Context Independent Unique Sequences Generation for Protocol Testing*

T. Ramalingom

Bell Northern Research Ltd
P.O. Box 3511, Station C
Ottawa, Ontario K1Y 4H7, Canada

K. Thulasiraman† Anindya Das

School of Computer Science
University of Oklahoma
Norman, OK 73019, U.S.A.

Abstract

A number of test sequence generation methods proposed recently for protocols represented as Extended Finite State Machines (EFSMs) use state identification sequences for checking the states. However, neither a formal definition nor a method of computation of these sequences for an EFSM state is known. In this paper, we define a new type of state identification sequence, called Context Independent Unique Sequence (CIUS) and present an algorithm for computing it. An unified method based on CIUSes is developed for automatically generating executable test cases for both control flow and data flow aspects of an EFSM. In control flow testing, CIUSes are very useful in confirming the tail state of the transitions. In data flow testing, CIUSes improve the observability of the test cases for the def-use associations of different variables used in the EFSM. Unlike general state identification sequences, the use of CIUSes does not increase the complexity of the already intractable feasibility problem in the test case generation.

1 Introduction

Automatic test case generation from protocol standard is a means of selecting high quality test cases efficiently. Recently, International Organization for Standards has established a working group for studying the application of formal methods in conformance testing [FMC93]. One of the primary aims of this group is to enable computer-aided test case generation from protocol standards specified in formal description techniques such as Estelle [IS9074], SDL [SDL88], and LOTOS [IS8807].

In this paper, we focus on the EFSM model [UY91] since it is the fundamental model for both Estelle and SDL. We need a few informal definitions before summarizing the salient features of the EFSM-based test case generation methods developed recently. The EFSM model is an extension of the FSM model [Koh78]. A transition in an EFSM may have an input interaction, a predicate, and can produce a sequence of output interactions. Also, the interactions can have a number of parameters. Moreover, the local variables of an EFSM can be defined and/or used in a transition. A walk in an EFSM is feasible if the underlying predicate of the walk is satisfiable. A walk W is executable if there exist a walk W_p from the initial state of the EFSM to the starting state of W such that the concatenated walk W_p followed by W is feasible. A test case is executable if its underlying walk is executable. The feasibility (executability) problem is to find if a given walk is feasible (executable) or not.

Chun and Amer are the first to apply the Unique Input Output (UIO) sequences for confirming the tail state of a transition in an EFSM [CA91]. However, they do not address either the definition or the computation of an executable UIO-sequence for an EFSM state. The test case generation method of Miller and Paul [MP92] covers both control and data flow aspects of an EFSM. This method takes the white-box approach of testing and assumes that the local contexts are part of an UIO-sequence. Hence the UIO-sequence has limited observability. Assuming the definition of a characterizing sequence, Chanson and Zhu [CZ93, CZ94] have introduced what is called a Cyclic Characterizing Sequence for identifying the states. They generate a set of test tours for covering both control and data flow aspects of testing. The executability of the tours are considered after their generation. As a result, the test coverage is affected since the unexecutable tours have to be discarded. In [LHHT94], Li *et al* propose a method for generating control flow test cases from an EFSM which has only integer type of local variables and input interaction parameters. It is the first method which explicitly addresses the executability of a state identification sequence. They have introduced a new type of UIO-sequencce, called Extended UIO-sequence (EUIO-sequence). It is basically an extension of an UIO-sequence so that the underlying walk is executable. However, the problem of finding if a given UIO-sequence has an EUIO-sequence is, in general, undecidable [LHHT94].

It is evident that the definitions provided for the state identification sequences in the FSM model[Koh78] is not adequate for the EFSM model.

*This work was done at Concordia University, Montreal, prior to T. Ramalingom joining Bell-Northern Research Ltd., and represents the views of the authors and not necessarily those of BNR Ltd.

†Currently on leave from Department of Electriacl and Computer Engineering, Concordia University, Montreal.

Moreover, the existing methods focus on the automatic test case generation rather than the issues related to the state identification sequences. A formal definition and the methods for computing these sequences are essential in order to successfully use the number of EFSM-based test sequence generation methods proposed in the literature. In this paper, we first formally define an Unique Input Sequence (UIS) for an EFSM state. Unlike a walk in an FSM model, the executability of a walk in an EFSM model can not be taken for granted. Moreover, the executability problem is inherently intractable. Therefore, generating executable test cases for a protocol represented in an EFSM is a challenging problem. It is known that the test case generation for an EFSM model is difficult when a general UIS is used [LHHT94]. One way of reducing the complexity of the test case generation problem is to avoid the feasibility problem as much as possible, without compromising the test selection criteria. With this motivation we define a special type of UIS called Context Independent Unique Sequence (CIUS) and present a method for computing CIUSes.

In order to demonstrate the use of CIUSes in test case generation, we establish control and data flow criteria and summarize a method [RDT95b] for automatically generating executable test cases satisfying these criteria. We show that the use of CIUSes enhances the observability (ability to observe the states and data flow in an implementation, which is viewed as a black-box) of the test cases without increasing the complexity of the feasibility problem.

2 The EFSM Model

The EFSM model presented in this paper is inspired from [UY91]. An EFSM M is a 6-tuple $M = (S, s_1, I, O, T, V)$, where S, I, O, T, V are a nonempty set of states, a nonempty set of input interactions, a nonempty set of formal output interactions, a nonempty set of transitions, and a set of variables, respectively. Let $S = \{s_j \mid 1 \leq j \leq n\}$; s_1 is called the **initial state** of the EFSM. Each member of I is expressed as $ip?i(parlist)$, where ip denotes an interaction point where the interaction of type i occurs with a list of input interaction parameters $parlist$, which is disjoint from V. Each member of O is expressed as $ip!o(outlist)$, where ip denotes an interaction point where the interaction of type o occurs with a formal list of parameters, $outlist$. Each parameter in $outlist$ can be replaced by a suitable variable from V, an input interaction parameter, or a constant. The interaction thus obtained from a formal output interaction is referred to as an **output interaction** or an **output statement**. We will assume that the variables in V and the input interaction parameters can be of types integer, real, boolean, character, and character string only. Each element $t \in T$ is a 5-tuple $t = (source, dest, input, pred, compute\_block)$. Here, $source$ and $dest$ are the states in S representing the starting state and the tail state of t, respectively. $input$ is either an input interaction from I or empty. $pred$ is a Pascal-like predicate expressed in terms of the variables in V, the parameters of the input interaction $input$ and some constants. The $compute\_block$ is a computation block which consists of Pascal-like assignment statements and output statements.

A component of a transition can also be represented by postfixing the transition with a period followed by the name of the component. For example $t.pred$ represents the predicate component of the transition t. Note that, unlike a variable, the scope of a parameter in an input interaction of a transition is restricted to the transition only. Let m denote the number of transitions in M. We will assume that $m \geq n$. A closed walk which starts and ends at the initial state is referred to as a **tour**. A transition in M with empty input interaction is called a **spontaneous transition**.

A walk W in M can be **interpreted symbolically** by assuming distinct symbolic values for the local variables at the beginning of W as well as distinct symbolic values for the input interaction parameters along W. When W is symbolically interpreted the predicates along W are also interpreted and is expressed in terms of the initial symbolic values for the local variables and the symbolic values for the input interaction parameters. W is said to be **feasible** if the conjunction of the symbolically interpreted predicates is satisfiable.

A **context** of M is the set $\{(var, val) \mid var \in V$ and val is a value of var from its domain$\}$. A **valid context** of a state in M is a context which is established when M's execution proceeds along a walk from the initial state to the given state.

Let t be a non-spontaneous transition in M. t is said to be **executable** if (i) M is in the state $t.source$, (ii) there is an input interaction of type i at the interaction point ip, where $t.input = ip?i(parlist)$, and (iii) the valid context of the state and the values of the input interaction parameters in $parlist$ are such that the predicate $t.pred$ evaluates to $true$. A spontaneous transition t is **executable** if (i) M is in the state $t.source$ and (ii) the valid context of the state is such that $t.pred$ evaluates to $true$. When a transition is executed, all the statements in its computation block get executed sequentially and the machine goes to the destination state of the transition. A walk W in M is said to be **executable** if all the transitions in W are executable sequentially, starting from the beginning of the walk. Note that every executable walk is feasible.

An EFSM is **deterministic** if for a given valid context of any state in the EFSM, there exists at most one executable outgoing transition from that state. An EFSM M is said to be **completely specified** if it always accepts any input interaction defined for the EFSM. An arbitrary EFSM M can be transformed into a completely specified one using what is called a **completeness transformation** described next. Given a valid context of a state and an instantiated input interaction, suppose that M does not have an executable outgoing non-spontaneous transition at the state for the given valid context and the input interaction, and that M does not have an outgoing spontaneous transition at the state such that it is executable for the given valid context, then a self-loop transition with an empty computation block is added at the state such that it is executable for the given context and the input interaction. The newly added transitions are called **non-core transitions**.

We assume that the EFSM representation of the specification is deterministic and completely specified. It is assumed that for every transition in the EFSM, it has at least one executable walk from the initial state to the starting state of the transition such that the transition is executable for the resulting valid context. Similarly, we assume that the initial state is always reachable from any state with a given valid context.

2.1 An Example

As an example of an EFSM , let us consider a major module (based on the *AP-module* in [Boc90]) of a simplified version of a class 2 transport protocol. This EFSM participates in connection establishment, data transfer, end-to-end flow control, and segmentation. It has the interaction point labeled U connected to the transport service access point and another interaction point labeled N connected to a mapping module. Here, we represent the EFSM by (S, s_1, I, O, T, V). This EFSM is used throughout the paper for illustrating various points. Let $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The set of input interactions and the set of output interactions are given below.

$I =$ {U?TCONreq(dest_add, prop_opt), U?TCONresp(accpt_opt), U?TDISreq, U?TDATreq(Udata, EoSDU), U?U_READY(cr), N?TrCR(peer_add, opt_ind, cr), N?TrCC(opt_ind, cr), N?TrDC, N?TrDR(disc_reason, switch), N?TrDT(send_sq, Ndata, EoTSDU), N?TrAK(XpSsq, cr), N?ready, N?terminated}

$O =$ {U!TCONconf(opt), U!TCONind(peer_add, opt), U!TDISind(msg), U!TDISconf, U!TDATAind(data, EoTSDU), U!error, U!READY, N!TrCR(dest_add, opt, credit), N!TrDR(reason, switch), N!terminated, N!TrCC(opt, credit), N!TrDT(sq_no, data, EoSDU), N!TrAK(sq_no, credit), N!error, N!TrDC}

V={*opt, R_credit, S_credit, TRsq, TSsq* }. All the variables in V are of integer type. The transitions as described in Table 1, and Table 2 are shown in Figure 1. The state s_1 is repeated in the figure merely for convenience.

3 Unique Input Sequences

An input sequence, a sequence of input interactions, is said to be **instantiated** if all the parameters in the sequence are properly instantiated with values. A **test sequence** is a sequence of input and output interactions. A sequence of zero or more output interactions between two successive input interactions in a test sequence is the sequence to be observed after applying the preceding input interaction to an EFSM and before applying the succeeding one.

The sequence of input and output interactions along a feasible walk W is denoted by **Trace(W)**, known as the **trace of the walk** W. The sequence of input (output) interactions along a feasible walk W is denoted by **Inseq(W) (Outseq(W))**. $Trace(W)$ and $Outseq(W)$ are actually obtained by symbolically interpreting W.

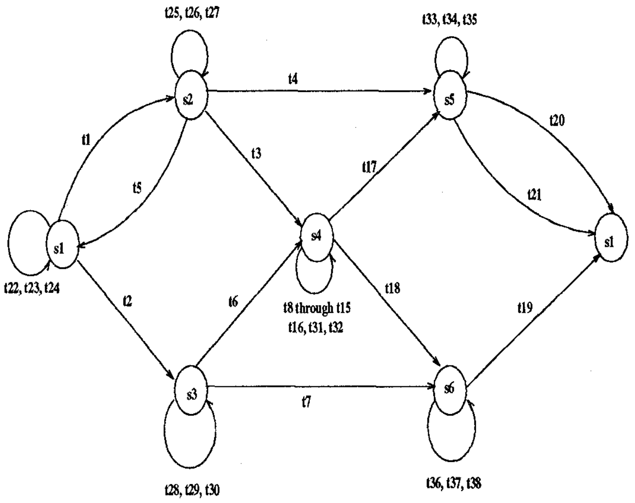

Figure 1: An EFSM in Class 2 transport protocol

Two input interactions are said to be **distinguishable** if: (i) they occur at two different interaction points or (ii) their interaction types are different. We say that two output interactions are **distinguishable** if at least one of the following is true: (i) they occur at two different interaction points, (ii) their interaction types are different, and (iii) if the parameters in a given position in both interactions are constants then they are different.

For example, the output interactions N!TrDR('procedure error', false) and N!TrDR('procedure error', true) are distinguishable. However, N!TrDT(TSsq, Udata, EoSDU) and N!TrDT(TRsq, Udata, EoSDU) are not distinguishable.

The **length** of a sequence is the number of interactions it contains. Two sequences are indistinguishable if they have the same length, and if the pairs of interactions in all positions are not distinguishable. Otherwise, the sequences are said to be distinguishable.

Let W be an executable walk at s_j. Let U be an instantiation of $Inseq(W)$. We define U as a **Unique Input Sequence (UIS)** of s_j if $Trace(W)$ is distinguishable from $Trace(W')$, for any feasible walk W' at state s_k, for $1 \leq k \leq n, k \neq j$. In this case, W is called an **UIS walk** for U.

As indicated in [LHHT94], automatic test case generation for an EFSM is difficult when a general UIS is used. For example, let U be an UIS for s_j, and let W be the UIS walk of U. Let t be a transition from a state s_i to s_j. In order to test t, one needs to compute an executable walk P_t from s_1 to s_i and associate values for the input interaction parameters along P_t and t such that $P_t\ t\ W$ is executable. For a given W, it is in general difficult to find a P_t so that the walk $P_t\ t\ W$ is executable. Moreover, if the general UISs are considered, then multiple UISs may be required for a state in order to test all the incoming transitions at that state. Hence a careful selection of

Tr.	Input	Predicate	Compute-block
t1	U?TCONreq(dst_add, *prop_opt*)		opt:= prop_opt; R_credit := 0; N!TrCR(dst_add,opt,R_credit)
t2	N?TrCR(peer_add, *opt_ind*, *cr*)		opt := opt_ind; S_credit := cr; R_credit := 0; U!TCONind(peer_add, opt)
t3	N?TrCC(opt_ind,cr)	opt_ind $\leq$ opt	TRsq:=0;TSsq:=0; opt := opt_ind; S_credit := cr; U!TCONconf(opt)
t4	N?TrCC(opt_ind, cr)	opt_ind > opt	U!TDISind(' procedure error'); N!TrDR('procedure error', false)
t5	N?TrDR(disc_reason, *switch*)		U!TDISind(disc_reason); N!terminated
t6	U?TCONresp(accpt_opt)	accpt_opt $\leq$ opt	opt := accpt_opt; TRsq := 0; TSsq := 0; N!TrCC(opt, R_credit)
t7	U?TDISreq		N!TrDR('User initiated' , true)
t8	U?TDATreq(Udata, *EoSDU*)	S_credit > 0	S_credit := S_credit−1; N!TrDT(TSsq, Udata, EoSDU); TSsq := $(TSsq + 1) mod 128$;
t9	N?TrDT(send_sq, Ndata, *EoTSDU*)	R_credit $\neq$ 0 $\wedge$ send_sq = TRsq	TRsq := $(TRsq + 1) mod$ 128; R_credit := R_credit − 1; U!TDATAind(Ndata, EoTSDU); N!TrAK(TRsq, R_credit)
t10	N?TrDT(send_sq, Ndata, *EoTSDU*)	R_credit = 0 $\vee$ *send_sq* $\neq$ *TRsq*	N!error; U!error
t11	U?U_READY(cr)		R_credit := R_credit+cr; N!TrAK(TRsq, R_credit)
t12	N?TrAK(XpSsq, cr)	TSsq$\geq$XpSsq $\wedge$ $cr + XpSsq - TSsq \geq 0$ $\wedge$ $cr + XpSsq - TSsq \leq 15$	S_credit := $cr + XpSsq - TSsq$
t13	N?TrAK(XpSsq, cr)	TSsq$\geq$XpSsq $\wedge$ $(cr + XpSsq - TSsq < 0 \vee cr + XpSsq - TSsq > 15)$	U!error; N!error
t14	N?TrAK(XpSsq, cr)	TSsq<XpSsq $\wedge$ $cr + XpSsq - TSsq - 128 \geq 0$ $\wedge$ $cr + XpSsq - TSsq - 128 \leq 15$	S_credit := $cr + XpSsq - TSsq - 128$
t15	N?TrAK(XpSsq, cr)	TSsq<XpSsq $\wedge$ $(cr + XpSsq - TSsq - 128 < 0 \vee cr + XpSsq - TSsq - 128 > 15)$	U!error; N!error
t16	N?ready	S_credit > 0	U!READY
t17	U?TDISreq		N!TrDR('User initiated', *false*)
t18	N?TrDR(disc_reason, *switch*)		U!TDISind(disc_reason); N!TrDC
t19	N?terminated		U!TDISconf
t20	N?TrDC		N!terminated; U!TDISconf
t21	N?TrDR(disc_reason, *switch*)		N!terminated

Table 1: Core transitions of the EFSM shown in Figure 1

Transitions	Input
t25, t28, t31, t33, t36	U?TCONreq(dest_add, prop_opt)
t23, t26, t34, t38	U?TDISreq
t22, t29, t37	N?TrDR(disc_reason, switch)
t24, t27, t30, t32, t35	N?terminated

Table 2: Non-core transitions of the EFSM shown in Figure 1

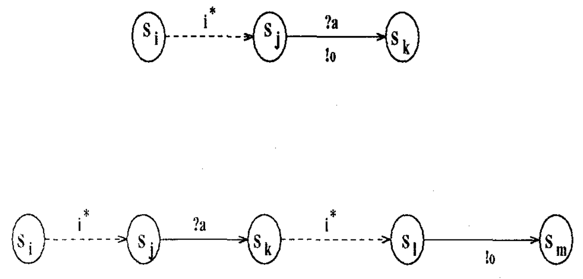

Figure 2: Different walks from s_i with behavior sequence $?a/!o$

the UISs is required.

A walk from a state is said to be **context independent** if the predicate of every transition along the walk, duly interpreted symbolically, is independent of the symbolic values of the local variables at the starting of the walk. Observe that every context independent feasible walk is executable.

We introduce a special type of UIS, called **Context Independent Unique Sequence (CIUS)**. Let U_i be an instantiated UIS of s_i and let $U(i)$ be the corresponding UIS walk at s_i. U_i is said to be a **CIUS** of s_i if $U(i)$ is context independent and executable.

Note that all the local variables used in the predicate of each transition in $U(i)$ are defined within $U(i)$ prior to their use. In other words, the predicates along $U(i)$ are independent of any valid context at s_i. Therefore, $U(i)$ can be postfixed to any executable walk from the initial state to s_i and the resulting walk is also executable. This property is very useful in handling the feasibility problem in test case generation. Also, one CIUS is sufficient for testing all the incoming transitions at a state. In the following section, we present an algorithm for computing a CIUS of a given state.

4 CIUS Computation Algorithm

The UIO-sequnce computation algorithms for an FSM [SD88] are, in general, not suitable for computing UIS in an EFSM. For instance, in order to find whether a state in an FSM produces an output o when an input a is applied, these algorithms simply check if the state has any outgoing transition with the label a/o. But in an EFSM, any of the two types of feasible walks from the state, say s_i, as shown in Figure 2, may have the same trace; therefore their presence has to be analyzed. In the figure, the dashed edges with label i^* denote walks consisting of a finite number (possibly zero) of silent transitions. $?a$ and $!o$ indicate the input interaction a and the output interaction o, respectively. In this section, we develop an algorithm for computing a CIUS of a given state in the EFSM . In order to guarantee the termination of the algorithm we assume that the EFSM has no feasible silent closed walk of length more than K for some integer $K \geq 1$.

Let $W1$ and $W2$ be feasible walks in the EFSM. Let t be the last transition in $W1$. We say that $Trace(W1)$ **subsumes** $Trace(W2)$ if there exists a sequence OS of zero or more output interactions at the end of t such that $Trace(W1)$ and $Trace(W2)@OS$ are indistinguishable. By a **null walk** at a state , we refer to an empty walk, a walk without any transition, starting and ending at that state. A higher level description of the proposed algorithm *ComputeCIUS* is given below.

```
Algorithm ComputeCIUS
Input: An EFSM and a state $s_k$ in the EFSM
Output: If it exists, a CIUS for $s_k$ such that its
underlying walk $W$ is of length $\leq 2n^2$ and $W$ has
only non-silent transitions.
Step 0 { Initialization }
    (i) $Wset$ := {null walk at $s_k$ }.
    (ii) $OWset$ := {null walk at $s_j$ | $1 \leq j \leq n, j \neq k$}.
    (iii) $L := 0$.
Step 1 { Iterative step }
    (i) repeat (a) to (c) until $L \geq 2n^2$
        (a) $L := L + 1$.
        (b) Set $TWset$ and $TOWset$ to the empty set.
        (c) Do Step 2.
    (ii) Stop.
Step 2
    (i) Do Step 2.1 for each walk $W \in Wset$ and for
        each non-silent outgoing transition $t$ in the
        EFSM at the tail state of $W$.
    (ii) Copy $TWset$ to $Wset$.
    (iii) Copy $TOWset$ to $OWset$.
Step 2.1
    If $W\ t$ is a context independent executable
    walk then do the following:
        (a) Let $W'$ be the walk $W\ t$.
        (b) Add $W'$ to $TWset$.
        (c) Initialize $NOWset$ to the empty set.
        (d) Do Step 2.1.1 for each walk $W_1$ in $OWset$.
        (e) If $NOWset$ is empty then Declare
            $Inseq(W\ t)$ as the CIUS and stop.
Step 2.1.1
    If there exists a feasible extended walk $W_2$ in
    the EFSM for $W_1$ such that $Trace(W_2)$
    subsumes $Trace(W')$ then do the following:
        (a) Add all such feasible extended walks of $W_1$
            to $TOWset$.
        (b) Add all such feasible extended walks of $W_1$
            to $NOWset$.
end ComputeCIUS
```

This algorithm computes a CIUS of the state s_k in the EFSM such that the walk from s_k which corresponds to the CIUS is of length at most $2n^2$. The same bound is used in the algorithm of Sabnani and Dahbura [SD88] for computing an UIO-sequence. For better observability of the test cases, these walks are allowed to have only non-silent transitions. However, the algorithm can easily be adapted to compute CIUS walks with silent transitions. We would like to note that there may exist a state which does not have a CIUS, in general, and the CIUS with this length and non-silent transition restriction, in particular.

At the beginning of the ith iteration of Step 1, *Wset* contains the set of all context independent executable walks of length $(i-1)$ which starts from s_k. At the same instant, *OWset* contains the set of all feasible walks from all the states other than s_k such that the trace of every walk in *OWset* subsumes the trace of a walk in *Wset*. In Step 0, *Wset* is initialized with the null walk at s_k and *OWset* contains the null walk at s_j, for all $j, 1 \leq j \leq n, j \neq k$. Step 1 is the iterative step which is repeated at most $2n^2$ times.

When the *i*th iteration of Step 1 invokes Step 2, the latter step computes a set (*TWset*) of context independent executable walks of length i which start from s_k. This is done by checking the executability of the walk obtained from each walk W in *Wset* by appending each non-silent outgoing transition from the tail state of W to W. Step 2 also computes the set (*TOWset*) of all feasible walks from any state other than s_k such that the trace along a walk in this set subsumes the trace along some walk in *TWset*. Step 2 does these computations by repeatedly calling Step 2.1 which in turn invokes Step 2.1.1 many times. *TWset* and *TOWset* become *Wset* and *OWset*, respectively, for the $(i+1)$th iteration. Step 2.1 and Step 2.1.1 are explained below.

Given a walk $W \in Wset$ and a non-silent outgoing transition t from the tail state of W, if the walk $W' = W\, t$ is executable and context independent, then Step 2.1 adds this walk to *TWset*. For each walk $W_1 \in OWset$, Step 2.1 invokes Step 2.1.1 for computing the set of all feasible walk extensions of W_1 such that the traces of the resulting walks subsume the $Trace(W')$. In *NOWset*, Step 2.1 stores the set of all feasible walks from any state other than s_k such that the trace along a walk in this set subsumes the trace along a walk $W' \in TWset$. If *NOWset* is empty, then the trace along W' is clearly a CIUS of s_k. And in this case the algorithm terminates. If *NOWset* is not empty for all the $2n^2$ iterations, then the algorithm terminates without finding a CIUS for s_k.

Given a walk $W_1 \in OWset$, and a walk $W' \in TWset$, Step 2.1.1 computes the set of all feasible walk extensions of W_1 such that the traces of the resulting walks subsume $Trace(W')$. The extended walks are added to *TOWset* as well as *NOWset*.

Following theorems summarize the time complexity and correctness of the algorithm. They are proved in [Ram94].

Theorem 1 *Suppose that the EFSM has no feasible closed silent walk of length more than K for some integer $K \geq 1$. Then the algorithm ComputeCIUS takes at most $2(d_{max}^{out})^{2n^2+1} + (n-1)(d_{max}^{out})^{(2n^2+2)+(1+O_{max})(K+1)(2n^2+1)}$ steps, where n, O_{max} and d_{max}^{out} are the number of states, the maximum number of output interactions in any transition in the EFSM, and the maximum number of outgoing transitions including self-loops in any state, respectively.*

Theorem 2 *Suppose that the EFSM has at least one walk W of length at most $2n^2$ at $s_k \in S$ such that (i) W is a context independent executable walk having only non-silent transitions, and (ii) $Trace(W)$ is distinguishable from the trace of any feasible walk from any state other than s_k. Then, the algorithm* ComputeCIUS *returns the input sequence U along a shortest walk at s_k which satisfies (i) and (ii). U is a CIUS of s_k.*

State	CIUS	Tr. Seq.
s_1	U?TCONreq(dst_add, prop_opt)	t1
s_2	N?TrDR(disc_reason, switch)	t5
s_3	U?TDISreq	t7
s_4	U?TDISreq	t17
s_5	N?TrDR(disc_reason, switch)	t21
s_6	N?terminated	t19

Table 3: CIUS for states in EFSM of Figure 1

Note that only a higher level complexity of the algorithm is given in terms of the number of times various basic steps are executed. The executions of some of these steps may themselves be complex. A detailed refinement of the algorithm is given in [Ram94]. The CIUS of every state in the EFSM of Figure 1 computed using the algorithm *ComputeCIUS* is presented in Table 3. The algorithm terminates after the first iteration of Step 2 for every state. The parameters in the CIUSes have to be instantiated with certain feasible values.

Though the given algorithm is exponential, for real life protocols which have CIUSes for all the states, the algorithm is expected to terminate within a few iterations, as in the above EFSM. We have also applied the algorithm on few other protocols. The EFSM representation of the class 0 transport protocol as specified in [UY91] has 4 states and 14 core transitions. The shortest CIUS walk for the initial state is of length 2. All other states have a CIUS walk of unit length. The EFSM representation of the abracadabra protocol as specified in [Tur93] has 5 states and 30 core transitions. It has a CIUS set such that the maximum length of a CIUS walk for a CIUS in this set is only 2. It should also be noted that there are protocols which may not have a CIUS for every state. For example, the initiator module of the INRES protocol [Hog92] does not have a CIUS for one state.

In the following section we shall study the use of CIUSes in the test sequence generation.

5 Automatic Test Case Generation

We consider the automatic generation of executable test cases for both data flow and control flow aspects of an EFSM. Our control flow fault coverage criterion is called trans-CIUS-set criterion and it is based on the Uv-method [CVI89]. For data flow coverage, we extend the "all-uses" criterion [RW85] to what is called the def-use-ob criterion. This extension is essential due to the black-box approach of protocol testing and it enhances the observability of the test cases for the def-use associations. As detailed below, both criteria use a CIUS set for identifying the states.

5.1 Control Flow Coverage Criterion

Let U_i be a CIUS for the state s_i, $1 \leq i \leq n$. Let $\mathcal{U} = \{U_i \mid 1 \leq i \leq n\}$. We call $\mathcal{U}$ as a **CIUS set.**

Our control flow coverage criterion, called the **trans-CIUS-set criterion** is to select a set $\mathcal{T}$ of executable tours such that for each transition t in the EFSM and for each $U_i \in \mathcal{U}$, $\mathcal{T}$ has a tour which traverses t followed by U_i. Apart from guranteeing the coverage of every transition in an EFSM, the test cases generated based on this criterion have the capability of observing and confirming the tail state of each transition. This criterion is similar to the one used in the Uv-method [CVI89] for the FSM model. As the entire CIUS set is applied at the tail state of every transition, the trans-CIUS-set criterion is superior to the existing control flow coverage criterion for the EFSM model. An executable walk W starting from the initial state is called a **preamble walk for** t if Wt is also executable.

5.2 Data Flow Coverage Criterion

The data flow testing is basically to check if the implementation has the right flow of information as its execution proceeds. A hierarchy of data flow coverage criteria including the "all-uses" criterion has been proposed in [RW85] for testing computer programs. For the data flow coverage, we extend the all-uses criterion to what is called a **def-use-ob criterion**. An useful property of the def-use-ob criterion is that the set of test cases selected as per this criterion facilitates the tester to observe every def-use association in the protocol. The observable extension is similar to the one proposed for the IO-def-chain criterion [UY91].

The data flow terminologies such as defínintion (def), use, computation use (c-use), predictate use (p-use), output use (o-use) of a variable or an input interaction parameter are taken from [UY91, CZ93].

A **def-use pair** D with respect to a variable/parameter v is an ordered pair of def and use of v such that there exists a walk in the EFSM which satisfies the following: (i) the first transition in the walk is the one where v is defined (i.e., where the def occurs) and the last transition of the walk is the one where v is used (i.e., where the use occurs) and (ii) v is not redefined in the walk between the location where it is originally defined and the location where it is used. Such a walk is called a **def-clear** walk for D. Let $\mathcal{D}$ be the set of all def-use pairs for all the variables and input interaction parameters.

Our **def-use-ob criterion** requires the selection of a set of executable tours such that for each feasible def-use pair $D \in \mathcal{D}$, the set has at least one tour, say T, satisfying the following conditions.

(a) If the use part in D is an o-use, then T contains a def-clear walk for D.

(b) If the use part in D is a p-use, then T contains a def-clear walk $W1$ for D followed by the CIUS walk $U(j)$, where s_j is the tail state of $W1$.

(c) If the use part in D is a c-use, then T contains a walk $W2$ followed by a walk $W3$, where $W2$ is a def-clear walk for D and $W3$ has an information flow chain [Ram94] from the variable which is defined at the location where the variable for D is c-used to a location where a variable is either o-used or p-used. Moreover, if the information flow chain terminates in a p-use variable, then, in T, $W3$ is followed by the CIUS walk $U(p)$, where s_p is the tail state of $W3$.

Def-Use Pair	Tour
(t3.c1, t11.c2)TRsq	t1t3t1117t20
(t6.c3, t14.c1)TSsq	t2t6t14t8t17t20
(t8.c1, t16.P)S_credit	t1t3t8t16t17t20
(t8.c3, t12.c1)TSsq	t1t3t8t12t8t17t20
(t9.c2, t9.c2)R_credit	t1t3t11t9t9t17t20
(t14.c1, t16.P)S_credit	t1t3t14t16t17t20

Table 4: Sample data flow test tours for EFSM given in Figure 1

Condition (a) takes care of the def-use association for all the def-use pairs in which the use part is an o-use. If the use part of D is a p-use, then apart from meeting the def-use association, by applying the CIUS of s_j, condition (b) enables the tester to observe and check if the predicate of the transition where the p-use occurs evaluates to **true** as expected. On the other hand, if the use part of D is a c-use, then condition (c) enables the tester to observe the effect of the value computed. Actually, this value flows through other intermediate variables along T until it is used in an output statement or in a predicate of a transition. In addition, the correct evaluation of the predicate is ensured by T as in condition (b). An executable walk W starting from the initial state is called a **preamble walk** for the def-use pair D if it satisfies conditions (a), (b), and (c) where T is replaced by W.

5.3 The test case generation method

We have developed an algorithm reported elsewhere [RDT95b] for generating a set of executable test tours for covering the trans-CIUS-set and def-use-ob criteria for a given EFSM. The algorithm has two phases and each phase incrementally constructs the test tours. Starting from the initial state, the first phase traverses the EFSM in a breadth-first fashion in order to compute preamble walks for every transition in the EFSM and for every feasible def-use pairs in $\mathcal{D}$. In the second phase, all preamble walks computed in the first phase are completed into executable tours. These tours are infact the required set of tours for the coverage criteria. Note that a CIUS walk of a state can be postfixed to any executable walk terminating at that state and the resulting walk is also executable. Therefore the use of CIUSes in test cases does not increase the complexity of the feasibility problem in the test case generation.

Table 4 and Table 5 show some sample test tours from the set of executable test tours generated by this algorithm for the EFSM given in Figure 1. In table 4, $(t14.c1, t16.P)S\_credit$, for example, denotes the def-use pair in which S_credit is defined at the first computation statement ($c1$) of transition $t14$ and it is p-used in the predicate of $t16$. [Ram94] has the complete set of test tours generated for this EFSM.

Transition	Preamble	Set of Tours
t8	t1t3	t1t3t8t17t20 t1t3t8t31t17t20 t1t3t8t32t17t20 t1t3t8t18t19
t17	t1t3	t1t3t17t21 t1t3t17t33t20 t1t3t17t34t20 t1t3t17t35t20

Table 5: Sample control flow test tours for the EFSM given in Figure 1

6 Summary

A new type of state identification sequence, namely, the Context Independent Unique Sequence, is defined and an algorithm for computing a CIUS of a given state in an EFSM is developed. While the CIUSes facilitate the observability of the test cases, they do not increase the complexity of the alreday complicated feasibility problem in the automatic generation of executable test cases. Also, the incremental nature of the feasibility problems encountered in the CIUS computation and the test case generation algorithms lends itself to an incremental solution.

In this paper we have focussed on the formalization and the computational aspects of UIO-sequences for the EFSM model. Similar work needs to be done for the other types of state identification sequences such as characterizing sequences and the distinguishing sequences [RDT95a, Koh78]. Extending our work for testing EFSMs which may not have CIUSes for certain states is another direction for further research.

References

[Boc90] G. v. Bochmann. Specifications of a simplified transport protocol using different formal description techniques. *Computer Networks and ISDN systems*, 18:335–377, 1989/1990.

[CA91] W. Chun and P. D. Amer. Test case generation for protocols specified in Estelle. In J. Quemada, J. Manas, and E. Vazquez, editors, *Formal Description Techniques, III*, pages 191–206. Elsevier Science Publishers B. V. (North-Holland), 1991.

[CVI89] W. Y. L. Chan, S. T. Vuong, and M. R. Ito. An improved protocol test generation procedure based on UIOs. In *ACM SIGCOMM*, pages 283–294, 1989.

[CZ93] S. T. Chanson and J. Zhu. A unified approach to protocol test sequence generation. In *Proc. IEEE INFOCOM*, pages 106–114, 1993.

[CZ94] S. T. Chanson and J. Zhu. Automatic protocol test suite derivation. In *Proc. IEEE INFOCOM*, pages 792–799, 1994.

[FMC93] ISO SC21 WG1 P54: Information Processing Systems - Open Systems Interconnection - Formal Methods in Conformance Testing, Working Document, June 1993.

[Hog92] D. Hogrefe. OSI formal specification case study: the Inres protocol and service, Revised. Technical report, Institute for Informatics, University of Berne, May 1992.

[IS8807] ISO/IEC 8807: Information Processing Systems - Open Systems Interconnection - LOTOS - a Formal Description Technique Based on the Temporal Ordering of Observational Behavior, June 1988.

[IS9074] ISO/IEC 9074: Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model, 1987.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[LHHT94] X. Li, T. Higashino, M. Higuchi, and K. Taniguchi. Automatic generation of extended UIO sequences for communication protocols in an EFSM model. In *7th International Workshop on Protocol Test Systems, Tokyo, Japan*, November 1994.

[MP92] R. E. Miller and S. Paul. Generating conformance test sequences for combined control and data flow of communication protocols. In *Proc. 12th International Symposium of Protocol Specification, Testing and Verification*, 1992.

[Ram94] T Ramalingam. *Test case generation and fault diagnosis methods for communication protocols based on FSM and EFSM models.* PhD thesis, Concordia University, Montreal, Canada, 1994.

[RDT95a] T. Ramalingam, A. Das, and K. Thulasiraman. Fault detection and diagnosis capabilities of test sequence selection methods based on the FSM model. *Computer Communications*, 18(2):113–122, February 1995.

[RDT95b] T. Ramalingom, A. Das, and K. Thulasiraman. A unified test case generation method for the EFSM model using context independent unique sequences. In *8th International Workshop on Protocol Test Systems, Evry, France*, September 1995.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Tr. Soft. Engg.*, SE-11(4):367–375, April 1985.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN systems*, 15:285–297, 1988.

[SDL88] CCITT/SGx/WP3-1, Specification and Description Language, SDL. CCITT Recommendations Z.100, 1988.

[Tur93] K. J. Turner, editor. *Using formal description techniques.* John Wiley & Sons, Chichester, England, 1993.

[UY91] H. Ural and B. Yang. A test sequence selection method for protocol testing. *IEEE Tr. Comm.*, 39(4):514–523, April 1991.