Automatic executable test case generation for extended finite state machine protocols

C. Bourhfir[1], R. Dssouli[1,2], E. Aboulhamid[1], N. Rico[2]
[1] DIRO, Pavillon André Aisenstadt, C.P. 6128, succursale Centre-Ville, Montréal, Québec, H3C-3J7, Canada.
E-mail: {bourhfir, dssouli, aboulham}@iro.umontreal.ca.
[2] Nortel, 16 Place du commerce, Verdun, H3E-1H6

Abstract
This paper presents a method for automatic executable test case and test sequence generation which combines both control and data flow testing techniques. Compared to published methods, we use an early executability verification mechanism to reduce significantly the number of discarded paths. A heuristic which uses cycle analysis is used to handle the executability problem. This heuristic can be applied even in the presence of unbounded loops in the specification. Later, the generated paths are completed by postambles and their executability is re-verified. The final executable paths are evaluated symbolically and used for conformance testing purposes.

Keywords
EFSM, Conformance testing, Control flow testing, Data flow testing, Executability, Cycle Analysis, Symbolic evaluation, Test case generation.

1 INTRODUCTION

In spite of using a formal description technique for specifying a system, it is still possible that two implementations derived from the same specification are not compatible. This can result from incorrect implementation of some aspects of the system. This means that there is a need for testing each implementation for conformance to its specification standard. Testing is carried out by using test sequences generated from the specification.

With EFSMs, the traditional methods for testing FSMs such as transition tours, UIOs, distinguishing sequences (DS), or W-Method are no longer adequate. The extended data portion which represents the data manipulation has to be tested also to determine the behaviors of the implementation. Quite a number of methods have been proposed in the literature for test case generation from EFSM specifications using data flow testing techniques (Sarikaya, 1986) (Ural, 1991) (Huang, 1995). However, they have focused on data flow testing only and control flow has been ignored or considered separately, and they do not consider the executability problem. As to control flow test, applying the FSM-based test generation methods to EFSM-based protocols may result in non-executable test sequences. The main reason is the existence of non-satisfied predicates and conditional statements. To handle this problem, data flow testing has to be used.

The generation of test cases in the field of communication protocols, combining both control and data flow techniques, has been well studied. In (Chanson, 1993), the authors presented a method for automatic test case and test data generation, but many executable test cases were not generated. This method uses symbolic evaluation to determine how many times an influencing self loop should be executed. An influencing transition is a transition which changes one or more variables that affect the control flow, and a self loop is a transition which starts and ends at the same state. The variables are called influencing variables. (Ural, 1991) does not guarantee the executability of the generated test cases because it does not consider the predicates associated with each transition. Also control flow testing is not covered. (Huang, 1995) generates executable test cases for EFSM-based protocols using data flow analysis and control flow is not tested. To handle the executability problem, this method uses a breadth-first search to expand the specification graph, according to the inputs read and to the initial configuration. It is a kind of reachability analysis. Hence, it has the same disadvantage, i.e. state explosion.

In this paper, we present a method which alleviates some of the existing problems. This method is different from (Huang, 1995) because it combines control and data flow testing instead of using only data flow testing. Unlike (Chanson, 1993) which verifies the executability after all the paths are generated and which considers only the self loops to solve the executability, our method verifies the executability during path generation which prevents from generating paths which will be discarded later. To make the non-executable paths executable, Cycle Analysis is performed in order to find the shortest cycle to be inserted in a path so that it becomes executable. A cycle is one or many transitions $t_1,t_2,..,t_k$ such that the ending state of t_k is the same as the starting state of t_1. Our method can also generate test cases for specifications with unbounded loops.

In the next section, concepts such as the FSM and EFSM models, conformance testing, data flow and control flow testing are described. Section 3 presents the general algorithm for executable test case and test sequence generation. In sections 4 and 5, the algorithm for executable definition-uses paths (or du-paths) generation is presented. This latter checks the executability during the du-path generation and uses cycle analysis to make the non-executable paths executable. Finally, in the last sections, we will compare the results obtained by our tool to those of another method and conclude the paper.

2 PRELIMINARIES

2.1 The FSM and EFSM models

Formalized methods for the specification and verification of systems are developed for simplifying the problems of design, validation and implementation. Two basically different approaches have been used for this purpose: modeling by FSMs, and specifications using high-level modeling languages.

The FSM model falls short in two important aspects: the ability to model the manipulation of variables conveniently and the ability to model the transfer of arbitrary values. For this reason, an FSM becomes cumbersome for simple problems (state explosion) because the number of states grows rapidly. This type of problems can be alleviated when EFSMs are used.

An EFSM is formally represented as a 6-tuple <S, s_0,I, O, T,V> where

1. S is a non empty set of states,
2. s_0 is the initial state,
3. I is a nonempty set of input interactions,
4. O is a nonempty set of output interactions,
5. T is a nonempty set of transitions,
6. V is the set variables.

Each element of T is a 5-tuple t=(initial_state, final_state, input, predicate, block). Here *initial_state* and *final_state* 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. *predicate* is a predicate expressed in terms of the variables in V, the parameters of the input interaction and some constants. *block* is a set of assignment and output statements.

We assume that the EFSM representation of the specification is deterministic and that the initial state is always reachable from any state. In order to simplify the determination of the control and data flow graphs of a formal specification, it is convenient to transform the specification into an equivalent form containing only the so-called "*Normal Form Transitions*" (NFT). A method for generating a normal form specification from an ESTELLE specification is given in (Sarikaya, 1986).

2.2 Conformance testing

There are two approaches for checking conformance between an implementation and a specification. One approach is verification and the other is conformance testing. While verification techniques are applicable if the internal structure of the implementation is known, conformance testing aims to establish whether an implementation under test (IUT) conforms to its specification. If the implementation is given as a *black box*, only its observable behavior can be tested against the observable behavior of the specification. During a conformance test, signals are sent to (inputs) and received from (outputs) the implementation. The signals from the implementation are compared with the expected signals of the specification. The inputs and the expected outputs are described in a so-called test suite. A test suite is structured into a set of test cases. The execution of a test case results in a test verdict. From the test verdicts a conclusion about the conformance relation is drawn.

In recent years, several approaches have been developed for conformance test generation; these techniques are based upon traditional finite automata theory and usually assume a finite-state machine (FSM).

2.3 Fault models and control flow testing

The large number and complexity of physical and software failures dictate that a practical approach to testing should avoid working directly with those physical and software failures. One method for detecting the presence or absence of failures is by using a fault model to describe the effects of failures at some higher level of abstraction (logic, register transfer, functional blocks, etc.) (Bochmann, 1991).

The purpose of control flow testing is to ensure that the IUT behaves as specified by the FSM representation of the system and the fault model used to test it is the FSM model. The most common types of errors it tries to find are *transition (or operation) errors* which are errors in the output function and *transfer errors* (errors in the next state function) in the IUT.

Many methods for control flow testing exist. They usually assume that the system to be tested is specified as an FSM (transition tours, DS, W, etc.). Many attempts were made to generalize these methods to EFSM testing (Ramalingom, 1995) (Chanson, 1993). For control flow testing, we choose the UIO sequence for state identification since the input portion is normally different for each state and the UIO sequence for a state distinguishes it from all other states.

2.4 Data flow analysis

This technique originated from attempts in checking the effects of test data objects in software engineering. It is usually based on a data flow graph which is a directed graph with the nodes representing the functional units of a program and the edges representing the flow of data objects. The functional unit could be a statement, a transition, a procedure or a program. Data flow analyzes the data part of the EFSM in order to find data dependencies among the transitions. It usually uses a data-flow graph where the vertices represent transitions and the edges represent data and control dependencies. The objective is to test the dependencies between each definition of a variable and its subsequent use(s).

Definitions

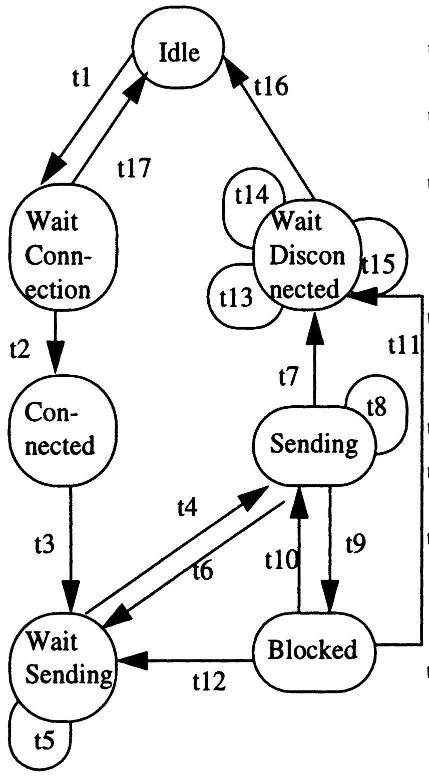
A transition T has an assignment-use or **A-Use** of variable x if x appears at the left hand side of an assignment statement in T. When a variable x appears in the input list of T, T is said to have an input-use or **I-Use** of variable x. If a variable x appears in the predicate expression of T, T has a predicate-use or **P-Use** of variable x. T is said to have a computational-use or **C-Use** of variable x if x occurs in an output primitive or an assignment statement (at the right hand side). A variable x has a definition (referred to as **def**) if x has an A-Use or I-Use.

We now define some sets needed in the construction of the path selection criteria: def(i) is the set of variables for which node i contains a definition, C-Use(i) is the set of variables for which node i contains a C-use and P-Use(i,j) is the set of variables for which edge (i,j) contains a P-use. A path (t_1,t_2...t_k,t_n) is **a def-clear-path** with respect to (w.r.t) a variable x if t_2,..,t_k do not contain definitions of x.

A path (t_1,...,t_k) is **a du-path** w.r.t a variable x if $x \in def(t_1)$ and either $x \in C-Use(t_k)$ or $x \in P-Use(t_k)$, and (t_1,...,t_k) is a def-clear path w.r.t x from t_1 to t_k.

When selecting a criterion, there is, of course, a trade-off. The stronger the selected criterion, the more closely the program is scrutinized in an attempt to detect program faults. However, a weaker criterion can be fulfilled, in general, using fewer test cases. As the strongest criterion *all-paths* can be very costly, we will use the second strongest criterion *all-du-paths* (see (Weyuker, 1985) for all the criteria). P satisfies the *all-du-paths* criterion if for every node i and every $x \in def(i)$, P includes every *du-path* w.r.t x.

The main difference between the "all definition-use" or "all du" criterion and a fault model such as FSM fault model is the following: in the case of the "all du", the objective is to satisfy the criterion by generating test cases that exercise the paths corresponding to it. Exercising the paths does not guarantee the detection of existing faults because of variable values that should be selected. If the right values are selected then certain "du" criteria are comparable to fault models.



t1 ?U.sendrequest
! L.cr

t2?L.cc
! U.sendconfirm

t3 ?U.datarequest(sdu, n,b)
number:=0;
counter:=0;
no_of_segment:=n;
blockbound:=b;

t4 ?L.tokengive
! L.dt(sdu[number])
start timer
number:=number+1;

t5 ?L/resume

t6 expire_timer
! L.tokenrelease

t7 ?l.ack()
number==no_of_segment
!U.monitor_complete(counter)
!token_release
!L.disrequest

t8 ?L.ack()
number<no_of_segment
not expire_timer
!L.dt(sdu[number])
number:=number+1

t9 ?L.block
not expire_timer
counter:=counter+1

t10 ?L.resume
not expire_timer and
counter<=blockbound

t11 counter>blockbound
!L.token_realease
!U.monitor_incomplete
(number)
!U.dis_request

t12 expire_timer
counter<=blockbound
!L.token_release

t13 ?L.resume

t14 ?L.block

t15 ?L.ack

t16 ?L.dis_request
!U.disindication

t17 ?L.disrequest
!U.disindication

Figure 1. Example of an EFSM specified protocol (same as in (Huang, 1995)).

For transition t_3 in figure 1, I-Use(t3)={sdu, n, b}, A-Use(t3)={number, no_of_segement, blockbound, counter}, C-Use(t3)={n, b} and P-Use(t3)=$\varnothing$.

3 TEST CASE GENERATION

3.1 Choosing the values for the input parameters

The choice of the values of the input parameters has a sure impact on the test cases. These values may influence the number of times a cycle should be repeated. The user may specify valid and invalid values for each input parameter and our tool will choose randomly a value within the valid domain. If no value is specified, then if the input parameter influences the control flow, the user will be asked to enter a value for that input parameter.

3.2 Test case and test sequence generation

As we mentioned earlier, our method combines both control and data flow testing techniques to generate complete test cases (a complete test case is a test case which starts and ends at the initial state). Also, it verifies the executability during the du-path generation. The following algorithm illustrates the process of generating automatically executable test cases.

```
Algorithm EFTG (Extended Fsm Test Generation)
Begin
  Read an EFSM specification
  Generate the dataflow graph G form the EFSM specification
  Choose a value for each input parameter influencing the control flow
  Executable-Du-Path-Generation(G)
  Remove the paths that are included in others
  Add state identification to each executable du-path
  Add a postamble to each du-path to form a complete path
  For each complete path
      Re-check its executability
      If the path is not executable
          Try to make it executable
      EndIf
      If the path is still not executable Discard it
      EndIf
  EndFor
  For each uncovered transition T
      Add a path which covers it (for control flow testing)
  EndFor
  For each executable path
      Generate its input/output sequence using symbolic evaluation
  EndFor
End;
```

```
Procedure Executable-Du-Path-Generation(flowgraph G)
Begin
  Generate the set of A-Uses, I-Uses, C-Uses and P-Uses for each transition in G
  Generate the shortest executable preamble for each transition
  For each transition T in G
      For each variable v which has an A-Use in T
          For each transition U which has a P-Use or a C-Use of v
              Find-All-Paths(T,U)
          EndFor
      EndFor
  EndFor
End.
```

Table 1 presents the shortest executable preambles for the transitions in the EFSM in figure 1 (both input parameters n and b are equal to 2).

Table 1. Executable preambles for the EFSM's transitions in figure 1

Trans	*Executable Preamble*	*Trans*	*Executable Preamble*
t2	t1, t2	t10	t1, t2, t3, t4, t9, t10
t3	t1, t2, t3	t11	t1, t2, t3, t4, t9, t10, t9, t10, t9, t11
t4	t1, t2, t3, t4	t12	t1, t2, t3, t4, t9, t12
t5	t1, t2, t3, t5	t13	t1, t2, t3, t4, t8, t7, t13
t6	t1, t2, t3, t4, t6	t14	t1, t2, t3, t4, t8, t7, t14
t7	t1, t2, t3, t4, t8, t7	t15	t1, t2, t3, t4, t8, t7, t15
t8	t1, t2, t3, t4, t8	t16	t1, t2, t3, t4, t8, t7, t16
t9	t1, t2, t3, t4, t9	t17	t1, t17

The reason we start by finding the shortest executable preamble for each transition is as follow: Suppose we want to find all executable du-paths between t_3 and t_7. Since t_3 needs a preamble, then any path from t_3 to t_7 cannot be made executable unless an executable (or feasible) preamble is attached to it.

When finding the preambles and postambles, we try to find the shortest path which does not contain any predicate. If we fail to find such a path, then we choose the shortest path and try eventually to make it executable.

4 EXECUTABLE DU-PATH GENERATION

In (Chanson, 1993), after adding preambles and postambles to the du-paths, their executability is verified. However, many paths remain non-executable and are discarded because the predicates associated with some transitions are not satisfied. To overcome this problem, we verify the executability of each path during its generation. Below is the algorithm which finds all the paths between two transitions.

```
Procedure Find-all paths(T1, T2, var)
Begin
  If a preamble, a postamble or a cycle is to be generated
     Preamble:=T1
  Else
     Preamble:= the shortest executable preamble from the first transition to T1
  EndIf
  Generate-All-Paths(T1,T2,first-transition, var, preamble)
End;
```

The following algorithm is the algorithm used to find all executable preambles and all executable du-paths between transition T1 and transition T2 with respect to the variable var defined in T1.

```
Procedure Generate-All-Paths(T1, T2, T, var, Preamble)
Begin
  If (T is an immediate successor of T1) (e.g. t3 is an immediate successor of t2)
    If (T=T2 or (T follows T1 and T2 follows T in G)) (e.g. t4 follows t2)
      If we are building a new path
        Previous:= the last generated du-path (without its preamble)
        If (T1 is present in the previous path)
          Common:= the sequence of transitions in the previous path before
          T1
        EndIf
      EndIf
      If we are building a new path
        Add Preamble to Path, Add var in the list of test purposes for Path
      EndIf
      If Common is not empty
        Add Common to Path
      EndIf
      If (T = T2)
        Add T to Path, Make-Executable(Path)
      Else
        If T is not present in Path (but may be present in Preamble) and T does
          not have an A-use of var
          Add T to Path
          Generate-All-Paths(T, T2, first-transition, var, Preamble)
        EndIf
      EndIf
    EndIf
  T:= next transition in the graph
  If (T is not Null) Generate-All-Paths(T1, T2, T, var, Preamble)
  Else
    If (Path is not empty)
      If (the last transition in Path is not an immediate precedent of T2)
        Take off the last transition in Path
      Else
        If (Path is or will be identical to another path after adding T2)
          Discard Path
        EndIf
      EndIf
    EndIf
  EndIf
End.
```

The algorithm used to find the postambles and the cycles is also similar, except that it does not call the procedure Make-Executable(Path).

Suppose P1=(t_1,t_2,..t_{k-1},t_k). Make-Executable(P1) finds the non-executable transition t_k in P1 if it exists. Then it finds if another executable du-path P2=(t_1,t_2,..,t_{k-1},...,t_k) exists. If such path exists, P1 is discarded. If not, the procedure Handle-Executability(P1) is called (see next section). This verification enables to save time generating the same path or an equivalent path (the same du-path with different cycles in it) more than once. Handle-Executability(Path) starts by verifying if each transition in Path is executable or not. In each transition, each predicate is interpreted symbolically until it contains only constants and input parameters and the algorithm can determine if the transition is executable or not (especially for simple predicates). However, for some specifications with unbounded loops, Handle-Executability may not be able to make a non-executable path executable.

Table 2 shows all the du-paths (with the preamble (t_1, t_2, t_3, t_4)) form t_9 to t_{10} w.r.t the variable counter and the reason why some paths were discarded. All the paths that were discarded because the predicate became (3=2) cannot be made executable, because the influencing transition (t_4 or t_8) appears more than it should be.

Table 2. All du-paths form t9 to t7 w.r.t. counter

Du-Path	*Discarded*	*Reason path is discarded*
1,2,3,4,9,10,6,4,7	no	-
1,2,3,4,9,10,6,4,8,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,6,5,4,7	no	-
1,2,3,4,9,10,6,5,4,8,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,7	yes	will be equivalent to the first path after solving the executability
1,2,3,4,9,10,8,6,4,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,8,6,5,4,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,8,7	no	-
1,2,3,4,9,12,4,7	no	-
1,2,3,4,9,12,4,8,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,12,5,4,7	no	-
1,2,3,4,9,12,5,4,8,7	yes	predicate in t7 become (3=2)

In the next section, we will show what cycle analysis is and how it can be used to make the non-executable paths executable.

5 HANDLING THE EXECUTABILITY OF THE TEST CASES

The executability problem is in general undecidable. However, in most cases, it can be solved. (Ramalingom, 1995) deals essentially with the executability of the preambles and postambles, and not with the executability of the du-paths covering the data flow. (Huang, 1995) overcame this problem by executing the EFSM. This method does not cover the control flow and may not deal with large EFSMs. (Chanson, 1993) used static loop analysis and symbolic evaluation techniques to determine how many times the self loop should be repeated so that test cases become executable. This method is not appropriate for specifications where the influencing variable is not updated inside a self loop, such as the EFSM in figure 1, and cannot be used if the number of loop iterations is not known. For these reasons, the following heuristic was developed in order to find the appropriate cycle to be inserted in a non-executable path to make it executable.

```
Procedure Handle_Executability(path P)
Begin
  Cycle:= not null
  Process(P)
  If P is still not executable Remove it
  EndIf
End;

Procedure Process(path P)
Begin
  T:= first transition in path P
  While (T is not null)
     If (T is not executable)
         Cycle:= Extract-Cycle(P,T)
     EndIf
     If (Cycle is not empty)
         Trial:=0
         While T is not executable and Trial<Max_trial Do
            Let Precedent be the transition before T in the path P
            Insert Cycle in the path P after Precedent
            Interpret and evaluate the path P starting at the first transition
            of Cycle to see if the predicates are satisfied or not
            Trial:= Trial+1
         EndWhile
     Else
         Exit
     EndIf
     T:= next transition in P
  EndWhile
End.
```

We would like to mention that our tool makes a difference between two kinds of predicates. A binary predicate has the following form: "var1 R var2", where R is a relational operator such as "<"; while a unary predicate can be written as F(x), where F is a boolean function such as "Even(x)" (see figure 2).

The heuristic "Handle-Executability" verifies if each non-executable path can be made executable and uses the procedure "Extract-Cycle(P,T)" to find the shortest cycle, if it exists, to be inserted in a non-executable path in order to make it executable. For this purpose, we find the first non-executable transition T in the path P. Two cases may arise: If the transition T cannot be executed because some unary predicate is not satisfied, we find a transition t_k, if it exists, among the transitions preceding transition T, which has the same predicate with a different value. An influencing cycle containing t_k is generated (if it exists) and inserted in the path P before transition T. If the predicate is not a unary predicate, we find out, using symbolic evaluation, what the variable causing the non-executability is, and whether it should be increased or decreased for the transition t_k to be executable. This variable must be an influencing one and transitions which update the variable must exist. If this is not the case, an empty cycle is returned, and the path is discarded. If the variable in the predicate is an influencing variable, we search among the transitions preceding T, for a transition t_k which updates properly the variable, generate a cycle containing this variable and insert it in the path. If a path cannot be made executable, it is discarded.

To illustrate the heuristic, suppose that in the EFSM of figure 1, both variables n and b have the value 2. The shortest preamble for t_{11} is (t_1, t_2, t_3, t_4, t_9, t_{11}), but t_{11} is not executable because its predicate "counter>2" becomes "1>2" after interpretation. Our tool finds that the influencing variable is "counter" and that among the transitions preceding t_{11}, t_9 is an influencing transition which may be adequate, because it increases the variable "counter". The cycle (t_{10},t_9) is generated and inserted twice after transition t_9. The path becomes (t_1, t_2, t_3, t_4, t_9, t_{10}, t_9, t_{10}, t_9, t_{11}).

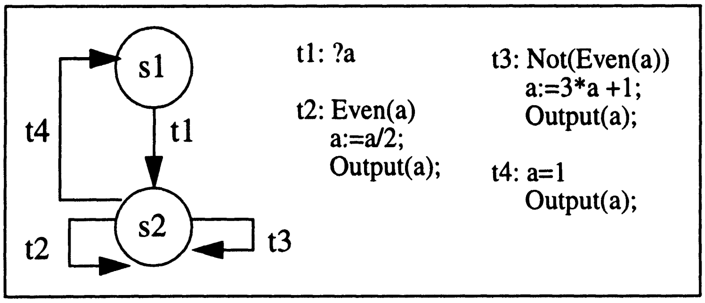


Figure 2. EFSM with unbounded loops and unary predicates.

Figure 2 presents an example of an EFSM with unbounded loops. Each loop is a self-loop with a unary predicate. For this example, since transitions t_2 and t_3 are not bounded, (Chanson, 1993) (Ramalingom, 1995) cannot generate any executable test case for this example.

In table 3, the executable test cases (without state identification) and test sequences for the EFSM in figure 2 are presented. Each test case is relative to one value for the input parameter a.

Table 3. Executable test cases for the EFSM in figure 2

Input parameter	*Executable test case*	*Input/Output sequence*
1	t1, t4	?1!1
5	t1, t3, t2, t2, t2, t2, t4	?5! 16! 8! 4! 2! 1!1
100	t1, t2, t2, t3, t2, t2, t3, t2, t3, t2, t2, t2, t3, t2, t3, t2, t2, t3, t2, t2, t2, t3, t2, t2, t2, t2, t4	?100!50!25!76!38!19!58!29! 88!44!22!11!34!17!52!26!13,!40, 20!10!5!16!8!4!2!1!1
125	-	-

For the EFSM in figure 2, our tool failed to generate any executable test case for a=125. But when we increased the value of the variable Max-Trial (in the procedure Process) , a solution was found. Giving our tool more time to let it find a solution does not mean that a solution will be found. In these cases, out tool cannot decide if a solution exists. After the generation of executable paths, input/output sequences are generated. The inputs will be applied to the IUT, and the observed outputs from the IUT will be compared to the outputs generated by our tool. A conformance relation can then be drawn.

6 RESULTS

Table 4 presents the final executable test cases (without state identification) generated by our tool on the EFSM in figure 1. In many cases, the tool had to look for the influencing cycle to make the test case executable. With state identification, the first executable path will look like: (t_1, t_2, t_3, t_5, t_4, t_8, t_7, $\mathbf{t_{15}}$, t_{16}). The last two paths are added to cover the transitions t_{13}, t_{14}, t_{15} and t_{17} which were not covered by the other paths.

Table 4. Executable test cases for the EFSM of Figure 1

No	*Executable Test Cases*	*Test Purposes*
1	t1, t2, t3, t5, t4, t8, t7, t16,	number, counter, no_of_segment
2	t1, t2, t3, t5, t4, t8, t9, t10, t7, t16	number, counter, no_of_segment, blockbound
3	t1, t2, t3, t5, t4, t9, t10, t8, t7, t16	number, counter, no_of_segment, blockbound
4	t1, t2, t3, t4, t8, t9, t10, t9, t10, t9, t11, t16	number, counter, no_of_segment, blockbound
5	t1, t2, t3, t5, t4, t8, t9, t10, t9, t10, t9, t11, t16	number, counter, no_of_segment, blockbound
6	t1, t2, t3, t5, t4, t9, t10, t9, t10, t9, t11, t16	number, counter, blockbound
7	t1, t2, t3, t5, t4, t9, t12, t4, t7, t16	number, counter, blockbound
8	t1, t2, t3, t4, t6, t4, t7, t16	number, counter, no_of_segment
9	t1, t2, t3, t4, t6, t5, t4, t7, t16	number
10	t1, t2, t3, t4, t8, t7, t16	number, counter, no_of_segment
11	t1, t2, t3, t4, t8, t9, t10, t7, t16	number, counter, no_of_segment, blockbound
12	t1, t2, t3, t4, t9, t10, t6, t4, t7, t16	number, counter, no_of_segment, blockbound
13	t1, t2, t3, t4, t9, t10, t6, t5, t4, t7, t16	number, counter, blockbound
14	t1, t2, t3, t4, t9, t10, t8, t7, t16	number, counter, no_of_segment, blockbound
15	t1, t2, t3, t4, t9, t12, t4, t7, t16	number, counter, blockbound
16	t1, t2, t3, t4, t9, t12, t5, t4, t7, t16	number, counter, blockbound
17	t1, t2, t3, t4, t9, t10, t9, t10, t9, t11, t16	number, counter, blockbound
18	t1, t2, t3, t4, t9, t10, t6, t4, t9, t10, t9, t11, 16	number, counter, blockbound
19	t1, t2, t3, t4, t9, t10, t6, t5, t4, t9, t10, t9, t11, t16	number, counter, blockbound
20	t1, t2, t3, t4, t9, t10, t8, t6, t4, t9, t10, t9, t11, t16	number, counter, no_of_segment, blockbound
21	t1, t2, t3, t4, t9, t10, t8, t6, t5, t4, t9, t10, t9, t11, t16	number, counter, no_of_segment, blockbound
22	t1, t2, t3, t4, t9, t10, t8, t9, t10, t9, t11, t16	number, counter, no_of_segment, blockbound
23	t1, t2, t3, t4, t9, t12, t4, t9, t10, t9, t11, t16	number, counter, blockbound
24	t1, t2, t3, t4, t9, t12, t5, t4, t9, t10, t9, t11, t16	number, counter, blockbound
25	t1, t2, t3, t4, t8, t7, t13, t14, t15, t16	-
26	t1, t17	-

The sequence of input/outputs is extracted from the executable test cases, and applied to test the IUT. For output parameters with variable (such as the output "dt"), symbolic evaluation is used to determine the value of the variable number which has an output use (see Table 3 for an example).

In order to compare our tool to other methods, we implemented an algorithm which generates all the du-paths (like in (Chanson, 1993)), to which we added Cycle Analysis to handle the executability problem instead of loop analysis. We shall call this algorithm "Ch+". Note that "Ch+" verifies the executability after all the du-paths are generated.

Table 5. Results obtained by Ch+ and by our tool

EFSM	*Ch+*		*Our tool*			
	du-paths	Exec	du-paths	discarded du-paths		Exec
fig 1	81	26	60	29	16	26
fig 2	9	1	1	0	0	1
INRES	54	25	24	4	4	22

In table 5, the results obtained by Ch+ and by our tool on three EFSMs are summarized. The third EFSM is a simplified version of the INRES protocol. It has four states, fourteen transitions, four loops two of which are influencing self-loops.

The first column of discarded "du-paths by our tool" specifies the total number of discarded paths during du-path generation. The second column specifies the number of paths that were discarded by the tool without trying to make them executable, because equivalent paths already existed. "Exec" stands for executable.

7 CONCLUSIONS AND FUTURE WORK

As me mentioned earlier, for the EFSM in figure 1, our tool discarded only twenty nine paths (during du-paths generation) while Ch+ discarded fifty five (after generating all the du-paths). Verifying the executability of the du-paths during their generation enables to generate only those paths which are more likely to be executable. Our method generates executable test cases for EFSM-specified systems by using symbolic evaluation techniques to evaluate the constraints along each transition, so only executable test sequences are generated. Also, our method discovers more executable test cases than the other methods and enables to generate test cases for specifications with unbounded loops.

This work is supported by an NSERC Strategic grant STR0167072.

8 REFERENCES

Bochmann, G. v. Das, A. Dssouli, R. Dubuc, M. Ghedamsi, A. and Luo, G. (1991) Fault models and their use in Testing. Proc. IFIP Intern. Workshop on Protocol Test Systems (invited paper), pp. (II-17)-(II-32).

Chanson, S. T. and Zhu, J.(1993) A Unified Approach to Protocol Test Sequence Generation. In Proc. IEEE INFOCOM.

Huang, C.M. Lin, Y.C. and Jang, M.Y. (1995) An Executable Protocol Test Sequence Generation Method for EFSM-Specified Protocols. International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September.

Ramalingom, T. Das, A. and Thulasiraman, K.(1995). A Unified Test Case Generation Method for the EFSM Model Using Context Independent Unique Sequences. International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September.

Sarikaya, B. and Bochmann, G.v. (1986) Obtaining Normal Form Specifications for Protocols. In Computer Network Usage: Recent Experiences, Elsevier Science Publishers.

Ural, H. and Yang. B. (1991) A Test Sequence Selection Method for Protocol Testing. IEEE Transactions on Communication, Vol 39, No4, April.

Weyuker, E.J. and Rapps, S. (1985) Selecting Software Test Data using Data Flow Information. IEEE Transactions on Software Engineering, April.

9 BIOGRAPHY

Chourouk Bourhfir is a Ph-d student in the Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal. She received the M.Sc. degree in Computer Science in Université Laval, Canada in June 1994. Her research interests include modeling and automatic test generation.

Rachida Dssouli is professor in the DIRO, Université de Montréal. She received the Doctorat d'université degree in computer science from the Université Paul-Sabatier of Toulouse, France, in 1981, and the Ph.D. degree in computer science in 1987, from the University of Montréal, Canada. She is currently on Sabbatical at NORTEL, Ile des Soeurs. Her research area is in protocol engineering and requirements engineering.

El Mostapha Aboulhamid, received his Ing. Degree from INPG, France, in 1974, The M.Sc. and Ph.D. degrees from Universite de Montréal in 1979 and 1985 respectively. Currently, he is an associate Professor at Université de Montréal. His current research interests include Hardware software codesign, testing, modeling and synthesis.

Nathalie Rico is currently working in Nortel, Ile des soeurs, Montréal. She is the manager of the DMS Network Applications group.