successor tree is exactly equal to the input size of the FSM. Hence, the successor tree is a full m-ary tree and d_{max} = (the input size of a FSM). Kohavi theoretically proved that the upper bound for the length of a UIO sequence is $(n-1)n^n$ where n is the number of states in the FSM (Kohavi 1978). Sabnani and Dahbura considers this upper bound to be meaningless since an FSM with n = 10 will have an upper bound of 9×10^{10}, which does not hold up for all protocols that they had examined (Sabnani and Dahbura 1988, 285-297). In the context of using a UIO sequence generation algorithm as part of the conformance testing algorithm, Sabnani and Dahbura gave the upper bound of the length of the sequence to be at most $2n^2$ (Sabnani and Dahbura 1988, 285-297). Sabnani and Dahbura, however, noted that the issue of a tight bound on the depth of the UIO successor tree (which corresponds to the length of the UIO sequence) is still open (Sabnani and Dahbura 1988, 285-297). Therefore, the algorithm for a UIO sequence (if it exists) for the state s of a FSM takes $O(nI)^{2n2+2}$, where n is the number of states in the FSM which corresponds to

the maximum number of states in each node of the successor tree. I is the size of the input set of the FSM and $2n^2$ is the depth of the successor tree.

2.4 Cyclomatic Number

McCabe's cyclomatic number is a metric used to establish the minimum number of independent paths through a program modeled as a control flow graph which is strongly connected. This metric is derived from mathematical techniques which are based on concepts of graph theory. A program control flow graph is a directed graph with unique entry and exit nodes with the assumption that each node in the graph can be reached by the entry node and each node can reach the exit node. By this assumption, a program control flow graph is a strongly connected directed graph. The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is defined as $v = e - n + 2p$ (McCabe 1983; Berge 1976). A connected component of a graph G is a maximal strongly connected subgraph of G. Because all nodes in a program control flow graph are reachable from the entry node and every node can reach the exit node, a program control flow graph has only one connected component; hence, $V(G) = e - n + 2$ (McCabe 1983). The cyclomatic number of a directed graph can also be defined in terms of the Euler formula (McCabe 1983). Formally defined, the Euler formula of a connected planar graph G with e edges v vertices and r regions in a planar representation of G is $r = e - v + 2$ (Rosen 2007). The number of regions is equal to the cyclomatic number.

CHAPTER III

RESEARCH HYPOTHESIS

In practice, Unique Input/Output (UIO) sequences are used successfully and very often as part of the conformance testing, even though finding a UIO sequence for a specific state ***s*** of a Finite State Machine (FSM) M is shown to be intractable. Using the proof by reduction technique, Lee and Yannakakis showed that the state verification decision problem is PSPACE-complete. Consequently, there must exist a threshold in the problem domain size that when exceeded would result in an exponential time solution. It would be useful to have a simple metric that identifies that point. One possible metric is the McCabe Cyclomatic Number.

If the State Verification FSM problem is PSPACE-complete as theoretically established by Lee and Yannakakis then the number of states in the FSM, the size of the input set and the transition types such as reflexive transitions, are factors of the FSM that contribute directly to the intractable nature of the problem. Lee and Yannakakis did not empirically substantiate their proof; hence, we shall provide empirical evidence to corroborate the findings of Lee and Yannakakis. Providing empirical data to a theoretical proof supports the findings of the proof. Assuming that an empirical analysis verifies the correctness of this proof, at some point, the time to generate the UIO sequences becomes exponential.

In the following sections, we shall highlight Lee and Yannakakis proof and explain why we believe the above hypothesis to be true and how it is possible to substantiate their findings. Also, we will provide an explanation of why McCabe’s cyclomatic number is a

good candidate to identify the point at which the solution to the state verification problem becomes exponential.

3.1 Complexity of the State Verification Problem

Lee and Yannakakis provide a proof that for a given machine M, it is PSPACE-complete to determine if a specific given state *s* of M have a UIO sequence, if all states of M have UIO sequences or if any state of M has a UIO sequence. They showed that this holds even in the case of machines with binary input and output alphabets (Lee and Yannakakis 1994, 306-320). The structure of their proof is as follows:

- They first showed membership in the PSPACE class by reducing an instance of the state verification problem to an equivalent instance of the decision problem, reachability in an exponentially large graph.
- Next, they showed that the problem of determining whether a specific state *s* of a given machine M has a UIO sequence is PSPACE-hard. They achieved this by reduction from an instance of the “Finite State Automata Intersection” problem known to be PSPACE–complete into an equivalent instance of the state-verification decision problem (Lee and Yannakakis 1994, 306-320; Broy and others 2005).

PSPACE-Complete problems are usually different from problems known to be NP-Complete. One rich source of PSPACE-Complete problems has been the area of combinatorial games. Outside the world of games, PSPACE-Complete problems also appear in areas associated with automata, programming, and languages, where they often are restricted versions of problems already known to be intractable or un-decidable (Garey and Johnson 1979). PSPACE-complete problems are more intractable than NP-

complete problems; it is widely believed that such problems do not have polynomial time algorithms, even when allowing for non-determinism (Broy and others 2005).

Intuitively, as the number of elements increases in PSPACE-complete problems the expectation is that the solution time will grow exponentially. It is assumed that the elements of an FSM that are likely to affect the solution time for the derivation of UIO sequences for a particular state in an FSM are the number of states, size of the input set, and possibly the transition types. In a fully specified FSM, the size of the input set establishes the number of transitions, based on the transition function. Subsequently, with systematic analysis of the dataset between the time requirement of UIO sequence generation and the number of states / input set size, the coefficient of determination (R^2) is expected to be ≥ 0.80 with a predictive fit to an exponential trend line. The coefficient of determination (R^2) is defined as the ratio of the explained variation in the dependent variable to the total variation (Allen 1990). A coefficient of determination (R^2) of 0.8 for scientific or engineering problems is considered evidence that the dependent values (y) can be predicted from the independent values (x) using the regression line (Allen 1990)

Further explanation of why we think the above hypothesis to be true, we would look at the time complexity function of the UIO sequence generation algorithm. The time complexity function of an algorithm expresses the amount of time needed by the algorithm to solve any possible problem instance as a function of the problem input size. A polynomial time algorithm is defined as an algorithm whose time complexity function is O (p (n)) for some polynomial function p and input length n (Garey and Johnson 1979). The time complexity function of the UIO sequence generation algorithm is $O(nI)^{2n2 + 2}$ as

specified in the background. Thus, the UIO sequence generation algorithm is not a polynomial time algorithm but rather it is an exponential time algorithm.

Given that Lee is correct and the solution to the state verification problem is PSPACE-complete, we introduce another very interesting question. Does the characteristic of the "pieces" affect the solution time of the problem? In an FSM, there are two major pieces: states and transitions. A transition can originate in one state, terminate in another state; or they can originate and terminate in the same state. Transitions originating and terminating in the same state are known as reflexive transitions. Reflexive transitions in an FSM cause it to have an arbitrarily high number of potential paths. These reflexive transitions can cause difficulty when trying to establish a path through a state machine because the number of times to traverse a reflexive transition is not limited. It is assumed that this would influence the time it takes to generate the UIO sequences. In the data analysis of FSMs with reflexive transitions and FSMs with no reflexive transitions, we expect to observe a steeper exponential trend line for FSMs with reflexive transitions than for FSMs with no reflexive transitions.

In addition to the investigation of the hypothesis, it would be expedient to explore the possibility of using McCabe's Cyclomatic Number to predict the performance of the algorithm generating the UIO sequences. This can be achieved by deriving a metric for determining the point in the trend of the solution time of the state verification problem in which it becomes exponential.

3.2 Cyclomatic Number as a Predictive Metric

In PSPACE-complete problems, the execution time of the algorithmic solution grows exponentially with the number of pieces. There are no known methods for predicting in advance, where an algorithm's performance becomes exponential. It is convenient for Software Engineers to have a metric predicting this threshold. McCabe's Cyclomatic Number, as previously discussed, is a metric used to determine the number of paths through a control graph. In other words, it attempts to identify the minimum number of independent paths in a control graph representing a program. This research is investigating the performance of the UIO sequence generation algorithm and since McCabe's Cyclomatic Number deals with the various components of a state machine (states and transitions), we will explore the suitability of the McCabe's Cyclomatic Number for predicting the performance of the algorithm generating the UIO sequence.

Directed graphs can be used to model a program control flow graph and FSM; hence, graph theory techniques used in analyzing a program control flow graph can also be applied to FSMs. We will assume that McCabe's metric can be used to derive a technique for predicting the point in which the solution time to the state verification problem becomes exponential. McCabe's cyclomatic number is used to determine the number of basic paths in a program, which in turn is used to limit the size of a program module. McCabe's recommends an upper limit of V (G) = 10 when defining program modules. Thus, by employing McCabe's cyclomatic number as a possible component in the derivation of the anticipated metric, the attempt is to specify the complexity of an FSM in terms of its cyclomatic number and establish an upper limit. If the PSPACE-complete characteristic is caused by the number and interconnection of the parts, then it

should be possible to demonstrate a relationship between the McCabe Cyclomatic Number and the time to generate the UIO sequence. In the data analysis between these two variables, we expect to observe a strong correlation and a point in the trend line in which the solution time becomes exponential. This threshold can be used as a predictor of the complexity of the FSM with respect to UIO sequence generation.

CHAPTER IV

EXPERIMENTATION

4.1 Overview

The primary objectives of this experiment are to demonstrate the exponential growth rate of the solution time of the state verification problem as the number of “pieces” (such as the number of states and input set size) increases. Another objective is to investigate the effect of some FSM “pieces” characteristic (such as the reflexive nature of the FSM transitions) on the solution time of the problem. A third objective is to identify the point where the exponential trend begins. Using the observed relationship between the cyclomatic number and the state verification solution time a metric can be derived for determining this point. To achieve the above objectives, the structure of the experiment conducted is discussed below.

4.2 Experiment Description

In this study, five experiments are designed and each experiment evaluates the performance of the UIO sequence generation algorithm against some characteristics of the FSM. For each experiment, a sample set of 10 randomly generated FSMs are employed and the time it takes to generate the UIO sequence for each of the FSMs in the sample set is measured and recorded. We would demonstrate how the individual and collective components of an FSM (transitions, number of inputs and states) affect the performance of the UIO sequence generation algorithm respectivelyThe first experiment evaluates the effect of a combination of components of the FSM on the UIO sequence

generation algorithm performance. To achieve this, the association between the solution time of the UIO sequence generation and the controlled collective components of the FSM is demonstrated. Each FSM in the sample set has features that are different from other FSMs in the set. The number of states and input of each FSM in the sample set is varied between the intervals 10 – 100. For example, an FSM may have 10 states and 10 inputs while another FSM in the same sample set has 20 states and 20 inputs. All FSMs used in this experiment are fully specified and contain reflexive transitions. Fully specified FSMs imply that the number of edges increases proportionally to the number of states and inputs; hence, an FSM with 20 states and 20 inputs would have 400 transitions.

The second experiment assesses the effect of the number of states of an FSM on the performance of the UIO sequence generation algorithm. All FSMs in the sample set are fully specified and contain reflexive transitions. To demonstrate the effect of the number of states on the UIO-sequence generation algorithm performance, all components of the FSM are kept constant while the number of states is varied for each FSM in the sample set. In this experiment, the cardinality of the input set is fixed at 10, while the number of states is varied between the intervals 10 – 100. Keeping the number of inputs constant for fully specified FSMs implies that the number of transitions would increase in proportion to the number of states.

The third experiment evaluates the effect of the number of inputs of an FSM on the performance of the UIO sequence generation algorithm. All FSMs in the sample set are fully specified and contain reflexive transitions. The number of states of each FSM in the sample set is kept constant while the number of inputs is varied between the intervals 10 -

100. The number of transitions for each FSM varies in direct proportion to the number of inputs because the FSMs considered in this experiment are fully specified.

The fourth experiment evaluates the effect of the number of transitions and transition types of an FSM on the performance of the UIO sequence generation algorithm. In this experiment, two sample sets were employed. One sample set contains partially specified FSMs with reflexive transitions while the other contains partially specified FSMs with no reflexive transitions. Each FSM in both sample sets has ten (10) states and fifteen (15) inputs while the number of transitions is controlled between the intervals 10 – 100. To control the number of transitions, the FSMs considered have to be partially specified with the number of transitions out of a state less than or equal to 15.

The fifth experiment investigates the suitability of the cyclomatic number as a viable component to use in the derivation of the metric that identifies the point where the exponential trend begins. The FSMs in the sample set are partially specified and contain reflexive transitions. Partially specified FSMs enable the number of states to be kept constant while the cyclomatic number is controlled. Each of the FSMs in the sample set has ten (10) states and their respective cyclomatic number is varied between the intervals 10 – 100. The number of transitions is dependent on the number of states and Cyclomatic Number.

To implement the experiments, two programs are constructed. The first program randomly generates FSMs represented as strongly connected directed graphs with unique entry and exit nodes. The generated directed graphs are represented in tabular forms

which are input parameters for the second program. The second program generates the successor tree for the UIO sequence generation and measures the execution time.

Table 1 and Figure 6 illustrate a randomly generated fully specified FSM with reflexive transitions in tabular form and state chart diagram respectively. Table 2 represents the corresponding set of UIO sequences generated for each of the states in the FSM shown in Table 1. The UIO sequence is an input sequence, which is a unique signature for a state in the FSM. The output sequence generated for this input sequence for a particular state is different from the output sequences generated when this input sequence is passed through other states in the FSM. From Table 2, it can be observed that the input sequence <1, 1> generates the output sequence <1, 0>, <1, 1> and <0, 0> for the states s0, s1 and s3 respectively which are distinct from other output sequences. States s2 and s4 generates the same output sequence <0, 1> when the input sequence <1, 1> is passed through them. Hence <1, 1> is a UIO sequence for states s0, s1 and s3.

Table 1 Randomly generated fully specified FSM with reflexive transitions

Input	Output	Origin	Destination
0	1	s2	s1
0	1	s1	s2
1	1	s1	s0
0	1	s0	s0
1	0	s4	s1
0	0	s4	s2
1	0	s2	s1
1	1	s0	s3
1	0	s3	s3
0	1	s3	s2

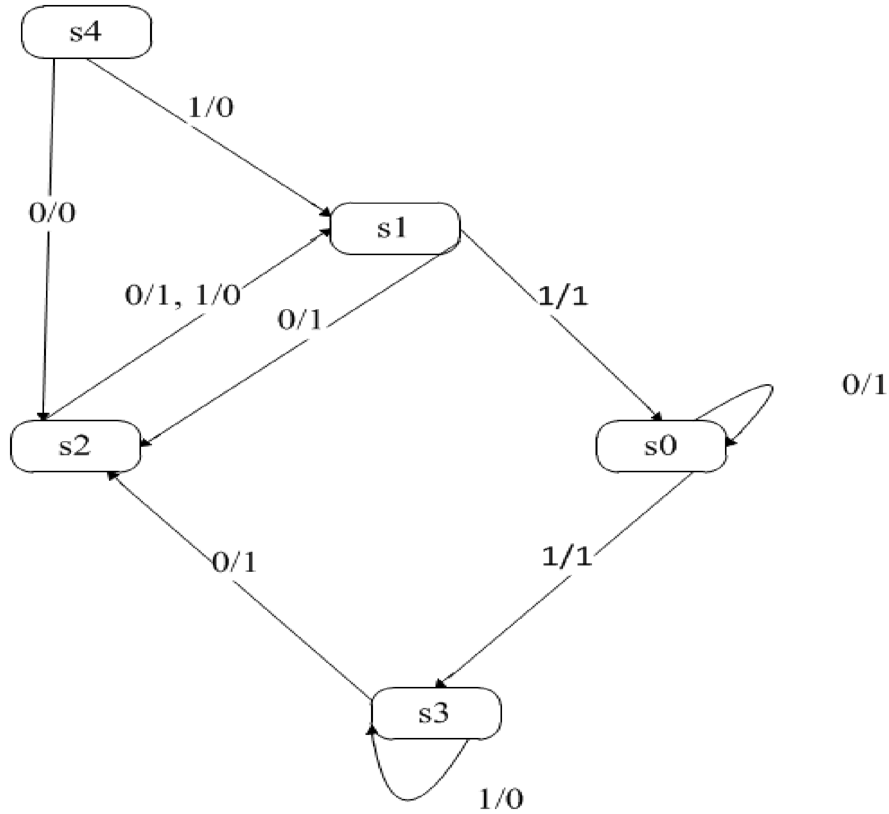

Figure 8 State Chart diagram for Table 1

Table 2 Set of UIO sequences generated for FSM in Table 1

State	Sequence
s0	<1/1, 1/0>
s1	<1/1, 1/1>
s2	<0/1, 1/1, 1/1>
s3	<1/0, 1/0>
s4	<0/0>

Table 3 and Figure 7 illustrate a randomly generated fully specified FSM with no reflexive transitions in tabular form and state chart diagram respectively. Table 4 represents the corresponding set of UIO sequences generated for each of the states in the FSM shown in Table 3.

Table 3 A randomly generated fully specified FSM with no reflexive transitions

Input	Output	Origin	Destination
1	0	s1	s0
0	0	s1	s3
0	2	s4	s1
0	2	s0	s1
1	1	s3	s4
1	0	s0	s1
1	0	s2	s0
0	2	s3	s1
1	2	s4	s2
0	2	s2	s0

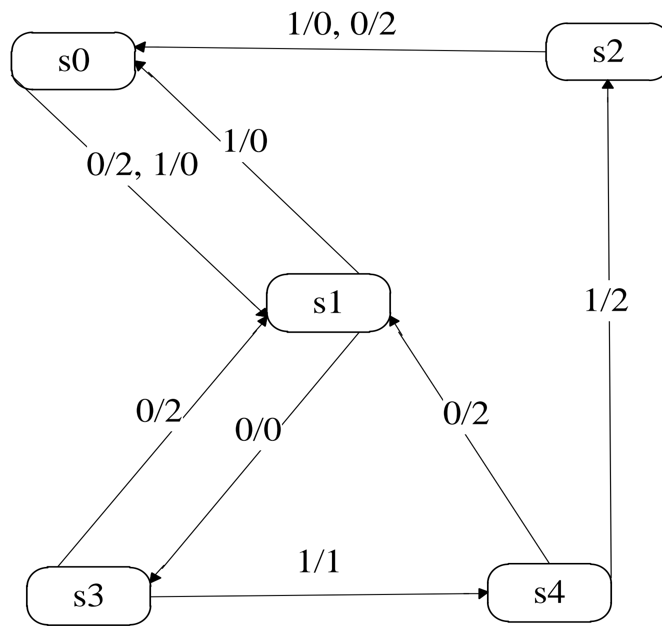

Figure 9 State Chart diagram for Table 3

Table 4 Set of UIO sequences generated for FSM in Table 3

State	Sequence
s0	<1/0, 0/0>
s1	<0/0>
s2	<0/2, 0/2>
s3	<1/1>
s4	<1/2>

To ensure that the underlying graph of each randomly generated directed graph is connected, Warshall's algorithm is applied. Warshall's Algorithm establishes reachability between any two nodes x and y of a graph. The program also checks that each directed graph generated is strongly connected by ensuring that each node can be reached by the unique entry node and each node can reach the unique exit node. The second program constructs a successor tree recursively in a breadth first search fashion, as described in chapter 2, for each state of a randomly generated FSM. Using the generated successor trees for an FSM, produces a set of UIO sequences. It also calculates the time used in the construction of the successor trees and the derivation of the UIO sequences.

4.3 Experiment Execution

In this research, all programs were executed on Windows platform. To collate the raw data needed for relevant analysis, the first program accepted the following input parameters: the number of states, the input set size, and the output set size and the cyclomatic number to produce randomly generated directed graphs as output. Source code for this program is available in Appendix A. The output obtained from the first program was used as an argument to the second program to generate sets of UIO sequences by first constructing a successor tree for each state of the FSM. Source code for this program is available in Appendices B and C. Standard system time was used to calculate the time it took to construct a successor tree but this proved to be inadequate because the standard clock resolution used in practice is too short. A Windows API high-resolution clock was employed to obtain time with the accuracy of microseconds; and this was used to compute the time it took to construct a successor tree and the

corresponding set of UIO sequences. Source code for this function is available in Appendix D. To reduce the influence of noise in the computed execution time, each execution time in the data set generated, represents the average of the execution time for the generation of five successor trees for a state. As specified in chapter 2, the successor tree algorithm requires that for partially specified FSMs, pseudo transitions are generated for transitions not specified for a given input to a state. Though this input is not specified in the transition function, it is part of the generated UIO sequence. In the experiments conducted, the input events that generated these pseudo transitions and null outputs constitute part of the UIO sequence derived. Most FSMs used in practice are partially specified; hence, using sequence driven test for FSM validation may be inadequate since these pseudo transitions do not exist.

4.4 Experiment Result

This section contains statistical data describing the results of the various experiments carried out in this study. The graphs below describe the findings of the various experiments conducted which is focused on the relationship of the UIO sequence generation algorithm performance and other measures of the FSM such as the number of states, transitions and the input set size. Linear regression was used to model the association between the execution time of the UIO sequence algorithm and the independent components of the FSM.

The set of data for Figure 10 is based on the time-complexity function O (n (I)$^{n+2}$) of the UIO sequence generation algorithm where n is the number of states in the FSM and I is the input size. Figure 9 illustrates the correlation expected between the parts of the FSM and the solution time for the UIO sequence generation. From the scatterplot, we observe a perfect fit to the exponential trend line.

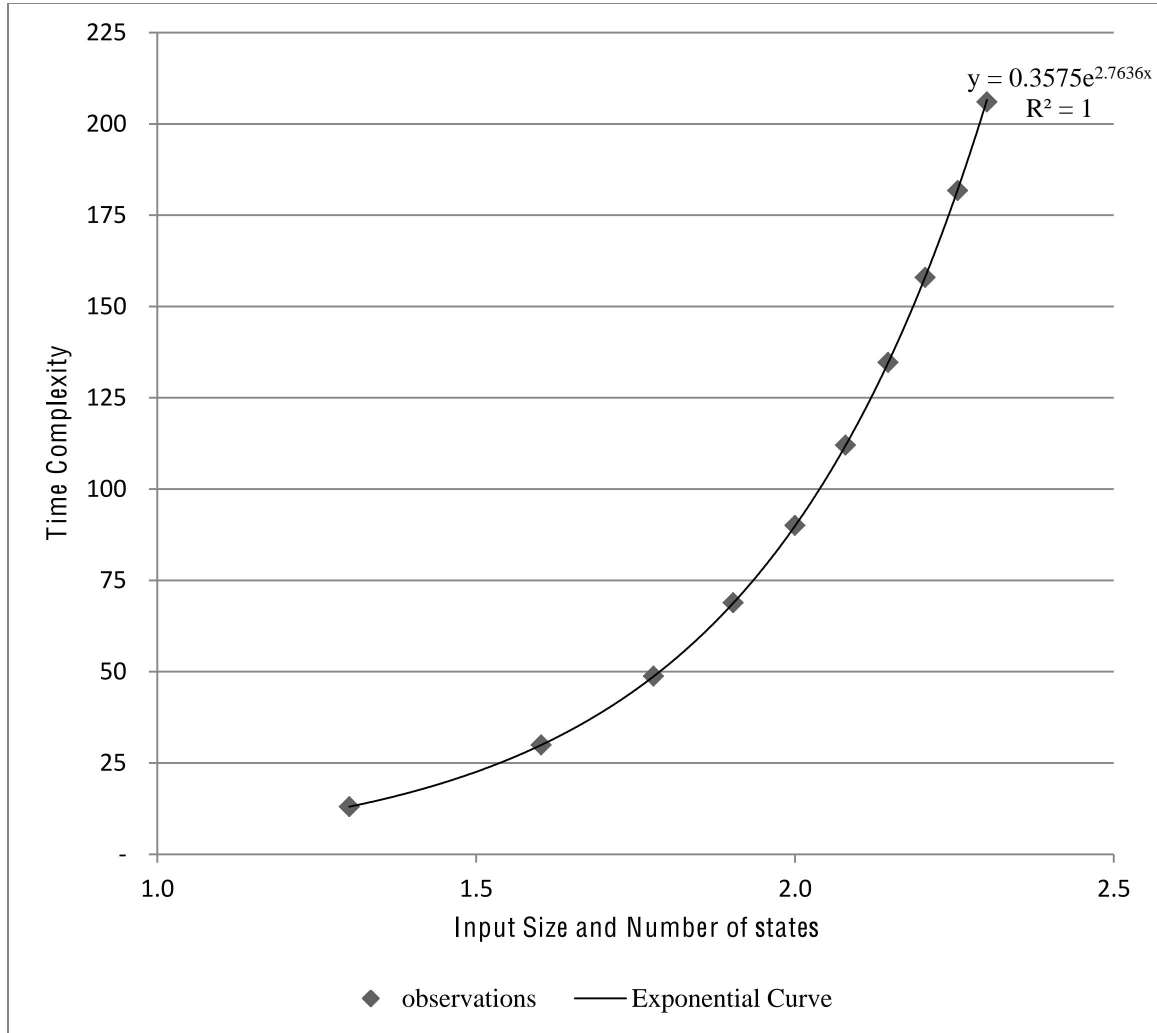

Figure 10 Log-Log Graph of Predicted Execution Time for UIO sequence

Figure 11 illustrates the relationship between the total FSM elements and the UIO sequence generation execution time in microseconds for 10 observations. Each data point in the scatter plot graph represents the total independent variables (input size,

number of states and transition). The scatter plot demonstrates a positive correlation (r) of 0.98 between the total independent variables and the dependent variable (the UIO sequence generation execution time). The R^2 values of 0.99 and 0.92, for the polynomial and exponential regression lines respectively, show that both regression lines have predictive fits. However, the polynomial regression line shows a stronger fit.

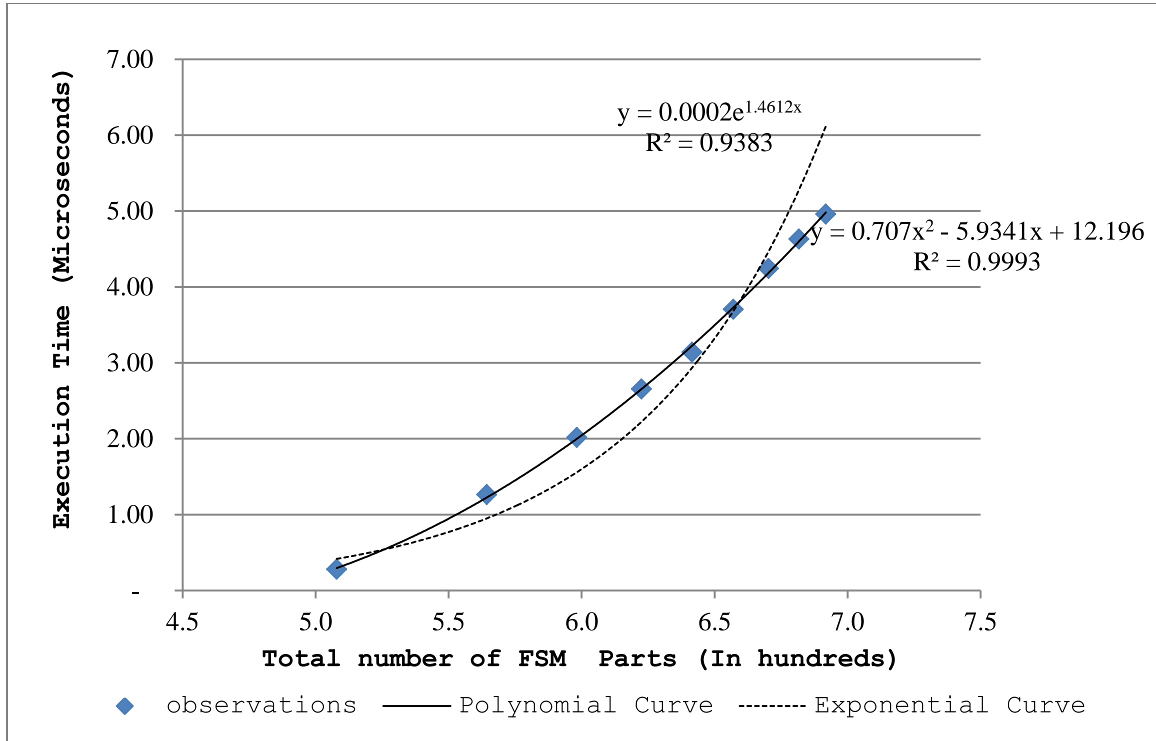

Figure 11 Log-Log Graph for Observed Execution Time for UIO Sequence.

Figure 12 illustrates the association between the number of states in the FSM and execution time (in Microseconds) for 10 observations. The data points in the scatter plot represent controlled increase for both the number of states and edges because the FSMs used for this experiment are completely specified. The scatterplot clearly indicates that there is a positive association between the number of states and execution time. It has a coefficient correlation (r) value of 0.98. Having coefficient of determination (R^2) values > 0.8 for both the polynomial and exponential trend line indicates that the association is

predictive. The polynomial and exponential regression lines both have predictive fits, however, the R^2 value of 0.9927 shows that the polynomial regression line has a stronger fit of the two. This indicates that the correlation between these two variables is polynomial, though it shows strong exponential characteristics.

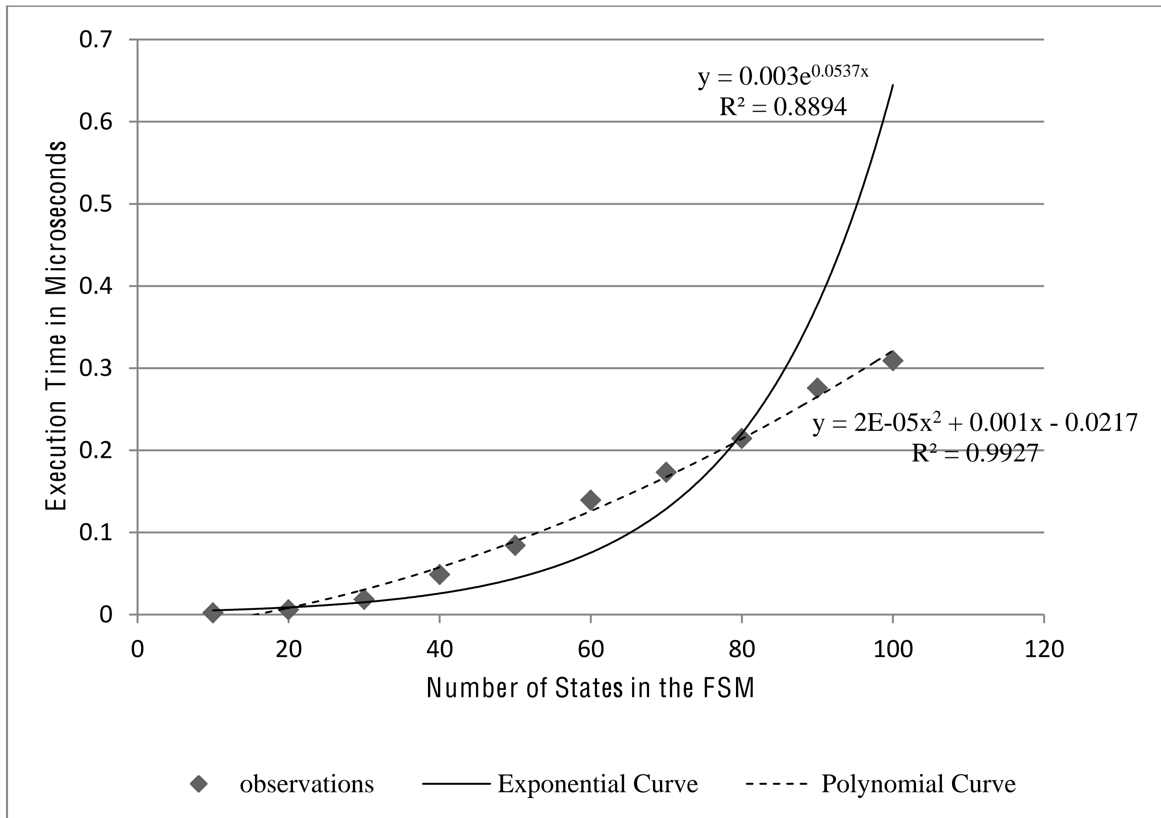

Figure 12 Observed Execution Time for UIO sequence.

Figure 13 illustrates the relationship between the input size of the FSM and the UIO sequence generation execution time in microseconds for 10 observations. The scatterplot graph indicates that there is a positive correlation (r) of 0.98 between the variables. This relationship is also predictive because its R^2 values are greater than the desired value of 0.8 for both the exponential and polynomial trend lines. The solution time of the UIO-sequence generation algorithm increases as the input size increases and the solution time

increase shows strong polynomial and exponential trends. However, the R^2 value of 0.9917 shows that the polynomial regression line has a stronger fit and this indicates that the correlation between these variables is polynomial, though it displays exponential characteristics.

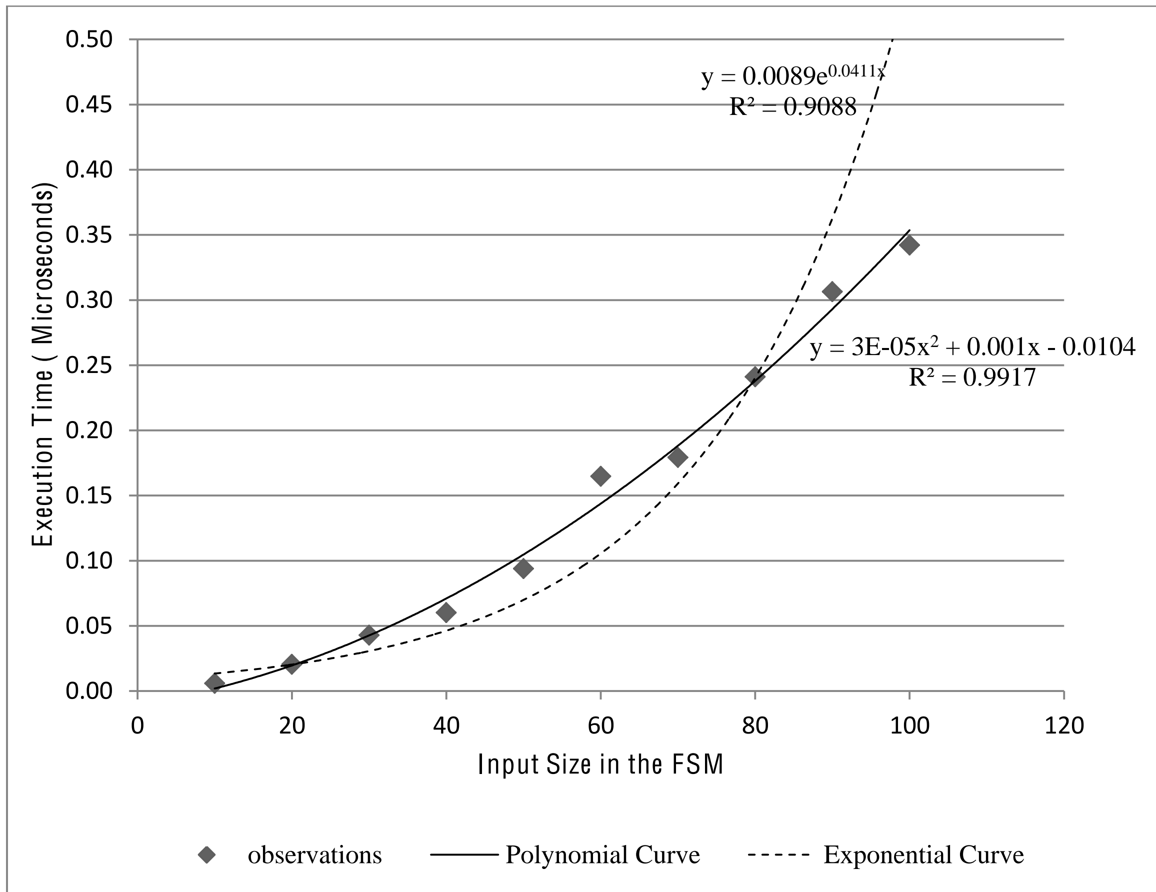

Figure 13 Observed Execution Time for UIO sequence

Figure 14 demonstrates the correlation between the transitions of FSM (number and types) and the solution time of the UIO sequence generation algorithm. From the scatter plot we observe a negative correlation between the variables. This implies that as the number of transitions increases, the solution time decreases. By keeping the number of states and input for each FSM considered constant while increasing the number of transitions, the probability of finding UIO sequences for a state in the FSM increases.

Thus, the number of transitions influences the solution time for the UIO sequence generation inversely. The R^2 values of the regression lines indicate that the transition type of the FSMs is inconsequential to the time required in generating the UIO sequences. The R^2 values for both the reflexive and non-reflexive transitions are less than 0.8, which implies that the regression lines for either transition type is effective in predicting the UIO-sequence generation solution time.

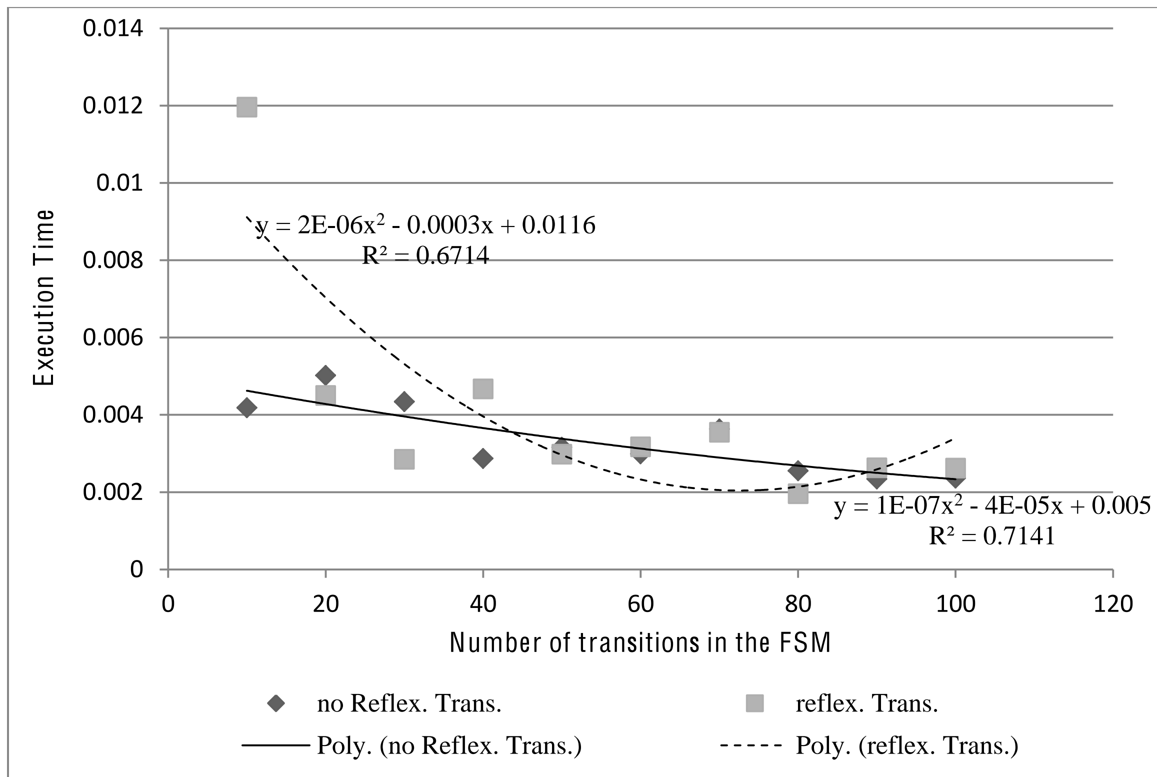

Figure 14 Transition Types on the performance of the UIO sequence generation

Table 5 illustrates a randomly generated partially specified FSM. Table 6 represents the transition function for the FSM. Each cell in Table 6 depicts the next state and output when an input represented on the column heading enters a state specified in the row heading. Table 7 shows the set of UIO sequences generated for each of the states in the FSM. From the transition function, we can observe that no transitions are specified when

there is an input of 0 to the states s1 and s3 respectively. However, the UIO sequence generated for state 1 is 0, which is a transition that does not exist in the FSM represented in table 5. The classic algorithm used in the UIO sequence generation assumes all FSMs to be fully specified. Where the FSMs are partially specified, pseudo transitions are generated, which originate and terminate in the same state with a null output. From Table 7 we can observe that the output sequence generated as a consequence of the UIO sequence is null.

Table 5 A partially specified FSM

Input	Output	Origin	Destination
0	1	s2	s2
1	1	s1	s2
1	1	s3	s1
0	0	s4	s3
0	0	s0	s4
0	1	s3	s0
1	0	s2	s2
1	0	s0	s2

Table 6 Transition Function for FSM in Table 5

	0	1
s0	s2/1; s4/0	s2/0
s1		s2/1
s2		s2/0
s3	s0/1	s1/1
s4	s3/0	

Table 7 Set of UIO sequences generated for FSM in Table 5

States	Sequences
s0	< 0/0, 0/0>
s1	<0/null>
s2	<0/1, 0/1>
s3	<0/1, 0/0>
s4	<1/null>

Figure 15 illustrates the relationship between the Cyclomatic number of the FSM and the UIO sequence generation execution time in microseconds for 10 observations. It has a negative correlation coefficient (r) of -0.75. This relationship is not predictive because the R^2 value for both the polynomial and exponential trend lines are below the desired R^2 value of 0.8, which is the value, used for scientific or engineering problems.

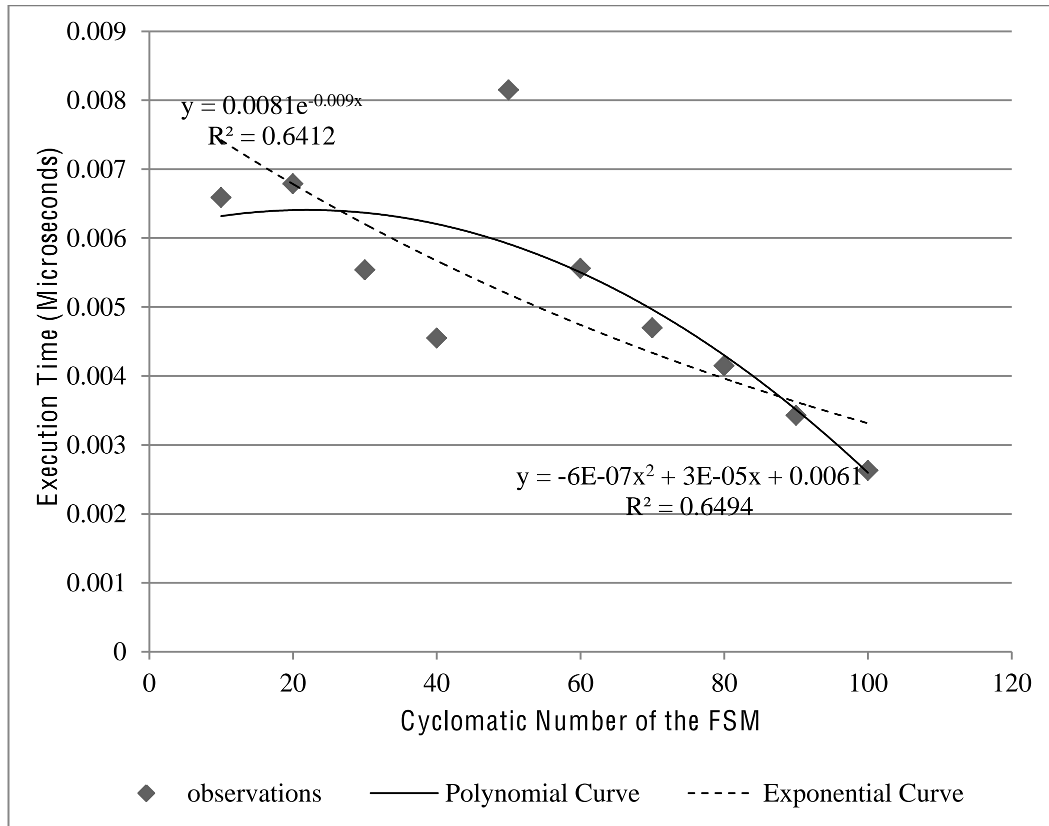

Figure 15 Observed Execution Time for UIO sequence generation.

CHAPTER V

CONCLUSION AND FUTURE RESEARCH

From the findings of the empirical experiment illustrated in Figures 11 and 12, the number of states and inputs are elements of a Finite State Machine (FSM) directly affecting the performance of the Unique Input / Output (UIO) sequence generation algorithm. In a fully specified FSM, the cardinality of the input set (I) determines the number of transitions originating from each state. The number of transitions affects the performance inversely in a non-fully specified FSM, and the transition types appear to have no significant effect on the performance; thus, they are inconsequential as factors in the time required to generate a UIO sequence algorithm, as illustrated in Figure 13.

Figure 10 illustrates that the performance of the UIO sequence generation algorithm is polynomial for finite inputs and states. Lee proved that the state verification problem is PSPACE-complete by reducing an instance of the finite-state automata intersection problem known to be PSPACE-complete to an instance of the state verification problem. However, Hopcroft and Ullman proved that the Finite State Automata Intersection problem is PSPACE-complete but solvable in polynomial time for a finite number of inputs (Garey 1979). This is consistent with the findings modeled in Figure 10, where the performance of the UIO sequence generation algorithm trends exponential but exhibits a stronger polynomial trend for a finite number of inputs of the FSM. The state verification testing problem is PSPACE and not PSPACE-complete. It is PSPACE because as the number of states and inputs of a FSM increases, the solution time of the

UIO sequence generation algorithm increases in a polynomial trend where the number of inputs of a FSM are finite. This indicates that Lee's theoretical proof, though accurate, is not precise. It is accurate because the algorithm exhibits an exponential trend, but it is not strictly an exponential time algorithm. Lee's proof failed to establish the distinction in the solvability of the problem for finite and infinite number of inputs and states of the FSM.

In the classic UIO sequence generation algorithm, it is assumed that the FSM to be tested is fully specified. To generate UIO sequences for a partially specified FSM, it must first be expanded to a fully specified FSM. This entails generating pseudo transitions (these are self loops or reflexive transitions that originate and terminate in the same state) with null outputs for inputs in a state that have no transition specification in the transition function of the FSM. Table 7 illustrates that the pseudo transitions constitute part of the UIO sequence path in a partially specified FSM. These pseudo transitions do not occur in physical systems and therefore are not possible to test. Thus, sequences that contain pseudo transitions cannot be used to accurately test a system with state behavior.

In practice, Mealy and Moore Finite State Machines (FSMs) are not powerful enough to model complex software systems succinctly anymore. To model complex software systems, FSMs are extended to include variables, which are known as Extended Finite State Machine (EFSM). To test an EFSM using sequences, it must first be expanded to an ordinary FSM. This is possible only if each variable in the variable set has a finite number of values, then each combination of a state and variable value constitute the states of the expanded FSM. However, this equivalent FSM may be too cumbersome to construct in practice. For an EFSM with infinite variable values such as real numbers, it

becomes impossible to expand it to a fully specified FSM; thus the use of sequences to test an EFSM becomes infeasible. Future research to develop new techniques for testing EFSMs is expedient.

Figure 14 indicates that the cyclomatic number of an FSM cannot be used in deriving a metric to predict the time to derive a set of UIO sequences, as proposed in the hypothesis. This is not a complete surprise since the intent of the cycolmatic number is to establish the minimum number of paths through a control graph (McCabe 1983).

In conformance testing, a test sequence must traverse each state and each state transition, of a FSM and must check that each state has a unique signature called the UIO sequence. This characteristic of the conformance testing procedure shows a strong correlation to the definition of the cyclomatic number; hence, the cyclomatic number could be a more viable component in predicting the complexity of an FSM where conformance testing is the focus.

The following further studies are necessary to fully substantiate some of the conclusions:

- An Empirical investigation into the suitability of employing McCabe's Cyclomatic Number in deriving a metric for predicting the performance of the checking sequence generation algorithm
- An Empirical investigation into the inadequacy of using sequences to test EFSMs. Develop techniques other than sequences that can be used to adequately test EFSMs.

- An Empirical investigation into the derivation of conformance sequences using UIO sequences. As with this study, it would be interesting to evaluate the effects of transition types in FSMs that are not fully specified.

APPENDIX A

UIO Sequence Generation Analysis Program

Main Analysis Program

```
#include "hdrFSMgeneratorStatistics_vs6.h"
#include "hdrUserInterface_vs6.h"
#include "hdrCStopWatch_vs6.h"
#include "hdrUIOsuccessorTree_vs6.h"
#include "hdrFSMrandomGenerator_vs6.h"

int main()
{

   const int RepeatRuns = 7;

   directedGraph FSM;
   int numEdges,
       numVertices,
       cyclomaticNum,
       IncrementalValue,
       intervalValue,
       lowerBound,
       upperBound,
       Input_Size,
        Output_Size,
       transitionType,
       temp,
       recordUIO_Seq , //Holds the length of the UIO Sequence
       Rank = 0,
```

```
     ReflexTranxExists,
    Rindex = 0;  /*Index for the vectorRandomlySelected_numVertices vector*/

double runTime_inSeconds,
       totalTime;

bool duplicate;

string strOutput;

vector<int*> AdjecencyMatrix;
vector<possibleEdges> AllPossibleEdges;
vector<string> OutputSet;
vector<char*> inputSet;
vector<int*> OutputTransitionTable;
vector<int*> StateTransitionTable;
vector<double> timerVector;
vector<int> vectorRandomlySelected_values;

char uniqueTerminalnode,
     response,
     cycloType,
     stateType,
     edgeType,
     inputType,
     writeToFile = 'y';

CStopWatch timer;
userInterface dInterface;

//To randomly select numbers
unsigned seed = time(0);
srand(seed);
```

```
ofstream Data_Gathering("_RunTimeUIO.txt", ios::out|ios::app);
ofstream Statistics_Analysis("_UIORunTime_Statistics_Analysis.txt",
                              ios::out|ios::app);
ofstream fout("Data.txt", ios::out|ios::app);
ofstream _FSMDetails ("FSM.txt", ios::out|ios::app);

//Display the User Interface
dInterface.displayInterface();

//Get the Output Set
do
{
   cin.clear();
   cin.ignore(numeric_limits<streamsize>::max(),'\n') ;
   cout << "\n\tEnter the output set size [MUST be > 0]: ";
   cin >> Output_Size;

}while(!cin.good()|| Output_Size < 1);

cout << "\n\tEnter the output symbols : ";
for(int i = 0; i < Output_Size; i++)
{
   duplicate = false;
   cin >> strOutput;
   //Check for duplicate entry
   for (int y= 0; y < i; y++)
       if (OutputSet[y] == strOutput)
       {
          cout << "Duplicate Entry...re-enter: ";
          duplicate = true;
          i--;
          break;
       }
```

```
   if(!duplicate)
      OutputSet.push_back(strOutput);
}

/*Get the cyclomatic type, number or interval*/
cycloType = dInterface.getCylcomaticStatus();
if (cycloType == 'i' || cycloType == 'I')
{
    lowerBound = dInterface.getCyclomaticLowerBound();
    upperBound = dInterface.getCyclomaticUpperBound();
}
else if (cycloType == 'c' || cycloType == 'C')
    cyclomaticNum = dInterface.getcyclomaticNumber();

/*Get the state type, number or interval*/
stateType = dInterface.getStateStatus ();
if (stateType == 'i' || stateType == 'I')
{
    lowerBound = dInterface.getStateLowerBound();
    upperBound = dInterface.getStateUpperBound();
}
else if (stateType == 'c' || stateType == 'C')
    numVertices = dInterface.getNumberOfState();

/*Get the edge type, number or interval*/
edgeType = dInterface.getEdgeStatus ();
if (edgeType == 'i' || edgeType == 'I')
{
    lowerBound = dInterface.getEdgeLowerBound();
    upperBound = dInterface.getEdgeUpperBound();
}
else if (edgeType == 'c' || edgeType == 'C')
    numEdges = dInterface.getNumberOfEdge();
```

```
/*Get the input type, number or interval*/
inputType = dInterface.getInputStatus ();
if (inputType == 'i' || inputType == 'I')
{
    lowerBound = dInterface.getinputLowerBound();
    upperBound = dInterface.getinputUpperBound();
}
else if (inputType == 'c' || inputType == 'C')
    Input_Size = dInterface.getinputSize();

/*Get the incremental Value*/
dInterface.setIncrementalValue();
IncrementalValue = dInterface.getIncrementalValue();

//Helps to keep the user engaged in the processing
cout << endl << setw(14) <<  "numVertices"
    << setw(14) <<  "numEdges"
    << setw(16) << "cyclomaticNum"
    << setw(16) << "Input_Size" << endl;

for (int transitionType = 0; transitionType < 2; transitionType++ )
{
    cout << "transitionType - " << transitionType << endl;
    //Write Transition Type of FSM to File
    if (transitionType == 1)
    {
          Statistics_Analysis << "FSM with No Reflexive transition\n\n";
          _FSMDetails << "FSM with No Reflexive transition\n\n";
          Data_Gathering << "FSM with No Reflexive transition\n\n";
    }
    else
    {
          Statistics_Analysis << "FSM with Reflexive transition\n\n";
```

```
      _FSMDetails <<"FSM with Reflexive transition\n\n";
      Data_Gathering << "FSM with Reflexive transition\n\n";
}

//Print header for Statistical File
Statistics_Analysis << fixed << setprecision(5)
                   << setw(15) << "|State |"
                   << setw(15) << " Edges |"
                   << setw(15) << " cyclomatic # |"
                   << setw(10) << " Input Size |"
                   << setw(25) << " Execution Time (Median) |"
                   << setw(25) << " Execution Time (Mean) |"
                   << setw(15) << " Length of Sequence |"
                   << endl;

//Clear the vectorRandomlySelected_values vector for a new interval run
 vectorRandomlySelected_values.clear();

//reset the Rindex variable for a new interval run
Rindex = 0;

/*Run for a specified interval*/
for (intervalValue = lowerBound; intervalValue <= upperBound;
                     intervalValue += IncrementalValue)
{
    //set the cyclomaticNum variable with intervalValue
    if (cycloType == 'i' || cycloType == 'I')
       cyclomaticNum = intervalValue;

    //set the numVertices variable with intervalValue
    if (stateType == 'i' || stateType == 'I')
       numVertices = intervalValue;

    //set the numEdges variable with intervalValue
```

```
if (edgeType == 'i' || edgeType == 'I')
   numEdges = intervalValue;

//set the Input_Size variable with intervalValue
if (inputType == 'i' || inputType == 'I')
   Input_Size = intervalValue;

//Generate Random Values for the states
if(stateType == 'r' || stateType == 'R')
{

   duplicate = false;

   do
   {
       /*Randomly select the number of vertices for
       a range of numbers between the upper and lower bounds*/
       temp = rand()% upperBound;

       //Confirm that the selected number has not been previously selected
        for(int i = 0; i < vectorRandomlySelected_values.size(); i++)
        {
           duplicate = false;
           //Check for duplicate entry
           if (vectorRandomlySelected_values[i] == temp || temp == 0
              || temp == 1)
           {
              duplicate = true;
              break;
           }
        }

        if(!duplicate && (numEdges >= (temp - 1)))
           vectorRandomlySelected_values.push_back(temp);
```

```
          //Minimum number of edges = n - 1 where n is the number of vertices
     }while(duplicate || (numEdges < (temp - 1)));

     cout << "Random number\t" << Rindex
           << "\t" << temp << endl;

      //return the randomly selected value
     numVertices = vectorRandomlySelected_values[Rindex++];
   }

  //Generate Random Values for input
  if(inputType == 'r' || inputType == 'R')
  {

     duplicate = false;

     do
     {
         /*Randomly select the number of vertices for
         a range of numbers between the upper and lower bounds*/
         temp = rand()% upperBound;

         //Confirm that the selected number has not been previously selected
          for(int i = 0; i < vectorRandomlySelected_values.size(); i++)
          {
             duplicate = false;
             //Check for duplicate entry
             if (vectorRandomlySelected_values[i] == temp || temp == 0
                || temp == 1)
             {
                duplicate = true;
                break;
             }
```

```
        }

        if(!duplicate && temp != 0 && temp != 1)
           vectorRandomlySelected_values.push_back(temp);

   }while(duplicate);

   cout << "Random number\t" << Rindex
         << "\t" << temp << endl;

    //return the randomly selected value
   Input_Size = vectorRandomlySelected_values[Rindex++];
}

//Calculate the required number of edges with respect to the input size
if (edgeType == 's' || edgeType == 'S')
   numEdges =  Input_Size * numVertices;

//Calculate the required number of states with respect to the input size
if (stateType == 's' || stateType == 'S')
   numVertices =  numEdges / Input_Size;

//Compute the cyclomatic Number where the cycloType = 'k'
if (cycloType == 'k' || cycloType == 'K')
   cyclomaticNum = numEdges - numVertices + 2;

//Calculate the number of states where stateType = 'k'
if (stateType == 'k' || stateType == 'K')
  numVertices  = numEdges - cyclomaticNum + 2;

/*Calculate the required number of edges with respect to the
  cyclomatic number and number of edges*/
if (edgeType == 'k' || edgeType == 'K')
   numEdges = numVertices -2 + cyclomaticNum;
```

```
//Compute the Input_Size where the inputType = 'k'
if (inputType == 'k' || inputType == 'K')
{
   Input_Size = numEdges/numVertices;
   if (numEdges % numVertices)
      Input_Size++;
}

//Write details to the statistical analysis file
Statistics_Analysis << setw(14) <<  numVertices
                    << setw(14) << numEdges
                    << setw(14) << cyclomaticNum
                      << setw(9) << Input_Size;

//Write details to the screen
cout << setw(14) <<  numVertices
     << setw(14) <<  numEdges
     << setw(14) << cyclomaticNum
     << setw(9) << Input_Size;

//Write details to the FSM.txt file
_FSMDetails << " cyclomaticNum - " << cyclomaticNum << endl
            << " Input Size - " << Input_Size << endl
            << " Number of States - " << numVertices << endl;

 /*Clear AllPossibleEdges vectors to generate edges of different
          transition Types*/
 AllPossibleEdges.clear();

 //Generate all possible edges
 generateEdges(AllPossibleEdges, numVertices,  transitionType);

 //Reset to write UIO sequence to file or console
```

```
writeToFile = 'y';

//Reset the inputSet
inputSet.clear();

 //Reset the FSM
FSM.resetFSM();

//Reset the OutputTransitionTable vector
OutputTransitionTable.clear();

//Reset the StateTransitionTable vector
StateTransitionTable.clear();

//get the Cyclomatic number and the number of vertices
getVertex_numCyclo_num(numVertices, inputSet, Input_Size, FSM);

cout << "\tPlease Wait ..." << endl;
do
{
      /*Reset the ReflexTranxExists variable. This variable indicates
     that a reflex Transition exists for an FSM that is required to
     have a refex transition*/
    ReflexTranxExists = 0;

    //Clear the edge list and cascade to incidences on nodes
    FSM.clearEdgeList();

    //Arbitrarily select edges
    selectEdgeRandomly(FSM, numEdges, AllPossibleEdges, inputSet,
                        Input_Size, OutputSet, ReflexTranxExists,
                       intervalValue);

    //Construct an adjancency Matrix
```

```
                constructAdjecencyMatrix(FSM, numVertices, AdjecencyMatrix);

                //Using Warshall's Algorithm construct a reachability matrix from the
adjacency matrix
                generateReachabilityMatrix(AdjecencyMatrix);

            }
            /*Check if the constructed graph is an FSM, clear the graph structure where
false
              Also confirm the correct transition*/
            while(!isFSM(AdjecencyMatrix, FSM) || !((ReflexTranxExists && transitionType
== 0)
                                                                      || transitionType == 1));

            //------------State Transition Table------------
            BuildStateTransitionTable(FSM, inputSet, Input_Size, StateTransitionTable);

            //------------Output Transition Table------------
            BuildOutputTransitionTable(FSM, inputSet, Input_Size, OutputTransitionTable,
                                                  OutputSet);
            _FSMDetails << "\n------------FSM Details------------" << endl;
            FSM.printEdges(inputSet, OutputSet, _FSMDetails);
            _FSMDetails << "\n\n";

            //Re-run the build of the successor Tree
            totalTime = 0;
            cout << "\n\n\nRepeat Run...\n";
            for (int j = 0; j < RepeatRuns; j++ )
            {

                  //Write data Statistic to file
                  Data_Gathering << setw(15) << numVertices
                                   << setw(15) << cyclomaticNum
                                   << setw(15) << Input_Size;
```

```
                timer.startTimer() ;

                buildUIOSuccessorTree(FSM, OutputTransitionTable, StateTransitionTable,
                inputSet, Input_Size, OutputSet, fout, writeToFile, recordUIO_Seq);
                writeToFile = 'n'; //To prevent writing to file during several iteration
to get the runtime

                  timer.stopTimer();

                  runTime_inSeconds = timer.getElapsedTime();

                 //Insert time into the timer vector
                 timerVector.push_back(runTime_inSeconds);
                  Data_Gathering << fixed << setprecision(5) << setw(15)
                              << runTime_inSeconds;
                  Data_Gathering << endl;
            }

             Data_Gathering << "From Timer......." << endl;
             for (int j = 0; j < timerVector.size(); j++)
             {
               //Get the total Time of every re-run
               totalTime += timerVector[j];
               Data_Gathering << timerVector[j] << endl;
           }
            Data_Gathering << endl;

             //First sort the timer vector
            sort (timerVector.begin(), timerVector.end());

            //Calculate the median instead of the mean for the run time
            if (timerVector.size()% 2)
                Statistics_Analysis << setw(25)
```

```
                                << timerVector[(timerVector.size()/2)];

             else
             {
                 double i, i1;
                  i = timerVector[(timerVector.size()/2) - 1];
                  i1 = timerVector[timerVector.size()/2];
                  Statistics_Analysis << setw(25)
                                   << (i + i1)/2 ;
             }

             //Display the average time but first normalize the data by removing the
outliers
             Statistics_Analysis << setw(25)
                              << (totalTime - timerVector[0] -
                            timerVector[timerVector.size()-1])/(RepeatRuns-2);

            //Write the length of the UIO Sequence to the statistic file
            Statistics_Analysis << setw(15)
                              << recordUIO_Seq;

              //Clear the timer vector for a rerun
              timerVector.clear();
              Statistics_Analysis << endl;
        }
    }

    //Delete the char Inputs
    for (int y = 0; y < inputSet.size(); y++)
          delete inputSet[y];
    cout << "The end...\n";
    system("pause");
    return 0;
}
```

Analysis Header File

```
#ifndef hdr_FSMgenerator_H
#define hdr_FSMgenerator_H

#include "hdrFSMrandomGenerator_vs6.h"

#include<iostream>
#include<fstream>
#include<vector>
#include <locale>
#include <iomanip>
#include<ctime>
using namespace std;


void generateEdges(vector<possibleEdges> &, int, int);
void printToScreen(vector<possibleEdges> &, directedGraph &);
void selectEdgeRandomly(directedGraph &, int, vector<possibleEdges> &,
                           vector<char*> &, int &, vector<string>&, int &, int);
void getVertex_numCyclo_num(int &, vector<char*> &, int &, directedGraph &);
void constructAdjecencyMatrix(directedGraph &, int, vector<int*> &);
void printToScreen(vector<int*> &);
void generateReachabilityMatrix(vector<int*>&);
void addEdge_UpdateVertex(directedGraph &, vector<possibleEdges> &, int &,
                                                  int, int &, int &);
void printColRow(vector<int*> &);
bool isFSM(vector<int*> &, directedGraph &);
void BuildStateTransitionTable(directedGraph &, vector<char*> &, int, vector<int*> &);
void printToScreen(vector<int*> &, int, directedGraph &);
void BuildOutputTransitionTable(directedGraph &, vector<char*> &, int ,
                                                         vector<int*> &, vector<string>
&);
void printToScreen1(vector<int*> &, vector<string> &, int);
```

```
void getSpecificFSM(int &, directedGraph &, char* &, int &, vector<string> &);

void buildUIOSuccessorTree(directedGraph &, vector<int*> &, vector<int*> &,
              vector<char*> &, int, vector<string> &, ostream &, char, int&);
void constructAdjecencyMatrix(int , vector<int*> &);
int pathExist (vector<int*> &, int *, int );
void generateReachabilityMatrix_noRef(vector<int*>);
int randomlySelectValue(int &, int &, vector<int>&);

#endif
```

Analysis Source File

```
//Generate all possible edges: Store position of vertices in the nodes
void generateEdges(vector<possibleEdges> &AllPossibleEdges, int numVertices,
                                                 int transitionType)
{
   possibleEdges edges;
   switch (transitionType)
   {
   case 0:
        for (int i = 0; i < numVertices; i++ )
        {
             for (int j = 0; j < numVertices; j++)
             {
                edges.node1 = i;
                edges.node2 = j;
                AllPossibleEdges.push_back(edges);
            }
        }
        break;
   case 1:
        for (int i = 0; i < numVertices; i++ )
        {
             for (int j = 0; j < numVertices; j++)
             {
                if (i != j)
                {
                    edges.node1 = i;
                    edges.node2 = j;
                    AllPossibleEdges.push_back(edges);
                }
            }
        }
```

```
    }
}

//Arbitrarily select e = v-2+cyclomaticNum edges
void selectEdgeRandomly(directedGraph &FSM, int numEdges,
       vector<possibleEdges> &AllPossibleEdges, vector<char*> &inputSet, int &Input_Size,
          vector<string> &outputSet, int &ReflexTranxExists, int randomizeSeed)
{
   int index,
       count = 0,
       edgePOS,
       pos;    /*Edge position in the edge list*/
   vertex state;
   int input,
        output,
        maxEdgesOut,
        *vertices,
       _noEdges = 0;

   vector<int*> AdjecencyMatrix,
                   ReachabilityMatrix;

   vector<possibleEdges> EdgesToConsider;
   possibleEdges _edge;

   bool Input_Used,
        vertexExist;

   vector<int> verticesUsed;

   unsigned seed = time(0);
   srand(seed + randomizeSeed);

   vertices = new int[FSM.getMaxVertices()];
```

```
//Initialize the dynamic array to -1
for (int y = 0; y < FSM.getMaxVertices(); y++)
     vertices[y] = -1;

//Construct an adjecency matrix
constructAdjecencyMatrix(FSM.getMaxVertices(), AdjecencyMatrix);
constructAdjecencyMatrix(FSM.getMaxVertices(), ReachabilityMatrix);

if (numEdges % FSM.getMaxVertices())
     maxEdgesOut = (numEdges / FSM.getMaxVertices())+ 1;
else
     maxEdgesOut = numEdges / FSM.getMaxVertices();

while (count < numEdges)
{

   //Select an edge randomly
   index = rand()% AllPossibleEdges.size();

   state = FSM.getVertex(AllPossibleEdges[index].node1);

   /*Transition out of the state can not exceed
    numEdges/numVertices + numEdges%numVertices and every vertex must be considered*/
   if ((state.Edge_incidenceOUT.size() < maxEdgesOut &&
        verticesUsed.size() >= FSM.getMaxVertices()) ||
      (verticesUsed.size() < FSM.getMaxVertices() &&
       state.Edge_incidenceOUT.size() < (numEdges / FSM.getMaxVertices())))
   {

       //randomly select an input string
       do
       {
           Input_Used = false;
```

```
        pos = rand()% Input_Size;
        for (int k = 0; k < state.Edge_incidenceOUT.size(); k++)
        {
            if (pos == FSM.getEdge(state.Edge_incidenceOUT[k]).Input)
                Input_Used = true;
        }
    } while(Input_Used);

    input = pos; //register the input index

    //randomly select an output string
    pos = rand()%outputSet.size();
    output = pos; //register the output index

    //Add Edge to EdgesToConsider vector
    _edge.node1 = AllPossibleEdges[index].node1;
    _edge.node2 = AllPossibleEdges[index].node2;

    EdgesToConsider.push_back(_edge);

    /*Update the constructed adjecency matrix incremently from
      randomly selected edges that meets specific criteria*/
 AdjecencyMatrix[AllPossibleEdges[index].node1]
                             [AllPossibleEdges[index].node2] = 1;

    //Reset the Reachability Matrix with the AdjecencyMatrix
    for (int i = 0; i < AdjecencyMatrix.size(); i++)
           for (int t = 0; t < AdjecencyMatrix.size(); t++)
                ReachabilityMatrix[i][t] = AdjecencyMatrix[i][t];

    generateReachabilityMatrix(ReachabilityMatrix);

    //Mark the occurrence of a vertex in the vertices dynamic array
    vertices[AllPossibleEdges[index].node1] = 1;
```

```
vertices[AllPossibleEdges[index].node2] = 1;


//Verify if a path exists with the generated edges
if (pathExist (ReachabilityMatrix, vertices, FSM.getMaxVertices()))
{
     addEdge_UpdateVertex(FSM, AllPossibleEdges, count, index, input, output);

     if (AllPossibleEdges[index].node1 ==
             AllPossibleEdges[index].node2)
         ReflexTranxExists++;

    /*Each Node must have at least a transition from it or to it.
           Keep count of vertices used*/
    vertexExist = false;
    for (int k = 0; k < verticesUsed.size(); k++)
        if (AllPossibleEdges[index].node1 == verticesUsed[k])
        {
            vertexExist = true;
            break;
        }
    if (!vertexExist)
       verticesUsed.push_back(AllPossibleEdges[index].node1);

    //Check for node 2
    vertexExist = false;
    for (int k = 0; k < verticesUsed.size(); k++)
        if (AllPossibleEdges[index].node2 == verticesUsed[k])
        {
            vertexExist = true;
            break;
        }
    if (!vertexExist)
       verticesUsed.push_back(AllPossibleEdges[index].node2);
```

```
          }
          else
          {
                 /*cout << "Remove Edge, Path does not Exists!!\n";
                 for (int g = 0; g < EdgesToConsider.size(); g++)
                      cout << EdgesToConsider[g].node1 << "\t"
                            << EdgesToConsider[g].node2;
                 cout << endl << endl; */

                 EdgesToConsider.erase(EdgesToConsider.end());

                 //Reset the vertices array
                 for (int y = 0; y < FSM.getMaxVertices(); y++)
                      vertices[y] = -1;

                for (int g = 0; g < EdgesToConsider.size(); g++)
                {
                     vertices[EdgesToConsider[g].node1] = 1;
                     vertices[EdgesToConsider[g].node2] = 1;
              }
                 //Remove the edge from the Adjecency Matrix
                 AdjecencyMatrix[AllPossibleEdges[index].node1]
                                        [AllPossibleEdges[index].node2] = 0;
           }
      }
   }
}

//
void addEdge_UpdateVertex(directedGraph &FSM,
            vector<possibleEdges> &AllPossibleEdges, int &count, int index,
                                               int &input, int &output)
{
```

```
    int edgePOS =FSM.addEdge(input, output, AllPossibleEdges[index].node1,
                         AllPossibleEdges[index].node2);
    FSM.update_VertexIncidence(-1, edgePOS-1, AllPossibleEdges[index].node1);
    FSM.update_VertexIncidence(edgePOS-1, -1, AllPossibleEdges[index].node2);
    //AllPossibleEdges[index].selected = 1;
    count++;
}

//Get the number of Vertices and the Cyclomatic number
void getVertex_numCyclo_num(int &numVertices, vector<char*> &inputSet,
                       int &Input_Size, directedGraph &FSM)
{
     string StateDesc;
     int index;
     char State_Desc[10],
            *charInput;


     //Specify a description for each state
     for (int y = 0; y < numVertices; y++)
     {
           StateDesc = "s";
           itoa(y,State_Desc, 10);
           StateDesc = StateDesc + State_Desc;
         FSM.addVertex(StateDesc);
     }

      //Get the Input set
     for(int i = 0; i < Input_Size; i++)
     {
         //Insert input symbol if not already inserted
        charInput = new char;

        inputSet.push_back(itoa(i,charInput,10));
```

```
     }
}

/*Get Details of a Specific FSM*/
void getSpecificFSM(int &numVertices, directedGraph &FSM, char* &inputSet,
                     int &Input_Size, vector<string> &outputSet)
{
     string StateDesc,
             strOutput;
     char charInput;
     bool duplicate;
     int Output_Size,
         numEdges,
         OriginNode,
         DestinationNode,
         inputSize = -1,
         input,
         output;
     do
     {
          cin.clear();
          cin.ignore(numeric_limits<streamsize>::max(),'\n') ;

          cout << "Enter the number of vertex: ";
          cin >> numVertices;
      }while(!cin.good() || numVertices <= 0 );

     //Add the vertices(Description)
     cout << "Enter Description for each State(Vertex)\n\n";
     for (int y = 0; y < numVertices; y++)
     {
         cout << "State(Vertex) " << y << ": ";
         cin >> StateDesc;
         cout << endl;
```

```
    FSM.addVertex(StateDesc);
}
//Get the Input Set
do
{
     cin.clear();
     cin.ignore(numeric_limits<streamsize>::max(),'\n');

      cout << "\nEnter the input set size : ";
      cin >> Input_Size;
 }while(!cin.good() || Input_Size <= 0 );

 inputSet = new char[Input_Size];

 //Initialize the Input set
 for (int i = 0; i < Input_Size; i++)
 {
      inputSet[i] = '?';
 }

//Get Edges
do
{
     cin.clear();
     cin.ignore(numeric_limits<streamsize>::max(),'\n') ;

      cout << "\nEnter the number of Edges : ";
      cin >> numEdges;

}while(!cin.good() || numEdges <= 0 );

for(int i = 0; i < numEdges; i++)
{
     do
```

```
 {
      cin.clear();
      cin.ignore(numeric_limits<streamsize>::max(),'\n') ;
     cout << "\nEdge " << i << ":\n"
          << "Origin Node: ";
     cin >> OriginNode;
 }while(!cin.good());

cout << "\nDestination Node: ";
cin >> DestinationNode;
cout << "\nInput: ";
cin >> charInput;

duplicate = false;
//Check for duplicate input entry
for (int y= 0; y < Input_Size; y++)
    if (inputSet[y] == charInput)
    {
       duplicate = true;
       input = y;
       break;
    }
//Update the input set
if(!duplicate)
{
    inputSet[i] = charInput;
    inputSize++;
    input = inputSize;
}

//Get the Output
cout << "\nOutput: ";
cin >> strOutput;
```

```
        duplicate = false;
        //Check for duplicate output entry
        for (int y= 0; y < outputSet.size(); y++)
            if (outputSet[y] == strOutput)
            {
                duplicate = true;
                output = y;
                break;
            }

         if(!duplicate)
         {
            outputSet.push_back(strOutput);
            output = outputSet.size() - 1;
         }
        int edgePOS =FSM.addEdge(input, output, OriginNode, DestinationNode);
        FSM.update_VertexIncidence(-1, edgePOS-1, OriginNode);
        FSM.update_VertexIncidence(edgePOS-1, -1, DestinationNode);

    }

}
/*Construct an adjecency matrix from the generated FSM*/

void constructAdjecencyMatrix(directedGraph &FSM, int numVertices,
                                            vector<int*> &AdjecencyMatrix)
{
    edge temphold;
    int numEdges;

    int *rowPtr;  /*Points to the Adjancency Matrix vector*/

    // Clear the AdjecencyMatrix Matrix
    AdjecencyMatrix.clear();
```

```
    for (int i = 0; i < numVertices; i++)
    {
        rowPtr = new int[numVertices];
        AdjecencyMatrix.push_back(rowPtr);
    }

    //Initialize the Adjecency matrix
    for (int i = 0; i < AdjecencyMatrix.size(); i++)
        for (int j = 0; j < AdjecencyMatrix.size(); j++)
            AdjecencyMatrix[i][j] = 0;

    //Insert edges
    numEdges = FSM.getMaxEdge();
    for (int j = 0; j < numEdges; j++)
    {
        temphold = FSM.getEdge(j);
        AdjecencyMatrix[temphold.origin][temphold.destination] = 1;
    }
}

/*Construct an adjecency matrix */
void constructAdjecencyMatrix(int numVertices, vector<int*> &AdjecencyMatrix)
{

    int *rowPtr;  /*Points to the Adjancency Matrix vector*/

    for (int i = 0; i < numVertices; i++)
    {
        rowPtr = new int[numVertices];
        AdjecencyMatrix.push_back(rowPtr);
    }

    //Initialize the Adjecency matrix
```

```
    for (int i = 0; i < AdjecencyMatrix.size(); i++)
        for (int j = 0; j < AdjecencyMatrix.size(); j++)
            AdjecencyMatrix[i][j] = 0;

}


/*Construct a reachability matrix from an adjecency matrix
  derived from the generated FSM*/

void generateReachabilityMatrix(vector<int*> &AdjecencyMatrix)
{
  //printToScreen(AdjecencyMatrix);
  //system("pause");
  for (int k = 0; k < AdjecencyMatrix.size(); k++)
      for (int i = 0; i < AdjecencyMatrix.size(); i++)
          for (int j = 0; j < AdjecencyMatrix.size(); j++)
              AdjecencyMatrix[i][j] = AdjecencyMatrix[i][j] || AdjecencyMatrix[i][k]
                                      && AdjecencyMatrix[k][j];
  //cout << endl;
  //printToScreen(AdjecencyMatrix);
  //system("pause");

}

//For every randomly selected edge, confirm that a path exists
int pathExist (vector<int*> &ReachabilityMatrix, int *vertices, int verticeSize)
{
      int path,
           node;

      vector<int> nodes;
```

```
     for (int k = 0; k < verticeSize; k++)
           if (vertices[k] != -1)
              nodes.push_back(k);

     for (int k = 0; k < nodes.size(); k++)
     {
           path = 1;
           for (int f = 0; f < nodes.size(); f++)
                if (k != f)
                    path = path && ReachabilityMatrix[nodes[k]][nodes[f]];
           if (path)
               break;
     }

     return path;

}

void printColRow(vector<int*> &AdjecencyMatrix)
{
  for (int i = 0; i < AdjecencyMatrix.size(); i++)
  {
      for (int j = 0; j < AdjecencyMatrix.size(); j++)
          cout << AdjecencyMatrix[i][j] << "\t";
      cout << endl;
      for (int j = 0; j < AdjecencyMatrix.size(); j++)
          cout << AdjecencyMatrix[j][i] << endl;
  }
}


bool isFSM(vector<int*> &AdjecencyMatrix, directedGraph &fsm)
{
  bool TerminalNode,
```

```
      startNode;
vector<int> startNodeList;
vector<int> TerminalNodeList;
int  count = 0,
     acceptState; /*Holds the number of Start or Terminal states*/

unsigned seed = time(0);
srand(seed);

//Test for terminal nodes:
for (int i = 0; i < AdjecencyMatrix.size(); i++)
{
   //Ensure that a transition to a node in position 0,0 is considered.
   if (i == 0 && AdjecencyMatrix[0][i] == 0)
      TerminalNode = !AdjecencyMatrix[0][i];
   else
      TerminalNode = AdjecencyMatrix[0][i];
   //A terminal node must be reachable from all nodes
    for (int j = 1; j < AdjecencyMatrix.size(); j++)
   {
         if (i == j && AdjecencyMatrix[j][i] == 0)
             TerminalNode = TerminalNode && !AdjecencyMatrix[j][i];
         else
             TerminalNode = TerminalNode && AdjecencyMatrix[j][i];
    }

    //Get a vector of all possible terminal nodes
   if (TerminalNode)
   {
       TerminalNodeList.push_back(i);
   }
}

//Test for a start node: A start node must reach all nodes
```

```
  //Ensure that a transition to a node in position 0,0 is considered.
  for (int i = 0; i < AdjecencyMatrix.size(); i++)
  {

     if (i == 0 && AdjecencyMatrix[i][0] == 0)
        startNode = !AdjecencyMatrix[i][0];
     else
        startNode = AdjecencyMatrix[i][0];

      for (int j = 1; j < AdjecencyMatrix.size(); j++)
     {
           if (i == j && AdjecencyMatrix[i][j] == 0)
               startNode = startNode && !AdjecencyMatrix[i][j];
           else
               startNode = startNode && AdjecencyMatrix[i][j];
      }

      //Get a vector of all possible start nodes
     if (startNode)
     {
         startNodeList.push_back(i);
     }
  }

  //Update the the start and accept state
  if (startNodeList.size() && TerminalNodeList.size())
  {
       fsm.update_Vertex(1, 0, startNodeList[rand() % startNodeList.size()]);
       fsm.update_Vertex(0, 1, TerminalNodeList[rand() % TerminalNodeList.size()]);
  }

  return (startNodeList.size() && TerminalNodeList.size());
}
```

```
//Build the State Transition Table of the FSM
void BuildStateTransitionTable(directedGraph &FSM, vector<char*> &inputSet,
                      int Input_Size, vector<int*> &StateTransitionTable)
{

     vertex State;
      edge Edge;
      int *ptrNextStateList;

      int StateSet_Size = FSM.getMaxVertices();

      for (int i = 0; i < StateSet_Size; i++)
     {
          //Get the nextstate list for the state transition table and initialize every
entry to -1
           ptrNextStateList = new int[Input_Size];
          for (int k = 0; k < Input_Size; k++)
          {
                ptrNextStateList[k] = -1;
           }

         //Insert the nextState vector into the StateTransitionTable vector
         StateTransitionTable.push_back(ptrNextStateList);

           /*Update the StateTransitionTable for each state with the next state
             for each input applied */
           State = FSM.getVertex(i);
          for (int k = 0; k < State.Edge_incidenceOUT.size(); k++)
          {
                Edge = FSM.getEdge(State.Edge_incidenceOUT[k]);
                StateTransitionTable[i][Edge.Input] = Edge.destination;
           }
      }
```

```
}

//Build the output function
void BuildOutputTransitionTable(directedGraph &FSM, vector<char*> &inputSet,
  int Input_Size, vector<int*> &OutputTransitionTable, vector<string> &OutputSet)
{
     vertex State;
      edge Edge;
      int *ptrOutputList;

      int StateSet_Size = FSM.getMaxVertices();

      for (int i = 0; i < StateSet_Size; i++)
     {
          //Get the nextstate list for the state transition table and initialize every
entry to -1
           ptrOutputList = new int[Input_Size];
          for (int k = 0; k < Input_Size; k++)
          {
                ptrOutputList[k] = -1;
           }

         //Insert the nextState vector into the StateTransitionTable vector
         OutputTransitionTable.push_back(ptrOutputList);

           /*Update the OutputTransitionTable for each state with the output
             for each input applied */
           State = FSM.getVertex(i);
         for (int k = 0; k < State.Edge_incidenceOUT.size(); k++)
          {
                Edge = FSM.getEdge(State.Edge_incidenceOUT[k]);
                OutputTransitionTable[i][Edge.Input] = Edge.Output;
         }
     }
```

```
}


//Print to screen
void printToScreen(vector<possibleEdges> &AllPossibleEdges, directedGraph &fsm)
{
  vertex state;

  for (int i = 0; i < AllPossibleEdges.size(); i++)
  {
     state = fsm.getVertex(AllPossibleEdges[i].node1);
     cout << state.Vertex << "\t";
     state =  fsm.getVertex(AllPossibleEdges[i].node2);
     cout << state.Vertex << "\t"
          << AllPossibleEdges[i].selected << endl;
  }
}

void printToScreen(vector<int*> &AdjecencyMatrix)
{

  for (int i = 0; i < AdjecencyMatrix.size(); i++)
  {
      for (int j = 0; j < AdjecencyMatrix.size(); j++)
          cout << AdjecencyMatrix[i][j] << "\t";
      cout << endl;
  }
}

void printToScreen(vector<int*> &STT, int Input_Size, directedGraph &FSM)
{
  vertex State;
  for (int i = 0; i < STT.size(); i++)
  {
```

```
      for (int j = 0; j < Input_Size; j++)
      {
          if (STT[i][j] != -1)
          {
              State = FSM.getVertex(STT[i][j]);
              cout << State.Vertex << "\t";
          }
          else
              cout << STT[i][j] << "\t";
      }
      cout << endl;
  }
}

void printToScreen1(vector<int*> &STT, vector<string> &OutputSet, int Input_Size)
{
  for (int i = 0; i < STT.size(); i++)
  {
      for (int j = 0; j < Input_Size; j++)
      {
          if (STT[i][j] != -1)
          {
              cout << OutputSet[STT[i][j]] << "\t";
          }
          else
              cout << STT[i][j] << "\t";
      }
      cout << endl;
  }
}

//Build the BFSSuccessor Tree
void buildUIOSuccessorTree(directedGraph &FSM, vector<int*> &OutputTransitionTable,
    vector<int*> &StateTransitionTable, vector<char*> &inputSet, int Input_Size,
```

```
    vector<string> &OutputSet, ostream &out, char writeToFile, int &recordUIO_Seq)
{
     UIOsuccessorTree *UIOseqSuccessorTree;
     Vertex_Node *Root_Node;

     int SeqLength,
           Path;
     vector<char> UIO_sEQ;
     queue<Vertex_Node*> FIFO_BFSqueue;

     //Create the initial state uncertainity Root Node
    int StateSize = FSM.getMaxVertices();
    for (int y = 0; y < StateSize; y++)
    {
       Path = 0;
       SeqLength = 0;

        UIOseqSuccessorTree = new UIOsuccessorTree; //New Tree for a specific State
        Root_Node = new Vertex_Node; //Root Node for the Tree

        UIOseqSuccessorTree->setRoot(Root_Node);

        if (writeToFile == 'y')
            out << "\t\tUIO Sequence for state - " << y+1 << endl;

       UIOseqSuccessorTree->setState(y);
       Root_Node->ptrParentNode = NULL;
       Root_Node->state.initialStateIndex = y;
       Root_Node->state.currentStateIndex = y;

       for (int t = 0; t < StateSize; t++)
       {
             if (t != y)
                 Root_Node->current_Set_Uncertainty.push_back(t);
```

```
         }
         FIFO_BFSqueue.push(Root_Node);
        //Build the UIO successor tree
        UIOseqSuccessorTree->generateUIO_Tree(FIFO_BFSqueue, Input_Size, inputSet,
                     StateTransitionTable, OutputTransitionTable, OutputSet, SeqLength,
                 Path, out, writeToFile, recordUIO_Seq);
        //Delete tree for state y
        UIOseqSuccessorTree->deleteTree(Root_Node);
    }
}


/*Randomly select values for the intervals in the main program*/
int randomlySelectValue(int &upperBound, int &temp ,
                            vector<int> &vectorRandomlySelected_values)
{
   int Rindex = 0;

   bool duplicate = false;

   do
   {
      //Confirm that the selected number has not been previously selected
       for(int i = 0; i < vectorRandomlySelected_values.size(); i++)
       {
          duplicate = false;
          //Check for duplicate entry
          if (vectorRandomlySelected_values[i] == temp || temp == 0)
          {
             duplicate = true;
             break;
          }
       }
```

```
        if(!duplicate)
            vectorRandomlySelected_values.push_back(temp);
    }while(duplicate);

    cout << "Random number\t" << Rindex
         << "\t" << temp << endl;

    //return the randomly selected value
    return vectorRandomlySelected_values[Rindex++];

}
```

APPENDIX B

FSM Random Generator Program

FSM Random Generator Header File

```
#ifndef hdr_FSMrandomGenerator_H
#define hdr_FSMrandomGenerator_H

#include<iostream>
#include<fstream>
#include<vector>
using namespace std;


struct possibleEdges
{
    int node1;
    int node2;
    int selected;

    possibleEdges(){selected = 0;}
};

struct vertex
{
    string Vertex;
    vector<int> Edge_incidenceIN;
    vector<int> Edge_incidenceOUT;
    int startState;
    int acceptState;
```

```
    vertex(){startState = 0; acceptState = 0;}
};

struct edge
{
    int Input;
    int Output;
    int origin;
    int destination;
};

//A directed graph implementation of an FSM
class directedGraph
{
    vector<vertex> vertexList;
    vector<edge> edgeList;

public:
   void addVertex(string v);
   int addEdge(int , int , int , int );
   void update_VertexIncidence(int , int , int );
   //Update vertex with the start State and Acceptance State
   void update_Vertex(int , int , int );

   //Return the vertex at position index
   vertex getVertex(int );

   //Return the edge at position index
   edge getEdge(int );

   //Return the size of the edge list
   int getMaxEdge ();
```

```
   //Return the size of the vertex list
   int getMaxVertices ();

   //Clear the edge data structure
   void clearEdgeList();

   //Reset FSM
   void resetFSM();

   void printEdges(vector<char*> &, vector<string> &, ostream &);

   void printVertices();

   void printGraph ();
};



#endif
```

FSM Random Generator Source File

```
   void directedGraph::addVertex(string v)
   {
       vertex temp;
       temp.Vertex = v;
       vertexList.push_back(temp);
   }
   int directedGraph::addEdge(int InputIndex, int OutputIndex, int origin, int
destination)
   {
       edge temp;
       temp.Input = InputIndex;
       temp.Output = OutputIndex;
       temp.origin = origin;
       temp.destination = destination;
       edgeList.push_back(temp);

       return edgeList.size();
   }

   void directedGraph::update_VertexIncidence(int in, int out, int index)
   {
       if (in != -1)
          vertexList[index].Edge_incidenceIN.push_back(in);
       if (out != -1)
          vertexList[index].Edge_incidenceOUT.push_back(out);
   }

   //Update vertex with the start State and Acceptance State
   void directedGraph::update_Vertex(int startState, int acceptState, int index)
   {
       vertexList[index].startState = startState;
       vertexList[index].acceptState = acceptState;
```

```
    }

    //Return the vertex at position index
    vertex directedGraph::getVertex(int index)
    {
        return vertexList[index];
    }

    //Return the edge at position index
    edge directedGraph::getEdge(int index)
    {
        return edgeList[index];
    }

    //Return the size of the edge list
    int directedGraph::getMaxEdge ()
    {
        return edgeList.size();
    }

    //Return the size of the vertex list
    int directedGraph::getMaxVertices ()
    {
        return vertexList.size();
    }

    //Clear the edge data structure
    void directedGraph::clearEdgeList()
    {
         for (int i = 0; i < edgeList.size(); i++)
         {
             //Remove edge incidence on the origin node
             for (int j = 0; j < vertexList[edgeList[i].origin].Edge_incidenceOUT.size();
j++)
```

```
               if (vertexList[edgeList[i].origin].Edge_incidenceOUT[j] == i)

vertexList[edgeList[i].origin].Edge_incidenceOUT.erase(vertexList[edgeList[i].origin].Edg
e_incidenceOUT.begin() + j);

            //Remove edge incidence on the destination node
            for (int j = 0; j <
vertexList[edgeList[i].destination].Edge_incidenceIN.size(); j++)
               if (vertexList[edgeList[i].destination].Edge_incidenceIN[j] == i)

vertexList[edgeList[i].destination].Edge_incidenceIN.erase(vertexList[edgeList[i].destina
tion].Edge_incidenceIN.begin() + j);

        }

        //Clear the edge list
        edgeList.clear();

   }

   //Reset FSM
   void directedGraph::resetFSM()
   {
       vertexList.clear();
       edgeList.clear();
   }

   void directedGraph::printEdges(vector<char*> &inputSet, vector<string> &OutputSet,
ostream &out)
   {
       out << "Input" << "\t" << "Output" << "\t" << "origin" << "\t" << "destination" <<
endl;
       for (int i = 0; i < edgeList.size(); i++)
          out << inputSet[edgeList[i].Input] << "\t"
```

```
            << OutputSet[edgeList[i].Output] << "\t"
               << vertexList[edgeList[i].origin].Vertex << "\t"
               << vertexList[edgeList[i].destination].Vertex
            << endl << endl;
}

void directedGraph::printVertices()
{
    cout << "vertex" << endl;
    for (int i = 0; i < vertexList.size(); i++)
    {
       cout << vertexList[i].Vertex << endl;
       cout << "Edge_incidenceIN: ";
       for (int j = 0; j < vertexList[i].Edge_incidenceIN.size(); j++)
               cout << vertexList[i].Edge_incidenceIN[j] << "\t";
       cout << endl << "Edge_incidenceOUT: ";
       for (int j = 0; j < vertexList[i].Edge_incidenceOUT.size(); j++)
               cout  << vertexList[i].Edge_incidenceOUT[j] << "\t";
        cout << "\n----------------------------------------------------" << endl;
    }
}

void directedGraph::printGraph ()
{
    ofstream fout("output.txt");
    bool entered;
    for (int i = 0; i < vertexList.size(); i++)
    {
        if (vertexList[i].startState == 1)
              fout << "Start state:\t" << vertexList[i].Vertex << endl;
        else if (vertexList[i].acceptState == 1)
              fout << "Accept state:\t" << vertexList[i].Vertex << endl;
    }
     //row represents the origin
```

```
    for (int i = 0; i < vertexList.size(); i++)
    {
          //column represenrs destination
        for (int j = 0; j < vertexList.size(); j++)
        {
            entered = false;
            for (int k = 0; k < vertexList[i].Edge_incidenceOUT.size(); k++)
            {
                 if (edgeList[vertexList[i].Edge_incidenceOUT[k]].destination == j+1)
                {
                   fout << 1 << "\t";
                   entered = true;
                   break;
                }
            }
            if (!entered)
               fout << 0 << "\t";
        }
        fout << endl;
    }
}
```

APPENDIX C

Unique Input Output (UIO) Generator Program

UIO Successor Tree Header File

```
#ifndef hdr_UIOsuccessorTree_H
#define hdr_UIOsuccessorTree_H

#include<iostream>
#include<fstream>
#include<vector>
#include <queue>
#include <iomanip>
using namespace std;

//Specifies the state for which a UIO seq. is generated
struct state4UIO
{
    int initialStateIndex;
    int currentStateIndex;
};

//A node of the successor tree: The current state uncertainity vector
struct Vertex_Node
{
    char* input;
    string output;
    state4UIO state;
    //vector<int> initial_Set_Uncertainty;
    vector<int> current_Set_Uncertainty;
```

```
    bool isStatesDistinguished,
         isSameAsParentVertex; //Set ON if two states are not distinguished
    vector<Vertex_Node*> ChildrenPtrsSet;
    Vertex_Node *ptrParentNode; //Ptr to the parent node

    Vertex_Node(){isStatesDistinguished = true; isSameAsParentVertex = false;}
};

//A (successor Tree) Tree Data structure
class UIOsuccessorTree
{
    Vertex_Node* Rootnode;
    int StateIndex;
    queue<Vertex_Node*> FIFO_BFSqueue;
    vector<char> UIO_sEQ;

public:
    UIOsuccessorTree();
    //~UIOsuccessorTree();
    void deleteTree(Vertex_Node*);
    void setState(int);
    void generateUIO_Tree(queue<Vertex_Node*> &, int &, vector<char*> &, vector<int*> &,
     vector<int*> &, vector<string> &, int, int&, ostream &, char, int&);
    void setRoot(Vertex_Node*);
};

#endif
```

UIO Successor Tree Source File

```
//The successor Tree constructor
UIOsuccessorTree::UIOsuccessorTree()
{
    Rootnode = NULL;
}

//Traverses the UIO successor Tree and deletes each node
void UIOsuccessorTree::deleteTree(Vertex_Node *Node)
{
     char input;
     if (Rootnode == NULL)
         return;
     else if (Node->ChildrenPtrsSet.empty())
         delete Node;
     else
     {
         for (int i = 0; i < Node->ChildrenPtrsSet.size(); i++)
         {
             deleteTree(Node->ChildrenPtrsSet[i]);
         }
         delete Node;
     }
}

//Each Tree is for a specified state in the FSM: This specifies the state
void UIOsuccessorTree::setState(int stateIDX)
{
    StateIndex = stateIDX;
}

//Set the Root Node
```

```
void UIOsuccessorTree::setRoot(Vertex_Node* R_node)
{
     Rootnode = R_node;
}

//Generate the UIO Successor tree
void UIOsuccessorTree::generateUIO_Tree(queue<Vertex_Node*> &FIFO_BFSqueue, int
&Input_Size,
      vector<char*> &inputSet, vector<int*> &StateTransitionTable, vector<int*>
&OutputTransitionTable,
      vector<string> &OutputSet, int Rank, int &Path, ostream &out, char writeToFile,
     int &recordUIO_Seq)
{
     Vertex_Node *node,
                    *ptrNode,
                *cursor;
     int ST_CurrentStateIdx,
          ST_outputIndex,
          nextStateIdx;
    string strOutput;
    bool flag,
         stateExist;

     //Base Case1: Terminate sucessor tree
    if (FIFO_BFSqueue.empty())
    {
          if (writeToFile == 'y')
           out << "FIFO queue is empty - No UIO Sequence" << endl;
       return;
     }
    else
    {
          //Pop out the oldest node in the FIFO queue and examine it
          node = FIFO_BFSqueue.front();
```

```
  FIFO_BFSqueue.pop();

  //Terminate node if state is not distinguished or node is repeated
 if (!(node->isStatesDistinguished) || node->isSameAsParentVertex)
       generateUIO_Tree(FIFO_BFSqueue, Input_Size, inputSet,
       StateTransitionTable, OutputTransitionTable, OutputSet, Rank+1, Path,
     out, writeToFile, recordUIO_Seq);

/*Base Case2: The state under consideration is completely distinquished
  Terminate sucessor tree*/
 else if (node->current_Set_Uncertainty.empty())
 {
       if (writeToFile == 'y')
       {
          out << "\n" << setw(18) << "Path" << setw(5) << "-" << "<";
        cursor = node;
          while (cursor->ptrParentNode != NULL)
          {
              out << cursor->state.currentStateIndex << " ";
              cursor = cursor->ptrParentNode;
         }
         out << cursor->state.currentStateIndex
            << ">\n";

        int UIO_Seq_Length = 0;//Gets the UIO sequence length

          out << setw(18) << "UIO Sequence" << setw(5) << "-" << "<";
        cursor = node;
          while (cursor->ptrParentNode != NULL)
          {
              out << cursor->input << " ";
              cursor = cursor->ptrParentNode;
              UIO_Seq_Length++;
         }
```

```
         out << ">\n"
            << setw(18) << "UIO Length" << setw(5) << "-"
            << UIO_Seq_Length
            << endl;

        /*Assign the UIO sequence length
        to the recordUIO_Seq variable*/
         recordUIO_Seq = UIO_Seq_Length;

          out << setw(18) << "Output Sequence" << setw(5) << "-" << "<";
        cursor = node;

          while (cursor->ptrParentNode != NULL)
          {
              out << cursor->output << " ";
              cursor = cursor->ptrParentNode;
         }

          out << ">\n\n" << endl;
       }
   //Clear the FIFO queue
   while(!FIFO_BFSqueue.empty())
   {
      FIFO_BFSqueue.pop();
   }
    return;
 }
 else
 {
      for (int n = 0; n < Input_Size; n++)
     {
         ptrNode = new Vertex_Node;
           node->ChildrenPtrsSet.push_back(ptrNode);
```

```
                    //push child nodes to the FIFO queue
                    FIFO_BFSqueue.push(ptrNode);

               ptrNode->input = inputSet[n];
                    ptrNode->ptrParentNode = node;

                  ST_CurrentStateIdx = StateTransitionTable[node-
>state.currentStateIndex][n];
                    ST_outputIndex = OutputTransitionTable[node-
>state.currentStateIndex][n];

                    //Update details for the state in question
                    ptrNode->state.initialStateIndex = node->state.currentStateIndex;

               if (ST_CurrentStateIdx != -1)
               {
                        ptrNode->state.currentStateIndex = ST_CurrentStateIdx;
                        ptrNode->output = OutputSet[ST_outputIndex];
                }
               else
               {
                        ptrNode->state.currentStateIndex = node->state.currentStateIndex;
                        ptrNode->output = "null";
                        ST_CurrentStateIdx = node->state.currentStateIndex;
                   }

                    for (int k = 0; k < node->current_Set_Uncertainty.size(); k++)
                    {
                    /*check if the state under consideration and any other state
                          gives the same output; for an unspecified input a null
                          output dented as "-1" is produced*/
                         if (ST_outputIndex == OutputTransitionTable[node-
>current_Set_Uncertainty[k]][n])
                         {
```

```
                    /*If the behaviour for the input is not specified, then
                        it is assumed that for the input the FSM remains in
                        the current state: Null output is permitted.
                  It is assumed that every transition and the production
                  of output takes place within a certain time [SD],
                  and thus the null output (absence of output) can be
                  treated as just another output symbol*/
                   if (StateTransitionTable[node->current_Set_Uncertainty[k]][n]
== -1)
                       nextStateIdx = node->current_Set_Uncertainty[k];
                  else
                       nextStateIdx = StateTransitionTable[node-
>current_Set_Uncertainty[k]][n];

                    /*Check if Input symbol distinguishes two states: Does the
                 state under consideration and any other state transit
                 to the same state?*/
                    if (ST_CurrentStateIdx == nextStateIdx)
                    {
                        ptrNode->isStatesDistinguished = false;
                        break;
                }
                    else
                         ptrNode->current_Set_Uncertainty.
                               push_back(nextStateIdx);
                }
           }

           //Check if parent vertex is same as child vertex
           /*First check if the node's current state is same as the current
          state in any of the blocks of the vertex*/
           ptrNode->isSameAsParentVertex = true;
        flag = false;
```

```
                if (node->state.currentStateIndex == ptrNode->state.currentStateIndex)
                    flag = true;
                else
                   for (int n = 0; n < ptrNode->current_Set_Uncertainty.size(); n++)
                   {
                        if (node->state.currentStateIndex == ptrNode-
>current_Set_Uncertainty[n])
                        {
                           flag = true;
                           break;
                        }
                    }
                ptrNode->isSameAsParentVertex = ptrNode->isSameAsParentVertex && flag;

                   /*Proceed to check if the parent vertex is same as the child vertex
                   where ptrNode->isSameAsParentVertex is true*/

                   if (ptrNode->isSameAsParentVertex)
                   for (int y = 0; y < node->current_Set_Uncertainty.size(); y++)
                   {
                     flag = false;
                     if (node->current_Set_Uncertainty[y] == ptrNode-
>state.currentStateIndex)
                     {
                         continue;
                        }
                   for (int n = 0; n < ptrNode->current_Set_Uncertainty.size(); n++)
                   {
                         if (node->current_Set_Uncertainty[y] == ptrNode-
>current_Set_Uncertainty[n])
                         {
                            flag = true;
                            break;
                         }
```

```
                        }
                        if(!(ptrNode->isSameAsParentVertex = ptrNode->isSameAsParentVertex
&& flag))
                            break;
                    }
                }
                generateUIO_Tree(FIFO_BFSqueue, Input_Size, inputSet,
                StateTransitionTable, OutputTransitionTable, OutputSet, Rank+1, Path,
              out, writeToFile, recordUIO_Seq);
            }
    }
}
```

APPENDIX D

Timer Routine

Time Routine Header File

```
#ifndef hdr_CStopWatch_H
#define hdr_CStopWatch_H

#include<iostream>
#include <windows.h>
#include <iomanip>
using namespace std;

typedef struct {
       _LARGE_INTEGER start;
       _LARGE_INTEGER stop;
} stopWatch;

class CStopWatch
{
    private:
            stopWatch timer;
            _LARGE_INTEGER frequency;
            double LIToSecs( _LARGE_INTEGER & L) ;
    public:
           CStopWatch() ;
           void startTimer( ) ;
           void stopTimer( ) ;
           double getElapsedTime() ;
};
#endif
```

Time Routine Source File

```
double CStopWatch::LIToSecs( _LARGE_INTEGER & L)
{
       return ((double)L.QuadPart /(double)frequency.QuadPart) ;
}

CStopWatch::CStopWatch()
{
    timer.start.QuadPart=0;
    timer.stop.QuadPart=0;
    QueryPerformanceFrequency( &frequency );//Gets how many times it is counting in a
second
    //cout << setprecision(35) << "frequency\t" << (double)frequency.QuadPart << endl;
}

void CStopWatch::startTimer()
{
   QueryPerformanceCounter(&timer.start) ;

}

void CStopWatch::stopTimer()
{
   QueryPerformanceCounter(&timer.stop) ;
}

double CStopWatch::getElapsedTime()
{
    LARGE_INTEGER time;
    time.QuadPart = timer.stop.QuadPart - timer.start.QuadPart;
    return LIToSecs( time) ;
}
```

BIBLIOGRAPHY

Allen, Arnold O. *Probability, statistics, and queueing theory with computer science applications*. San Diego, CA: Academic Press, Inc., 1990.

Beizer, Boris. *Software testing techniques*. New York NY: International Thomson Computer press, 1990.

Berge, Claude. *Graphs and hypergraphs*. New York: North-Holland Publishing Company Amsterdam New York Oxford., 1976.

Bolton, David. How do I do high resolution timing in C++ on windows? http://cplus.about.com/od/howtodothingsi2/a/timing.htm (accessed July 23, 2010).

Broy, Manfred, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems*. Germany: Springer-Verlag Berlin Heidelberg, 2005.

Gallagher, T. THERAC-25 computerized radiation therapy. 20 famous software disasters. http://www.devtopics.com/20-famous-software-disasters (accessed August 04, 2009).

Garey, Micheal R. and David S. Johnson. *Computers and intractability. A guide to the theory of NP – completeness*. New York: W. H. Freeman and Company, 1979.

Gibbs, Wayt, W. Software's chronic crisis. http://www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1994.html (accessed August 04, 2009).

Gleick, James. ARIANE-5 A bug and a crash. http://www.around.com/ariane.html (accessed August 04, 2009).

Kohavi, Zvi. *Switching and finite automata theory*. second ed. New York: McGraw-Hill, Inc, 1978.

Lee, D. and M. Yannakakis. Principles and methods of testing finite state machines A survey. *Proceedings of the IEEE* 84, no. 81996). : 1090-1123.

Lee, D. and M. Yannakakis. Testing finite-state machines: State identification and verification. *Proceedings of the IEEE* 43, no. 31994). : 306-320.

Leveson, Nancy. Medical devices: The therac-25. http://sunnyday.mit.edu/papers/therac.pdf (accessed August 04, 2009).

McCabe, Thomas J., ed. *Structured testing*. New York: IEEE Computer Society Press, 1983.

Mueller, Carl J. 2003. Automated Interface Probing Applied To Cots Component Evaluation. Doctor of Philosophy in Computer Science, Illinois Institute of Technology.

Rich, Elaine. *Automata, computability, and complexity theory and application*. Upper Saddle River NJ: Pearson Prentice Hall, 2008.

Rosen, Kenneth H. *Discrete mathematics and its applications*. sixth ed. New York: McGraw-Hill, 2007.

Sabnani, Krishan and Anton Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems* 15, no. 41988). : 285-297.

Sakharov, Alexander. Finite state machines: Quick tutorial. http://www.sakharov.net/fsmtutorial.html (accessed July 08, 2009).

Sipser, Micheal. *Introduction to the theory of computation*PWS Publishing, 1997.

Sun, Xiao, Yinan Shen, and Chao Feng. *Protocol conformance testing using Input/Output sequences*. first ed.World Scientific Publishing Co. Pte. Ltd, 1997.

Tian, Jeff. *Software quality engineering: Testing, quality assurance, and quantifiable improvement*Jo