Conformane Testing of Protocols Represented as Communicating Finite State Machines

B.Tech. Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor of Technology

by

V Arun Kumar
Roll No: 95005014

under the guidance of

Dr. S Ramesh

Department of Computer Science and Engineering
Indian Institute of Technology
Bombay
April 2, 2001

Acknowledgement

April 2, 2001

I wish to express my heartfelt gratitude to my guide Prof. S Ramesh for his invaluable guidance, support and encouragement. He has been very helpful in the various discussions we have had and has been a constant source of motivation for me to improve myself. I also wish to thank S Vemuri for all the help and information he provided to get me started with this project. Due thanks also to Prof. R Heirons for providing a related paper of his promptly.

V Arun Kumar
95005014

Abstract

In recent times, conformance testing of implementation of network protocols to their standard specification by expressing them in the form of communicating Mealy machines has come to be an active field of research. Relevant literature in this area has traditionally confined itself to the problem of test case generation for a single finite state machine. This approach expresses a set of communicating finite state machines as a single machine by constructing the cross product. This however may lead to combinatorial state explosion.

An alternate method for generating test sequences for CFSMs by combining test sequences of the component FSMS, thereby avoiding cross product computation has been proposed in this paper. This method works for restricted set of CFSMs participating in bidirectional synchronous communication. We also present an adaption of this new approach in the form of a heuristic algorithm for the more general case of arbitrary synchronous communication. The proposed algorithm has also been implemented and experimental analysis performed to verify the claims of efficiency.

Contents

1 Introduction **1**

1.1 Motivation and Problem Statement . . . 1

1.2 The UIO Method . . . 2

1.3 Scope of the Project . . . 4

1.4 Organisation of the Report . . . 4

2 The Compositional Approach **6**

2.1 Definitions . . . 7

2.2 Composing Algorithms . . . 9

2.3 Projection Strings . . . 10

2.4 Composing Algorithms . . . 12

2.4.1 Synchronously interacting Machines . . . 13

3 Constrained Synchronous Communication **15**

3.1 Path Compatibility . . . 16

3.2 Computaion of UIO sequences . . . 18

3.3 Complexity Analysis . . . 19

3.3.1 Space Complexity Comparision . . . 19

3.3.2 Time Complexity . . . 22

3.4 Length of Test sequences . . . 24

3.5 Conclusion . . . 26

4 Heuristics and Generalizations **29**

5 Design and Implementation **32**

5.1 Design of Data Structures . . . 32

5.2 Module Descriptions . . . 33

6 Conclusion and Future Work **34**

References **40**

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Finite state machines have been widely used to model systems in diverse areas such as sequential circuits, some classes of programs involving lexical analysis, pattern matching and more recently, communication protocols. This motivated the study of testing finite state machines to ensure their correct functioning. The literature in this area has traditionally confined itself to the problem of test sequence generation for a single finite state machine. However, most network protocol specifications are expressed in the form of communicating Mealy machines. Languages like SDL and Promela support concurrent processes interacting through signals and channels and are useful for expressing protocol specifications. The motivation for this work arises out of the need to generate test-sets from these specifications in the most efficient way possible.

Testing is carried out by means of test sequences. A test sequence is a sequence of I/O pairs, which is derived from the specification FSM. For conformance testing, we treat the implementation as a black box and apply the input sequence. The outputs given out by the implementation are now compared to the outputs expected from the test sequence. This gives us a way of checking whether the implementation conforms to the specification or not. The next section describes the well known UIO method for test sequence generation.

Conventional methods of protocol test generation would first compute the cross product and then derive the test sequences by any of the standard methods in the literature. However, our aim is to explore possibilities of extending the UIO method to communicating FSMs without taking the cross product. One such approach, which we shall refer to as the *compositional approach* was proposed by S Vemuri in [Vem98]. He proposed a method of composing test sequences generated by the UIO method to obtain the test sequences of the product machine for the class of non-interacting machines and machines participating in unidirectional, synchronous communication with certain constraints. In this report we present an improved

adaptation of his idea backed by a complete complexity analysis. The main contributions are:

1. We formally introduce the compositional methodology backed by proofs of various results used.

2. We give an efficient procedure to generate UIO sequences for a system of CFSMs by composing UIO sequences of the component FSMs.

3. We describe extensions of the compositional approach to handle an arbitrary number of CFSMs and also allow for feedback transitions.

4. We also given a heuristic algorithm to handle the case of unconstrained synchronous communication.

5. Finally, we describe the design and implementation of the compositional approach for purposes of experimenatl analysis.

1.2 The UIO Method

In this section, we describe the UIO method for test generation. For a detailed description, refer to [SD88] The UIO method is used to detect transition faults in Mealy machines. An example of a Mealy machine is given in fig 1.1.

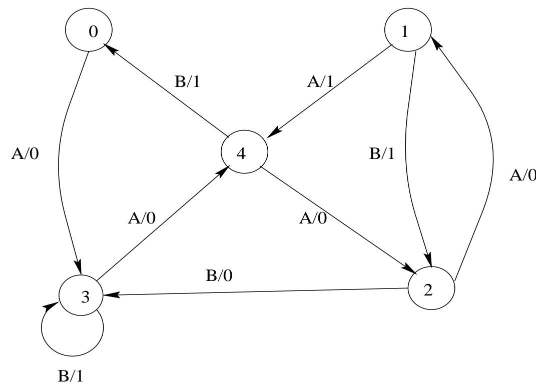

Figure 1.1: Transition diagram for machine M

A UIO sequence for a state is a sequence of valid input/output symbols which is unique for that state. It is used for state verification by applying the input sequence to the state and verifying that the outputs tally with the outputs specified by the UIO sequence. The UIO sequences for the states of the machine shown in figure 1.1 are given in table 1.1. In this table λ stands for nil ouput. (We assume that the FSM outputs λ whenever the transition is not specified on a given input symbol)

State	UIO
0	B/λ
1	A/1 A/1
2	B/0
3	B/1 B/1
4	A/1 A/0

Table 1.1: UIO sequences for the states of machine M

The test sequence T for testing the transition $S_i \rightarrow S_j$ using UIO sequences is generated by using Algorithm 2.1

Algorithm 1.1 Generation of UIO test sequences

Input : FSM M and states S_i and S_j.

Output : Test sequence for transition $S_i \rightarrow S_j$.

Let T1, T2 and T3 are input/output sequences initialized to NULL.

T1 = The shortest sequence to reach S_i from S_0, the initial state of the FSM.

T2 = The transition S_i to S_j.

T3 = A UIO sequence for S_j.

T = Concatenate T1, T2, T3.

The procedure described in [SD88] tries to obtain shortest length UIO sequences for each state, by enumerating the paths from a state in a breadth-first manner. If it fails to obtain a UIO sequence of length less than or equal to $2n^2$, n being the number of states in the FSM, then it computes an alternate signature for the states as described below.

Since all the states S_i are distinct, for every pair S_i, S_j of states there must be a distinguishing sequence $IO(i,j)$ of length less than n. The formula for the signature is

$$Sig(S_i) = \begin{cases} IO(s_i,1)@\#_{k=2}^{n} T_i(k-1)@IO(i,k) & for \quad k \neq i \quad and \quad i \neq 1 \\ IO(1,2)@\#_{k=3}^{n} T_1(k-1)@IO(1,k) & for \quad i = 1 \end{cases}$$

In the above equation $IO(i,k)$ is the shortest sequence that distinguishes S_i from S_j. $T_i(k)$ is the shortest sequence which brings back the machine to S_i from the state the machine enters after application of the sequence IO(i,k). '@' and '#' denote concatenation operators. Note that the length of this signature does not exceed $2n^2$ where n is the number of states.

There is no good upper bound on UIO sequences but the signature computed by using (1) has an upper bound of $2n^2$. Therefore, if we are not able to generate a UIO sequence of length smaller than $2n^2$, then we can use the alternate signature. Note however that the alternate signature may not be a UIO sequence.

The procedure for UIO computation described in [SD88] UIO sequences has time com-

plexity $O(n^2 d_{max}^{2n^2+2})$ where d_{max} is the maximum degree of a vertex in the FSM. Once the UIO sequences for all the states have been computed, an optimized set of test sequences can be generated in $O(mn^2)$ time where m is the number of edges in the FSM. The upper bound on the length of the test sequences computed by this method is shown to be $O(n^2m)$.

The results of applying the UIO method on the finite state machine in 2.1 are shown in table 2.2. (The symbol r is the reset symbol that brings the FSM from any state back to the initial state.)

State	Test Sequence
0-3	r A B B
0-0	r B B
1-4	r A A A A A A A
1-2	r A A A A B B
2-1	r A A A A A A
2-3	r A A A B B B
3-4	r A A A A
3-3	r A B B B
4-2	r A A A B
4-0	r A A B B

Table 1.2: Test sequences by the UIO method for M

1.3 Scope of the Project

The broad aim of the project is to generate test cases for protocols specified in a high level formal description language like SDL. SDL supports concurrent processes and these can be translated to finite state machine specifications. Each of the concurrent processes can be mapped to a finite state machine. The immediate aim of the project is to develop a method for generating test suites for the combined FSM without actually combining the individual FSMs. The method uses conventional methods to generate UIO test sequenecs for each of the component and integrates them in a certain manner to generate the test suite for the global FSM.

1.4 Organisation of the Report

The rest of the report is organised as follows. In chapter 2, we introduce basic definitions and prove some theorems pertaining to composition of UIO sequences. In chapter 3, we give the actual procedure to generate UIO sequences for FSMs participating in bidirectional

synchronous communication. Chapter 4 describes heuristic techniques and generalisations to arbitrary number of FSMs. In chapter 5, we describe the design and implementation of the compositional technique to analyse it experimentally. Chapter 6 concludes the report with a discussion of issues that still need to be resolved in future work.

Chapter 2

The Compositional Approach

We now propose a new approach for producing test sequences for communicating finite state machines. This approach works for non-interacting FSMs and a restricted set of FSMs participating in synchronous, bidirectional communication. The pitfalls and possibilities of extension of this method to a wider range of communicating FSMs are also pointed out.

The usual procedure for generating test sequences for communicating FSMs follows the path (1) in the figure 3.1 in which the cross product is first computed and then test sequences are generated for the product machine by any of the well known test generation methods for a single FSM. In contrast, the *c*ompositional approach follows the path (2) in the figure 3.1 in which the computation of the cross product machine and the resultant state explosion is avoided. The test sequences for the component machines are first generated and then combined to generate test sequences for the product machine.

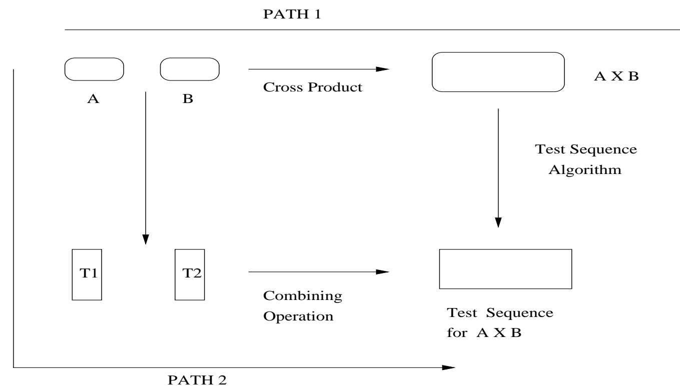

Figure 2.1: Alternative strategy for test sequence generation

2.1 Definitions

Formally, a communicating FSM, like any other FSM is a four-tuple $F = (\sum, V, \rho, s_0)$, where $\sum$ is an alphabet consisting of all of the machine's I/O operations; V is a finite set of states; $\rho : VX\sum \rightarrow 2^V$ is a deterministic state transition function; and $s_0 \in V$ is the initial state. In order to specify the conditions under which test sequences can be combined, a few more definitions are required. For the following, assume that A and B are two FSMs with the following parameters.
$I_A \cup C2$ = set of all inputs of A.
$O_A \cup C1$ = set of all outputs of A.
$I_B \cup C1$ = set of all inputs of B.
$O_B \cup C2$ = set of all outputs of B .

The symbols in $C1$ are communication symbols output by A and accepted by B as input symbols. The symbols in $C2$ are used for communication in the reverse direction, i.e., from B to A. The FSMs communicate only through the symbols of $C1$ and $C2$, and therefore $I_A \cap I_B = O_A \cap O_B = \phi$. Let T_1 and T_2 be I/O sequences in machines A and B respectively. Then a criterion for compatibility (also called full compatibility) between T_1 and T_2 is defined as follows.

- If A and B are non-interacting machines then T_1 and T_2 are always compatible.
- If A and B are synchronously communicating machines, then T_1 and T_2 are defined to be compatible iff there exists a combination of the edge-labels (I/O pairs) of T_1 and T_2 satisfying the following properties.
  1. **C1 :** The combination must contain every edge that is present in either T_1 or T_2 exactly once.
  2. **C2 :** The combination preserves the relative order of the edges occurring in T_1 and T_2.
  3. **C3 :** Every edge of the form a/x where a $\in I_A$ and x $\in C1$ is immediately succeeded by an edge of the form x/b where b $\in O_B$. Similarly every edge of the form x/b where x $\in C1$ and b $\in I_B$ is immediately preceded by an edge of the form a/x where a $\in I_A$. The symmetrically oposite condition holds for edges involving symbols of $C2$.

If M_1 and M_2 are two compatible I/O sequences, then we define an operation, Compose(M_1, M_2). This operation returns another I/O sequence.
If A and B are non-interacting machines, then Compose(M_1, M_2) is just the concatenation $M_1 @ M_2$.

If the FSMs are synchronously interacting, then Compose(M_1, M_2) is obtai ned from the I/O pairs of M_1 and M_2 as follows.
Since M_1 and M_2 are compatible, there must exist a combination of M_1 and M_2 having the properties specified in the definition of compatibility. We obtain the composed sequence from a suitable interleaving of the I/O labels in this combination. Every edge of the form a/x where a $\in I_A$ and x $\in C1$ must have an immediate successor of the form x/b where b $\in I_B$. Merge these edges into the single edge a/b. We thus iteratively merge all edges involving symbols from the set $C1$. The edges involving symbols from $C2$ are merged similiarly.

A constructive definition in the form of a simple algorithm which computes *a* composition of the sequences, Compose(T_1, T_2), if T_1 and T_2 are compatible and reports failure otherwise, is given below. Let T_1[1..n] and T_2[1..m] be the arrays which store the test-sequences to be composed. The algorithm is given as a recursive procedure *ComposeSub*(α, β) which returns the composition of the sub-sequences $T_1[\alpha$, n] and $T_2[\beta$, m]. The composition Compose(T_1, T_2) is calculated by invoking this recursive procedure with $\alpha = 1$ and $\beta = 1$. Assume that there is a procedure *Cancel(i,j)* which returns the composition of the IO pairs $T_1[i]$ and $T_2[j]$, if they are compatible and returns nil if they are not. For example, if $T_1[i] = a/x$ and $T_2[j] = x/b$ where x is a common symbol involved in the interaction between the FSMs, then Cancel(i,j) would return a/b.

Algorithm 2.1 Procedure ComposeSub(α, β)

Input : Test sub-sequences $T_1[\alpha, n]$ and $T_2[\beta, m]$
Output : Compose($T_1[\alpha, n], T_2[\beta, m]$)
Let $T_1[i]$ be the first IO pair in $T_1[\alpha$,n] which has a common symbol.
Let $T_2[j]$ be the first IO pair in $T_2[\beta$,m] which has a common symbol.
t = Cancel(i,j).
If t==nil
    Print "Not compatible"; exit(-1);
Return($T_1[\alpha..i-1]$@$T_2[\beta..j-1]$@t@ComposeSub(i+1,j+1));
/* Recursively invoke the procedure*/

Note that Algorithm 3.1 is one particular way of obtaining a composition of the sequences $T_1[\alpha$,n] and $T_2[\beta$,m]. In general, there may be more than one combination of the sequences, satisfying the conditions C1, C2 and C3. This algorithm gives us a unique way of obtaining the composition of the test sequences. The validity of any of the results stated in this report does not depend on the choice of the composed sequence. A simple property of the composition operation is the following:
Given a path labeled by p_A starting from state S_A of FSM A, a path p_B starting from state S_B of FSM B , if the state [S_A, S_B] is reachable in the cross-product and p_A and p_B are

compatible, then there exists a path labeled by Compose(p_A, p_B), starting from state [S_A, S_B] in the cross-product. This property is self-evident from the way in which the composition operation has been defined.

2.2 Composing Algorithms

It is interesting to observe that in the case of non-interacting machines if the sequences UIO(S_A) and UIO(S_B) are compatible, the composed sequence, Compose(UIO(S_A),UIO(S_B)) is a UIO sequence for the product state $[S_A, S_B]$ in the product machine, provided it is reachable. The proof of this fact is presented below as a theorem.

Theorem 2.1 Let M_3 be the combine of M_1 and M_2 where M_1 and M_2 are UIO sequences for states X and Y of *non-interacting* machines A and B respectively. If there is a path labeled M_3 starting from a state S of the product machine, it can only be [X,Y], i.e., M_3 is a UIO sequence for state S.

Proof: Since M_1 is a path starting from S_A and M_2 is a path starting from S_B, the global state [S_A, S_B] must have a path labeled by M_3, because M_3 is the composition of the sequences M_1 and M_2. Now each edge on this path causes a transition in only one of the two components of the global state. If it changes the component of A, then it corresponds to an edge with the same label in FSM A. If it causes a state change in the component corresponding to B, then there must exist a corresponding edge in the FSM B with the same edge-label.

Thus the path labeled M_3 in the product machine splits into two paths labeled M_1 and M_2 in the component machines A and B starting from the states S_A and S_B. Since M_1 and M_2 are UIO sequences for the states X and Y, it follows that S_A = X, S_B = Y By virtue of the above facts, it follows that M_3 is a UIO sequence for the state [S_A, S_B]. □

Note however that if the machines A and B are communicating synchronously, then the composed sequence of the UIO sequences may not be a UIO sequence for the product state. An example of this can be readily seen in the figure 3.2. Clearly the sequence a/x is a UIO sequence for $A1$ in machine A and the sequence x/b is a UIO sequence for the state $B1$ of machine B. But the combined sequence a/b is not a UIO sequence for the product state $[A1, B1]$ because even the state $[A2, B2]$ of the product machine exhibits the same behavior. However, a/b would be a UIO sequence for $[A1, B1]$ if the state $[A2, B2]$ were unreachable.

In a few special cases, if the sequences UIO(S_A) and UIO(S_B) are compatible, the composed sequence, Compose(UIO(S_A),UIO(S_B)) is a UIO sequence for the product state $[S_A, S_B]$ in the product machine. Our interest in the preservation of UIO property on composition

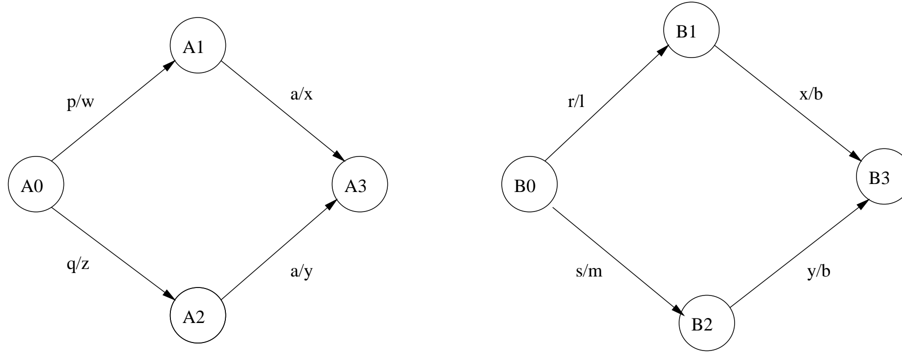

Figure 2.2: Composed sequence of two UIO sequences

stems from the intention to obtain the UIO sequences for states in the product FSM system by composition of UIO sequences of the corresponding constituent component FSM states. The UIO property is preserved on composition if the machines A and B satisfy the following constraint which we shall call as the *Communication Constraint* or CC in the rest of the discussion:

Definition 1: For all $a \in I_A$ and $b \in O_B$, if a/x_1 and a/x_2 are edge labels for some two transitions in FSM A, and and x_1/b and x_2/b are edge labels of some two transitions in FSM B, where x_1 and x_2 are common symbols, then it must be the case that $x_1 = x_2$. Symmetrically, for all $c \in I_B$ and $d \in O_A$, if c/y_1 and c/y_2 are edge labels for some two transitions in FSM B, and and y_1/d and y_2/d are edge labels of some two transitions in FSM A, where y_1 and y_2 are common symbols, then it must be the case that $y_1 = y_2$.

2.3 Projection Strings

To prove the UIO preserving property of the composition operation, when the machines A and B satisfy the communication constraint, we define a Projection of a path in the global finite state machine. Let M be any path in the global FSM of lenght n starting at state $[X_0, Y_0]$ and ending at state $[X_n, Y_n]$, with every r-length prefix of M taking the machine from $[X_0, Y_0]$ to $[X_r, Y_r]$. We define two strings called Projection strings, depicting the local behaviors of the machines A and B. For every r-length prefix of $M(r \leq n)$ define the Projection strings M_A^r and M_B^r as follows:

1. If $r = 0$, then $M_A^r = M_B^r = nil$.

2. If $r \neq 0$ then recursively obtain the Projection strings M_A^{r-1} and M_B^{r-1}. Look at the rth I/O pair p_r/q_r.
   If $p_r \in I_A$ and $q_r \in O_A$, then $M_A^r = {M_A}^{r-1}$ @ $p_r/q_r, M_B^r = M_B^{r-1}$;

If $p_r \in I_B$ and $q_r \in O_B$ then $M_B^r = M_B^{r-1}@\ p_r/q_r, M_A^r = M_A^{r-1}$;
If $p_r \in I_A$ and $q_r \in O_B$, then there must be an x such that $x \in C1$ and in FSM A, $X_{r-1} \rightarrow X_r$ on p_r/x; and in FSM B, $Y_{r-1} \rightarrow Yr$ on x/q_r. Choose any such x and define $M_A^r = M_A^{r-1}\ @\ p_r/x$, $M_B^r = M_B^{r-1}\ @\ x/q_r$;
If $p_r \in I_B$ and $q_r \in O_A$, then there must be an y such that $y \in C2$ and in FSM B, $Y_{r-1} \rightarrow Y_r$ on p_r/y; and in FSM A, $X_{r-1} \rightarrow X_r$ on y/q_r. Choose any such y and define $M_A^r = M_A^{r-1}\ @\ y/q_r$, $M_B^r = M_B^{r-1}\ @\ p_r/y$;

The following two properties can be easily verified to be true from the definition of projection strings given above:
(i) If M_A^n and M_B^n are the projection strings of a path M in the global machine, then $Compose(M_A, M_B) = M$.
(ii) It also follows that if M_1 and M_2 are compatible paths paths in the machines A and B respectively , and $M = Compose(M_1, M_2)$, then $[M_1, M_2]$ itself is a possible pair of projection strings (though not necessarily unique) for M and this pair can be obtained by a *suitable* application of the constructive definition given above.
Note that the definition of projection strings itself is independent of whether or not the component machines satisfy the communication constraint or not. We introduce this constraint in the folowing lemma to obatin a more useful result.

Lemma 2.2 *Let the given Mealy machines A and B satisfy the Communication Constraint specified above. For any path labeled M in the product machine A X B starting at state $[X, Y]$ and ending at state $[X_n, Y_n]$, there exists a unique pair of projection strings $[M_A^n, M_B^n]$ such that $X \rightarrow X_n$ on M_A^n in machine A and $Y \rightarrow Y_n$ on M_B^n in machine B.*

Proof: For proving the above result we make use of induction. The inductive hypothesis is the following: If the FSM is started in state $[X, Y]$ and goes to $[X_r, Y_r]$ on the path corresponding to the r-length prefix of M, then there must exist a unique pair of projection strings $[M_A^r, M_B^r]$ for the r-length prefix of M such that M_A^r is a path from X to X_r in FSM A and M_B^r is a path from Y to Y_r in FSM B respectively. This is vacuously true for $r = 0$.
Assume the result is true for $r = k$. We wish to prove it for $r = k + 1$;
Suppose $[X, Y] \rightarrow [X_k, Y_k] \rightarrow [X_{k+1}, Y_{k+1}]$ on the $k + 1^{th}$ length prefix of M. Let $[M_A^k, M_B^k]$ be the unique pair of projection strings of the k-length prefix of M. Thus

$$X \rightarrow X_k \text{ on } M_A^k \text{ and } Y \rightarrow Y_k \text{ on } M_B^k$$

Consider the $(k + 1)$'th symbol p_{k+1}/q_{k+1}. There are three cases:
Case 1: If $p_{k+1} \in I_A$ and $q_{k+1} \in O_A$, then by the definitions of the Projection strings,

$$M_A^{k+1} = M_A^k\ @\ p_{k+1}/q_{k+1} \text{ and } M_B^{k+1} = M_B^k$$

Clearly $X_k \rightarrow X_{k+1}$ on p_{k+1}/q_{k+1} and $Y_{k+1} = Y_k$.
Since $[M_A^k, M_B^k]$ is a unique pair of projection strings for the k-lenght prefix of M, it follows

from the above that $[M_A^{k+1}, M_B^{k+1}]$ is a unique pair of projection strings for the $(k+1)$-length prefix of M such that

$$\text{X} \rightarrow X_{k+1} \text{ on } M_A^{k+1} \text{ and Y} \rightarrow Y_{k+1} \text{ on } M_B^{k+1}$$

Case 2

If $p_{k+1} \in I_B$ and $q_{k+1} \in O_B$, the proof is similar

Case 3

If $p_{k+1} \in I_A$ and $q_{k+1} \in O_B$, then there must exist an $x \in C1$ such that $X_k \rightarrow X_{k+1}$ on p_{k+1}/x and $Y_k \rightarrow Y_{k+1}$ on x/q_{k+1}.

The Communication Constraint ensures that this x is unique.

Thus, by the uniqueness of x and the definition of projection strings:

$$M_A^{k+1} = M_A^k \text{ @ } p_{k+1}/x \text{ and } M_B^{k+1} = M_B^k \text{ @ } x/q_{k+1}$$

Since $[M_A^k, M_B^k]$, the pair of projection strings for the k-length prefix of M is unique, it follows from the above that, for a unique pair of projection strings $[M_A^{k+1}, M_B^{k+1}]$:

$$\text{X} \rightarrow X_{k+1} \text{ on } M_A^{k+1} \text{ and} \qquad \text{Y} \rightarrow Y_{k+1} \text{ on } M_B^{k+1}$$

Case 4

$p_{k+1} \in I_B$ and $q_{k+1} \in O_A$. The proof for this case is similar to case 3.

The following theorem establishes a result on the preservation of the UIO property in the global FSM on composition of compatible UIO sequences in the component FSMs.

Theorem 2.3 *: If M_1 and M_2 are UIO sequences of states X and Y of FSMs A and B, M_1 and M_2 are compatible and A and B satisfy the criterion (*) then t he composed sequence M =Compose(M_1, M_2) is a UIO sequence for the state [X,Y] in the FSM A X B if it is reachable.*

Proof: Assume on the contrary that M isn't a UIO sequence for the global state $[X, Y]$. Then, there exists some global state $[X'], Y']$ such that it shows the same I/O behaviour as $[X, Y]$. From Lemma 1, it follows that the global path M generates a unique pair of projection strings onto the component machines A and B. This combined with fact(ii) following the definition of Projection strings implies that $[M_1, M_2]$ is the only pair of paths in the component machines such that $Compose(M_1, M_2) = M$ and X' exhibits the I/O behaviour M_1 and Y' exhibits the I/O behaviour M_2 in machines A and B respecyively. But this means that both X and $[X']$ in FSM A shows the same I/O behaviour as the projection string M_1 which is in contradiction with the UIO property of M_1.

2.4 Composing Algorithms

Now we describe how to obtain test sequences for the product machine by combining test sequences for the component machines. The problem of test case generation using UIO sequences for every edge can be broken up into the the following two sub-problems:

(i) Generating a path to the head state of the edge to be tested. (ii) Generating a UIO sequence for the tail state of that edge.

Before giving the general algorithm, we illustrate for the special case of non-interacting FSMs, how the compositional technique can be applied. Let A and B be two non-interacting Finite State machines. We assume that we already have available using conventional methods for a single FSM, test sequences for the individual component machines A and B. Thus, for each transition in the component machines we maintain the following information:

1. The state transition $S_1 \rightarrow S_2$ which the sequence tests.
2. An index for the prefix of the sequence which brings the machine A to state S_1. This will be denoted by P(S_1). Since, this is usually computed by the Dijkstra's shortest path algorithm, we shall also refer to it as "Dijkstra path".
3. UIO(S_2) This is the UIO signature for state S_2 computed by the UIO algorithm.

Let $[S_1^A,\ S_1^B] \rightarrow [S_2^A, S_2^B]$ be the transition to be tested in the product machine, and let E be its edge I/O label. Let N_1, N_2 and N_3 be empty sequences to start with. Algorithm 3.2 describes a method to generate the test sequence for the global transition given the test sequences for the local transitions.

Algorithm 2.2 Test generation for non-interacting machines

Input : Test sequences for transitions $S_1^A \rightarrow S_2^B$ and $S_1^B \rightarrow S_2^B$
Output : Test sequence for the transition $[S_1^A, S_1^B] \rightarrow [S_2^A, S_2^B]$
N_1 = compose(P(S_1^A), P(S_1^B));
$N_2 = E$;
N_3 = compose(UIO(S_2^A), UIO(S_2^B));

2.4.1 Synchronously interacting Machines

In this case, test sequence generation by composing will give valid test sequences provided the machines A and B satisfy communication constraint and the test sequences are combinable. The following approach is adopted for test case generation for a transition $[S_A^1, S_B^1] \rightarrow [S_A^2, S_B^2]$, in the concurrent FSM system, consisting of synchronously communicating FSMs A and B satisfying the communication constraint:

1. Obtain a pair of compatible paths p_A and p_B such that p_A is a path in FSM A from S_A^0 to S_B^1 and p_B is a path in FSM B from S_B^0 to S_B^1, where S_A^0 and S_B^0 are the initial states of the FSMs A and B respectively. Let P = Compose(p_A, p_B).

2. Similarly, compute a pair of compatible UIOs u_A and u_B for the states S_A^2 and S_B^2 in FSMs A and B respectively. Let $U = \text{Compose}(u_A, u_B)$.

3. The test sequence is given by concatenating P, the edge label to be tested and U.

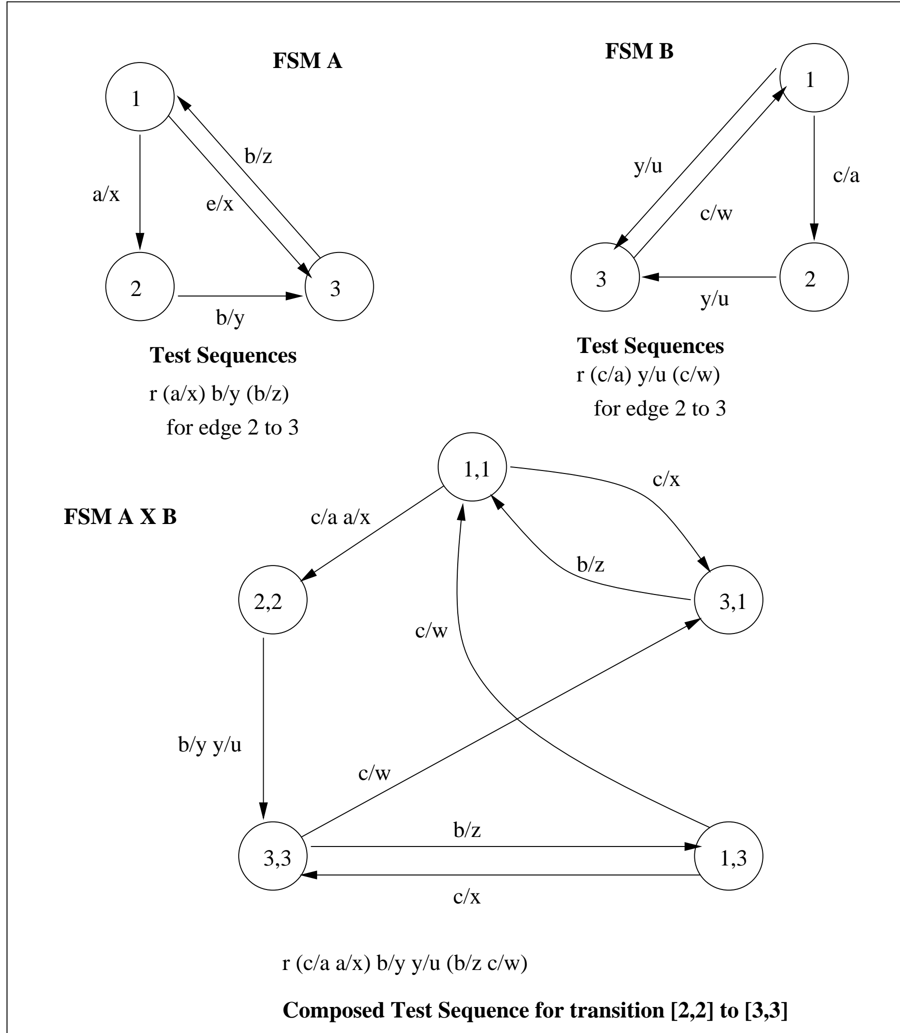

Figure 2.3: Composed sequence of two UIO sequences

As a motivational example of the compositional approach consider the FSMs depicted in figure 3.3 Sequence **r (a/x) b/y (b/z)** tests transition b/y from state 2 to 3 of FSM A. a/x takes the FSM A to state 2, b/y is the transition to be tested and b/z is the UIO sequence for state 3.

r (c/a) y/u (c/w) tests transition y/u from state 2 to state 3 of FSM B.

c/a takes the FSM to state 2, y/u is the transition to be tested and c/w is the UIO sequence for state 3.

By following the approach indicated above, we obtain the composed sequence, **r (c/x) b/u**

(b/z c/w) which tests the transition from state [2,2] to state [3,3] in the product machine. Observe that in this example, a pair of compatible paths to the head state of the transition and a pair of compatible UIO sequences for the tail state were readily available.

Chapter 3

Constrained Synchronous Communication

In this chapter we present an algorithm to compute compatible UIO sequenes in the component FSMs that can be composed as illustrated in the previous chapter to obtain UIO test sequences for the states of the global FSM. Bidirectional synchronous communication is assumed to be the mode of interaction. The following are the hurdles to be tackled in the compositional approach:
Given the edges $S_1^A \rightarrow S_2^A$ in FSM A and $S_1^B \rightarrow S_2^B$ in FSM B and the transition $[S_1^A, S_1^B] \rightarrow [S_2^A, S_2^B]$ for which we wish to obtain a test sequence,

1. ***Path to the Head state*** : The Dijkstra paths for S_1^A and S_2^B in the component may be incompatible, because of which we have to find an alternate method for constructing a path to the global state $[S_1^A, S_1^B]$, if it is reachable in the cross-product.

2. ***Incompatibility of UIO sequences*** : The UIO sequences UIO(S_2^A) and UIO(S_2^B) may be incompatible because of which we have to look for another way to obtain the UIO sequences for the global state $[S_2^A, S_2^B]$.

3. ***UIO property preservation*** : Compatible UIO sequences may not retain their UIO property in the concurrent FSM system on composition.

The first of the hurdles listed above is the well known reachability problem that is intractable due to the problem of state explosion.
Proposition 1: The following problems are PSPACE complete: i) Is a state or transition of the composite machine reachable from the initial state? ii) Given two states of a composite machine, is one state reachable from the other through internal transitions only?

The problem of obtaining compatible UIO sequences in the component FSMs is solved by performing an exhaustive search over all possible *minimal* UIO sequences for every pair

of states in the individual FSMs. For computing all possible minimal UIOs for a given state we use the method of UIO trees described in [Nai97]. We then give an $O(n^2)$ time algorithm to compute a compatible path in the other FSM for a given path in one of the FSMs. Our interest in this stems from the fact that the UIO sequences in the individual FSMs that we are trying to combine may have a different number of communicating transitions. This point will be made more precise in section 4.2.

The problem of preservation of the UIO property on composition of compatible UIO sequences is avoided by considering only those classes of FSMs, which satisfy a communication constraint, that is given below. In the rest of this chapter we assume that the given pair of FSMs obey the communication constraint. A Heuristic method for the general case will however be given in the next chapter.

In the rest of this chapter we assume A and B to be two synchronously interacting finite state machines, satisfying the following conditions.

- *A and B are completely specified.*
- *A and B are deterministic.*
- *Reduced*
- *Satisfy the following Communication Constraint.*

3.1 Path Compatibility

In contrast to the special case of independent FSMs, an exhaustive search for compatible paths in the individual FSMs is necessitated because of the problem of incompatibility. This can be done by constructing a shortest path tree with the initial global state as the root and all reachable global states as its nodes. This in the worst case will take time exponential in the number of interacting FSMs. This is precisely the state explosion problem. However, we concentrate on moderately sized protocols in which the number of reachable product states is manageable and the bottleneck lies in the computation of UIO sequences on the bulky product machine. Assumptions like reachability of every state in the individual FSMs through non communicating transitions only help trivialise this problem. Heuristics and random walk approaches [Lee96] are better suited for tackling this problem. For the case of just two FSMs under discussion, the naive approach is reasonable.

From the definition of projection and composition it is clear that if p is a UIO sequence for the global state $[S_A, S_B]$ of the FSM AXB, then the projections p_A and p_B are UIO sequences for the states S_A and S_B. Thus, if state $[S_A, S_B]$ has a UIO sequence, there must exist two UIO sequences, p_A and p_B for states S_A and S_B which are compatible and whose composition results in the UIO sequence for $[S_A, S_B]$.

This suggests a strategy of constructing a UIO sequence for state $[S_A, S_B]$ by looking for UIO sequences for S_A and S_B which are compatible and then taking their composition. Observe that if p is a UIO sequence for state S of FSM M, then any extension of p by a non-empty valid string of I/O pairs is also a UIO for state S. Therefore the number of possible UIO sequences for any state is infinite, if a cycle is reachable from it in the digraph representation of the FSM. We only consider all possible pairs of $m$inimal UIO sequences.

We mention a theorem that suggests a method to compute a UIO sequence for the global state $[S_A, S_B]$, given all possible minimal UIO sequences of S_A and S_B (generated by the UIO tree method).

Theorem 3.1 *If there is a UIO sequence 'M' for state $[S_A, S_B]$ in AXB, then there exist two sequences p_A and p_B such that p_A is a minimal UIO sequence for S_A, p_B is a minimal UIO sequence for state S_B and either p_A is compatible with a prefix q_B of p_B or p_B is compatible with a prefix q_A of p_A.*

Proof : Let the length of M be n. Since M is a UIO sequence for state $[S_A, S_B]$, the projections M_A^n and M_B^n are compatible and are UIO for states S_A and S_B respectively. Also, for each i, the projections M_A^i and M_B^i are compatible. (Refer to Chapter 3)

Let p_A denote the shortest length prefix of M_A^n which is also UIO for state S_A and let p_B denote the shortest prefix of M_B^n which is UIO for S_B.

Let i be the smallest integer ($i \leq n$) such that $M_A^i = p_A$ and let j be the smallest integer ($j \leq n$) such that $M_B^j = p_B$.

If ($i \leq j$), then p_A is compatible with M_B^i which is a prefix of $M_B^j = p_B$.

If ($j < i$) then p_B is compatible with M_A^j which is a prefix of $M_A^i = p_A$. Hence the theorem. □

Thus, we go about constructing UIO sequences for the states in the product machine by first computing a pair of partially compatible UIO sequences in the individual FSMs. Let p_A be the UIO with greater number of communicating transitions. Then we compute a path say q_B in FSM B that is compatible with the incompatible suffix of p_A, so that the concatenated sequence $p_B@q_B$ is fully compatible with p_A.

Next, we present an algorithm (algorithm 3.2) to compute a path p_B in say FSM B that is compatible with a given path p_A in FSM A. As has been indicated above, our interest in this stems from the need to compute a pair of compatible UIO sequences given a pair of partially compatible UIO sequences.

In the following $T[\alpha, \beta]$ denotes a path composed of the transitions $T[\alpha]$, $T[\alpha + 1]$,...$T[\beta]$. As before $Apply(s, x)$ returns the state obtained by performing the transition defined by input x in state s. Apply(S, x) returns the set of states reachable from any of the states in S on input x.

Similarly we define another function $Generate(s, y)$ as the set of all states that are reachable from state s through a single transition outputting y.
Generate(S, y) for a set of states S is analogously defined as the set of states reachable from any of the states in S through a single transition with output y.

The algorithm also uses a function called $NCTreachable(s)$ that returns the set of all states reachable through paths totally devoid of of communicating transitions(CTs) or common symbols for a given state s in either FSM. This function may be simply computed by a breadth first search starting from state s as the root in the corresponding FSM.

Similarly we define NCTreachable$(\{S\}$ for a set of states S as the set of states reachable from any state in S along NCTs alone.

We define an array Reachable$[1...k]$ of size k such that Reachable[i] holds the set of states in which FSM B could possibly be after traversing a path compatible with the $i-1$ lenght prefix of the given path in FSM A. The array Transition[i] of the same size holds the set of states in which FSM B could possibly be after traversing some path compatible with the $i-$length prefix of the given path in FSM A, and ending in a communicating edge.

***E*xplanation** : It is necessary to perform a complete search in FSM B for some compatible path. Since NCTs in one FSM don't effect a state transition in the other FSM, we are maintaining the set Reachable[i] which is the set of all states in which FSM B could possibly be along some compatible path, just before the application of the ith transition in FSM A. Transition$[i]$ is the set of all states whch FSM B could possibly reach by making a transition along an arbitrary edge in FSM B that is compatible to the edge T$[i]$ in FSM A. Observe that there is excatly one transition from a given state in FSM B that is compatible to a transition of the form a/x in FSM A, where x is a common symbol and $b \in O_B$, whereas there could be several or no transitions from a given state in FSM B compatible to a transition of the form y/b in FSM A, where y is a common symbol. Therefore, for any state s, Apply(s, x) returns one state, whereas Generate(s, y) returns a set of states.

3.2 Computaion of UIO sequences

Based on the results obtained in the previous section, we propose an approach for constructing UIO sequences for the states of the product machine using the UIO trees of the FSMs A and B.

First we construct the UIO trees U_A and U_B for the FSMs A and B respectively. For every state S_A of FSM A, maintain a pointer to all singleton leaf-nodes of U_A having S_A as the initial vector. All such nodes represent shortest length UIO sequences for state S_A. Similarly, for each state S_B of FSM B, maintain pointer s to all singleton leaf-nodes, of U_B having S_B as the initial vector. The procedure to obtain UIO sequences for states of the product machine

is described in algorithm 3.3. Algorithm 3.3 attempts to construct a UIO sequence for the product-state $[S_A, S_B]$, by finding a compatible pair of UIO sequences, for S_A andS_B.

This algorithm is similar to a $cross - product$ construction of the UIO trees of A and B, with the nodes treated as states and the edge labels as transitions, because we are looking for paths in the UIO trees, which are compatible and hence can be composed. Since, the UIO tree contains only a subset of the total paths in the FSM, we expect this computation to be less expensive.

3.3 Complexity Analysis

In this section we investigate the impact of the new method on the test sequence generation process in terms of space and time efficiency. In the following discussion, we use the following notation :

- n_A and n_B denote the number of states in the machines A and B respectively.
- E_A and E_B denote the number of edges in the machines A and B respectively.
- d^A_{max} and d^B_{max} stand for the maximum degree of a vertex in the machines A and B respectively.

3.3.1 Space Complexity Comparision

The conventional method of generating test cases for a concurrent FSM system computes the cross product of the individual FSMs and then uses the method proposed in [SD88] to obtain UIO sequences. We can compare the space complexities of the two methods as described below.

FSM storage

The storage of the transition relation of the two communicating machines A and B consumes the storage equivalent of $E = |E_A| + |E_B|$, where E_A and E_B are the number of edges in the machines A and B respectively. This minimum storage requirement is demanded by both the conventional method and the compositional approach.
However, the conventional method necessitates the storage of the transition relation of the cross product machine. It may be observed that in the worst case (i.e., when the FSMs are non-interacting) each edge in either FSM is replicated as many times as there are states in the other FSM. Thus, the upper bound on the number of edges in the product FSM and therefore the space required for storing the product FSM is $|E_X| = n_B.|E_A| + n_A.|E_B|$

Dijkstra Path computation

An inspection of the standard Dijkstra algorithm for computing shortest paths to states in the global FSM shows that it has a space complexity of O($n_A n_B$), where n_A and n_B are the number of states in the individual FSMs, since the number of reachable nodes in the Dijkstra tree could be as large as $n_A n_B$ in the worst case.

UIO computation

The Compositional Approach

- **UIO trees**: In our method the UIO trees for the interacting FSMs are computed separately. We are generating UIO trees in the hope of obtaining all minimal UIO sequences of each FSM. However we stop growing the trees beyond a depth of $2n^2$ as in [SD88]'s method. In the UIO tree, for FSM A say, at each level, each node could lead out to a maximum of d^A_{max} other nodes. Therefore, the UIO tree thus generated could potentially have as many as ${d^A_{max}}^{2n_A^2}$ nodes. Thus the storage requirement for generating UIO trees is bounded by M_{UIO} = maximum$({d^A_{max}}^{2n_A^2}, {d^B_{max}}^{2n_B^2})$. Finally, to obtain a pair of partially compatible minimal UIO sequences for the product state $[S_A, S_B]$ we may have to try out all possible combinations of minimal UIOs for the states S_A and S_B in the individual FSMs. However, since we will be trying to test only one pair at a time for compatibility, this doesn't introduce any additional memory costs.

- **Compatible path computation - Algorithm 3.2**: The following are the sets used in the algorithm:

  1. Reachable[i], i = 1,...,k, where k is the length of the path for which we are trying to obtain a compatible path in the other FSM.
  2. Transition[i], i = 1,...,k.
  3. NCTreachable(s_i), i = 1,...,n_B.

  We represent each of these sets as an array of size n_B. The justification of this will be given in the time complexity analysis of this algorithm. The space complexity of the algorithm is thus given by the sum of the amounts of storage required for these sets. Space required for maintaining NCTreachable(s) for each of the n_B states = O(n_B^2). Space required for maintaining the array Reachable[i], of size k = O(kn_B). Thus, the total space complexity is given by O($n_B^2 + kn$). Observe that k may be as large as $2n^2$, since we are allowing UIOs/state signatures as long as $2n^2$. Therefore, in the worst case computing a compatible path may take as much as $M_{comp} = O(n^3)$ space.

  However, we can consider the pairs of minimal UIOs for compatibility test strictly in increasing order of length without any additional increase in the space complexity. So,

unless there is no pair of compatible UIOs with the length of the UIO with greater number of common transitions less than $2n^2$, this upper bound will not be reached. Of course, even in this worst case the conventional method will be far costlier!

Thus, the total cost for the UIO computation step in the conventional approach is: Space required for storing UIO trees + Space required for computing compatible paths $= M_{UIO} + M_{comp} = \text{maximum}({d_{max}^A}^{2n_A^2}, {d_{max}^B}^{2n_B^2}) + O(n^3) = M_{UIO}$, since the second term doesn't make a difference to the order.

The Conventional Approach

Computation of UIO sequences by the conventional method proceeds by first taking the cross-product and then applying the procedure described in [SD88]. Since the product machine could have as many as $n_A n_B$ states, and the maximum degree could be as large $(d_{max}^A + d_{max}^B)$ the memory requirement for this method is bounded above by

$M_{UIO}^{'} = (d_{max}^A + d_{max}^B)^{2n_A^2 n_B^2}$

Of course, even this is a rather pessimistic bound and is attained only when the FSMs are non-interacting and all sequences of length less than $2n^2$ have to be considered before arriving at a UIO for each state.

Thus, the cumulative cost of the compositional approach is given by the sum of the space requirements for FSM storage, Dijkstra path tree computation , and the UIO computation method, i.e.,

$O(|E_A + E_B|) + O(n_A n_B) + O(M_{UIO}) = O(M_{UIO})$.

The total memory cost for the conventional approach is determined by the costs of the storage of the product FSM and the UIO tree generation step i.e.,

$O(|E_X|) + M_{UIO}^{'} = O(M_{UIO}^{'})$

From the above, it is clear that the cumulative cost of memory usage in our method, M_{UIO} *is significantly smaller than the usage for the conventional method, which is* $M_{UIO}^{'}$.

Observe that even in the conventional approach, the cost of storing the product FSM doesn't form a significant part of the total memory requirement. This is because the computation of UIO sequences for the states of an FSM itself is a PSPACE complete problem and occupies a significantly higher amount of memory. In fact there are machines whose states have UIO sequences only of exponential length, as proved by Yannakakis and Lee in [LY94]). The UIO sequence generation method given in [SD88] stops if the tree attains a depth of $2n^2$. The resulting UIO/state signature could be as long as $2n^2$. State explosion due to cross product computation is avoided in the sense that the UIO sequence generation method isn't used for the product FSM and interesting properties of concurrency are being exploited to generate test case more efficiently. Computations related the PSPACE complete problem of finding UIO sequences are restricted to the component FSMs.

3.3.2 Time Complexity

- The time complexity of computing UIO sequences by the procedure described in [SD88] is $O(n^2 d_{max}^{2n^2+2})$, where n is the number of states in the given FSM and d_{max} is the maximum degree of a vertex in the machine.

- The time complexity of Dijkstra path tree computation algorithm is $O((n_A n_B)(|I_A| + |I_B|))$ since every node is visited exactly once and time proportional to $|I_A| + I_B|$ is spent in computing its neighbours.

- The time complexity of Algorithm 3.2 to compute compatible paths can be calculated as follows:
  Let n be the size of the FSM in which we wish to obtain a compatible path. NCTreachable(s) takes $O(n^2 d)$ time. Since this is a costly step, we calculate and store NCTreachable(s) for each state in a precomputation step. However, then NCTreachable(S) of a set of states S will involve taking the union of all NCTreachable(s_i), $s_i \in S$. This could potentially take $O(n^2)$ steps if the sets were defined as a list of states. We could however represent a set of states as a bit vector of size 'n' which has a 1 in the ith place if state s_i is a member of the set and 0 otherwise.

  Then, computation of NCTreachable(S) for a set of states S is also possible in just $O(n)$ time.
  Apply(S, i) has a time complexity of $O(n)$.
  Generate(S, i) has a time complexity of $O(nd)$, where d is the maximum degree of the FSM under consideration.

  Thus each iteration of the first for loop in the algorithm can be performed in $O(nd)$ time. It's easily seen that the reconstruction step can be performed in $O(n^2)$ time. Thus, the time complexity of Algorithm 3.2 is given by
  the sum of the time complexities of the Precomputation, Looping and Reconstruction steps, i.e., $O(n^2 d + n^2 d + n^2) = O(n^2 d)$.

The Compositional Approach

The total cost of computing test sequences by our method can be computed as follows:

1. The cost of computing the UIO sequences for the states of the machines A and B is $n_A^2 {d_{max}^A}^{2n_A^2+2} + n_A^2 {d_{max}^B}^{2n_B^2+2}$.

2. The total number of reachable states in the product machine could be as large as $O(n_A n_B)$. We will have to compute Dijkstra paths to each of those states. The time complexity of creating a Dijkstra tree has already been shown to be $O(n_A n_B(|I_A| + |I_B|)$

in section 3.1. Observe that we need not perform a breadth first search of the Dijkstra tree for every product state. We will consider states in the product machine for Dijkstra path generation in the order in which they appear along a breadth first search. Thus, the time complexity of generating Dijkstra paths to all reachable states in the product machine is $O(n_A n_B(|I_A| + |I_B|)$.

3. Next, we obtain an upper bound on the time required to obtain a pair of partially compatible UIO sequences from the UIO trees of A and B. The number of paths in the UIO tree of say FSM A could be as large as $O({d^A_{max}}^{2n^2})$. Thus, performing an exhaustive search over all possible pairs of minimal UIOs in the two FSMs could potentially take $O(2n^2).O({d^A_{max}}^{2n^2}{d^B_{max}}^{2n^2})$, since the time taken to test for partial compatibility is linear in the length of the UIOs which are bounded by $2n^2$. On finding a partially compatible pair we also have to find a compatible *tail* in the UIO with lesser number of communicating transitions. This has already been shown to be of time complexity $O(n^2 d_{max})$ in the analysis of Algorithm 3.2, where $n = max(n_A, n_B)$ and $d_{max} = max(d^A_{max}, d^B_{max})$. Thus, the total complexity of the UIO sequence computation for all product states is bounded by $O((n^2 d_{max}).(2n^2)({d^A_{max}}^{2n^2}{d^B_{max}}^{2n^2}))$, which is also $O(n^4 d_{max}^{(4n^2+1)})$.

Thus, the total time required for test sequence generation for all reachable states in the product machine is bounded by
$O([n_A^2 {d^A_{max}}^{2n_A^2+2} + n_A^2 {d^A_{max}}^{2n_A^2+2}] + [(n_A n_B(|I_A| + |I_B|)] + [(n^4 d_{max}^{(4n^2+1)})])$
which because of the bulky last term may be written simply as $O(n^4 d_{max}^{(4n^2+1)})$. Thus, we see that the time complexity of our method is determined essentially by the step involving the exhaustive search of all possible UIOs for compatibility.

The Conventional Approach

The time complexity of the conventional method is determined by the following costs:

1. The cost of *computing cross-product* is proportional to the number of edges in the product machine and hence is bounded above by $|E^x| = n_A|E_B| + n_B|E_A|$. Note again that this upper bound is only attained when the machines are non-interacting.

2. The cost of *computing UIO sequences* for the product machine by procedure in [SD88] is bounded above by
$O(n_A^2 n_B^2 (d^A_{max} + d^B_{max})^{(2n_A^2 n_B^2+2)})$.
This is based on the estimate that the maximum degree of a vertex in the cross-product is the sum of the maximum degrees of vertices in A and B respectively.

3. The cost of *obtaining the test-sequences* after the UIO sequences have been computed is $n^2|E^x| = n_A^2 n_B^2 (n_A|E_B| + n_B|E_A|)$

Therefore the cumulative cost of test-generation by the conventional method is
$n_A|E_B| + n_B|E_A| + n_A^2 n_B^2 (d^A_{max} + d^B_{max})^{(2n_A^2 n_B^2+2)} + n_A^2 n_B^2 (n_A|E_B| + n_B|E_A|)$
It is clear that the middle term directs the time complexity and could be as large as $O(n^4 {d_{max}}^{(2n^4+2)})$

By comparing the expressions for the conventional method and our method, it can be seen that the time involved in test generation for our method is considerably shorter than the conventional method. In our method d_{max} is raised to the power of n^2 whereas in the conventional method it is raised to the power of n^4.

3.4 Length of Test sequences

The length of the test sequence determines the runtime of the test experiment. Therefore, it is essential to minimise the length of the test sequences.

The conventional method always generates the shortest possible UIO sequence for every product state. This is because the UIO method proceeds in a shortest-first manner in which a sequence of length 'm' is considered only after all sequences of length less than 'm' have been considered.

However, our method aims at finding *some* pair of partially compatible UIO sequences, because of which we may not close in on the *best* such pair. Suppose, we make the following modifications in our method:

1. Instead of stopping at the first compatible pair of minimal UIO sequences, try out all possible pairs of minimal UIO sequences and select the best pair.

2. Make Algorithm 3.2 return the shortest possible compatible path.

Both of the above modifications may be made without changing the worst case complexity of our method. In Algorithm 3.2 to find a compatible path, we can store the path lengths as well to the states in NCTreachable(s). In the path reconstruction step at every iteration choose the state that yields the shortest NCT path. Neither of the above modifications affect the worst case complexity of our method because this is exactly what we assume would have to be done in the worst case. However, the average case complexity becomes equal to the worst case complexity with these modifications, especially because of modification 1. Modification 2, i.e., tinkering with Algorithm 3.2 to always give the worst case time complexity shouldn't worry us since the total time complexity is anyway determined by the term corresponding to the search of a partially compatible pair of UIOs.

Claim: With the above modifications, the compositional method also always generates the shortest possible UIO sequences for states in the product machine.

Proof: Note that any UIO in the product machine projects itself as UIOs in the individual FSMs. Suppose the shortest possible UIO sequence for some state $[S_A, S_B]$ in the product machine is P. Then its projections P_A and P_B are UIOs for the states A and B in FSM A and FSM B respectively. We claim that both P_A and P_B are minimal UIO sequences in the individual FSMs. To see this assume the contrary. Let q_A and q_B be the minimal UIOs corresponding to P_A and P_B respectively. They are partially compatible. Combine q_A and q_B using Algorithm 3.2 for finding a compatible path in the FSM with fewer communicating transitions. Call this combine as Q. Because of the communication constraint Q is a UIO for $[S_A, S_B]$ Since Algorithm 3.2 returns the shortest possible compatible path, it is clear that Q is strictly smaller than P which contradicts our initial assumption that P is the smallest UIO for the product state $[S_A, S_B]$. This contradiction proves that both P_A and P_B are minimal UIOs which means that they can always be detected by our modified method performing an exhaustive search over all possible minimal UIOs for S_A and S_B. Hence the claim. □

With or without these modifications, let's calculate the length of the combined UIO for the product state by our method. Let p_A and p_B be two partially compatible minimal UIOs. Suppose that p_A has the greater number of communicating transitions. Then we have to find a compatible *tail* for p_B. However, it is possible that the suffix of p_A for which we wish to obtain a compatible path in FSM B may have all its transitions as communicating ones. Let the length of this suffix be k. Even worse, every pair of consecutive communicating transitions in FSM B thrown up by Algorithm 3.2 for corresponding CTs in the suffix of p_A could have an NCT path of length $n_B - 1$ between them. In this case the length of the compatible path will be $k(n_B - 1)$. Each of p_A and p_B could be of length $2n^2$. Hence, k could be as large as $2n^2 - 1$, (when say, p_A is of length $2n^2$ and p_B is of unit length). The length L of the combined UIO sequence by our method is thus given in the worst case by:

$L = p_A + (2n^2 - k) + k + k(n_B - 1)$

$\leq 2n^2 + (2n^2 - k) + kn$

Thus, the maximum value of L could be as large as $O(n^3)$.

Of course, this is a very pessimistic calculation. In most case length of the combined UIO obtained by our method will be far lesser than this. In any case we could adopt a strategy of either discarding or chopping down the UIO thus computed to some acceptable bound l_b and satisfy ourselves with the *state signature* thereby obtained. In case we are not able to find any UIO of length smaller than l_b by our method, we can assert that there exists no UIO for that product state smaller than l_b, and so even the conventional method will only yield a state signature.

The conventional method will chop down the UIO sequence of the product state if it exceeds a length of $2(n_A n_B)^2$, since the number of states in the product machine cannot exceed $n_A n_B$. Thus, the length of the UIOs computed by the conventional method are

bounded by $O(n^4)$.

Note that the Dijkstra paths computed by our method are the shortest possible. But, since their length is bounded by n, they don't affect the order of the length calculated above.

3.5 Conclusion

In this chapter we have proposed a method to generate test sequences for detecting transition faults in a concurrent FSM system consisting of a pair of two synchronously interacting FSMs obeying the communication constraint. The problem of testing an edge transition in the product FSM system was split into the following three parts (as in [Vem98]):

1. Computation of Dijkstra paths to the initial state of the transition
2. Performing a search for a pair of partially compatible minimal UIO sequences.
3. Finding a compatible tail in the FSM with the UIO sequence with lesser number of communicating transitions.

We saw that the composition method is far superior to the conventional method in terms of space and time complexity when the communication constraint is obeyed by the interacting FSMs. Also, the lengths of the UIOs so obtained stay within a reasonable upper bound. Given any bound l_b, our method can always calculate UIOs of length less than that bound if they exist.

Several of the arguments used in the proofs in chapters 2 and 3 assume that we have available to us minimal UIOs for all states in FSMs A and B. However, [SD88]'s method calculates UIOs only if it is of length less than or method in terms of space and time complexity when the communication constraint is obeyed by the interacting FSMs. Also, the lengths of the UIOs so obtained stay within a reasonable upper bound. Given any bound l_b, our method can always calculate UIOs of length less than that bound if they exist.

Several of the arguments used in the proofs in chapters 2 and 3 assume that we have available to us minimal UIOs for all states in FSMs A and B. However, [SD88]'s method calculates UIOs only if it is of length less than or equal to $2n^2$ and returns only state signatures otherwise. The same holds for UIO trees which we grow only up to a length of $2n^2$. But, there are FSMs with states that have all minimal UIOs of exponential length [LY94]. This means that even an exhaustive search for a pair of partially compatible *minimal UIOs/state signatures* in the UIO trees by our method may not yield a UIO for the product state on composing. But, then even the conventional method suffers from this disadvantage. Computation of UIO sequences is basically a PSPACE complete problem and hence the above mentioned method of bounding the length of state signatures cannot be done away with.

Algorithm 3.1 Algrithm for Computation of Compatible Paths

Input: : State S_A in FSM A, state S_B in FSM B and a path p_A of length k defined by the transition array $T[1,k]$ in FSM A starting from S_A.

Output: : A path p_B in FSM B compatible to p_A and starting from S_B.

begin

Initialisation Step : Reachable[1] = NCTreachable(S_B), Transition[0] = ϕ.

for $i \in 1$ **to** k **do**

Case 1: $T[i]$ is an NCT

Transition[i] = Reachable[i].

Reachable[i+1] = Reachable[i].

Case 2: $T[i]$ is of the form a/x where 'x' is a common symbol and $a \in I_A$

Transition[i] = Apply(Reachable[i],x);

Reachable[$i+1$] = NCTreachable[Transition[i]];

Case 3: $T[i]$ is of the form y/b where 'y' is a common symbol and $b \in O_A$

Transition[i] = Generate[Reachable[i],y];

Reachable[$i+1$] = NCTreachable[Transition[i]];

if Transition[i] = NULL **then**

Printf("No compatible path exists");

end if

end for

Path Reconstruction :

Select some transition $t_k = s_1^k \rightarrow s_2^k$ such that $s_1^k \in$ Reachable[k] and $s_2^k \in$ Transition[k]

for $i \in k-1$ **down to** 1 **do**

Select some transition $t_i = s_1^i \rightarrow s_2^i$ such that $s_1^k \in$ Reachable[k] and $s_2^k \in$ Transition[k] and $s_2^{i+1} \in$ NCTreachable(s_2^i).

end for

Output the path obtained by concatenating the edge labels $t_1, t_2, ..., t_k$.

end

Algorithm 3.2 Generation of UIO sequences for product states

Input : UIO tree of A, UIO tree of B, State S_A, State S_B

Output : UIO($[S_A, S_B]$)

Step 1 : To compute UIO(S_A, S_B), consider all possible pat hs in the UIO tree U_A which are UIO for A and all possible paths in U_B which are UIO for state S_B.

Step2 : If there exists a pair p_A , p_B such that p_A and p_B are compatible, then we can readily compute a UIO sequence for $[S_A, S_B]$ by using Compose(p_A, p_B).

Step 3 : If there does not exist such a pair, then there must exis t a pair p_A, p_B such that p_A is compatible with a prefix q_B of p_B or vice-versa. Obtain the pairs p_A and q_A

Step 4 : Assume that the pair is such that p_A is compatible w ith a prefix q_B of p_B. Decompose p_B into q_B and w_B where $p_B = q_B@w_B$.

Step 5 : Compute $S_A^1 = \delta(S_A, p_A)$.

Step 6 Construct path w_A starting from state S_A^1 which is compatible with w_B. using algorithm 3.2.

Step 7 : Set UIO(S_A, S_B) = Compose($p_A@w_A, p_B$).

Step 8 : If the pair p_A, p_B is such that p_B is compatible with a prefix q_A of p_B, the construction is symmetrical.

Chapter 4

Heuristics and Generalizations

Heuristic Methods for Unconstrained Communication

A complete methodology for synchronously communicating FSMs obeying the communication constraint was described in the previous chapter. However, the communication constraint is a *sufficient* condition for preservation of UIO property on composition, not a necessary one. Even if the communication constraint isn't satisfied, one may use the idea of composition and obtain test sequences of reasonably good quality. Also, this constraint is rather strict and may therefore exclude a large class of network protocols/FSMs from the set for which our method is *s*urely applicable.

These reasons motivate us to look for some heuristic algorithms which may not always return perfect state verification sequences in all cases, but still provide us with some kind of state signatures.(as in [SD88]'s method) We have developed one such heuristic algorithm (algorithm 4.1) that returns state signatures in the product FSM, by an extension of the compositional approach described in the previous chapters. The efficiency of the test cases generated by this heuristic algorithm will be evaluated experimentally.

Generalization to $K \geq 2$ FSMs

In representations of network protocols as finite state machines, we may, in general obtain a concurrent system consisting of more than two FSMs. The compositional method has a straightforward generalization for the case of K synchronously CFSMs. The communication constraint is suitably redefined for K CFSMs. First we have to formally specify the model and the specification of the mode of interaction between the K FSMs for the generalised problem.

We place the following restrictions on the nature of the alphabet and communication between the K FSMs:

- The external(or non-communicating) input alphabet of each FSM is distinct. The same holds for the external output alphabet. No symbol can be both an external input symbol of some FSM and an external output symbol of some other FSM.

Algorithm 4.1 Heuristic Algorithm - Unconstrained Communication

Input: : FSM A, FSM B.

Output: : State signatures/verifiers for all states of the product FSM.

begin

1. First we compute as in the previous chapter a pair of compatible UIOs for every product state $[S_A^i, S_B^j]$ by first searching for partially compatible UIOs and then applying Algorithm 3.2 for generating compatible tails. Call these paths as $p_{ij}^0, 1 \leq i \leq n_A, 1 \leq j \leq n_B$. Assign $p_{ij} := p_{ij}^0$.

2. **Repeat**

- Compute the unpredictability set of each of the sequences p_{ij}. The unpredictability set of a sequence p_{ij} is the set of product states, all of which show the I/O behaviour specified by p_{ij}. Call this set S_{ij}.

- Compute $E_{ij} = \delta(S_{ij}, p_{ij})$, i.e., the set of states where control in the concurrent FSM system could be on the application of p_{ij} from any state in S_{ij}. Let $[S_A^k, S_B^l] = \delta([S_A^i, S_B^j], p_{ij})$

- p_{ij} = Concatenate p_{ij} and p_{kl}^0

till S_{ij} ***contains exactly one product state***

or S_{ij} ***stops decreasing in size and goes into a cycle.***

end

- Every communication symbol is the input or output symbol of exactly one machine. Thus, given a communication, there is a unique pair of FSMs that communicate through that symbol.

- Every rendevous or communication involves exactly two FSMs. Thus, edge labels in the individual FSMs can only be of the following types:

  - a/c, where a is an external input symbol and c a communication symbol.

  - c/a, where a is an external output symbol and c a communication symbol.

  - a/b, where both a and b are external input and output symbols respectively.

The above constraints imply that communications cannot nest beyond a single level, i.e., a communication cannot involve three or more participating edges of the form $a/c_1, c_1/c_2, ..., c_k/b$, wherein a is an external input symbol, b an external output symbol and the rest are communication symbols. Such a model is restricted but is very useful

for representing the CSP (Communicating Sequential Processes) model that has a wide variety of applications.

We also redefine the following terms introduced in chapter 2.

- **Composition**: The operation *Compose* is now defined over a maximum of K paths as arguments. The composition of these paths is a sequence of edge labels such that the order of edges belonging to any particular FSM is preserved and every edge of the form a/c, where a is an external input symbol is adjacent to an edge of the form c/b, where b is an external output symbol of another FSM. As before such an adjacent pair of edges is understood to yield the composed edge a/b in the global FSM.

- **Communication Constraint**: The communication constraint has a natural extension to the K FSM problem. Since every communication involves exactly two FSMs, it is enough if every pair of FSMs mutually satisfy the communication constraint defined in chapter 2.

Keeping in mind the above definitions, we state without proof the following:
Proposition: The composition of any number of paths that are UIO sequences in the corresponding component machines yields a UIO sequence for the product state in the global FSM.

We follow the same approach for the K FSM problem as for two FSMs described in the previous chapter. We proceed by constructing UIO trees for each of the componenct FSMs. However, the algorithm described in the previous chapter for generating compatible tails cannot be extended directly to K FSMs and doesn't yield to a plynomial time algorithm. The generation of compatible tails takes space whose complexity is exponential in the number of participating FSMs. Though this method can be used for test suite generation for several real protocols, for some protocols like the ATM protocol, in which the global FSM could have as many as 10^{20} states, generating compatible paths becomes intractable. We describe an approach exploiting the non-deadlocking property to avoid an exhautive search

Chapter 5

Design and Implementation

In this chapter we will discuss the design and implementation details of the method described in the previous chapter. The subsequent sections give an overview of the data structures used and the major modules in the implementation are discussed. The complexity calculations done in the previous chapter show the compositional approach to be more efficient in terms of both space and time required. However, it must be cautioned that one must not attach too much importance to the expressions for complexity obtained. Even for moderate values of d and n, say $n = 10$ and$d = 5$, the expression for space complexity of the compositional approach - $O(n^4 d_{max}^{4n^2+1})$ shoots up to uncontrollably huge values. This necessitates the need to experimentally verify the claims of efficiency made in the previous chapters. The aim is to compare the space and time complexity of the compositional approach with that of the conventional one.

5.1 Design of Data Structures

The FSMs are stored in the form of a matrix with the number of rows equal to the number of states and the number of columns equal to the total number of input symbols. This is different from an adjacency list or even an adjacency matrix, the data structure normally used for representing FSMs. Its in fact an inefficient storage structure compared to the adjacency list especially if the matrix is sparse. However, this is not a concern since the space occupied by the component FSMs themselves is an insignificant fraction of the total memory usage for UIO tree construction. The advantage of this data structure lies in the speed of accessing the output symbol and final state of a transition given the head state and the input symbol, since all it takes is the constant time for accessing an array element. In an adjacency list for every transition, one has to traverse a linked list to obtain the result of making that transition. This will show up as increased time taken in computing the UIO trees. Note that the space requiring for storing the UIO tree for either of the component FSMs or the product

FSM isn't affected by the representation of the FSM itself.

5.2 Module Descriptions

The implementation may be divided into the following modules :

- **User_Spec**: For analysis, we need to randomly generate FSMs. This module reads specifications from the user about the FSMs to be randomly generated. The user is expected to enter the number of states in either FSM, the number of communicating symbols in each direction and finally the edge percentage. The latter refers to the expected percentage of input symbols for which a transition is defined for each state. For input symbols for which no transition is defined, we introduce what is commonly known as a non-core edge, i.e., a transition to itself with a null output. This is only an implementation trick to *complete* the FSM and doesn't affect the algorithm, since these edges cannot be part of test sequences.

- **Gen_FSM**: This module generates randomly a pair of FSMs that satisfy the communication constraint. One of the FSMs is generated randomly first in accordance with the user specifications. While randomly generating the second FSM, only those random edges are permitted which satisfy the communication constraint. Finally, non-communicating edges are suitably added to make the FSMs strongly connected.

- **Cross_Product**: This simply computes the cross product of the two FSMs generated.

- **UIO_tree**: This takes as input an FSM and returns the UIO tree corresponding to it. There are some optimisations that are performed to suitably prune the UIO tree, as given in [Nai97].

- **Compose**: This module is the most time consuming of all. For every state of the global FSM, it searches for a pair of partially compatible UIOs.

- **Compatible_Tail**: This module implements algorithm 3.1 to generate a compatible path in the other FSM for any given path in one of the FSMs.

A number of experiments are being performed over a large set of randomly generated examples to confirm the efficiency of the compositional approach. It is also intended to take up some readily availbale real life prorocols like the FABP protocol [Lee96]. In case the communication constraint isn't satisfied, the heuristic version will be analysed. The preliminary results are very encouraging. For quite a few cases, the conventional method is seen to run out of memory or use up enormous amount of it to generate the UIO tree for the cross product machine. The experimentation is still in the process of completion and results will be published very soon.

Chapter 6

Conclusion and Future Work

In this report we have given a new compositional approach to test sequence generation for CFSMs. We have also performed a complete complexity analysis of our method and proved the efficiency of this method over the conventional one, for the case of synchronous CFSMs obeying some constraints. The theoretical results are being supplemented with experimental analysis. The extension of the compositional method to the general case of K FSMs is not a smooth one. We need a more efficient method to compute compatible *tails* for any given set of paths in the K FSM system. We believe that the non-deadlocking property of real protocols naturally yields to an efficient procedure for generating compatible tails without performing an exhaustive search. This is because the non-deadlocking property of protocols forces certain communications edges to be present at strategic points, thereby obviating the need for an exhaustive search. In the case of the two FSM problem, this implies the following, which we state without proof:
Proposition: Let $[S_A, S_B]$ be any reachable state in the glolbal FSM system. Further, let a/c be any edge label in FSM A for state S_A. Then, there exists a state $S_B^{'} \in NCTreachable(S_A)$, such that $S_B^{'}$ has an edge with the label c/a.

We also need to consider communications of depth greater than one, i.e., involving more than two FSMs in a single communication. This naturally brings up the possibililty of feedback transitions. It may be shown that even if feedback transitions were to be allowed, the algorithm for computing compatible paths may be easily extended to accomodate this possibility. However, since it has space complexity exponential in the number of FSMs, we need to investigate more efficient search procedures.
It also remains to be seen how efficiently heuristic methods work for unconstrained CFSMs through further experimental analysis.
It is also necessary to analyse more real life protocols for the nature of communication involved and the special properties that we can hope to take advantage of like the non-deadlocking property. Future directions of research will be directed towards extending the compositional

approach to asynchronously communicating FSMs. Work to develop software is already underway which would enable us to apply this approach on a large number of randomly generated interacting FSMs and record the effect on test-generation parameters like time, space complexity and test sequence length.

Computation of UIOs and hence the UIO test sequence method is basically a PSPACE complete problem and a perfect solution is too much to hope for. We are essentially looking for certain interesting classes of CFSMs for which test sequence generation may be done avoiding state explosion.

Appendix A

UIO Tree Computation - The Naik Method

The main motivation for studying the Naik method is to compute all possible shortest length UIO sequences for a given state of the Finite State Machine. This would enable us to try all possible pairs of UIO sequences for a pair of states S_A in FSM A and S_B in FSM B, enabling us to compute the UIO sequence for the product state $[S_A,\, S_B]$.

The main idea used in this algorithm is the notion of a UIO tree that is a tree representation of all possible UIO sequences of the Finite State machine M.

We shall briefly describe the nayak method. For a detailed discussion, refer [Nai97]. In the rest of this section, we follow the usual notation of $\delta(s, x)$ to denote the "next state" reached by state s on application of input string "x" and $\lambda(s, x)$ to denote the output string generated during the transition when "x" is applied to state s.

Description of method

Definitions

The concepts of a path vector and perturbation are central to the idea of the algorithm for UIO tree construction.

Given an FSM M, a path vector is a collection of state pairs $(s_1/s_1^{'},\ s_2/s_2^{'},\ ....s_k/s_k^{'})$ with the following properties.

1. s_i and $s_i^{'}$ denote the head and tail state of a path where a path is a sequence of state transitions.

2. An identical sequence of input/output is associated with all the paths in the path vector.

Given a path vector PV, the notation IV(PV) is used to denote the set of all head states of the path vector while CV(PV) is used for the collection of tail states of PV. A path vector is said to be *Singleton* if it contains exactly one state pair, and it is said to be *homogenous*

if all members of CV(PV) are identical. Note that every singleton vector is by definition homogenous.

Perturbation

Given PV = $(s_1/s_1^{'}, s_2/s_2^{'}, ...s_k/s_k^{'})$ and an edge-label a/b, we define the perturbation of PV with respect to edge-label a/b Pert(PV, a/b) as follows:

Pert(PV, a/b) = $s_i/s_i^{''}$ — $s_i^{''} = \delta(s_i^{'}$,a) and $\lambda(s_i^{'}$, a) = b and $s_i/s_i^{'} \in$ PV

Starting from an initial path vector $(S_1/S_1, S_2/S_2, \ldots S_k/S_k)$, (this includes all the states of the FSM M), where a null transition is assumed to exist with all these state pairs, one can infinitely perturb all the path vectors for all the edge labels. Given the operation of perturbation and the initial path vector, once could construct a tree in the following manner. For each unperturbed path vector, PV, construct the path vectors $PV^{'} = pert(PV, a/b)$ for all edge labels a/b and add an edge from PV to $PV^{'}$ labelled by the edge a/b. This tree construction can be carried on indefinitely. Hence, we define pruning conditions.

- **C1 :** CV($PV^{'}$) is homogenous
- **C2 :** On the path from the root (initial node) to PV, there exists $PV^{''}$ such that $PV^{'} \subseteq PV^{''}$.

If one of the above pruning conditions is satisfied, then $PV^{'}$ is declared to be a terminal node. A terminal node is not perturbed further. Let us justify the above conditions. Suppose C1 is satisfied. In this case, $PV^{'}$ contains either exactly one element or identical elements. If $PV^{'}$ contains exactly one element, we have found a UIO sequence for the head-state of this path vector. So, we need not perturb further, having already achieved our goal of obtaining a UIO sequence. If CV($PV^{'}$) contains identical elements, then any further perturbation of the path from the initial node to $PV^{'}$ will not lead to any UIO sequence. Suppose condition C2 is satisfied. Since a superset of the node appears above on the path from the root to this node, it has already been perturbed in all possible ways.

Algorithm

Based on the above notions of perturbation and the pruning conditions, we can construct an algorithm for UIO tree construction as follows The above algorithm generates all possible UIO sequences of a given length and hence generates all multiple UIO sequences for a given state.

To illustrate the algorithm, we have shown the construction of the UIO tree for the FSM g given in figure 1. The UIO tree is shown in figure . For representing the UIO tree, each path vector is represented by two rows. The first row is the initial vector of the path vector

Algorithm .1 Algorithm for UIO tree construction

Step 1 : Let ψ be the set of nodes denoting path vectors in the UIO tree. Initially, Ψ contains the initial vector marked as non-terminal.

Step 2: Find a non-terminal member $\psi \in \Psi$ which has not yet been perturbed. If no such member exists, then the algorithm terminates.

Step 3: Compute $\psi^{'}$ = Pert(ψ,a_i/b_i) for all distinct edge labels a_i/b_i and add $\psi^{'}$ to Ψ. Mark ψ as perturbed. Update the UIO tree.

Step 4: If $\psi^{'}$ computed in step3 satisfies conditions C1 and C2, then mark $\psi^{'}$ as a terminal node and go to step 2.

corresponding to that node i.e. IV(PV) and the second row is the current vector of the path vector of that node i.e. CV(PV).

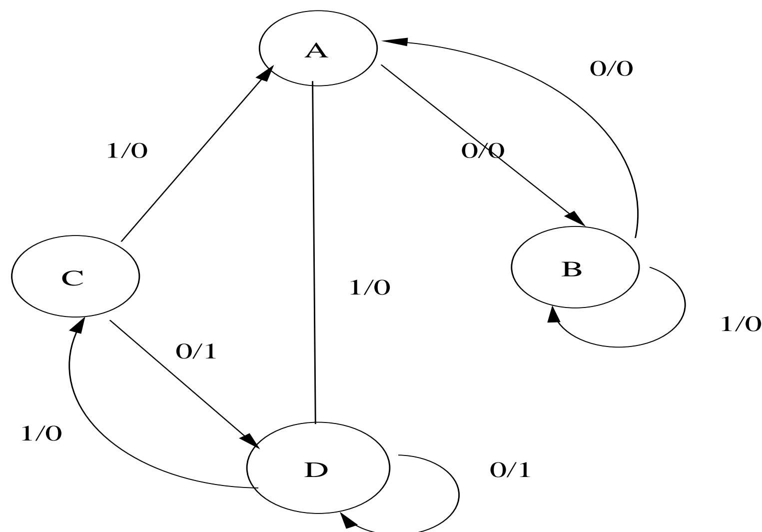

Figure 1: Example FSM g for the UIO tree method

That the algorithm generates all possible UIO sequences is stated and proved in the form of a theorem in [Nai97]. theorem.

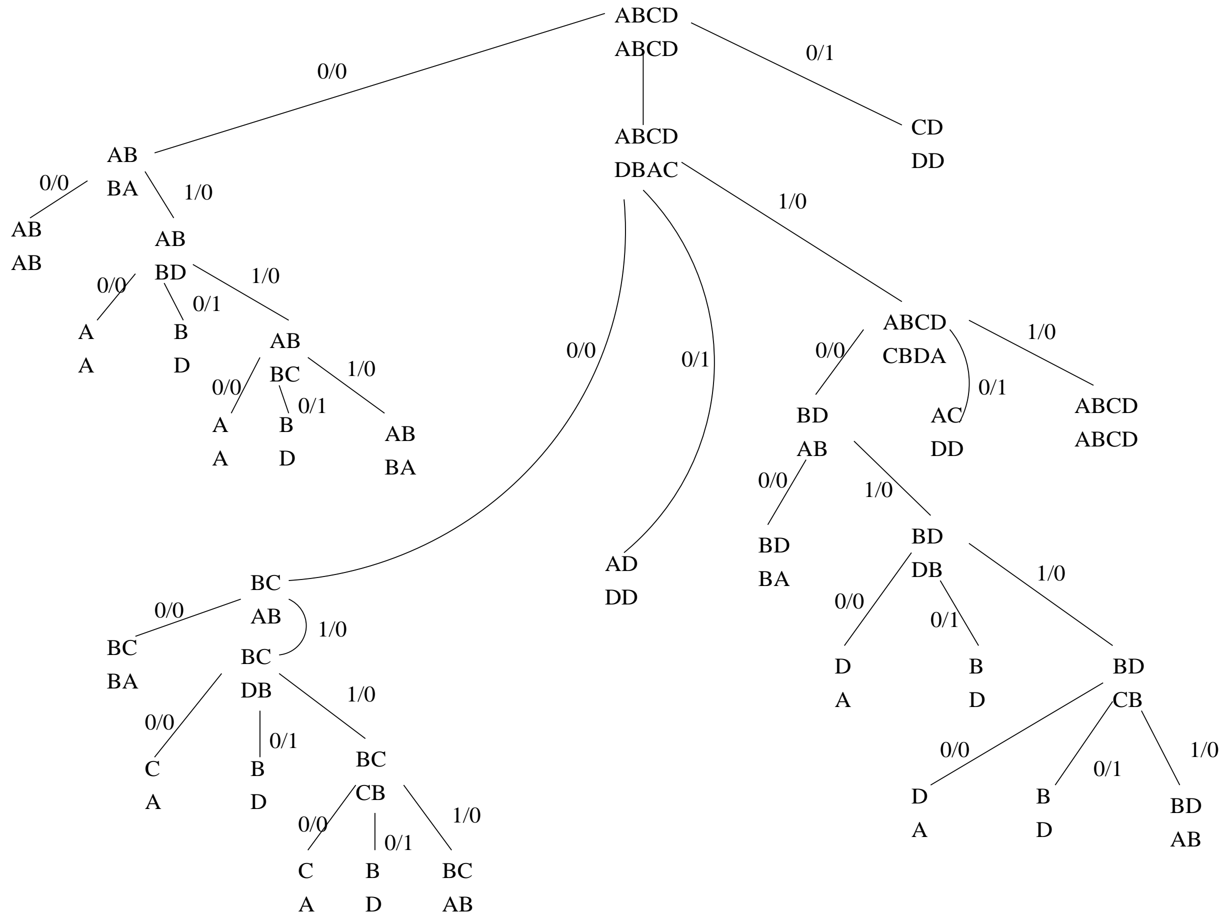

Figure 2: UIO tree for the FSM g

Bibliography

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Inc., 1991.

[SL89] Deepinder P Sidhu and Ting-Kau Leung. Formal Methods for Protocol Testing, a Detailed Study. *IEEE Transactions on Software Engineering*, 15(4):413-426, April 1989.

[Vem98] Srinivas Vemuri. Test case Generation for Communicating Finite State Machines. *B.Tech Project Report, IIT Bombay*, April 1998

[LY94] David Lee and Mihalis Yannakakis. Testing Finite-State Machines: State Identification and Verification. *IEEE Transactions on Computers, Vol 43, No. 3, March 1994.*

[SD88] Krishnan Sabnani and Anton Dahbura. A protocol Test Generation Procedure. *Computer Networks and ISDN systems*, 15:285-297, 1998.

[SM97] S M Mahesh. Test Case Generation for SDL-92 Programs. *M.Tech II stage Dissertation, IIT Bombay*, August 1997.

[Nai97] Kshirsagar Naik. Efficient Computation of UIO Sequences in Finite State Machines. *IEEE Transactions on Software Engineering* 5:585-599, August 1997.

[Hie97] Rob M. Hierons. Testing from Semi-independant Communicating Finite State Machines with a Slow Environment. *IEE Proceedings on Software Engineering*, 1998.

[TC78] T Chow. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering*, SE-4:178-187, March 1978.

[ZK78] Zvi Kohavi. *Switching and Finite State Automata Theory.* McGrawHill, 1978.

[Lee96] David Lee, Krishan Sabnani, David Kristol and Sanjoy Paul. *Conformance Testing of Protocols Specified as Comunicating Finite State Machines - A Random Walk Approach. IEEE Transactions on Communications*