18

A Unified Test Case Generation Method for the EFSM Model Using Context Independent Unique Sequences[1]

T. Ramalingom[a] Anindya Das[b] and K.Thulasiraman[c]

[a]Bell-Northern Research Ltd., Ottawa, Canada K1Y 4H7 Tel: (613) 765-5377
E-mail: ramaling@bnr.ca Fax: (613) 763-5782
[b]D.I.R.O., University of Montreal, Montreal, Canada H3C 3J7
[c]School of Computer Science, University of Oklahoma, Norman, OK 73019, U.S.A. On leave from Dept. of Electrical Engineering, Concordia University, Montreal, Canada

A unified method for generating test cases for both control flow and data flow aspects of a protocol represented as an Extended Finite State Machine (EFSM) is presented. Unlike most of the existing methods, the proposed method considers the feasibility of the test cases during their generation itself. In order to reduce the complexity of the feasibility problem without compromising the control flow coverage, a new type of state identification sequence, namely, the Context Independent Unique Sequence (CIUS) is defined. The trans-CIUS-set criterion used in the control flow test case generation is superior to the existing control flow coverage criteria for the EFSM. In order to provide observability, the "all-uses" data flow coverage criterion is extended to what is called the def-use-ob criterion. A two-phase breadth-first search algorithm is designed for generating a set of executable test tours for covering the selected criteria. The approach is also illustrated on an EFSM module of a transport protocol.

Automatic test case generation from protocol standards is a means of selecting high quality test cases efficiently. Recently, International Organization for Standards (ISO) has established a working group for studying the application of Formal Methods in Conformance Testing (FMCT) [5]. One of the primary aims of this group is to enable computer-aided test case generation from protocol standards specified in Formal Description Techniques (FDT) such as Estelle [2], SDL [3], and LOTOS [4]. In this paper, we present a new method for automatically generating test cases for both control flow and data flow aspects of a protocol which is represented as an Extended Finite State Machine (EFSM) as defined in [21].

In order to have better fault coverage [7], some of the test sequence generation methods proposed recently [11, 13, 14] for the EFSM model apply state identification sequences for confirming the states. However, the state identification sequences defined for the FSM model are inadequate for the EFSM model. In this paper, we define a general Unique Input Sequence (UIS) for an EFSM state. We then consider a special type of UIS, called Context Independent Unique Sequence (CIUS) in order to reduce the complexity of the well known feasibility problem associated with the EFSM model that arises during the application of UISs for confirming states.

The test case generation method proposed in this paper addresses both control and data flow aspects of an EFSM. It is known from Finite State Machine (FSM) testing methods that those which use state identification sequences for confirming the tail state of a transition under test have better fault coverage [16, 10, 8]. In particular, the Uv-method has the capability of detecting both label faults and tail state faults in transitions [8]. The control flow fault coverage criterion established in this paper is called **trans-CIUS-set criterion** (defined later) and it is based on the Uv-method. For the data flow coverage, we extend the "all-uses" criterion [17]

[1]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.

to what is called a **def-use-ob criterion**. We shall see that this new criterion is required due to the so called black-box approach of protocol testing and it enhances the observability of the def-use associations. Thus our aim is to generate a set of feasible test cases for the trans-CIUS-set criterion and the def-use-ob criterion. Each test case in the proposed approach corresponds to a test tour which starts and ends at the initial state of the protocol. In the worst case, the cardinality of the set of tours generated is only quadratic in terms of the number of transitions in the protocol.

Most of the existing methods first generate a set of test tours which satisfy the coverage criteria and then check if the generated test tours are feasible [12, 21, 11, 20]. This strategy results in discarding infeasible tours, which in turn affects the coverage criteria. Therefore, an important requirement of our method is to consider the feasibility of the tours during their generation itself. We present a two-phase breadth-first search algorithm which generates a set of feasible test tours which adequately covers the required control flow and data flow criteria. The combined testing method by Miller and Paul [14] addresses the feasibility problem while selecting the test tours. This method does not however handle the feasibility issue effectively while joining different types of test subsequences into a single feasible sequence. Moreover, the trans-CIUS-set criterion and the def-use-ob criterion established in this paper are superior to the respective criterion in [14].

1 The EFSM Model

The EFSM model presented in this paper is inspired from [21]. 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 **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. 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. Let W be a symbolically interpreted walk. Clearly the conjunction of the predicates along W is 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 **satisfiable** if the conjunction of the interpreted predicates is satisfiable. Note that a walk which is executable is always satisfiable. However, its converse is not true. This is because none of the possible values for the variables which made W satisfiable may be a valid context at the starting state of the walk. That is, these values are not 'settable' by any of the executable walks from the initial state to the starting state of W.

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 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** and they do not have computation blocks.

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.

1.1 An Example

As an example of an EFSM , let us consider a major module (*AP-module* in [6]) of a simplified version of a class 2 transport protocol [1]. This module 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). We would like to note that the EFSM is obtained from the *AP-module* by eliminating a few non-determinisms in certain transitions starting from the data transfer state. 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?TrDR(disc_reason, switch), N?TrDT(send_sq, Ndata, EoTSDU),
N?TrAK(XpSsq, cr), N?ready, N?terminated, N?TrDC }

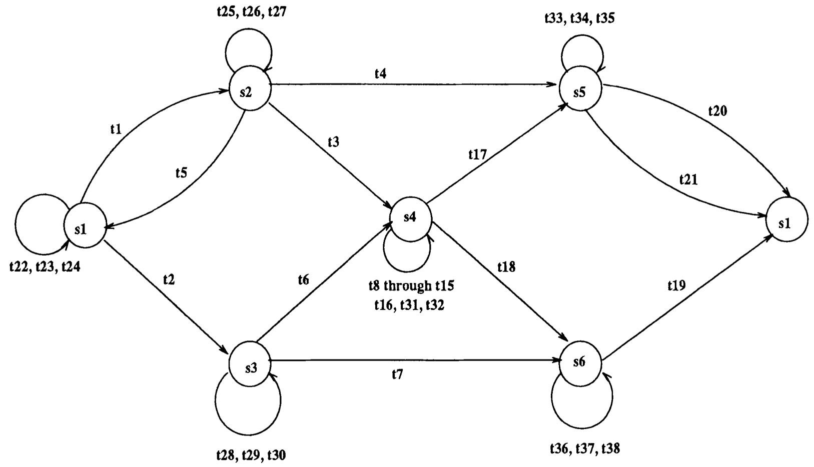

Figure 1: An EFSM for the AP-module in the Class 2 transport protocol

O = {U!TCONconf(opt), U!TCONind(peer_add, opt), U!TDISind(msg), U!TDATAind(data, EoTSDU), U!error, U!READY, U!TDISconf, 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.

1.2 Unique Input Sequence

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. Given an instantiated input sequence X, a state s_i and a valid context C at s_i, $Ewalk(i, X, C)$ denotes the unique walk traversed when X is applied to the EFSM which is currently at s_i with the context C.

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 satisfiable walk W is denoted as **Trace(W)**, known as the **trace of the walk** W. The sequence of input (output) interactions along a walk W is denoted by **Inseq(W)** (**Outseq(W)**). $Trace(W)$ and $Outseq(W)$ are actually obtained by symbolically interpreting W. Suppose that the actual value of a symbol is known, then the corresponding sequences can be obtained from the above sequences by replacing the symbol by the value throughout the sequences.

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

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 in the transport protocol

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 in the transport protocol

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.

An input interaction is obviously distinguishable from an output interaction. The total number of input and output interactions - each occurrence of an interaction is counted - in a sequence is called the **length** of the sequence. Let S_1 and S_2 be two sequences of input and/or output interactions. Assume that they are of the same length. In order to check for distinguishability of the two sequences, starting from the first position the interactions in S_1 and S_2 are checked position-wise. S_1 and S_2 are said to be **distinguishable** if the interactions in at least one position in S_1 and S_2 are distinguishable. Otherwise, they are said to be **indistinguishable**. Two sequences of different lengths are always 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 satisfiable 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.

2 Test Case Selection Criteria

2.1 Control Flow Coverage Criterion

We would like to apply an UIS of every state at the tail state of the transition under test. As indicated in [13], 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 an incoming transition at s_j and s_i be the starting state of t. In order to test t, one needs to compute an executable preamble 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 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 satisfiable 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 computing

State	CIUS	Transition 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: CIUSs for the states in the EFSM of Figure 1

feasible test cases for the control flow coverage. Also, one CIUS of a state is sufficient for testing all the incoming transitions at that state.

In [15], we have developed an algorithm for computing a CIUS for a given state. Table 3 shows the CIUSs for all the states of the EFSM of Figure 1 computed using the algorithm. Note that the parameters in the CIUSs have to be instantiated with certain valid values. We have also found that a few other protocols such as a class 0 transport protocol as specified in [21] and the abracadabra protocol [19] have a CIUS for every state. The maximum length of the CIUSs computed for these protocols 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 as modeled in [9] does not have a CIUS for one state.

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, namely, 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. An executable walk from the initial state to the starting state of a transition t is called a **preamble walk for** t if Wt is also executable. Due to the requirement of applying the entire UIS set at the tail state of a transition under test, the trans-CIUS-set criterion is superior to the existing control flow coverage criteria for the EFSM.

2.2 Data Flow Coverage Criterion

A hierarchy of data flow coverage criteria has been proposed in [17]. It is interesting to know that the "all-uses" is the best criterion among those which can be satisfied by a set of test cases with polynomial order cardinality [17]. Ural and Williams [20] have recently used the all-uses criterion for generating test cases for protocols specified in SDL. Due to the black-box approach of protocol testing, the set of test cases which satisfy the all-uses criterion may not be observable. Therefore, we extend the all-uses criterion to what is called a **def-use-ob criterion**. This criterion facilitates the tester to observe every def-use association in the protocol.

We introduce some definitions before presenting the def-use-ob criterion. A parameter v occurring in the input interaction of a transition t is referred to as a **def** and is denoted by $t.I.v$. Similarly, a variable v in the left side of an assignment statement at the location c in the computation block of a transition t is also said to be a **def** and it is denoted by $t.c.v$. The use of a variable or input interaction parameter v in the predicate of a transition t is called a **p-use** and is denoted by $t.P.v$. The variable/input interaction parameter v used on the right side of an assignment statement at the location $c1$ in the computation block of a transition t is referred to as a **c-use** and is denoted by $t.c1.v$. Similarly, the variable/input interaction parameter v appearing as a parameter in the output interaction at the location $c2$ in the computation block of a transition t is referred to as a **o-use** and it is denoted by $t.c2.v$. By an **use**, we refer to a p-use, a c-use or a o-use.

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 and the last transition of the walk is the one where v is used 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. Note that a def-clear walk could be a single transition. A def-use pair is said to be **feasible** if the EFSM has at least one executable tour which contains a def-clear walk for this pair. The def-use pairs can be classified into five types as follows.

type 1: An input parameter v is defined in the input interaction of a transition t_1 and is used in the predicate of the same transition. Such a pair is denoted by $(t_1.I, t_1.P)v$.

type 2: An input parameter v is defined in the input interaction of a transition t_1 and is used in an output statement c_2 in the computation block of the same transition. Such a pair is denoted by $(t_1.I, t_1.c_2)v$.

type 3: An input parameter v is defined in the input interaction of a transition t_1 and is used in an assignment statement c_3 in the computation block of the same transition. Such a pair is denoted by $(t_1.I, t_1.c_3)v$.

type 4: A variable v is defined in an assignment statement c_1 in the computation block of a transition t_1 and is used in the predicate of another transition t_2. Such a pair is denoted by $(t_1.c_1, t_2.P)v$.

type 5: A variable v is defined in statement c_1 in the computation block of a transition t_1 and is used in statement c_2 in the computation block of a transition t_2. Such a pair is denoted by $(t_1.c_1, t_2.c_2)v$.

Let l (l') be a location in transition t (t') where a variable/parameter v (v') is defined (used). Suppose that $X = D_1 D_2 \ldots D_k$, where $k \geq 1$, is a sequence of def-use pairs such that (i) D_i is a def-use pair for variable v_i, $i = 1, 2, \ldots, k$, (ii) $v_1 = v$ and $v_k = v'$ and the source of D_1 is $t.l$ and the destination of D_k is $t'.l'$, (iii) the use part of D_i is for defining v_{i+1}, where $i = 1, 2, \ldots, k-1$ and (iv) if $k = 1$, then $v = v'$. Then, X is called an **information flow chain** from the definition of v at the location l of transition t to the use of v' at the location l' of transition t'. Further, if a walk W has a subwalk W' with t and t' as the first and the last transition such that W' can be expressed as $W' = W_1 @ W_2 @ \ldots @ W_k$, where W_i is a def-clear walk for D_i, for $i = 1, 2, \ldots, k$, then, we say that X is an **information flow chain along** W. In this case, we also say that W has an information flow chain from the definition of v at the location l of transition t to the use of v' at the location l' of transition t'. We would like to note that the information flow chain is somewhat similar to the IO-def-chain proposed in [21].

Let $\mathcal{D}$ be the set of all def-use pairs for all the variables and input interaction parameters in the EFSM. A minor modification of the algorithm presented in [9] would suffice to obtain $\mathcal{D}$. This modification is to consider the def-use pairs within a transition. 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 $W3$ such that $W2$ is a def-clear walk for D and $W3$ has an information flow chain 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 $W3$.

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 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** D if it satisfies conditions (a), (b) and (c) where T is replaced by W.

We know that, as per the trans-CIUS-set criterion, each transition followed by the CIUS of the tail state of the transition will be covered by at least one tour. Clearly, this tour also covers all the def-use pairs of types 1 and 2 for the def-use-ob criterion. Henceforth, we assume that $\mathcal{D}$ consists of types 3 4 and 5 only.

We define a new type of Data Flow Graph (DFG) to represent the data flow information on a particular executable walk starting from the initial state. This graph is useful in computing the subset of $\mathcal{D}$, for which this walk is a preamble walk, except possibly for the CIUS walk extension. The data flow graph has four types of nodes: i-node, c-node, p-node and o-node.

- An **i-node** is labeled as (t, I, v) and it corresponds to the definition of the parameter v in the input interaction of the transition t.
- A **c-node** is labeled as (t, c, v) and it corresponds to the definition of the variable v in the assignment statement c of the transition t.
- A **p-node** is labeled as (t, P) and it indicates that the node corresponds to the predicate of the transition t.
- A **o-node** is labeled as (t, c) and it simply denotes that it corresponds to the output statement c in the computation block of the transition t.

The **data flow graph** for the transition t with respect to the walk W which contains t is denoted by DFG$[t, W]$. It contains the data flow information along W for all the input interaction parameters and local variables defined in t. It has one connected directed subgraph, say G, for each definition of a variable or an input interaction parameter, say v, in t. G has a designated node, called the **root node** which identifies the definition of v. A given node in G is considered to be in one of three different levels. The root node is the unique node in the first level. Nodes in level 2 correspond to the direct use of v in statements/predicates in W and W contains a def-clear walk for every def-use pair consisting of the root node and a node in level 2. The root node is connected to all the nodes of level 2. A node is in level 3 if there exists a data flow along W from at least one assignment statement which corresponds to a c-node in level 2 to a predicate, assignment statement, or an output statement corresponding to this level 3 node. A c-node in level 2 is connected to a level 3 node if there exists an information flow chain along W from the level 2 node to the level 3 node.

Figure 2 shows the data flow graph DFG$[t3, t1t3t8]$, for the transition $t3$ in the walk $t1t3t8$ of the EFSM given in Figure 1. In Figure 2, rectangles represent i-nodes as well as o-nodes, whereas the circles and diamonds represent c-nodes and p-nodes, respectively. The second subgraph in this data flow graph, for instance, corresponds to the definition of the input interaction parameter *cr*. Observe that the edges from $(t3, c4, S\_credit)$ to the level 3 nodes $(t8, P)$ and $(t8, c1, S\_credit)$ indicate that the variable *S_credit* defined in $t3.c4$ is p-used at the predicate of transition $t8$ and c-used in the definition of *S_credit* at the first statement in the computation block of $t8$, respectively.

The size of the test cases required for satisfying the coverage criteria is summarized in the following theorem.

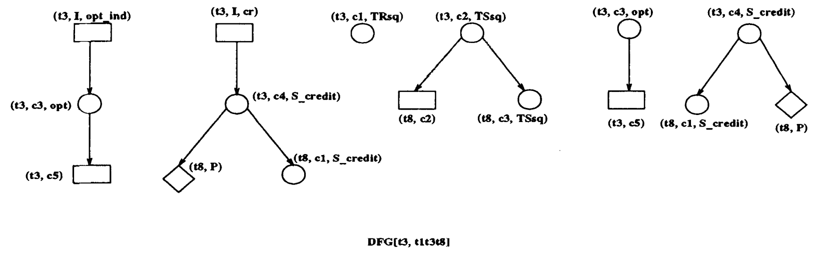

Figure 2: A data flow graph for $t3$ with respect to the walk $t1t3t8$

Theorem 1 *The order of the set of test tours required to satisfy the trans-CIUS-set and the def-use-ob criteria together is quadratic in the number of transitions in the EFSM.*

3 Data Flow Graph Manipulation

In this section, we briefly describe the procedures for constructing and manipulating the data flow graph DFG$[t, W]$ for a given transition t which is a part of a given executable walk W starting from the initial state of an EFSM. These procedures are used in our test case generation algorithm for checking if a walk is a preamble walk for some def-use pairs.

Our first procedure *PredExtendGraph* is for processing a predicate in a given transition. The procedure accepts a walk $W2$, a transition $t2$, where $t2$ is the last transition in $W2$, and a partial subgraph G of DFG$[t3, W2]$, for some transition $t3$ in $W2$. Let G correspond to a variable/parameter u defined at $t3$. G is partial since it does not have the data flow information corresponding to the transitive use of u in $t2$. As described below, *PredExtendGraph* extends the graph G if the value of u is eventually used in the predicate of $t2$. The variable *inlevel*2 (*inlevel*3) is used to ensure that the p-node $(t2, P)$ is created atmost once in level 2 (level 3) of G. This procedure also checks if $W2$ is a preamble walk for a def-use pair along $W2$ where the definition corresponds to the root node of G. For notational convenience, we denote a node at a given level by attaching the level number as a subscript to the label of the node. For example, a c-node (t, c, v) at level 3 is also denoted by $(t, c, v)_3$. Comments are enclosed in braces.

```
procedure PredExtendGraph(G:graph; t2:transition; W2:walk);
begin
   inlevel2 := false; inlevel3 := false;
   Let (t1,x1,u) be the root node of G; { x1 = 'I' or assignment stmt. no. }
   for each variable v used in t2.pred do begin
      Let (t, c) = W2.recentdef(v); {Recent definition of $v$ in $W2$ is at $t.c$}
      if ((t,c,v) is the root node of G) then begin { (t,c,v)= t1,x1,u) }
         if (not inlevel2) then begin
            Create a p-node (t2,P) at level 2 in G; inlevel2 := true;
         end;
         Add an edge from $(t1,x1,u)_1$ to $(t2,P)_2$ in G;
         if (D = (t1.x1, t2.P)(u)$\in \mathcal{D}$ is not yet covered) then begin
            Mark D as covered;
            Obtain a preamble walk for $D$ by appending U(j) to W2,
               where $s_j$= t2.dest & U(j) is the CIUS walk for $U_j$;
         end
      end;
      if ((t,c,v) is a node at level 2 in G) then begin
         if (not inlevel3) then begin
```

```
            Create a p-node (t2,P) at level 3 in G; inlevel3 := true;
        end;
        Add an edge from $(t,c,v)_2$ to $(t2,P)_3$ in G;
        if (D = (t1.x1, t.c)(u)∈ $\mathcal{D}$ is not yet covered) then begin
            Mark D as covered;
            Obtain a preamble walk for $D$ by appending U(j) to W2,
                where $s_j$= t2.dest & U(j) is the CIUS walk for $U_j$;
        end
    end;
    if ((t,c,v) is a node at level 3 in G) then begin
        if (not inlevel3) then begin
            Create a p-node (t2,P) at level 3 in G; inlevel3 := true;
        end;
        for each incoming edge e to (t, c, v) do begin
            Let $(t', c', v')_2$ be the starting node of e;
            Add an edge from $(t', c', v')_2$ to $(t2,P)_3$ in G;
            if (D = (t1.x1, $t'.c'$)(u)∈ $\mathcal{D}$ is not yet covered) then begin
                Mark D as covered;
                Obtain a preamble walk for $D$ by appending U(j) to W2,
                where $s_j$= t2.dest & U(j) is the CIUS walk for $U_j$;
            end
        end
    end
  end { for each variable v }
end { PredExtendGraph }
```

StmtExtendGraph and *OutputExtendGraph* are the other two procedures for extending a subgraph of a data flow graph with respect to an assignment statement and an output statement, respectively. They are similar to *PredExtendGraph* [15].

We shall now describe procedure *ExtendDFG*. This procedure accepts a walk $W1$, a transition $t1$ in $W1$, and a transition $t2$ which starts from the tail state of $W1$ and it computes DFG[$t1, W1\ t2$], the data flow graph for $t1$ with respect to the walk $W1\ t2$. *ExtendDFG* achieves this by extending the already known data flow graph DFG[$t1, W1$] as per the data flows along $W1\ t2$ from the variables/parameters defined in $t1$ to the variables used in the predicates and the statements in $t2$. Let $W2 = W1\ t2$. Let us assume that the set of def-use pairs in $\mathcal{D}$ which are yet to be covered for the def-use-ob criterion is known at the starting of the procedure. After copying DFG[$t1, W1$] into DFG[$t1, W2$], it manipulates each subgraph in DFG[$t1, W2$] with respect to the variables used in the predicate of $t2$. It calls the procedure *PredExtendGraph* for this purpose. It then sequentially selects every statement in the computation block of $t2$, and updates every subgraph in DFG[$t1, W2$] by considering all the variables/parameters used in the statement. If it is an assignment statement, then *ExtendDFG* calls the procedure *StmtExtendGraph*; otherwise it invokes *OutputExtendGraph* for updating a given subgraph. The formal description is given below.

```
procedure ExtendDFG(t1:transition;W1:walk;t2:transition);
begin
    Let W2 be the walk obtained by appending t2 to the walk W1;
    DFG[t1,W1] := DFG[t1,W2];
    for each subgraph G in DFG[t1,W2] do
        PredExtendGraph(G, t2, W2);
    { Sequentially process the statements in the compute-block of t2 }
    for each statement c2 in the compute-block of t2 do
        for each subgraph G in DFG[t1,W2] do
            if (c2 is an assignment statement) then
                StmtExtendGraph(G, t2, c2, W2)
            else OutputExtendGraph(G, t2, c2, W2);
end; { ExtendDFG }
```

Our final procedure for DFG manipulation is *ConstructDFG* for constructing DFG[t, t] for every transition t in an EFSM. It is very similar to *ExtendDFG* but for the fact that it starts with an empty data flow graph. It is easy to see that the data flow graph DFG$[t, W]$ of a transition t with respect to a walk W which contains t can be constructed using *ConstructDFG* and *ExtendDFG.*

4 Automatic Test Case Generation

4.1 The Two-Phase Algorithm

We have already established the trans-CIUS-set criterion for the control flow testing and the def-use-ob criterion for data flow testing. The next step is to generate a set of test cases satisfying these criteria. The algorithm presented in this section systematically generates a set of executable test tours for covering the above criteria. It has two phases and it traverses the EFSM in a breadth-first fashion in both phases. The first phase constructs a preamble walk for every transition in the EFSM and for the feasible def-use pairs in $\mathcal{D}$. In the second phase, all preambles computed in the first phase are completed into a set of executable tours.

The step-wise description of the first phase of the algorithm is given below. The salient points in the algorithm are then discussed. For ease of understanding, each step is embedded with comments.

Phase I

Input: An EFSM, CIUS-set $\mathcal{U} = \{U_j \mid 1 \leq j \leq n\}$, Def-use pairs set $\mathcal{D}$. A positive integer K_1.

Output: UFset: set of preamble walks for the coverage criteria.

Step 0 { Data flow graphs initialization }

(i) Construct the data flow graph of each transition with respect to itself.

Step 1 { null walk initialization }

(i) Let P be a null walk at s_1; Let $\mathcal{P} = \{P\}$.

Step 2 { ith iteration of this step computes the set of all executable walks of length i starting from s_1. They are computed from the executable walks of length $i-1$ computed in the previous iteration. This step marks all transitions & def-use pairs covered by the new walks.}

(i) Let $\mathcal{T} = \emptyset$.

(ii) Do Step 2.1 for each $P \in \mathcal{P}$ and for each outgoing transition t from the tail state of P.

(iii) If all the transitions in the EFSM are covered for control flow and all the def-use pairs in $\mathcal{D}$ are covered for data flow or the number of iterations of Step 2 exceeds K_1, a fixed positive integer, then proceed to Step 3.

(iv) Consider $\mathcal{T}$ as $\mathcal{P}$ and repeat Step 2.

Step 3 { For every transition t, and for every CIUS, postfix t followed by the walk along the CIUS to the preamble walk. Also collect the resulting walks for the transitions as well as the preamble walks for the def-use pairs into *UFset.*}

(i) Let both *CFset* and *DFset* to be the empty set.

(ii) For each transition t covered by Step 2 and for each CIUS $U_k, 1 \leq k \leq n$, add $W@t@Ewalk(j, U_k, C)$ to CFset, where W is the preamble walk computed for t, s_j is the tail state of t and C is the context after executing $W@t$.

(iii) For each def-use pair $D \in \mathcal{D}$ covered by Step 2, add the preamble walk for D computed in Step 2 to *DFset.*

(iv) Let *UFset* = *CFset* $\cup$ *DFset.* Delete each walk $W \in$ *UFset* such that W is a prefix of some other walk in *UFset.*

(v) Stop.

Step 2.1

(i) Let $Q = P\ t$. If Q is executable and t is not yet covered for control flow then mark t as covered and take P as the preamble walk for t.

(ii) If Q is executable and either t is not a self-loop or t has at least one assignment statement in its computation block then add Q to $\mathcal{T}$.

(iii) If Q is executable then do Step 2.1.1.

Step 2.1.1

(i) For each $t' \in P$, (a) Compute DFG$[t', Q]$ from DFG$[t', P]$, (b) Mark all the def-use pairs covered by Q, and (c) Construct an appropriate preamble walk for each such pair.

(ii) Consider DFG$[t, t]$ to be DFG$[t, Q]$.

Observe that the first phase starts by constructing DFG$[t, t]$, for every transition t in the given EFSM. This can be done using the procedure *ConstructDFG*. Starting from the initial state, Step 2 traverses the EFSM in a breadth-first fashion, in order to compute the preambles for each transition and for each feasible def-use pair in $\mathcal{D}$. At the starting of the kth iteration of Step 2, $k \geq 1$, $\mathcal{P}$ consists of the set of all executable walks of length $k - 1$ which start from the initial state. The kth iteration of this step computes the set of all executable walks of length k by extending the walks in $\mathcal{P}$ by single transitions. The executability of the extended walk is checked only with respect to the last transition since the rest of the walk is known to be executable at this point. This reduces the complexity of the feasibility problem to a great extent.

For each walk $P \in \mathcal{P}$ and for each transition t from the tail state of P, Step 2.1 checks if the walk Q obtained by postfixing t to P is executable. When Q is executable, Step 2.1 uses Step 2.1.1 for computing the data flow graphs pertaining to Q, for determining the def-use pairs in $\mathcal{D}$ covered by Q, and for selecting a preamble walk for every def-use pair covered by Q. Step 2.1.1 can be achieved using the procedure *ExtendDFG* which extends DFG$[t', P]$ to DFG$[t', Q]$, for all t' in P.

Step 2 is repeated until the preambles for all the transitions are computed and all def-use pairs in $\mathcal{D}$ are covered or the number of iterations of Step 2 exceeds a fixed positive integer K_1. K_1 depends on the given EFSM. It has to be chosen in such a way that the preambles for all the transitions are computed in K_1 iterations of Step 2. Recall that, for every transition, the EFSM is assumed to have at least one feasible walk from the initial state such that the transition is executable for the resulting context. Therefore, the preambles for all the transitions are computable in a finite number of iterations of Step 2. Observe that some of the def-use pairs in $\mathcal{D}$ may not be feasible. Also, the problem of finding whether a given pair is feasible or not is undecidable. If $\mathcal{D}$ has some infeasible pairs, then this phase terminates after K_1 iterations of Step 2.

Phase II described below is essentially for completing each walk in *UFset*, computed in Phase I, into an executable tour. These tours are in fact the ones required for the trans-CIUS-set and the def-use-ob criteria. The algorithm is self-explanatory and further description is omitted.

Phase II

Input: The EFSM considered in Phase I and the UFset returned by Phase I

Output: UFTourset, a set of tours for the selection criteria

Step 1 { Initialization }

(i) Let P be a null walk at s_1; Let $\mathcal{P} = \{P\}$.

(ii) Let *UFTourset* be the empty set.

Step 2 { ith iteration of this step computes the set $\mathcal{T}$ of all satisfiable walks of length i ending at s_1. The set of all preambles in *UFset*, which are executable in conjunction with a walk in $\mathcal{T}$ which starts at the tail state of the preambles, are declared to be covered by the tour obtained by prefixing the preamble to the walk. }

(i) Let $\mathcal{T}$ be the empty set.

(ii) Do Step 2.1 for each $P \in \mathcal{P}$ and for each transition t starting from a state other than s_1 and ending at the starting state of P.

(iii) If all the walks in *UFset* are covered, then stop.

(iv) Consider $\mathcal{T}$ as $\mathcal{P}$ and repeat Step 2.

Step 2.1

(i) Let $Q = t\ P$. If Q is satisfiable, then add Q to $\mathcal{T}$.

(ii) Do Step 2.1.1 for each walk W in *UFset* such that $W\ Q$ is a tour provided Q is satisfiable.

Step 2.1.1

(i) If $W\ Q$ is executable then Add $W\ Q$ to *UFTourset* and mark W as covered.

The time and space complexities and correctness of the algorithm are summarized below. The proof of the theorem and a detailed refinement of the above algorithm is presented in [15].

Theorem 2 *Let K_2 (K_1) be the number of times (maximum number of times) Step 2 of Phase II (Phase I) is executed. The time complexity of the algorithm is $O((d^{out}_{max})^{K_1+1} + (d^{in}_{max})^{K_2+1})$ steps, where d^{in}_{max} (d^{out}_{max}) denotes the maximum number of incoming (outgoing) transitions including the self-loops at any state in the EFSM. The algorithm also requires $O((d^{out}_{max})^{K_1} + (d^{in}_{max})^{K_2})$ units of memory. It successfully computes an executable tour for those transitions which have at least one preamble walk of length at most K_1. The algorithm computes an executable tour for every feasible def-use pair in $\mathcal{D}$ which have at least one preamble walk of length at most K_1 excluding their CIUS subwalk extension.*

□

Corollary 1 *For a suitable value of $K_1, 1 \leq K_1 < \infty$, the algorithm successfully computes a set of tours such that (i) the set satisfies the trans-CIUS-set criterion, and (ii) the set satisfies the def-use-ob criterion if $\mathcal{D}$ has only feasible def-use pairs.*

4.2 Fault Coverage

Let us assume that the Implementation Under Test (IUT) is represented as a deterministic, completely specified EFSM having the same set of input interactions and states as the specification EFSM. It is known that some of the FSM-based test sequence generation methods achieve complete fault coverage capability by including the verification of the state identification sequences in the IUT [7, 10, 8]. In the EFSM model, in order to establish that an input sequence is an UIS of a state in the IUT, one has to show that for any valid context of the IUT at that state, the output sequence produced by the IUT while applying the input sequence is different from the output sequence obtained by applying the input sequence at any other state with every valid context. Due to the black-box approach of testing, it is, in general, difficult to achieve this UIS verification requirement. For each incoming transition at a state s_i, our test case generation method generates one feasible tour for applying the CIUS U_i at s_i to see if it provides the expected output, and a tour for applying the CIUS U_j of the

Def-Use Pair	Preamble	Tour
(t3.c4, t8.c1)S_credit	t1 t3 t8 t8 **t17**	t1 t3 t8 t8t17t20
(t6.c2, t9.P)TRsq	infeasible	
(t6.c2, t9.c1)TRsq	infeasible	
(t6.c3, t12.P)TSsq	t2 t6 t12 **t17**	t2 t6 t12t17t20
(t6.c3, t12.c1)TSsq	t2 t6 t12 t8 **t17**	t2 t6 t12 t8t17t20
(t6.c3, t13.P)TSsq	t2 t6 t13 **t17**	t2 t6 t13t17t20

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

Transition	Preamble	Set of walks	Tour
t6	t2	t2t6t17	t2t6t17t20
		t2t6t31	t2t6t31t17t20
		t2t6t32	t2t6t32t17t20
		t2t6t18	t2t6t18t19
t7	t2	t2t7t19	t2t7t19
		t2t7t36	t2t7t36t19
		t2t7t37	t2t7t37t19
		t2t7t38	t2t7t38t19

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

state s_j, $j = 1, 2, \ldots, n, j \neq i$ at s_i to check if it produces the output different from the one obtained when U_j is applied at s_j. Further, these tours can be exercised for different data in their feasible domain. Thus our method establishes the CIUS verification requirement at least partially, while the existing EFSM based test generation methods do not consider this issue. In addition, the test tours selected are all feasible and for a suitable value for K_1, they satisfy the control flow criterion. Therefore, the control flow fault coverage of this method is the same or better than those guaranteed by the existing EFSM based test sequence generation methods.

5 Transport Protocol Test Case Generation

In [15] we have illustrated our test case generation algorithm on the transport protocol given in Figure 1. We shall summarize the results here. Only core transitions are considered for the coverage criteria. There are 80 def-use pairs satisfying the all-uses criterion. Among them 7 are infeasible. Some of the def-use pairs are shown in the first column of Table 4. Phase I computes the preamble walks for all the transitions by the fourth iteration of Step 2. The preamble walks selected for some of the transitions are shown in the second column in Table 5. Note that the walks in the third columns in this table are obtained by appending the preamble walk with the transition followed by a CIUS walk. By the fifth iteration pramble walks for all the feasible def-use pairs have been computed. The second column in Table 4 shows the preamble walks for the selected def-use pairs. Observe that the bold faced transition appended to a walk in the table is for confirming the tail state of the last transition whose predicate transitively uses the value of the variable in the corresponding def-use pair. After deleting the duplicate walks, Phase I produces 128 walks. Phase II for completing these walks in to feasible tours is fairly straight forward for the EFSM in Figure 1. For instance, since none of the incoming transitions ($t5, t19, t20$ and $t21$) at state s_1 has predicate, in the first iteration, all the walks output by Phase I which terminate at the starting states (s_2, s_5 and s_6) of these transitions are completed into executable tours by concatenating the appropriate transitions from $\{t5, t19, t20, t21\}$. With in two iterations of Step 2, Phase II successfully finds a set of executable tours for all the walks selected in the first phase. The last columns of Table 4 and Table 5 show some of the selected tours. This set of tours satisfies both the trans-CIUS-set and the def-use-ob criteria.

Let us examine the fault detection capability of the generated test tours through examples. Suppose that an IUT has a simple control flow fault at the transition $t6$, which originally ends at s_4. Let the tail state of this transition in the IUT be s_2. While applying a test data along the tour $t2t6t17t20$ which is one of the tours for covering the trans-CIUS-criterion for $t6$ (refer to Table 5), it shows an output mismatch. Therefore the fault is detected.

Suppose that the IUT has a variable definition fault at $t3.c4$ where the variable *S_credit* is defined. That is , in $t3.c4$, *S_credit* is replaced by some other variable, say *R_credit*. Let us assume that the default value for all the integer variables is zero. Take the def-use pair $D = (t3.c4, t8.c1)S\_credit$. From Table 4, we see that $T = t1t3t8t8t17t20$ is the required tour for covering D with respect to the def-use-ob criterion. Observe that for any feasible test data for T, the expected sequence along the tour is different from the one observed in the IUT. Thus, the presence of the fault is detected.

6 Conclusion

The Context Independent Unique Sequence defined in this paper is very useful in generating executable test cases for both control and data flow in an EFSM. The trans-CIUS-set criterion is superior to the existing control flow coverage criteria for the EFSM. In order to provide observability, the "all-uses" data flow coverage criterion is extended to what is called the def-use-ob criterion. Finally, a two-phase breadth-first search algorithm is designed for generating a set of executable test tours for covering the selected criteria.

In order to generate the control flow test cases for EFSM model with only integer variables, Li *et al* have recently defined an Extended UIO-sequence (EUIO-sequence, in short)[13]. We observe that if an UIO-sequence is also an EUIO-sequence, then the input part of this sequence becomes a CIUS. While a number of EUIO-sequences are required to test all the incoming transitions at a given state one CIUS is sufficient for this purpose. Also, there is no algorithm presently available for computing EUIO-sequences.

The problem of finding a set of test data for executing each tour selected by a test case generation algorithm such that the data-oriented faults are detected is certainly an interesting research problem. We believe that the set of tours generated by our approach is a good candidate for the test data selection problem, since (i) all the tours generated are executable and (ii) it provides observability of the data flow. The fault based techniques as described in [18] would be helpful to gain more insight on this problem.

Since the EFSM model considered in this paper is similar to a module in Estelle or SDL, an interesting area for future study is to integrate our test case generation method with the existing tools for these FDTs. Such an integrated tool will be useful to automatically generate test cases for real-life protocols specified in Estelle and SDL.

Extending our work to EFSMs which may not have CIUSs for certain states is another direction for further research.

References

[1] ISO TC97/SC6 8073: Information Processing Systems - Open Systems Interconnection - Connection Oriented Transport Protocol Specification.

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

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

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

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

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

[7] G. v. Bochmann, A. Petrenko, and M. Yao. Fault coverage of tests based on finite state models. In *7th International Workshop on Protocol Test Systems, Tokyo , Japan*, November 1994.

[8] 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.

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

[10] T. S. Chow. Testing software design modeled by finite state machine. *IEEE Tr. Soft. Engg.*, SE-4(3):178–187, March 1978.

[11] 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.

[12] B. Forghani and B. Sarikaya. Semi-automatic test suite generation from Estelle. *IEE/BCS Software Engineering Journal*, 7(4):295–307, July 1992.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. C. Thompson, D. J. Richardson, and L. A. Clarke. An information flow model of fault detection. In *Proc. International Symposium on Software Testing and Analysis*, pages 182–192, Cambridge, USA, June 1993. ACM press.

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

[20] H. Ural and A. Williams. Test generation by exposing control and data dependencies within system specifications in SDL. In *Proc. FORTE'93*, October 1993.

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