A test case generation approach for conformance testing of SDL systems[1]

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

Abstract

This paper presents an approach for automatic executable test case and test sequence generation for a protocol modeled by an SDL system. Our methodology uses a unified method which tests an Extended Finite State Machine (EFSM) based system by using control and data flow techniques. To test an SDL system, it extracts an EFSM from each process then the system is tested by incrementally computing a partial product for each EFSM C, taking into account only transitions which influence (or are influenced by) C, and generating test cases for it. This process ends when the coverage achieved by the generated test cases is satisfactory or when the partial products for all EFSMs are tested. Experimental results show that this method can be applied to systems of practical size.

Keywords

CEFSM, Testing in context, Embedded testing, Partial product, Reachability analysis, Control and data flow testing, Executability

1 INTRODUCTION

To ensure that the entities of a protocol communicate reliably, the protocol implementation must be tested for conformance to its specification. Quite a number of methods and tools have been proposed in the literature for test case generation from EFSM specifications using data flow testing techniques and/or control flow testing techniques [4, 6, 11, 16]. However, these methods are applicable when the protocol consists only of one EFSM. For CEFSM specified protocols, other methods

1. This work is supported by Nortel grant.

should be used. To our knowledge, very few work has been done in this area, and most existing methods deal only with communicating finite state machines (CFSMs) where the data part of the protocol is not considered. An easy approach to testing CFSMs is to compose them all-at-once into one machine, using reachability analysis, and then generate test cases for the product machine. But we would run into the well known state explosion problem. Also, applying this approach to generate test cases for CEFSMs is unpractical due to the presence of variables and conditional statements. To cope with the complexity, methods for reduced reachability analysis have been proposed for CFSMs [8, 13, 14]. The basic idea consists in constructing a smaller graph representing a partial behavior of the system and allowing one to study properties of communication. In this paper, we present a methodology which can be used to test a CEFSM based system or a part of it after a correction or enhancement. Our method does not compose all machines but decomposes the problem into computing a partial product (defined later) for each CEFSM and generating test cases for it.

Another problem is that most existing methods generate test cases from specifications written in the normal form [18] which is not a widely used specification approach. For this reason, our test generation method should start from specifications written in a high level specification language. Since we are interested in testing communication protocols and since SDL is one of the most used specification language in the telecommunication community, our method starts the test generation process form SDL specifications.

The objective of this paper is to present a set of tools which can be used to generate automatically executable test cases for SDL systems and to show the architecture of each tool.

The organization of this paper is as follows. Section 2 presents some existing tools for test generation from SDL systems most of which are semi-automatic. Section 3 describes the EFSM and CEFSM models. In section 4, our methodology for automatic test generation is presented. Section 5 describes the EFTG tool for automatic test generation for EFSM based systems. In Section 6, the process of testing an SDL system is presented as well as some examples. Finally, section 7 concludes the paper.

2 SOME TEST GENERATION TOOLS FOR SDL SPECIFICATIONS

Over the past ten years, tools have become available that seek to automate the software testing process. These tools can help to improve the efficiency of the test execution process by replacing personnel with test scripts that playback application behavior. However, it is the up-front process of deciding what to test and how to test it that has the dominant impact on product quality. Likewise, the cost and time to develop tests is an order of magnitude greater than that required for test execution. Today, manual methods are still the primary tools for this critical stage, however, tools exist which automate some parts of the testing process. In the following, some existing tools for test case generation or tools that help the test suite developer in the test generation process are presented.

TESDL [5] is a prototype tool for the automatic generation of test cases from SDL specifications in the context of the OSI Conformance Testing Methodology and Framework. TESDL implements a heuristic algorithm to derive the global behavior of a protocol as a tree, called Asynchronous Communication Tree (ACT), which is based on a restricted set of SDL diagrams (one process per block, no two processes are able to receive the same kind of signal, etc.). The ACT is the global system description as obtained using reachability analysis by perturbation. In the ACT, the nodes represent global states. A global state contains information about the states of all processes in the specification. Tests are derived from the ACT of a specification by a software tool, called TESDL. Input for the tool is a SDL specification (SDL-PR), the output are the test cases in TTCN-Notation.

TTCN Link (LINK for short) [12] is an environment for efficient development of TTCN test suites based on SDL specifications in SDT3.0 (SDL Description Tool) [15]. LINK assures consistency between the SDL specification and the TTCN test suite. It increases the productivity in the development by automatic generation of the *static parts* of the test suite and specification-based support for the test case design. The intended user of the LINK tool is a TTCN test suite developer. His inputs are an SDL specification and a test suite structure with test purposes and his task is to develop an abstract TTCN test suite, based on this input. This tool is semi-automatic.

SAMSTAG [9, 10] is developed within the research and development project “Conformance Testing a Tool for the Generation of Test Cases” which is funded by the Swiss PTT. The allowed behavior of the protocol which should be tested is defined by an SDL specification and the purpose of a test case is given by an MSC which is a widespread mean for the graphical visualization of selected system runs of communication systems. The SaMsTaG method formalizes test purposes and defines the relation between test purposes, protocol specifications and test cases. Furthermore, it includes the algorithms for the test case generation.

TOPIC V2 [1] (prototype of TTC GEN) works by co-simulating the SDL specification and an observer representing the test purpose. This co-simulation enables to explore a constrained graph, i.e., a part of the reachability graph of the specification, which enables to use this method for infinite graphs. The observer is described in a language called GOAL (Geode Observation Automata Language). In order to facilitate the use of TOPIC, it is possible to generate the observers from MSC’s. From the constrained graph, some procedures are executed in order to generate the TTCN test.

Tveda V3 [7] is a tool for automatic test case generation which incorporates several features:

- A modular architecture, that makes it possible to choose between the specification languages (Estelle or SDL), test description languages (TTCN or Menuet) and test selection strategy (single transition, extended transition tour, etc.)

- A semantic module, which can be called from the strategy modules to compute feasible paths.
- Functional extensions, such as hypertext links between tests and specification, test coverage analysis, etc.

To compute execution paths, two techniques can be used, symbolic computation techniques or reachability analysis. Symbolic computation techniques put strong restrictions on which constructs are accepted and the path computation requires an exponential computation with respect to the length of the path to be computed. On the contrary, reachability analysis puts almost no restriction on the Estelle or SDL constructs which are accepted, and it is implemented by interfacing Tveda with a powerful commercial tool for reachability analysis, Véda.

Most of these tools are semi-automatic. Also, the test purposes have to be defined by the user. In our case, our tool is completely automatic and the specification is used to generate test cases (instead of the observers). Also, it uses control flow and data flow testing techniques to generate test cases.

Since we are interested in testing SDL protocols, we present, in the next section, the EFSM and CEFSM models which are the underlying models for SDL systems.

3 THE EFSM AND CEFSM MODELS

Definition1. An EFSM is formally represented as a 8-tuple <S, s_0, I, O, T, A, δ, 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. A is a set of actions,
7. δ is a transition relation $\delta : S \times A \to S$,
8. V is the set variables.

Each element of A 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.

Definition2 . A communicating system is a 2k-tuple (C_1, C_2,...,C_k, F_1, F_2,..., F_k) where

- C_i = <S, s_0, I, O, T, A,δ , V> is an EFSM
- F_i is a First In First Out (FIFO) for Ci, i=1..k.

Suppose a protocol Π consists of k communicating EFSMs or CEFSMs for short: C_1, C_2,...,C_k. Then its state is a k-tuple <$s^{(1)}$, $s^{(2)}$,..., $s^{(k)}$, m_1, m_2,...,m_k> where

$s^{(j)}$ is a state of C_j.and m_j, j=1..k are set of messages contained in F_1, F_2,...,F_k respectively. The CEFSMs exchange messages through bounded storage input FIFO channels. We suppose that a FIFO exists for each CEFSM and that all messages to a CEFSM go through its FIFO. We suppose in that case that an internal message identifies its sender and its receiver. An input interaction for a transition may be internal (if it is sent by another CEFSM) or external (if it comes from the environment). The model obtained from a communicating system via reachability analysis is called a global model. This model is a directed graph G = (V, E) where V is a set of global states and E corresponds to the set of global transitions.

Definition 3. A global state of G is a 2k-tuple $<s^{(1)}, s^{(2)},..., s^{(k)}, m_1, m_2,...,m_k>$ where m_j, j=1..k are set of messages contained in F_1, F_2,...,F_k respectively.

Definition 4 . A global transition in G is a pair t = (i, α) where $\alpha \in A_i$ (set of actions). t is firable in $s = <s^{(1)}, s^{(2)},..., s^{(k)}, m_1, m_2,...,m_k>$ if and only if the following two conditions are satisfied where α = (initial_state, final_state, input, predicate, compute-block).

- A transition relation $\delta_i\ (s,\alpha)$ is defined
- $input = null$ and $predicate = True$ or $input = a$ and $m_i = aW$,

  where W is a set of messages to C_i, and $predicate = True$.

After t is fired, the system goes to $s' = <s'^{(1)}, s'^{(2)},..., s'^{(k)},m'_1, m'_2,...,m'_k>$ and messages contained in the channels are m'_j where

- $s'^{(i)} = \delta(s^{(i)},\alpha)$ and $s'^{(j)} = s^{(j)}\ \ \forall(j \neq i)$
- if $input = \varnothing$ and $output = \varnothing$, then $m'_j = m_j$
- if $input = \varnothing$ and $output = b$ then $m'_u = m_u b$ (C_u is the machine which receives b)
- if $input \neq \varnothing$ and $output = \varnothing$ then $m_i = W$ and $m'_j = m_j\ \ \forall(j \neq i)$
- if $input \neq \varnothing$ and $output = b$ then $m_i = W$ and $m'_u = m_u b$

In the next section, our test case generation methodology is presented.

4 OUR METHODOLOGY FOR TEST GENERATION FROM SDL SYSTEMS

Our methodology uses a set of tools developed at Université de Montréal. Some of the tools use an underlying FSM model and were adapted so that they can be used for an underlying EFSM model.

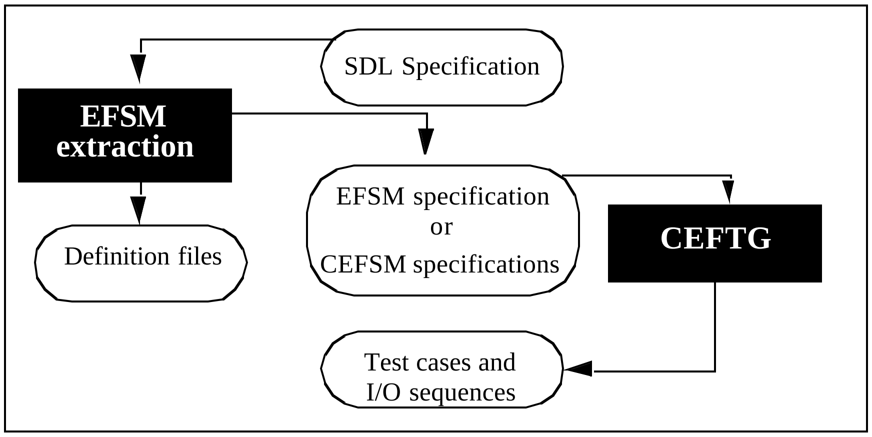

SDL Specification
EFSM extraction
EFSM specification or CEFSM specifications
CEFTG
Definition files
Test cases and I/O sequences

FIGURE 1. The process of test case generation from SDL specifications

Figure 1 shows the process of generating test cases from SDL systems. First, the EFSMs corresponding to the processes of the SDL system are extracted. At the same time, files containing SDL declarations of interactions (or signals) and channels are created, as for CEFTG, it generates automatically executable test cases for CEFSM based systems.

4.1 The extended finite state machine extraction

The process of extracting an EFSM form an SDL process is based on the FEX tool [2]. This latter was developed at Université de Montréal in order to extract an FSM representing a partial behavior of an SDL system by applying a normalization algorithm. One or more transitions in the generated FSM correspond to a given input at a given state in the SDL specification. This is due to the fact that the tool uses partial unfolding to preserve constraints on the values of message parameters. In addition, FEX generates additional files that can be used in order to complete the test cases. For these reasons, and due to the availability of FEX, we decided to use FEX and to modify the output of FEX so that it can be processed by CEFTG. Even though FEX was originally built to extract an FSM from an SDL system, it can also extract the corresponding EFSM from it. In [2], the output file generated by FEX contains assignment statements as well as the predicates of the transitions but these were not used by the tool described in the same paper, while this information will indeed be used by CEFTG. Also, CEFTG does not use the Definition files generated by FEX (which were used by the tools presented in [2]).

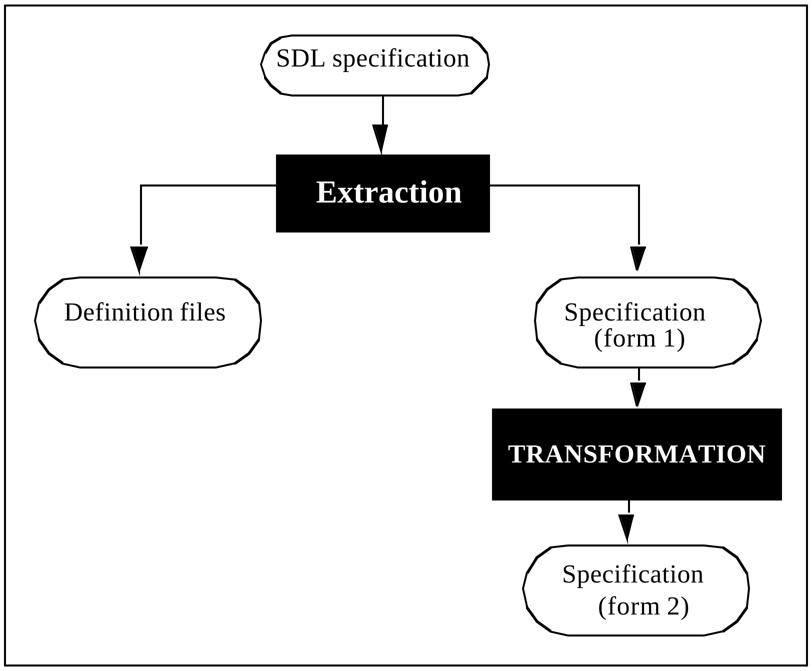


FIGURE 2. The EFSM extraction

As we already mentioned before, the input of CEFTG is a specification in the normal form, which is different from the output generated by FEX. For this reason, the output of FEX, Specification (form 1), is transformed to the normal form (Specification (form 2)).

To summarize this section, CEFTG was originally built to generate test cases from normal form specifications. But as we want our tool to start form SDL systems, we used the FEX tool which extracts the corresponding normal form specifications from the SDL processes.

Figure 3 below shows an example of an SDL specification, the output generated by FEX and the input accepted by CEFTG.

```
SDL Specification:
state S0;
     input I1;
     output O1;
     nextstate S1;

     input I2(n);
     decision n;
     (0): output O2;
           nextstate S2;
     (1): output O3;
           nextstate S3;
     enddecision;

Transitions generated by FEX:
S0?I1!O1 > S1;
S0?I2(n = 0)!O2 >S2;
S0?I2(n = 1)!O3 >S3;

Transitions generated by Transformer and accepted by CEFTG:

When I1
From S0
To S1
Name t1: Begin
Output(O1);
End;

When I2(n)
From S0
To S2
Provided (n = 0)
Name t2: Begin
Output(O2);
End;

When I2(n)
From S0
To S3
Provided (n =1)
Name t3: Begin
Output (O3);
End;
```

FIGURE 3. Example of an SDL specification and its corresponding form1 and form2

The next section presents the architecture of the CEFTG tool. This latter generates executable test cases for CEFSM based systems by generating test cases for each CEFSM in context.

4.2 Test generation for communicating extended finite state machines

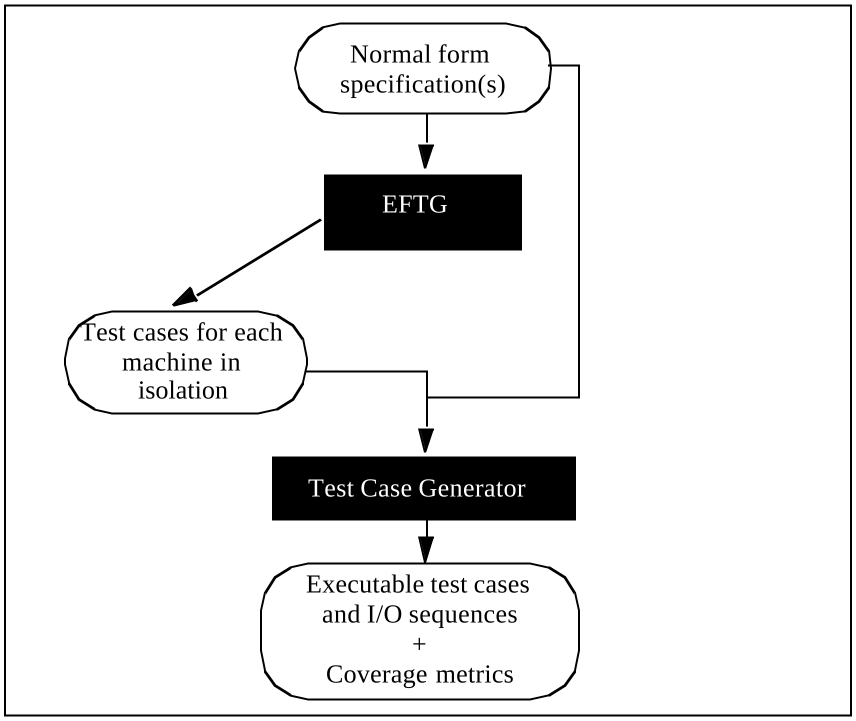

Normal form specification(s)
EFTG
Test cases for each machine in isolation
Test Case Generator
Executable test cases and I/O sequences + Coverage metrics

FIGURE 4. Architecture of CEFTG

CEFTG is the CEFsm Test Generator and can generate test cases for systems modeled by one EFSM using the EFTG tool. For CEFSM based systems, the user can generate test cases for the global system by performing a complete reachability analysis, i.e., taking into consideration all transitions of all CEFSMs, and generating test cases for the complete product (or reachability graph). CEFTG can also generate test cases for each CEFSM in context. In that case, the process ends when the coverage achieved by the generated test cases is satisfactory or after the generation of the test cases for the partial products of all CEFSMs.

CEFTG includes all the activities starting from the specification and leading to the test cases and to the input/output sequences. The first task performed by

CEFTG is to generate test cases for each CEFSM in isolation, i.e, without considering its interaction with the other CEFSMs. During this first phase, the tool EFTG is applied in order to generate test cases for each normal form specification in isolation. The next section presents our method for test generation for EFSM specified systems. The EFTG tool (Extended Finite state machine Test Generator) presented in [4] implements this method. This latter uses control flow testing techniques, data flow testing techniques and symbolic evaluation to generate test cases for systems modeled by one EFSM.

5 EFSM TEST GENERATION

Our method generates executable test cases for EFSM specified protocols which cover both control and data flow. The control flow criterion used is the UIO (Unique Input Output) sequence [17] and the data flow criterion is the "all-definition-uses" criterion [20] where all the paths in the specification containing a definition of a variable and its uses are generated. A variable is defined in a transition if it appears at the left hand side of an assignment statement or if it appears in an input interaction. It is used if it appears in the predicate of the transition, at the right hand side of an assignment statement or in an output interaction.

Step 1: From the normal form specification of the CEFSM, a dataflow graph is built. During this step, all definitions and uses of all variables are identified.

Step 2: For each state S in the graph, EFTG generates all its executable preambles (a preamble is a path such that its first transition's initial state is the initial state of the system and its last transition's tail state is S) and all its postambles (a postamble is a path such that its first transition's start state is S and its last transition's tail state is the initial state). To generate the "all-definition-uses" paths, EFTG generates all paths between each definition of a variable and each of its uses and verifies if these paths are executable, i.e., if all the predicates in the paths are true. To evaluate the constraints along each transition in a definition-use path, EFTG interprets symbolically all the variables in the predicate backward until these variables are represented by constants and input parameters only. If the predicate is false, EFTG applies a heuristic in order to make the path executable. The heuristic uses "Cycle Analysis" in order to find cycles (a sequence of transitions such that the start state of the first transition and the ending state of the last transition are the same) which can be inserted in the path so that it becomes executable. After this step, EFTG removes the paths which are included in the already existing ones, completes the remaining paths (by adding postambles) and adds paths to cover the transitions which are not covered by the generated test cases. EFTG discovers more executable test cases over the other methods [6, 11, 16] and enables to generate test cases for specifications with unbounded loops.

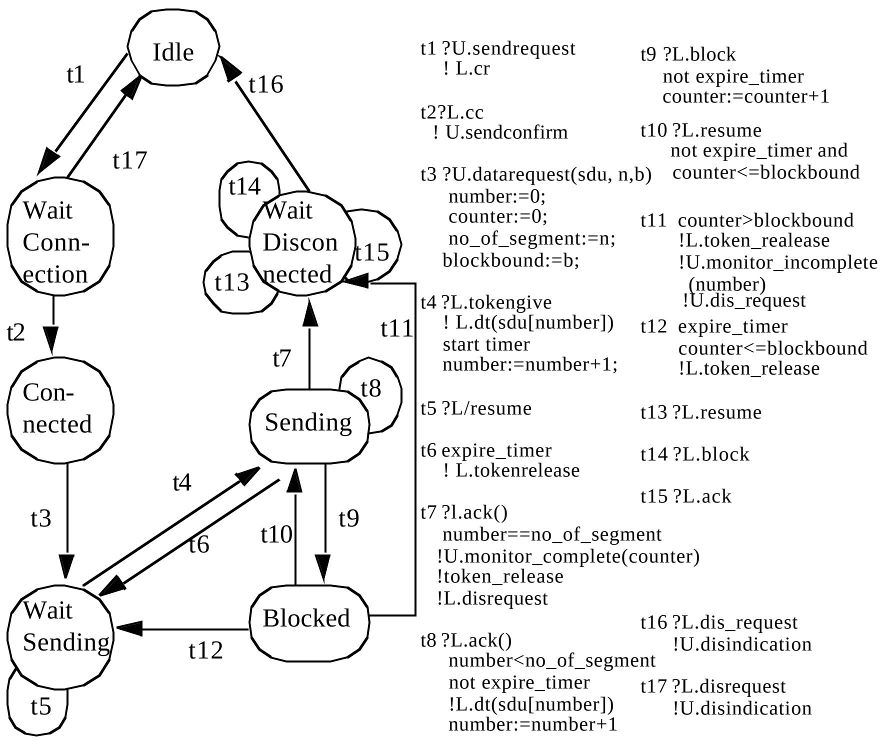

Idle
t1
t16
t17
Wait Conn- ection
t14
Wait Discon nected
t15
t13
t11
t2
t7
Con- nected
t8
Sending
t4
t3
t6
t10
t9
Wait Sending
Blocked
t12
t5
t1 ?U.sendrequest ! L.cr
t2?L.cc ! U.sendconfirm
t3 ?U.datarequest(sdu, n,b) number:=0; counter:=0; no_of_segment:=n; blockbound:=b;
t4 ?L.tokengive ! L.dt(sdu[number]) 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 5. Example of an EFSM specified protocol.

The following algorithm illustrates the process of generating automatically executable test cases.

5.1 The EFTG algorithm

```
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 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.

```
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
    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)

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

Du-Path	*Discarded*	*Reason path is discarded*
1,2,3,4,9,10,6,5,4,7	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 executa-bility
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.

5.2 Handling the executability of the test cases

The executability problem is in general undecidable. However, in most cases, it can be solved. [11] overcame this problem by executing the EFSM. This method does not cover the control flow and may not deal with large EFSMs. [6] 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.
```

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. If the transition T cannot be executed because some 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 a variable is causing the predicate to be false, we find out, using symbolic evaluation, what this variable 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 5, 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}).

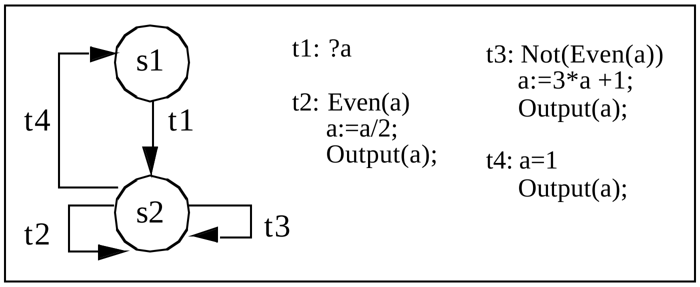


Figure 6. EFSM with unbounded loops and unary predicates.

Figure 6 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, [6] 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 6 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 6, 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 the executable paths, the 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.

5.3 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.

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).

Table 4. Executable test cases for the EFSM of Figure 5

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

Table 4. Executable test cases for the EFSM of Figure 5

No	*Executable Test Cases*	*Test Purposes*
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	-

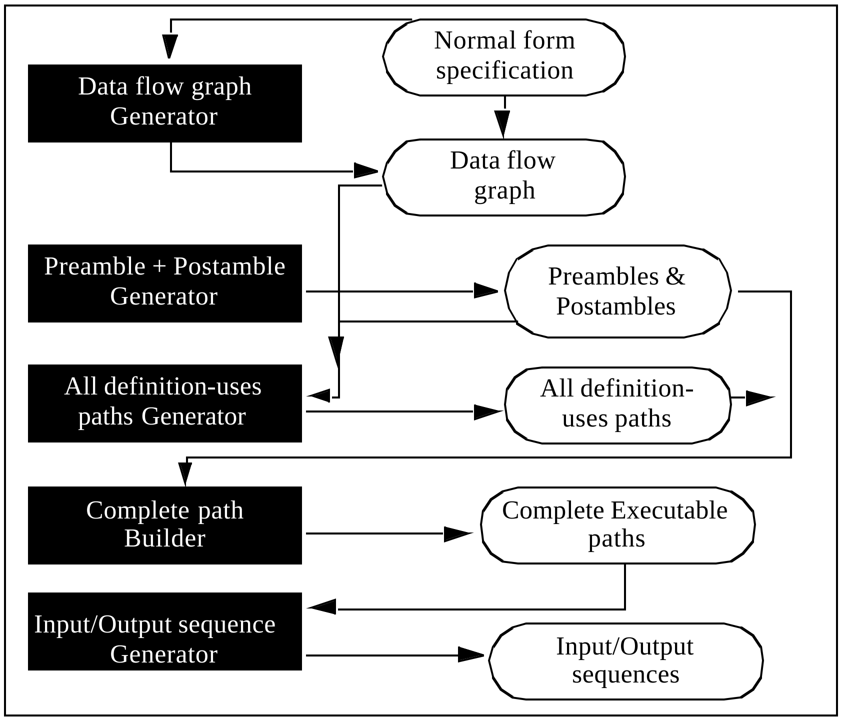

Normal form specification
Data flow graph Generator
Data flow graph
Preamble + Postamble Generator
Preambles & Postambles
All definition-uses paths Generator
All definition-uses paths
Complete path Builder
Complete Executable paths
Input/Output sequence Generator
Input/Output sequences

FIGURE 7. Architecture of EFTG

When testing CEFSM based systems, the first task performed by CEFTG is to apply the EFTG tool to each CEFSM in order to generate its test cases in isolation, i.e., when the communication of this CEFSM with the others is not considered. After this latter is performed, CEFTG generates test cases for the global system. The next section explains this part.

6 TEST CASE GENERATION

Before presenting the choices that the user have in the second step, we would like to explain what a partial product for one CEFSM is and how it can be computed.

6.1 Computing a partial product

The following steps summarizes the process of computing a partial product for one CEFSM:

Step 1: Marking process. Suppose that the system to be tested is modeled by C_1, C_2,...,C_k and suppose that we want to test the machine C_n in context. We use the test cases generated by EFTG, for each machine in isolation, to mark the transitions in all the paths which trigger C_n (or are triggered by C_n). We shall call the first set of transitions Pr(C_n) and the latter Po(C_n). Determining Pr(C_n) and Po(C_n) can be very costly if exhaustive reachability analysis is considered. For this purpose, our method uses the test cases generated by EFTG for each machine in isolation as a guide. If a transition in C_n receives (sends) a message from (to) a CEFSM C_i and since this message is sent (received) by some transitions in C_i which belong necessarily to test cases generated by EFTG for C_i in isolation, we mark all the transitions in these test cases. By marking all the transitions in each test case, we insure that transitions preceding (following) the transition which sends (receives) the message participate in the partial product generation. When the test case which contains the transition sending (receiving) a message to (from) C_n is marked, we verify if it contains transitions receiving (sending) messages from (to) other CEFSMs. If this is the case, for each such transition T, the same procedure is repeated in order to mark the paths in the machine sending (receiving) the message received (sent) by T.

At the end of this step, all marked transitions will participate in the generation of the partial product.

```
Algorithm Marking (*machine under test* $C_n$)
Begin
   For each transition T in $C_n$ Do
       Mark T;
       If T receives an internal input M from another machine
           Marking-Backward(Sender of M,$C_n$,M)
       For each internal output statement O in T Do
           Marking-Forward($C_n$,Receiver of O,O)
End;
```

At this stage, we suppose that the test cases for each CEFSM in isolation are already generated by EFTG (this step was explained in the previous section). The goal of procedure Marking-Backward (Sender,Receiver,M) is to mark all the paths in Sender which contain a transition sending M to Receiver (i.e., determine the set Pr(C_n)).

In the same way, procedure Marking-Forward(Sender,Receiver,M) marks the paths in Receiver which have transitions receiving message M from Sender (i.e., determine the set Po(C_n)).

```
Procedure Marking-Backward(*Sender,Receiver,M*)
Begin
  For each test case TC in Sender
  If one or more transitions in TC send M to Receiver Then
     For each **unmarked** transition T in TC Do
       If T receives an internal message M' and a call to the recursive
       procedure was not made with (Sender of M',Sender, M') Then
          Marking-Bakward(Sender of M', Sender, M')
     **Mark** all the transitions in TC
End;
```

```
Procedure Marking-Forward(*Sender,Receiver,M*)
Begin
  For each test case TC in Sender
    If one or more transitions in TC receive M from Sender Then
       For each unmarked transition T in TC Do
          For each internal output statement O in T such that a call to the recursive
          procedure was not made with (Receiver,Receiver of O, O) Do
               Marking-Forward(Receiver,Receiver of O, O)
       Mark all the transitions in TC
End;
```

At the end of this step, all marked transitions will participate in the generation of the partial product.

Step 2: Partial product generation.

Definition 4 . A partial product for C_n is defined as:

PP(C_n) = $C_n \times \{marked{-}transitions\}$. PP($C_n$) is also called C_n in context.

After the marking process, the partial product for C_n is performed. At each time, among the possible transitions which can be picked, only the marked ones are chosen (in the partial reachability analysis).

The unmarked transitions do not participate in the partial product computation because they do not influence the machine under test. Procedure ComputePP computes a partial product for a certain CEFSM and is similar to an ordinary algorithm for reachability analysis. The main difference is that among the possible transitions which can be picked, only the marked ones are chosen.

Procedures AddGlobalState and AddGlobalTransition add a new global state and a new global transition to the graph of the partial product respectively.

```
Procedure ComputePP(CEFSM M)
Begin
  Create a new global state NewState <$s0^{(1)}$, $s0^{(2)}$,..., $s0^{(k)}$, $m_1$, $m_2$,...,$m_k$>
  AddGlobalState(NewState)
  Push(NewState,Stack)
  While Stack is not empty Do
      $CurrentState \leftarrow Head(Stack)$
      If a **marked** firable transition T exists in some machine Ci then
        Create a new Global State NewState
        If transition T has an internal Input Then Remove the Input from its FIFO
        If T has internal outputs Then Add the outputs to the corresponding FIFOs
        AddGlobalState(NewState) if it does not exist   , AddGlobalTransition(T)
        Push(NewState)
      Else  Pop(Stack)
 End;
```

Step3: Test case generation for the partial product. After the partial product for C_n is computed, its test cases are generated automatically with EFTG since the result of the product of C_n with the marked transitions is also an EFSM. These test cases the “all-definition-uses + all transitions” criterion in the partial product. We should also note that our method is adapted to global systems made of more than two CEFSMs because if the system is made of only 2 CEFSMs, then the partial product for one CEFSM is in general the complete product.

6.2 An example

Consider the example in Figure 8.

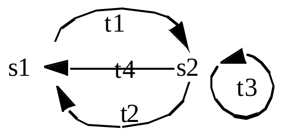
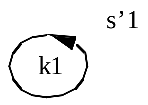
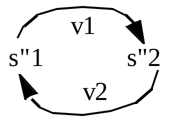
 (a) B1	 (b) B2	 (c) B3
t1: ?Ext1(c) R:=c*c; t2: ?int1 !Int2(R); t3:(R is odd) R:=R+1; t4: (R is even) L:=R/2; !Int3(L);	k1: ?Int3(L) !Ext4(L);	v1: ? Ext2 !Int1 v2: ?Int2(R) !Ext3(R)

FIGURE 8. A system of CEFSMs: B1, B2 and B3.

We have 3 processes B1, B2 and B3 which communicate with each other and with the environment.B1 reads an external input with parameter c and assigns c*c to R in t1. In state s2, if B1 receives input Int1 from B3, it responds with Int2 containing the value of R. R can be modified in t3 before being sent to B3. In t4, if R is even, a message Int3 is sent to B2 with the computed value of L (“Ext” means a message to/ from the environment while “Int” means an internal message; also, “?msg” means an input interaction while “!msg” means an output interaction). The next section illustrates the process described in this section.

6.2.1 Test case generation for the partial product for B3

Step 1: Test case generation for each machine in isolation. Here are the test cases generated by EFTG for each CEFSM in isolation.

- Test cases for B1: (t1, t2), (t1, t3, t2), (t1, t3, t4) and (t1, t4).
- Test case for B2: (k1).
- Test case for B3: (v1, v2).

Step 2: The marking process. First, all transitions in B3 are marked. Since v1 sends a message to B1, which can be received by t2 then all transitions in all test cases of B1 which contain t2 are marked (i.e., t1, t2 and t3). Before marking these transitions, we first take a look at the test cases containing t2. t1 receives an external transition and does not receive nor send any internal message. t2 sends int2 to B3, but since all transitions in B3 are marked, nothing happens. t3 does not contain any internal input or output message. Transition k1 in B2 is not marked because it is not influenced by any transition in B3, nor does it influence any transition in B1.

Step 3: Generation of the partial product for B3. After the Marking process, the partial product for B3 is computed. This latter is shown in Figure 9.

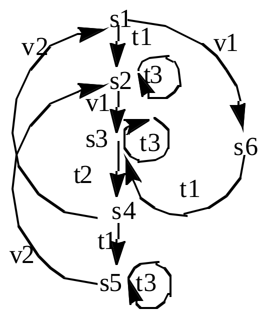


FIGURE 9. The partial product for B3.

It contains 11 transitions and 6 states. For clarity purposes, we kept the old names of the transitions. The detail about each transition can be found in Figure 8.

Step 4: Test cases of the partial product for B3. After the partial product for B3 is computed, its test cases are generated by EFTG. These test cases cover all the definition-uses criterion in the partial product for B3. In term of transition coverage, they cover all transitions in B3, 75% of transitions of B1 and no transition in B2. These test cases are as follow:

1)	t1, t3, v1, t3, t2, v2	6)	t1, v1, t2, t1, v2, v1, t3, t2, v2
2)	t1, t3, v1, t2, v2	7)	t1, v1, t2, t1, v2, v1, t2, v2
3)	t1, v1, t3, t2, v2	8)	t1, v1, t2, v2
4)	t1, v1, t2, t1, t3, v2, v1, t2, v2	9)	v1, t1, t3, t2, v2
5)	t1, v1, t2, t1, v2, t3, v1, t2, v2	10)	v1, t1, t2, v2

In the same manner, to compute the partial product for B2, t1, t3, t4 and k1 are marked. This partial product has 7 transitions and 4 states and 6 test cases. In terms of transition coverage, the test cases cover 75% of the transitions in B1, 100% of the transitions in B2 and 0% of the transitions in B3. If we consider both partial products of B2 and B3, we find that they cover 100% of the transitions in each machine as well as the “all-definition-uses” criterion for the partial products for B2 and B3.

6.2.2 Test case generation for the complete product of B1, B2 and B3

We computed the complete product machine (see Figure 10).

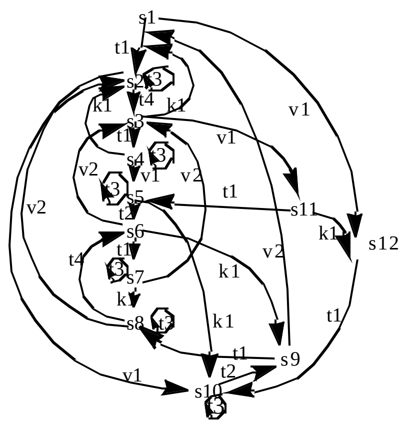


FIGURE 10. The complete product of CEFSMs in Figure 8.

This latter has 30 transitions and 12 states and 1483 executable definition-uses paths. This number corresponds to the product of 3 small CEFSMs. For big communicating systems, testing the product machine becomes unpractical. Therefore testing each CEFSM in context may be the only solution for large systems.

6.3 A test case generation algorithm for CEFSM specified protocols

In this section, we will present the general algorithm CEFTG for automatic test generation for CEFSM based systems. CEFTG generates executable test cases incrementally by generating test cases for the partial product for each CEFSM until the desired coverage is achieved. The process of generating test cases for the global system may stop after generating the partial product for *some* CEFSMs. In that case, these test cases cover the all definition-uses criterion for the partial product machines (this include all transitions in the CEFSM as well as transitions in other CEFSMs). In term of transition coverage, these test cases cover 100% of the transitions for each CEFSM for which a partial product was computed and p% of the transitions of each other CEFSMs, where $p = (marked - transitions) / (all - transitions)$ for some CEFSM.

First, lets take a look at the example in Figure 11. The example is inspired from a real communicating system. The global system is made of two blocks A and B. Block A communicates with block B and with the environment, while block B communicates only with block A.

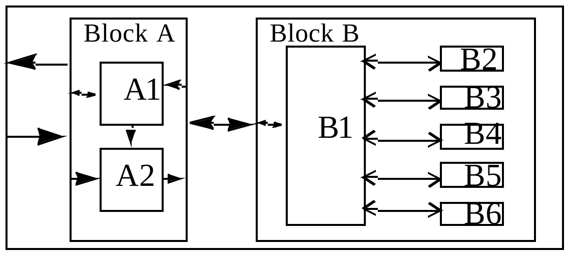


FIGURE 11. Architecture of a communicating system.

In this particular case, B1 controls all other processes in block B and none of the processes Bi $2 \leq i \leq 6$ is influenced by another, i.e., none of the messages sent by a process does influence the control flow of the other processes. Suppose we want to compute the partial product for B1, when its context is B; then all the transitions in B will be marked and will participate in the partial reachability analysis. In other words, the partial product for B1 is equivalent to $B1 \times B2 \times B3 \times B4 \times B5 \times B6$ .This is due to the fact that B1 communicates with all the other processes in block B. To test block B, if instead of starting with B1, we generate the partial products for B2,..,B6 then we can avoid generating the partial product for B1, because after generating the partial product for B6, the test cases for all the partial products cover all the transitions in all processes (even B1). In that case, if our goal is to simply cover each transition in each process without generating a partial product for each machine, then the order in which CEFSMs are chosen is very important. In the next section, we present some metrics which help choosing, among several CEFSMs, the CEFSM which has less interaction with the others.

6.3.1 Metrics for measuring complexity of communication

In Section 6.3.2, we present an algorithm which builds progressively the test cases for a communicating system by generating test cases for the partial product for each component machine. Normally, a partial product is generated for each CEFSM. But in some cases, we may not need to do so because the test cases of the partial product for certain machines may cover all transitions in all CEFSMs (in term of transition coverage and not all-definition-uses coverage). For systems having an architecture similar to that of block B in Figure 11, it is more interesting not to generate a partial product for B1 and a method should be used to avoid computing the partial product for the primary block which in this case is the complete product of all machines.

Let M be a set of CEFSMs. We define the following metrics for each CEFSM C.

- Input/Output Count (IOC): The number of internal input and output interactions in C is considered as a metric for measuring the complexity of communication that may take place during the composition of the CEFSMs in M. IOC of C is defined as, IOC(C) = # of input/output messages involving CEFSMs in M (number of internal input and output messages).
- CEFSM Count (CC): The CC for C is the number of CEFSMs which communicate with C.

To prevent the generation of the complete product when the goal is to cover each transition in each CEFSM, we first compute an IOC for each CEFSM. In each iteration, the algorithm will compute the partial product for the machine with the smallest IOC and eventually the smallest CC. After the partial product for a CEFSM is computed, its test cases are generated and the user can stop the execution of the algorithm when the desired coverage is achieved. In that case, for Block B, the execution of the algorithm may stop after the partial products of B2, B3,.., B6 are computed since, by that time, all transitions in B1 are already covered by the generated test cases of the partial products of B2,..,B6. This is particularly interesting if B1 is huge since it communicates with all machines. Also, the architecture of the system presented in Figure 11 is very common, and the method we are presenting in this paper can be very useful for such systems.

In the next section, we present an algorithm for automatic test generation for a CEFSM based system which uses the metrics described above.

6.3.2 An incremental algorithm for test generation for CEFSM based systems

The next algorithm presents our method to generate test cases for a communicating system. The user can either let the algorithm generate a partial product for each CEFSM and its test cases or stop the execution of the algorithm when the coverage achieved by the test cases for some partial products is satisfactory ("all definition-uses + all transitions" coverage for each partial product and "transition coverage" only of each CEFSM for which a partial product was not computed. After computing the partial product for one or many machines, Compute Coverage determines how many transitions in each machine are covered by the generated test cases (in term of transition coverage). Functions Push (resp. Pop) are functions which add (remove) a global state to (from) the stack. Add Global State and AddGlobalTransition add a new global state (resp. a new global transition) to the graph of the partial product. After computing the partial products of all CEFSMs and generating the test cases for them, we verify if some machines are still not completely covered. If this is the case, this means that some transitions in these machines may not be accessible when the communication between the machines is considered (possible specification errors or an observability and controllability problem).

```
Algorithm CEFTG(Nb: the number of CEFSMs)
Begin
  For $i = 1$ to Nb Do
    Read each CEFSM specification
    Generate Executable test cases for each CEFSM in isolation using EFTG
  $i \leftarrow 1$
  While $i \leq Nb$ Do
    Choose the CEFSMi with the lowest IOC (and CC) which has not been
    chosen yet
    Marking(CEFSM$_i$)
    $Ai \leftarrow ComputePP(CEFSM_i)$
    Generate executable test cases for A$_i$ using EFTG
    ComputeCoverage
    If the user is satisfied with the coverage then Break
    Else   $i \leftarrow i + 1$
End;
```

6.4 Result analysis

First, we would like to mention that all algorithms presented in this paper were implemented in C++. In the example in Figure 8, we can clearly see that k1 in B2 and t4 in B1 have no influence on B3. In this case, the marking algorithm marked only the transition t1, t2, t3, v1 and v2. Now lets compare the partial product for B3 with the complete product of B1, B2 and B3. Removing k1 from the complete product leads to remove states s8, s9, s10 and s12 and their transitions. Also, removing t4 from the resulting machine leads to remove states s2 and s3 and their transitions. The new resulting machine is nothing else but the partial product for B3 (Figure 4). In other words, the partial product for B3 is the ***projection*** of the complete product on {t1, t2, t3, v1, v2.}. Also, if we consider the test cases of the partial product and the test cases of the complete product, then removing all test cases containing k1 and t4 leads to remove 1475 test cases. To test B3 in context, we only need 10 test cases and not all 1483 test cases. The test cases of the partial products for B2 and B3 can be used to test the global system, and we may not need to generate the partial product for B1 which is the product of B1, B2 and B3. It is important to note that the test cases generated by our method cover all the global states and global transitions in each partial product (not global transitions in the complete product) since not all transitions participate in the partial product generation. They also cover all the “definition-uses” paths in each partial product since test case generation is performed by EFTG. Compared to other methods [Lee 1996], our coverage criterion is very strong. In that case, for huge systems, our incremental test generation strategy can be combined with a reduced reachability analysis technique. In Procedure ComputePP, at each step, we can choose a marked transition which has not been tested yet to reduce the size of the partial product. But, the coverage criterion induced by this method is weaker.

Since we divide the problem of generating test cases for a CEFSM based system by incrementally generating test cases for each partial product, we divide also the complexity of the problem by not considering all transitions in all CEFSMs at once.

6.5 Testing the global system

To test the global system (set of several CEFSMs), CEFTG generates executable test cases incrementally by generating test cases for the partial product for each CEFSM until the desired coverage is achieved. The process of generating test cases for the global system may stop after generating the partial product for some CEFSMs. In that case, these test cases cover the all definition-uses criterion for the partial product machines (this include all transitions in the CEFSM as well as transitions in other CEFSMs). In term of transition coverage, these test cases cover 100% of the transitions for each CEFSM for which a partial product was computed and p% of the transitions of each other CEFSM, where $p = (marked - transitions) / (all - transitions)$ for some CEFSM.

6.6 Test case generation

At this step, the user can make a choice among the following:

- Generate executable test cases for a certain CEFSM M in context: the marking process explained in section 6.1 is performed. Once this process is over, a reduced reachability analysis is performed considering only marked transitions. The result of this analysis is a partial product for which test cases are generated using EFTG. These latters test the machine M when its context is considered, unlike the test cases generated for M in isolation.

- Generate test cases for the complete product of all CEFSMs of the system: in that case, the marking process is not called because all transitions of all CEFSMs participate in the reachability analysis. CEFTG builds the complete product using a classical reachability analysis algorithm and generates test cases for the complete product using EFTG. This option is used when the CEFSMs as well as the complete product are small. However, The test case generation for the complete product may be impossible due to state explosion.

- Generate test cases for each CEFSM in context until the generated test cases reach a certain coverage or until test cases are generated for each partial product: this choice can be taken when the user wants to generate test cases which cover each transition of each CEFSM instead of each global transition (of the global system). In that case, CEFTG generates the partial products, starting by the CEFSM which has less interactions with the other machines. The goal here is to avoid generating large partial products. Each time, test cases are generated for a partial product, the tool computes the coverage achieved in order to determine how many transitions are covered by the generated test cases. If the user is satisfied with the coverage, the algorithm ends. Otherwise, CEFTG computes another partial product and generates test cases for it until the user is satisfied or all partial products are generated.

Once test cases for the partial products are generated, input/output sequences are extracted and used to test the conformance of the implementation against the specification. For the input and output messages with parameters, symbolic evaluation is used in order to determine their values.

6.7 Experimentation

We tried the algorithm CEFTG on a real SDL system whose architecture is similar to that of figure 11 and we obtained the following results: the first CEFSM of the system, with 2 states and 17 transitions, communicates with all other CEFSMs and when computing its partial product, all transitions were marked. Its partial product is also the complete product, i.e., the reachability graph for the global SDL system and this latter could not be computed due to state explosion. The second CEFSM has 1 state and 3 transitions and its partial product has 8 states, 13 transitions and 9 executable def-use paths. The third CEFSM has 4 states, 10 transitions and its partial product has 24 states, 55 transitions and 43 executable def-use paths. The fourth machine has 5 states, 12 transitions, and its partial product has 75 states,139 transitions and 95 executable all-use paths. The fifth CEFSM has 11 states and 24 transitions. Its partial product has 1027 states and 3992 transitions. CEFTG could not generate test cases for the partial product.This is because the criterion used by CEFTG, the all-definition-uses (or def-use) criterion is too expensive. For this reason, weaker criteria should be implemented so that test case generation may be possible for large systems or for large partial products. In fact, this solution may be more practical for the industry. Such criterion may be the “all-uses criterion” which generates one path from one definition to a use of a variable instead of all paths as in the “all-definition uses” criterion used in this paper.

7 CONCLUSION

In this paper, we presented an approach for test generation from SDL systems. Unlike other methods or tools, our method is completely automatic and does not need any assistance from the user. It enables the user to generate the complete reachability graph of the system. The user can also test the SDL system by testing each of its processes in context. Also, he can test one process in context after a modification or enhancement without testing the entire system. Our method has many advantages. First, it may take less time and space than the all-at-once reachability analysis. Second, incremental test generation can significantly reduce the effort for re-testing of a large protocol due to a correction or enhancement so that we won’t have to re-test the entire system. To find the paths containing transitions triggering (or triggered by) the machine under test, our method uses some of the information already generated by EFTG when used to test the machines in isolation. This method can be used to detect inaccessible transitions, by detecting the transitions which are not covered by the generated test cases. Moreover, it computes the coverage achieved by the generated test cases. As we mentioned earlier, our method can easily incorporate a reduced reachability analysis technique such as [13] which will decrease the size of the partial product for each CEFSM. Finally, since many published methods can only be applied to simple examples or to CFSM based systems, we consider this work as a step towards data-flow testing of real communicating systems. We are currently working in transforming the test cases generated by CEFTG in TTCN to test the implementation under test. The next step consists in improving the FEX tool so that more complex SDL systems can be processed by CEFTG. We also intend to link our tool to a set of tools such as ObjectGeode [19] or SDT.

8 REFERENCES

[1] Bernard Algayres, Yves Lejeune, Florence Hugonnet (Verilog), “GOAL: Observing SDL behaviors with ObjectGeode”, 7th SDL Forum, Oslo, Norway, September 1995, 26-29.

[2] G. v. Bochmann, A. Petrenko, O. Bellal, S. Maguiraga, “Automating the process of test derivation from SDL specifications”, SDL forum, Elsevier, 1997, pp. 261-276.

[3] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, “A guided incremental test case generation method for testing CEFSM based systems”, IWTCS’98, 1998.

[4] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, “Automatic executable test case generation for EFSM specified protocols”, IWTCS, Chapman & Hall, 1997, pp. 75-90.

[5] Lars Bromstrup, Dieter Hogrefe, “TESDL: Experience with Generating Test Cases from SDL Specifications”, Fourth Proc. SDL Forum, 1989, pp. 267-279.

[6] Chanson, S. T. and Zhu, J., “A Unified Approach to Protocol Test Sequence Generation”, In Proc. IEEE INFOCOM, 1993.

[7] Marylene Clatin, Roland Groz, Marc Phalippou, Richard Thummel, “Two approaches linking a test generation tool with verification techniques”, International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September, 1995.

[8] M. G. Gouda, Y. T. Yu, “Protocol Validation by Maximal Progressive Exploration”, IEEE Trans. on Comm. Vol. 32, No. 1, January, 1984.

[9] Jens Grabowski, Dieter Hogrefe, Robert Nahm, “A Method for the Generation of Test Cases Based on SDL and MSCs”, Institut fur Informatik, Universitat Bern, April, 1993.

[10] Jens Grabowski, “SDL and MSC Based Test Case Generation: An Overall View of the SaMsTaG Method”, Institut fur Informatik, Universitat Bern,

May, 1994.

[11] Huang, C.M. Lin, Y.C. and Jang, M.Y., "An Executable Protocol Test Sequence Generation Method for EFSM-Specified Protocols", (IWPTS), Evry, 4-6 September, 1995.

[12] "ITEX User Manual", Telelogic AB, 1995.

[13] David Lee, Krishan K. Sabnani, David M. Kristol, Sanjoy Paul, "Conformance Testing of Protocols Specified as Communicating Finite State Machines- A Guided Random Walk Based Approach", IEEE Trans. on Comm. Vol. 44, No. 5, May, 1996.

[14] J. Rubin, C. H. West (1982), "An Improved Protocol Validation Technique", Computer Networks, 6, April.

[15] "SDT User Manual", Telelogic Malmö AB, 1995.

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

[17] K.Sabnani, A.Dahbura (1985), "A new Technique for Generating Protocol Tests", ACM Comput. Commun. Rev. Vol 15, No 4, September.

[18] B.Sarikaya, G.v. Bochmann, "Obtaining Normal Form Specifications for Protocols", In Computer Network Usage: Recent Experiences, Elsevier Science Publishers, 1986.

[19] Verilog, Geode Editor- Reference Manual, France, 1993.

[20] Weyuker, E.J. and Rapps, S., "Selecting Software Test Data using Data Flow Information", IEEE Transactions on Software Engineering, April, 1985.