A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols[1]

C. Bourhfir[2], R. Dssouli[2], E. Aboulhamid[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 incremental method for automatic executable test case and test sequence generation for a protocol modeled as communicating extended finite state machines (CEFSMs) with asynchronous communication. Instead of testing the protocol by computing the product of all CEFSMs, we test it by incrementally computing a partial product for each CEFSM C, taking into account only transitions which influence (or are influenced by) C, and generating test cases for it. The partial product for C represents the behavior of C when composed with parts of the other CEFSMs. Experimental results show that this method can be applied to systems of practical size. We also propose a method which reduces the size of the product machine for certain systems.

Keywords

CEFSM, Partial product, Reachability analysis, Control and data flow testing, Executability

---

1. This work is supported by Nortel grant.

This work is published in IWTCS'98, Tomsk, Russia.

1 INTRODUCTION

To ensure that the entities of a protocol communicate reliably, the protocol implementation must be tested for conformance to its specification. In principle, finite state machines (FSMs) model appropriately the control portions of communication protocols. However, in practice most protocol specifications include variables and conditional statements based on variable values. Therefore the Extended Finite State Machine (EFSM) like model should be used. Quite a number of methods have been proposed in the literature for test case generation from EFSM specifications using data flow testing techniques and/or control flow testing techniques [Ural 1991, Huang 1995, Chanson 1993, Bourhfir 97]. However, these methods are applicable when the protocol consists only of one EFSM. For CEFSM specified protocols, other methods 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 [Rubin 1982, Gouda 1984, Lee 1996]. 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 method 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. To compute the partial product for a certain CEFSM, we compute the product of this CEFSM with parts of the other CEFSMs which influence (or are influenced by) it, i.e., we consider all transitions in the CEFSM and a subset of the transitions of each CEFSM which are involved in the communication with it when computing the partial product. We call it partial product since not all transitions in all machines are considered. By doing so, the complexity of test generation for a communicating system is significantly reduced which enables to test real communicating systems. In the rest of the paper, we use the term “CEFSM in context” to describe the partial product for a CEFSM. Our objective is not to cover all global transitions in the product of all CEFSMs but to cover all transitions in all CEFSMs and all global transitions as well as all data-flow paths in each partial product.

The organization of this paper is as follows. Section 2 describes the EFSM and CEFSM models. Section 3 presents the EFTG tool for automatic test generation for EFSM based systems. In Section 4, an algorithm for computing the partial product for a CEFSM is presented. Section 5 presents an example. In section 6, the algorithm for generating executable test cases for a system modeled by a set of CEFSMs is

presented. Section 7 analyzes the results. Finally, section 8 concludes the paper.

2 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 $\delta : S \times A \rightarrow S$,
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 2n-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>
- F_i is a First In First Out (FIFO) for Ci, i=1..n.

Suppose a protocol Π consists of k communicating CEFSMs: 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 2n-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 ac-

tions). 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 = (input, predicate, output, compute-block).

- A transition relation δ_i (s,α) 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 $m'_k = m_k b$ (C_k 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 $m'_k = m_k b$

The next section summarizes the approach for automatic test generation for EFSM based systems presented in [Bourhfir 1997]. This approach uses control flow testing techniques, data flow testing techniques and symbolic evaluation.

3 AN EFSM TEST CASE GENERATION ALGORITHM

The algorithm EFTG (Extended Finite state machine Test Generator) presented in [Bourhfir 1997] 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 [Sabnani 1986] and the data flow criterion is the "all-definition-uses" criterion [Weyuker 1985] 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. After reading the specification, EFTG generates a dataflow graph. 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 predicates 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 others, 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 [Ural 1991, Chanson 1993, Huang 1995] and enables to generate test cases for specifications with unbounded loops.

In the next section, we present a method for testing a CEFSM in context. This method uses the information computed by EFTG, when used to generate test cases for the CEFSM in isolation.

4 GUIDED TEST CASE GENERATION FOR A CEFSM

Testing a CEFSM C in isolation reduces the cost of testing. However, the key problem in testing C resides in its interaction with the other CEFSMs because the test cases generated for C do not consider its interaction with the other CEFSMs; also, some executable test cases for C may no longer be executable when it is combined with the others. Therefore, since testing each CEFSM separately is not sufficient, and since computing the complete product is too costly, we use a middle approach. We test a CEFSM based system by testing each CEFSM in context. We are not aware of any similar work for CEFSMs, but the problem of testing in context is studied for CFSMs in [Petrenko 1996] where a basic framework for testing in context has been given. The method computes a so-called approximation of the specification in context and the idea consists in reducing testing in context to testing in isolation. The method tries to give the most general solution to this problem.

In this paper, we generate a partial product for a CEFSM and then use the EFTG tool to generate test cases for it. In general, in order to test a system modeled by several CEFSMs, we first test each CEFSM in isolation to make sure that it is correct, then the global system is tested. The next algorithm computes the partial product for a CEFSM and generates executable test cases for it using EFTG. Moreover, the algorithm uses the test cases generated by EFTG for that CEFSM in isolation, as a guide, to compute its partial product. In that case, the generated test cases of the partial product cover the data and control flow criteria used in EFTG.

We call our method a *guided* procedure because it uses the already generated test cases as a guide to choose the transitions which will participate in the partial product generation. The process of generating the partial product for CEFSM C_n is divided in the following four steps:

Step1: Test case generation for all CEFSMs in isolation. This step consists in calling EFTG to generate test cases for all CEFSMs in isolation. The test cases

generated for each CEFSM are executable test cases which cover both the control and data flow criteria used in EFTG. This step is done once.

Step 2: The marking process. Suppose that the system to be tested is modeled by C1, C2,...,Ck and suppose that we want to test the machine Cn in context. We use the test cases generated by EFTG in Step 1 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 in Step 1 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. The next algorithm marks all the transitions in the machine C_n, then marks all the transitions in the other machines which can be part of the preambles and postambles of transitions in C_n. If there are cycles between 2 machines, the algorithm Marking detects them enabling procedures Marking-Backward and Marking-Forward to end.

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

At this stage, we suppose that the test cases for each CEFSM in isolation are already generated by EFTG. The goal of procedure Marking-Backward (Sender,Receiver,M) is to mark all the paths in Sender which contain a transition sending message 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 3: Partial product generation.

Definition 5 . A partial product for C_n is defined as:

$PP(C_n) = C_n \times \{marked\text{–}transitions\}$. $PP(C_n)$ 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.

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 (according to definition 4 given in Section 2).

Procedures AddGlobalState and AddGlobalTransition add a new global state and a new global transition to the graph of the partial product respectively. Functions Push (resp. Pop) are functions which add (remove) a global state to (from) the stack.

```
Procedure ComputePP(CEFSM M)
Begin
 Create a new global state NewState <s0(1), s0(2),..., s0(k), m1, m2,...,mk>
 AddGlobalState(NewState)
 Push(NewState,Stack)
 While Stack is not empty Do
    CurrentState ← 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;
```

Step4: 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 (according to definition 1 presented in Section 2). In that case, we do not have the problem which is inherent to the composition of FSMs: is the product of two FSMs still an FSM? These test cases cover both control and data flow in the partial product and we guarantee the coverage of the "all-definition-uses + all transitions" criterion in the partial product. We should also note that our method is more adapted to test a CEFSM in context when the global system is 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. In the next section, the process of testing in context each CEFSM in a communicating system is presented.

5 AN EXAMPLE

Consider the example in Figure 1. 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).

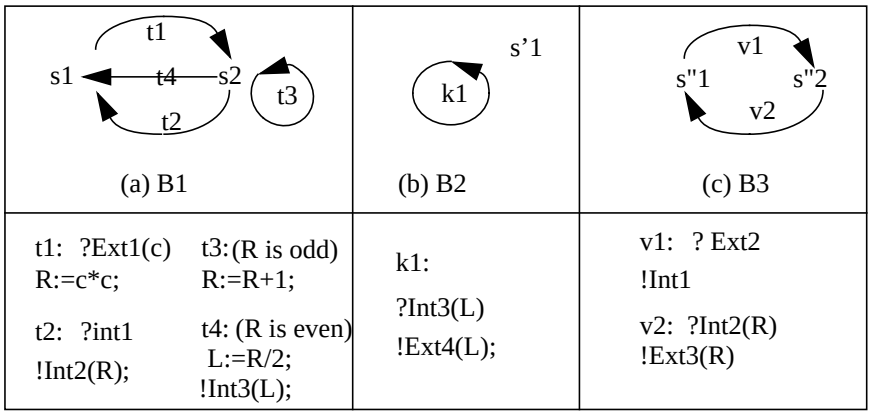

FIGURE 1. A system of CEFSMs: B1, B2 and B3.

The next section illustrate the process described in Section 4.

5.1 Test case generation for the partial product for B3

Step 1: Test case generation for each machine in isolation. Table 1 presents the executable test cases generated by EFTG for each CEFSM in isolation (for more detail on this phase, see [Bourhfir 1997]).

Test cases for B1	Test cases for B2	Test cases for B3
• t1, t2 • t1, t3, t2 • t1, t3, t4 • t1, t4	• k1	• v1, v2

TABLE 1. Executable test cases for B1, B2 and B3 in isolation.

Step 2: The marking process. Suppose that we want to generate the partial product for B3. Then, 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 2. 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 1.

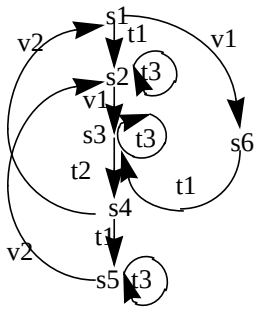

FIGURE 2. partial product for B3.

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.

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

We computed the complete product machine (see Figure 3). 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.

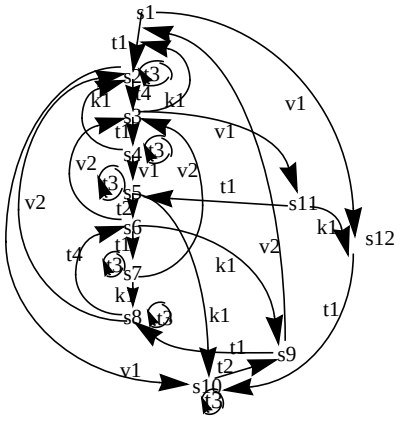

FIGURE 3. The complete product of CEFSMs in Figure 1.

6 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. This algorithm uses EFTG as well as the algorithm described in Section 4. 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 (not all of them). 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 4. The example is inspired from an existing 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.

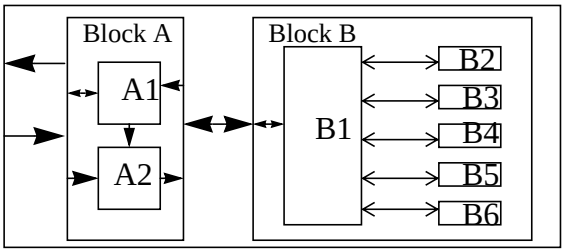

FIGURE 4. Architecture of a communicating system.

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.

6.1 Metrics for measuring complexity of communication

In Section 6.2, 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 4, 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 4 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.

6.2 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). 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;

7 RESULT ANALYSIS

First, we would like to mention that all algorithms presented in this paper were implemented in C++ and integrated to EFTG. In the example in Figure 1, 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 2). 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 and not 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. As we use the test cases generated by EFTG for the machines in isolation to determine the preambles and postambles for the transitions in the machine under test (MUT), we may not generate all preambles and postambles for the MUT (because, in the test cases generated by EFTG, we attach the shortest executable preamble to each definition-use path and not all preambles). In that case, the test cases for the partial product for a CEFSM may not be exhaustive. For this reason, we implemented another marking algorithm which uses all the preambles and all the postambles of each state, instead of the test cases, to guide the marking process of the transitions which will participate in the partial reachability analysis.The results obtained on the examples presented in this paper were similar. For systems where transitions may have many preambles and postambles, the second marking algorithm will achieve better coverage. Also, in order to find the shortest executable preamble to one transition, EFTG generates all its executable preambles and chooses the shortest executable one.

For the product in Figure 3, EFTG generates more than 20 000 preambles and postambles. For this reason, we implemented and instrumented the Dijkstra Algorithm [Dijkstra 1959] to find the shortest preambles and the shortest postambles for each state. As the shortest preamble to a certain state may not be executable, we intend to explore this part further in the near future by looking at algorithms which find all shortest paths between two nodes or paths of length k [Monien 1985, Watanabe 1981].

8 CONCLUSION

In this paper, we presented the algorithm CEFTG which generates automatically and incrementally executable test cases for a CEFSM specified protocol by generating test cases for each CEFSM in context. It is important to note that our method computes a “partial product” for each CEFSM with parts of the other CEFSMs. Also, since the method is guided by the test cases generated by EFTG when used to test each CEFSM in isolation, and since in [Bourhfir 1997] we demonstrated on some examples that EFTG generates more executable test cases than other existing methods, we are quite confident that the partial products computed by our method are representative of the behavior of each machine in its context. 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 also detect deadlocks in the CEFSMs and can

be used to detect inaccessible transitions, by detecting the transitions which are not covered by the generated test cases. As we mentioned earlier, our method can easily incorporate a reduced reachability analysis technique such as [Lee 1996] 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.

9 REFERENCES

C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico (1997) , “Automatic executable test case generation for EFSM specified protocols”, IWTCS, Chapman & Hallpp. 75-90.

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

E. W. Dijkstra (1959) , “A note on two problems in connection with graphs”, Numer. Math. 1 pp. 269-271.

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

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

David Lee, Krishan K. Sabnani, David M. Kristol, Sanjoy Paul (1996), “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.

Monien, B. (1983) “The complexity of determining paths of length k”, Proc. Int. Workshop on Graph Theoretical Concepts in Computer Science, Trauner B Verlag, Linz, 241-251.

A. Petrenko, N. Yevtushenko, G. Bochmann, R. Dssouli (1997), “Testing in context: framework and test derivation”, a Special Issue on Protocol Engineering of Computer Communication.

J. Rubin, C. H. West (1982), “An Improved Protocol Validation Technique”, Computer Networks, 6, April.

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

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

Watanabe (1981), “A fast algorithm for finding all shortest paths”, Inform. Process. Lett. 13, 1-3.

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