



## Validation of SDL specifications using EFSM-based test generation <sup>☆</sup>

W. Eric Wong <sup>a,\*</sup>, Andy Restrepo <sup>a</sup>, Byoungju Choi <sup>b</sup>

<sup>a</sup> Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, United States

<sup>b</sup> Department of Computer Science and Engineering, Ewha Women's University, Seoul, South Korea

### ARTICLE INFO

#### Article history:

Available online 5 July 2009

#### Keywords:

SDL  
EFSM (Extended Finite State Machine)  
Coverage testing  
Dominator analysis  
Hot spot  
Constraint solver

### ABSTRACT

Existing methods for testing an SDL specification mainly allow for either black box simulation or conformance testing to verify that the behavior of an implementation matches its corresponding model. However, this relies on the potentially hazardous assumption that the model is completely correct. We propose a test generation method that can accomplish conformance verification as well as coverage criteria-driven white box testing of the specification itself. We first reformat a set of EFSMs equivalent to the processes in an SDL specification and identify “hot spots” – nodes or edges in the EFSM which should be prioritized during testing to effectively increase coverage. Then, we generate test sequences intended to cover selected hot spots; we address the possible infeasibility of such a test sequence by allowing for its rejection decided by a constraint solver and re-generation of an alternate test sequence to the hot spot. In this paper, we present our test generation method and tool, and provide case studies on five SDL processes demonstrating the effectiveness of our coverage-based test sequence selection.

© 2009 Elsevier B.V. All rights reserved.

### 1. Introduction

It is well-known that the cost of fixing a bug discovered in a software product that has already been integrated and deployed can be much greater than repairing the same bug at an earlier stage of development. As the technology to automatically generate code from an architectural design becomes more powerful, such a design specification should ideally be subject to rigorous testing prior to the implementation stage. SDL (Specification and Description Language) [18] is widely used for architectural design in the telecommunications industry. An SDL system consists of a set of blocks – high level subsystems – which can be further decomposed into a set of low-level processes. Signal channels allow for communication between these processes. Tools exist [20] which test SDL specifications through simulation – a type of black box testing. We have previously proposed a technique to transform an SDL system into a set of EFSMs (Extended Finite State Machines) [22]; this enables the use of more generalized model-based testing techniques. However, most of these techniques [4,12,13,21] are testing for conformance between the model and the corresponding implementation. That is, testing proceeds under the assumption that the model is correct. This could prove to be costly if the model is

in fact found to contain defects after the implementation is already being tested.

We propose a test generation method for SDL-derived EFSM models which, in addition to the conformance testing described above, also allows for white box-based coverage testing of the model itself. This testing could be driven by various coverage criteria with respect to the EFSM, such as all-node,<sup>1</sup> in which every node is covered at least once, or all-edge, which requires that each edge is traversed at least once. For each EFSM, we calculate the “weight” of each node or edge with regard to coverage. When a node other than the start node is covered by a test sequence,<sup>2</sup> it follows that some nodes preceding it in the EFSM must have been covered. In fact, when any uncovered node is covered by a test sequence, there is a guaranteed minimum number of uncovered nodes that will also be covered by the test. We consider this the “weight” of the node. Stated differently, we use a *conservative* approach to calculate the weight of a node as the number of nodes (including itself) that will be guaranteed to be covered. A similar analysis can also be performed with regard to edges. Refer to Section 2: Methodology for more details. By determining the weights of all uncovered nodes and edges in the EFSM, we can guide testing by identifying so-called “hot spots” [1,24] which have the highest weights and should be covered first during testing. Our case studies indicate that node or edge coverage can be effectively increased by generating test cases that target these higher weight nodes or edges before those with a

<sup>☆</sup> This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement): IITA-2007-(C1090-0801-0032).

\* Corresponding author. Tel.: +1 972 883 6619; fax: +1 972 883 2399.  
E-mail address: [ewong@utdallas.edu](mailto:ewong@utdallas.edu) (W.E. Wong).

<sup>1</sup> A “node” of an EFSM is used interchangeably with a “state” of an EFSM, and an “edge” is used interchangeably with a “transition”.

<sup>2</sup> In this paper, “test sequences” and “test cases” are used interchangeably.

lower weight. A test set so generated can achieve high coverage with fewer test sequences and hence can be more easily maintained. Since a simulation tool, such as Telelogic Tau, can quickly reset an executable SDL specification to its initial state, a suite of fewer test sequences is also cheaper to execute. We notice that the advantage of “cheap execution” may not hold for testing a system with multiple distributed and heterogeneous hardware and software components (databases, file systems, etc.) such that some cannot be easily restored to their initial state. For such a system, the execution cost of a test set cannot be determined only by the number of test sequences.

We implement a tool, TGSP (Testing sequence Generation for SDL Process), which automatically reformats EFSMs from a textual SDL specification and allows visualization of the resulting model, including color-coded weights of the nodes and edges. Our tool also allows for the automatic generation of a test sequence that covers a selected node or edge. A known obstacle to the auto-generation of test sequences based on an EFSM is that some sequences may be infeasible; this problem could be mitigated by using a constraint solver such as that presented in [25]. We provide the capability to export the test sequence constraints to a format readable by this solver; however, since it is not yet robust enough, for now we still need some manual intervention. Also, we can easily reformat and export our test sequence constraints so that they can be read by any other constraint solver. This makes it possible to minimize (or ideally eliminate) all the manual intervention if a more powerful constraint solver than the one in [25] is used. Refer to Section 4.3 for more discussion.

Our approach expands on the earlier study [22], which describes how “hot spot” nodes in an EFSM should be identified and prioritized for testing, but does not elaborate on the test generation process. We provide a means for actual test generation for EFSMs, similar to that proposed for reachability graphs in [23]. However, we tackle the additional problem of test sequence feasibility, which is not an issue when dealing with reachability graphs.

The remainder of this paper is organized as follows. Section 2 discusses our methodology, including EFSM reformatting and test generation. Section 3 describes the tool, TGSP, which implements the methodology. Section 4 presents case studies on five SDL processes from an intelligent gateway call server (IGCS) system developed by Telcordia Technologies (formerly Bellcore), the Bluetooth TCSBIN (Telephony Control Specification – Binary) protocol by Motorola, and a railway system from the SDL Forum website [18]. Section 5 gives some related work. Conclusions and future research appear in Section 6.

## 2. Methodology

An SDL system can be considered a CEFSM (Communicating Extended Finite State Machine); each process is an EFSM which can communicate with another in the system via signal channels. For the purposes of this study, we consider each process to be an EFSM with the assumption that all the incoming signals will arrive and outgoing signals will be sent properly. Note that for a selected path, all its incoming and outgoing signals are included as part of the constraints that have to be satisfied in our test generation. Accounting for the communication aspect of the SDL system is a subject for future research.

During EFSM reformatting (which will be further explained in Section 3), we map SDL states directly to nodes in the equivalent EFSM. An SDL state can have multiple outgoing transitions, each with a unique signal that triggers it. To model this behavior, a node representing an SDL state is given an outgoing edge for each input signal accepted at that state. In addition to this type of branching, it is possible that each SDL transition specifies some additional

```
state s1;
input x;
decision x > 0
(true):
  decision x > 10
  (true):
    nextstate s2;
  (false):
    nextstate s3;
  enddecision;
(false):
  nextstate s4;
enddecision;
endstate;
```

```
state s1;
input x;
decision x > 0
(true):
  task x := x - 2;
  decision x > 10
  (true):
    nextstate s2;
  (false):
    nextstate s3;
  enddecision;
(false):
  nextstate s4;
enddecision;
endstate;
```

Fig. 1. Two sample SDL fragments.

control logic based on conditional statements over certain variables. Simple transitions which specify an input signal with no such additional logic can be mapped directly to a single edge in the EFSM. More complicated transitions that specify decisions or loops can potentially be translated into multiple nodes and edges. For each decision statement in a transition, we insert a decision node with two or more outgoing edges to represent the possible paths of execution. Each label (commonly used to implement loops in an SDL specification) is also represented by a corresponding label node in the EFSM. This process is intended to break down the complexity of a single transition into a more understandable form; however, it can result in additional nodes and edges being added to the EFSM. Note that in some cases, it may not be necessary to insert decision nodes to correctly represent the branching behavior of a transition. We optimize our EFSM structure by first performing a systematic insertion of a decision node for each SDL decision statement. Then, we make a second pass to eliminate any decision nodes that can be removed without changing the behavior of the EFSM. Referring to Fig. 1, both decision nodes for the SDL code on the left can be removed after optimization (see Fig. 2), whereas one decision node (namely, Decision\_2) for the SDL code on the right cannot (see Fig. 3). In the latter case, an assignment statement ( $x := x - 2$ ) occurs after the first decision but before the second, changing the value of the variable  $x$ . To maintain the correct ordering of these statements in the EFSM, the second decision node cannot be removed.

Without losing generality, we use the SDL specification of a *kitchen timer* as an example to help us explain the details of our methodology. The timer contains a user interface process, a ringer process, and a counter process. The SDL code fragment for the counter process is displayed in Fig. 4 with the equivalent EFSM shown in Fig. 5. Note that we have three states, two labels, and one decision in the SDL specification, leading to a total of six nodes in the EFSM.

For each EFSM, we conduct a dominator analysis [1] to calculate the weight of each node, which we define as the number of uncovered nodes that will be *guaranteed* to be covered if the node under consideration is covered. The selected node itself is also included in this calculation, so the weight of any uncovered node must be at least 1. A node  $\alpha$  is said to dominate node  $\beta$  if covering  $\beta$  implies that  $\alpha$  has also been covered. That is, if  $\alpha$  dominates  $\beta$ , then a test execution<sup>3</sup> cannot reach  $\beta$  without going through  $\alpha$ . Considering the counter process EFSM shown in Fig. 5, the state *grst4* is dominated by states *start* and *idle*. This implies that if *grst4* is covered, it is guaranteed that at least *start* and *idle* are also covered. Note that there exist other paths to *grst4* which cover more than three nodes, e.g., *start* → *idle* → *grst5* → *counting* → *grst4*. However, the coverage of

<sup>3</sup> This is under the assumption that the underlying hardware does not fail during the execution of a test.

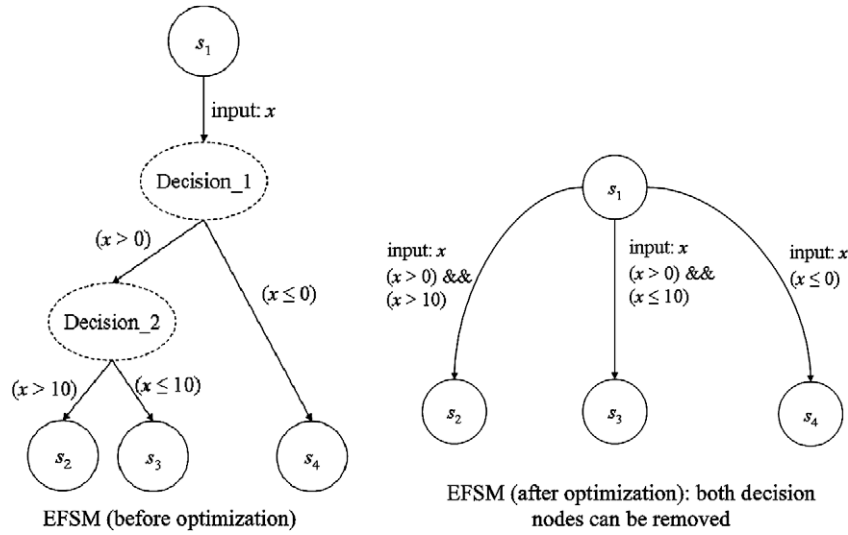


Fig. 2. EFSM for the SDL code in the left part of Fig. 1.

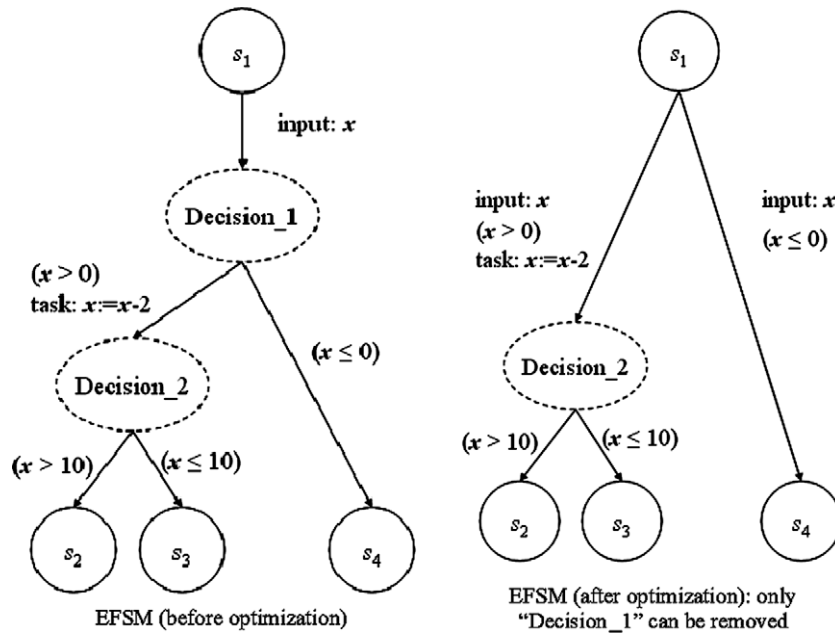


Fig. 3. EFSM for the SDL code in the right part of Fig. 1.

*grst5* and *counting* are not guaranteed if a different path is selected. This is because *grst5* and *counting* do not dominate *grst4*. A very important note is that when calculating the weight to a given node, we use a *conservative* approach by counting only the nodes that will definitely be covered as a consequence of covering that node. Therefore, the weight of *grst4* would be calculated as 3 assuming that *start* and *idle* have yet to be covered. In summary, a test sequence covering a node of weight  $\omega$  is guaranteed to cover “at least”  $\omega$  uncovered nodes, although the sequence could potentially cover more. However, the coverage of any additional nodes is not guaranteed.

Once the weight of each node has been calculated, a “hot spot” can be identified – the node in the EFSM with the highest weight. If multiple nodes happen to share the highest weight, during the test generation we randomly select one of them as the hot spot to be covered. The goal of our approach is to allow the automatic generation of a test sequence which covers a selected hot spot. We define a test sequence as consisting of two components: a path,

and the set of test constraints. A path is an ordered sequence of EFSM nodes and edges executed by the test sequence. The test constraints represent the sequence of SDL input signals that must be received, conditional statements that must be satisfied, and variable assignments that must be executed for the path to be traversed.

To generate a test sequence for a selected node, we first select a path via backward tracking. Starting from the selected node, we perform a greedy search based on the weights of head nodes of incoming edges. That is, if we are presented with a choice of multiple incoming edges which have not yet been searched, we choose the one leading from the node with the highest weight and mark that edge as searched. If there is more than one such node, we randomly select one to be backtracked. For explanatory purposes, let’s refer to Fig. 6. The number next to each node is its weight. We start from the hot spot  $\alpha$ . Suppose node  $\omega$  is already covered and node  $\eta$  does not have an edge going to  $\alpha$ . As a result, we only need to

```

process counterprocess;
timer minute := 60.0;
dcl min integer;
start ;
  task min := 0;
  nextstate idle;
state idle;
  input clearmsg ;
  task min := 0;
  output d(min);
grst4:
  nextstate idle;
  input incrmsg;
  task min := min + 1;
  output d(min);
  join grst4;
  input startmsg;
  set(minute);
grst5:
  nextstate counting;
endstate;

state counting;
input stopmsg ;
reset(minute);
join grst4;
input minute;
task min := min - 1;
decision min = 0;
(true):
  output ringmsg;
(false):
  output d(min);
  set(minute);
  join grst5;
endstate;
endprocess counterprocess;
    
```

Fig. 4. SDL code for the kitchen timer counter process.

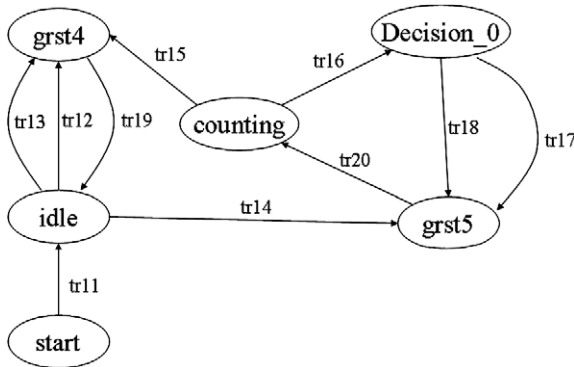


Fig. 5. EFSM for the kitchen timer counter process.

covering the hot spot  $\alpha$  containing the edge from  $\beta$  to  $\alpha$  as part of the sequence.

This search continues until the start node is reached. We then reverse the order of the node/edge sequence obtained by the search to create a path forward. For instance, one possible backward tracking from *grst4* in the counter process might be *grst4*  $\rightarrow$  *tr15*  $\rightarrow$  *counting*  $\rightarrow$  *tr20*  $\rightarrow$  *grst5*  $\rightarrow$  *tr14*  $\rightarrow$  *idle*  $\rightarrow$  *tr11*  $\rightarrow$  *start*. The reverse ordering of these nodes and edges is a path to *grst4*. Since our EFSM data structure contains all the inputs and conditions that must be satisfied for each edge to be traversed as well as the executable tasks associated with that edge, we can easily derive the corresponding test constraints.

After a test sequence has been generated, it is necessary to forward validate it to ensure that it is feasible. For example, if the test intends to exercise a decision branch where the value of  $x$  must be positive, but the decision was preceded by an assignment statement which set  $x$  equal to  $-1$ , then this test sequence would be infeasible. A tool such as a constraint solver could automatically detect this contradiction. We provide the capability to export our test constraints to a format recognizable by the constraint solver proposed in [25], and our tool could easily be extended to support the formats of other available constraint solvers.

If a test sequence is rejected as infeasible, an attempt is made to identify an alternate path that still covers the same selected hot spot. Our test sequence data structure saves the results of the backward tracking search which generated the path from the hot spot to the start node, including all the edges that had already been searched. Therefore, it is possible to reverse the search and return to a point where more than one choice of edge is available. From here, the edge which led to the infeasible path is marked as searched, and like previously described, among the remaining unsearched incoming edges we choose the one leading from the node with the highest weight to generate a new path. Note that if a node has already been covered by a previously generated test sequence, it will be the last node to be backtracked. In the event that all the paths are exhausted without finding one that is feasible, the target hot spot node itself is ruled infeasible.

Once a test sequence is validated as feasible, we mark all the nodes on that sequence as *covered*. Then, we recalculate the weights of all the remaining uncovered nodes (except those infeasible ones) using the dominator analysis described above and identify a new hot spot for test sequence generation. With respect to the all-node coverage criterion, the generation process continues until every feasible node is covered by at least one test sequence.

compare the weights of nodes  $\sigma$ ,  $\epsilon$ ,  $\beta$  and  $\omega$  (but not  $\eta$ ) with each other. Since node  $\beta$  has the highest weight, it is selected as the next node to be backtracked and the corresponding edge is marked as searched. This implies we are going to generate a test sequence

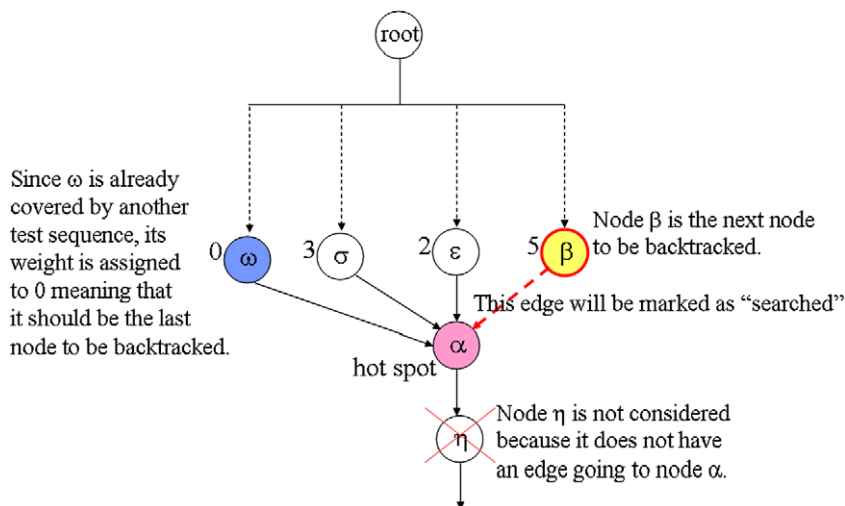


Fig. 6. An illustration of backtracking.

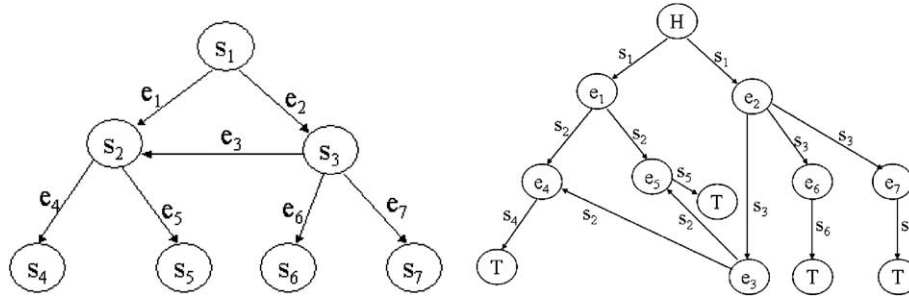


Fig. 7. A sample graph (left) and its dual graph (right). Nodes T and H are “pseudo” nodes in the dual graph.

We can easily extend this approach to allow for test generation that targets a specific “hot spot” edge. A dual graph can be generated according to the process described in [23]. Each edge  $e_i$  in the original EFSM has one corresponding node in the dual graph, and each node  $s_i$  in the original EFSM results in the insertion of  $(incoming(s_i) \times outgoing(s_i))$  corresponding edges into the dual graph, where  $incoming(s_i)$  and  $outgoing(s_i)$  are defined as the number of incoming and outgoing edges respectively for node  $s_i$ . Note that for the special case of a node  $s_j$  with no incoming edges (the start node), we assume  $incoming(s_j) = 1$ . In this case, we insert a pseudo node “H” into the dual graph to act as the head of each edge corresponding to the start node. Similarly, for a node  $s_k$  with no outgoing edges in the original EFSM, we assume  $outgoing(s_k) = 1$ , and we create a pseudo node “T” to act as the tail of the each edge in the dual graph corresponding to node  $s_k$ . Fig. 7 gives a sample graph and its corresponding dual graph. Note that pseudo nodes have no equivalent edges in the original graph and do not factor into any weight or coverage calculations on the dual graph.

The same dominator analysis can be performed on a node in the dual graph, and the resulting weight can then be assigned to the equivalent edge in the original EFSM. Similarly, to generate a test sequence for a specific edge, we apply the same test generation technique to the corresponding node in the dual graph and then translate the dual graph’s node/edge sequence back into its equivalent with respect to the original EFSM. In summary, to achieve all-edge coverage in the original graph, we only need to achieve all-node coverage in the dual graph (excluding any pseudo nodes that have been inserted as part of the dual graph generation process). Since all-edge coverage in the dual graph is not a priority, the number of edges in the dual graph is not anticipated to have much of a negative impact on the practicality of our approach. With the capability for this transformation in mind, any discussion of the all-node coverage criterion can also be applied without loss of generality to the all-edge criterion, unless otherwise noted.

### 3. The TGSP tool

A tool, TGSP, for automatic Test sequence Generation on SDL Processes was implemented. It not only automatically reformats EFSMs from SDL specifications but also represents these EFSMs either in the XML format or in a user friendly graphical interface. The former can be easily converted into different formats using a standard XML parser. This can be very useful as the EFSMs reformatted by TGSP can be used for other research opportunities. The latter has nodes (states) and edges (transitions) displayed in different colors based on their weights calculated using the dominator analysis. Such information is then used for generating test sequences to effectively improve the all-node and all-edge coverage of EFSMs. The focus of this paper is on the test generation including backtracking and forward validation.

The TGSP tool consists of three components: an SDL-2000 parser and syntax tree generator, an EFSM reformatter, a graphical

front-end supporting EFSM display, XML export, test generation, validation and summary, coverage report, etc.

#### 3.1. SDLPT: an SDL parser and syntax tree generator

An SDL-2000 grammar for the tool ANTLR (ANother Tool for Language Recognition) [15] was obtained from the SDL Forum Society website [18]. The grammar was modified to output Java code rather than C++ to facilitate development of a more portable and platform-independent tool. ANTLR processes the grammar file and generates a set of source files for the lexer and parser corresponding to the grammar. The compiled application parses a textual SDL-2000 specification and generates an abstract syntax tree that can be accessed and traversed as a Java object.

#### 3.2. EFSM reformatter

Our EFSM reformatter explores the abstract syntax tree of an SDL specification, extracts relevant nodes (e.g., SDL blocks and processes) and builds a data structure representing the SDL system. The behavioral elements of each process (such as state definitions, inputs, transitions, etc.) are translated into an EFSM as described in Section 2. The hierarchy of blocks and processes is maintained, with each process containing an EFSM object corresponding to the process behavior defined in the SDL specification. Also, the network of communication channels between blocks and processes, as well as the valid signals that can be sent along these channels, is extracted from the specification and included in the data structure. This structure can be output in multiple formats, such as an XML document or a series of Graphviz dot files [7].

#### 3.3. EFSM display with test generator

The graphical EFSM display provides an interface for navigating the blocks and processes in the SDL system data structure generated by the EFSM reformatter. The processes can then be expanded to display the derived EFSMs. Each node and edge of an EFSM is treated as a separate drawing object so that we can define a set of attributes associated with each of them. For example, one attribute is the weight and another attribute is the color. If the cursor is moved onto a node, the corresponding SDL code for that node is displayed. Also, when we click on a node (or an edge), it generates a “potential” (still needs to be forward validated) test sequence to cover that node (or edge). We use the Graphviz (graph visualization software) package from AT&T [7,9] to calculate the layout of the nodes and edges on the screen.<sup>4</sup> The input for this step is a dot file. An example of this is in Part (a) of Fig. 8 where state1[label = “start”] and state2 [label = “E\_IDLE”] imply state1 has a name “start” and state2 has a name “E\_IDLE,” and

<sup>4</sup> The graph generated by Graphviz is an image file, and the nodes and edges are non-interactive. Hence, these graphs are not suitable for our use.

```

state1 [label="start"];
state2 [label="E_IDLE"];
state1 -> state2 [label="tr1"];
state3 [label="E_WCOT"];
state4 [label="E_DONE"];
state2 -> state3 [label="tr2"];
state2 -> state4 [label="tr3"];

state1 [label=start, pos="1188,3338", width="0.75", height="0.50"];
state2 [label=E_IDLE, pos="1188,3246", width="1.11", height="0.50"];
state3 [label=E_WCOT, pos="842,1406", width="1.28", height="0.50"];
state1 -> state2 [label=tr1, pos="e,1188,3264 1188,3320 1188,3307 1188,3288
1188,3273", lp="1212,3292"];
state2 -> state3 [label=tr2, pos="e,955,3169 1151,3239 1120,3233 1076,3223
1039,3210 1012,3200 984,3186 962,3173", lp="1070,3200"];
    
```

Fig. 8. Part (a): a sample of a partial dot file generated by the EFSM reformatter. Part (b): the corresponding Tdot file generated by Graphviz.

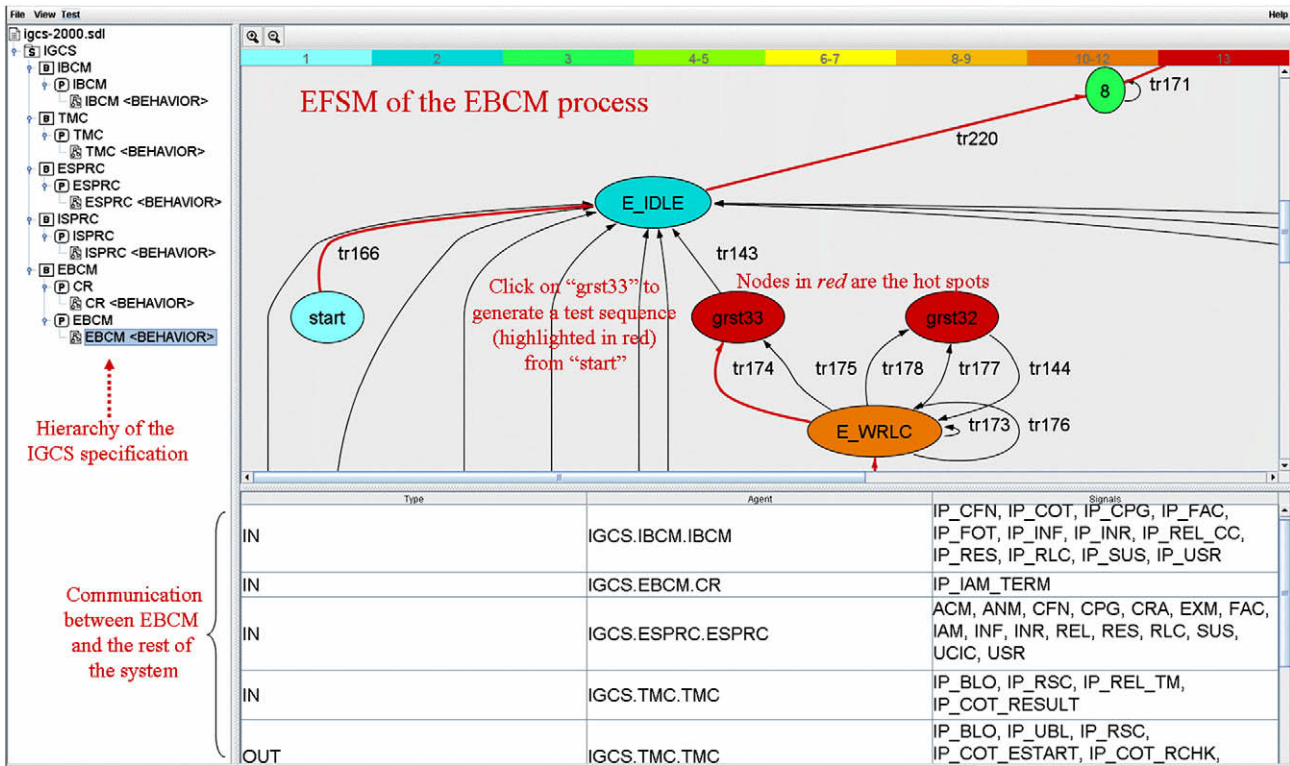


Fig. 9. Partial display of the EBCM process of IGCS.

state1 → state2[label = "tr1"] implies the transition from state1 to state2 is labeled as "tr1," etc. The output from Graphviz is saved as a Tdot file. Part (b) of Fig. 8 shows the Tdot file corresponding to the input in Part (a). The position of each node is decided by its "pos" (the coordinates of the upper-left corner), "width" and "height". Each edge is represented by a curve going through all the points whose coordinates are listed along with the edge. The actual drawing is done by using Java2D which draws all the objects (nodes and edges) of an EFSM on the display canvas of TGSP with their locations specified by the corresponding entries in the Tdot file generated by Graphviz.

The results of the dominator analysis on the EFSM are used to assign a weight and a color to each visible node.<sup>5</sup> At most eight different colors are displayed; if there are more than eight distinct weights, each color represents a range of weights. The colors are arranged on a spectrum from light blue to dark red. Referring to Fig. 9, covering a light blue node implies that only the node itself is guaranteed to be covered, while covering a red node (a hot spot) will cover a number of nodes equal to or greater than the highest node weight found in the EFSM.

As an example, we consider the SDL specification of the IGCS call agent software. It was developed by Telcordia Technologies (formerly Bellcore) as part of a VoIP softswitch technology package. The purpose of the system is to act as a virtual switch and help to deliver data, voice, and video services to subscribers. The IGCS specification contains about 2200 lines of SDL code, with five blocks and six processes. Fig. 9 displays the IGCS specification loaded into our tool, with part of the EFSM for the EBCM process displayed graphically. The left portion shows the hierarchy of IGCS. The lower right portion of this figure gives a partial list of the communication channels between the EBCM process and other processes in IGCS. For example, it receives incoming signals (IP\_CFN, IP\_COT, etc.) from the IBCM process and sends outgoing signals (IP\_BLO, IP\_UBL, etc.) to the TMC process. Since different colors in Fig. 9 are used to show the weights of different nodes, it is better viewed in color.<sup>6</sup> This is also the case for other figures in the remainder of the paper. Note that the two dark red "hot spot" nodes are visible; a test case that covers one of these nodes is guaranteed to cover at least thirteen nodes in total, the maximum value of the current node weights.

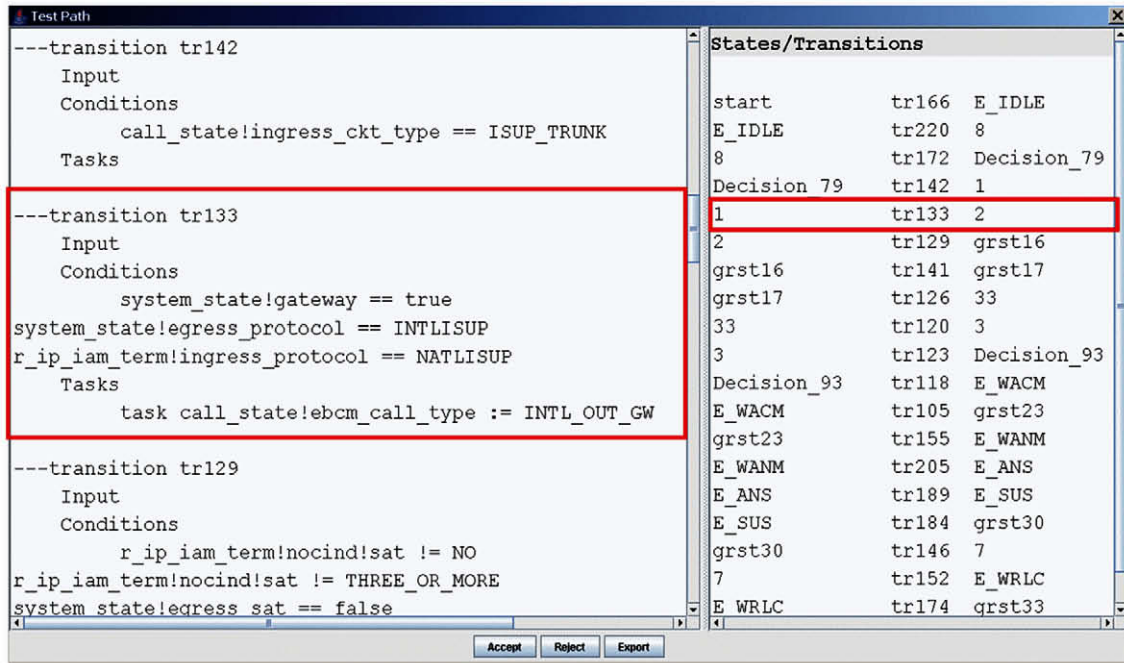
<sup>5</sup> As explained in Section 2, with the capability for the transformation between a graph and its dual graph, we focus our discussion only on the "nodes" as it can also be applied without loss of generality to the "edges."

<sup>6</sup> The colors provide additional visualization benefits when the paper is presented in a full-color format, such as online versions posted at the IST web site.

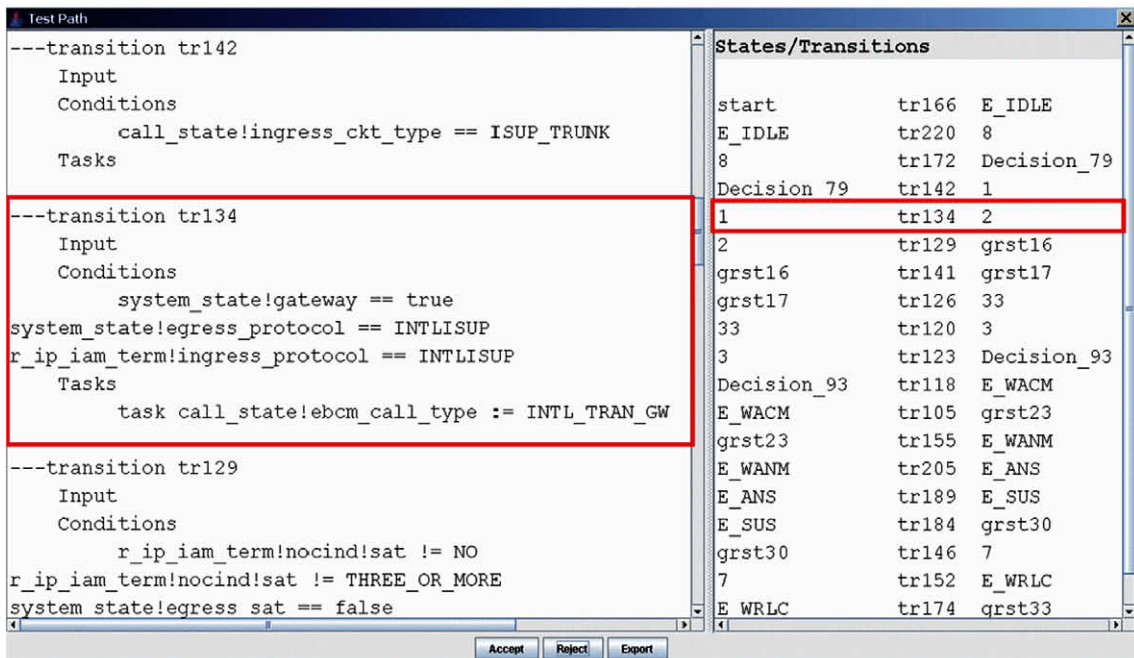
To generate a test sequence for a selected node in the EFSM (say grst33), the user simply clicks on that node. Backward tracking as described above is used to choose a path from the start node to the desired location. The user is then presented with both a graphical and textual display of the generated test sequence. In the EFSM pane, the generated path is highlighted in red (see Fig. 9). Also, as shown in Part (a) of Fig. 10, the user is presented with a text box which shows the path taken by the test sequence, as well as the details of each edge transition along the path: any inputs that must be received, conditions that must be satisfied, or tasks (e.g.,

variable assignments) that are executed. The user is then given the choice to accept or reject the generated test sequence. We provide an option to export constraints of a test sequence to a textual format readable by the constraint solver proposed in [25], as shown in Fig. 11.

If the test sequence is rejected, a different path (if one exists) is generated using the approach discussed in Section 2 and displayed in the EFSM pane. Parts (a) and (b) of Fig. 10 compare the original test sequence with an alternate test sequence generated if the original is rejected. The difference between the original and

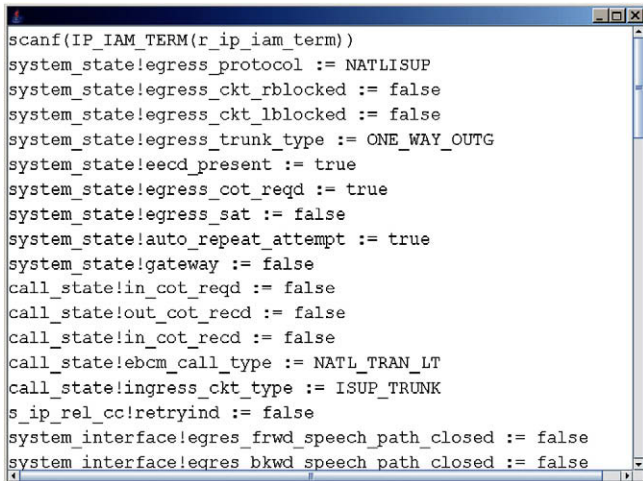


(a) Textual test sequence presented to the user



(b) Alternate test sequence generated if that in (a) is rejected. Elements in the red boxes indicate differences from the original test sequence.

Fig. 10. Two test sequences for the node "grst33" in the EBCM process.



```
scanf(IP_IAM_TERM(r_ip_iam_term))
system_state!egress_protocol := NATLISUP
system_state!egress_ckt_rblocked := false
system_state!egress_ckt_lblocked := false
system_state!egress_trunk_type := ONE_WAY_OUTG
system_state!eecd_present := true
system_state!egress_cot_reqd := true
system_state!egress_sat := false
system_state!auto_repeat_attempt := true
system_state!gateway := false
call_state!in_cot_reqd := false
call_state!out_cot_reqd := false
call_state!in_cot_reqd := false
call_state!ebcm_call_type := NATL_TRAN_LT
call_state!ingress_ckt_type := ISUP_TRUNK
s_ip_rel_cc!retryind := false
system_interface!egres_frwd_speech_path_closed := false
system_interface!egres_bkwd_speech_path_closed := false
```

Fig. 11. Test constraints in Part (a) of Fig. 10 exported to a constraint solver textual format.

the alternate test sequences is that the former has a path from node “1” to node “2” via edge “tr133”, whereas the latter has node “1” go to node “2” via edge “tr134.” In the event that no alternate path can be generated, the user is notified with a message indicating that the node is infeasible. If the user accepts the second generated test sequence, it is added to the current test set, and the coverage of the EFSM is updated to reflect the addition of the new test. That is, the weights of the uncovered nodes are updated and re-colored to identify new hot spots. Refer to Part (a) of Fig. 12 for the updated display of the EBCM process after the test sequence in Part (b) of Fig. 10 is accepted. Note that the originally selected hot spot (grst33) has its color changed from red to gray indicating that it has already been covered. In the EFSM pane, the accepted test sequence is highlighted in blue. We have new hot spots (e.g., grst20) as shown in Part (b) of Fig. 12. Compared with the original hot spot (grst33 in Fig. 9) which has a weight of 13, the weight of grst20 is only 3 because 20 of the 42 nodes in EBCM have already been covered by the test sequence in Part (b) of Fig. 10. We can generate a test sequence, as the one highlighted in red in Part (b) of Fig. 12, to cover the new hot spot grst20. Suppose we also accept this test sequence as a feasible sequence. Fig. 13 gives the current test set after accepting two test sequences from Part (b) of Figs. 10 and 12, respectively. We can report the state and edge coverage with respect to each individual test sequence and the cumulative coverage with respect to a set of selected test sequences. We can also display the details of a selected test sequence.

Due to the space limitation, other features of TGSP such as viewing the weight of each edge, displaying a selected portion of an EFSM, zooming in and out, etc. are not included in the paper.

#### 4. Five case studies

Five case studies were conducted to demonstrate the feasibility of using our methodology in automatic generation of test sequences to effectively improve the all-node and all-edge coverage of SDL processes. The first two studies used the EBCM and IBCM processes of the IGCS system introduced in Section 3.3. The system is part of a VoIP softswitch package developed at Telcordia Technologies (formerly Bellcore). We also obtained a specification for a railway crossing system from the SDL Forum website for the third study. This specification was the winning entry in an SDL design contest held at the third SAM workshop ([http://www.sdl-forum.org/SAM\\_contest/](http://www.sdl-forum.org/SAM_contest/)). The purpose of this system is to safely regulate traffic at a location where a roadway crosses the railroad tracks.

The system accomplishes this by raising or lowering the crossing gate and issuing start and stop commands to any trains present. For our case study, we chose the *controller* process, which triggers the appropriate action by considering the number of cars on the road as well as the number of oncoming trains. The final two studies used processes, *l2adapter* and *ccproc*, obtained as part of an SDL specification of the Bluetooth TCSBIN (Telephony Control Specification – Binary) protocol layer implemented by Motorola. This protocol supports the establishment of voice and data calls between two Bluetooth devices. Its functions include connection creation and management, call control, and other supplementary services. Table 1 gives the number of nodes and edges of each process. Below, we first present the studies for the all-node coverage criterion including the data collected and the analysis in Section 4.1, followed by the same for the all-edge criterion in Section 4.2. Additional discussion is in Section 4.3.

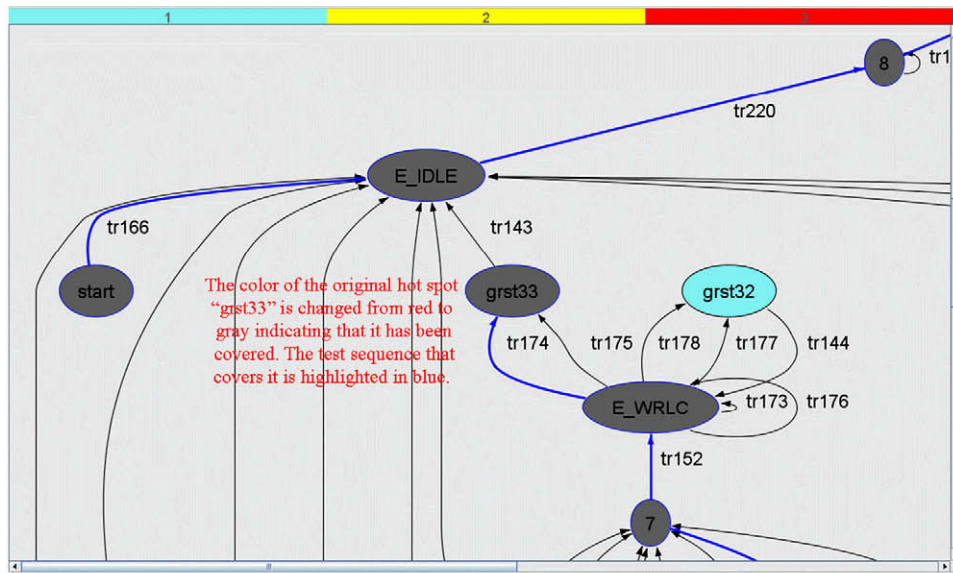
##### 4.1. Node coverage

While generating a test set that satisfies the all-node coverage criterion on the EFSM, a tester first selects the hot spot (the node with the highest weight) and generates a test sequence to cover it. The hot spot is highlighted in red and can be easily visualized in the EFSM pane of the TGSP graphical interface as shown in Fig. 9. In the event that the highest weight is shared by multiple nodes, the tie is broken by arbitrarily choosing one of them. If the test sequence has no contradictions, it is accepted. Otherwise, it will be rejected and an alternate test sequence will be generated to cover the same hot spot. This backtracking and forward validation continues until either a generated test sequence is accepted and included in the current test set or all the possible sequences for covering the selected hot spot have been identified as infeasible. If it is the latter, the selected hot spot will be marked as an infeasible node. Based on the updated coverage, a new hot spot is then identified, and the generation process is repeated.

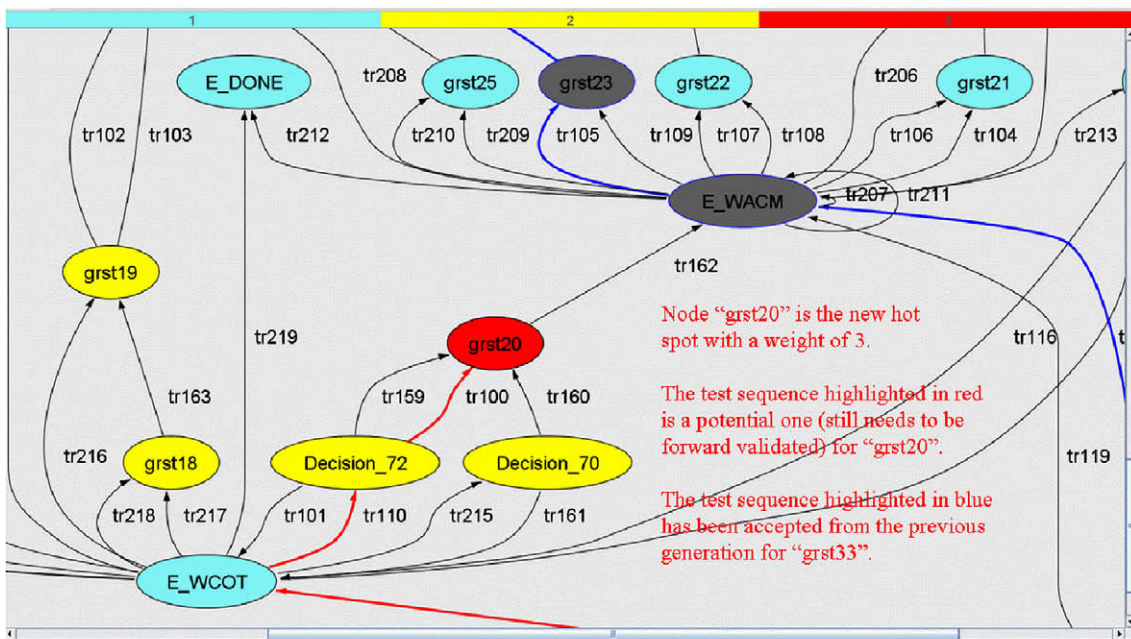
Since there may be more than one hot spot at each step of the test generation, the hot spot which is selected first may have an impact on the subsequent test sequences generated. Recall that the weight of a node is calculated, using a *conservative* approach, as the number of uncovered nodes that will be *guaranteed* to be covered if the node under consideration is covered. If there are, for example,  $n$  hot spots with the same weight of  $\omega$ , then no matter which of these  $n$  spots is selected and which test sequence is generated to cover the selected hot spot, the coverage will increase by *at least*  $\omega$  nodes. To eliminate the possible bias from using only one set of test sequences, three testers working independently each generated their own all-node adequate test sets (giving 100% node coverage) for all five of the SDL processes using their best judgment. This implies that when there is more than one hot spot, different testers are very likely to select different hot spots for test generation. Let  $T_{EBCM}^{n,1}$ ,  $T_{EBCM}^{n,2}$  and  $T_{EBCM}^{n,3}$  be the all-node adequate test sets for the EBCM process generated by testers 1, 2 and 3, respectively. Table 2 gives the first three sequences of these test sets.<sup>7</sup> Column 1 indicates whether it is the first, second or third sequence generated during the generation process. Columns 2–4 show the cumulative number of nodes covered. For example, the first two sequences of  $T_{EBCM}^{n,1}$  cover a total of 23 nodes, whereas the first three sequences of  $T_{EBCM}^{n,3}$  cover 25 nodes. Columns 5–7 give the number of nodes in each test sequence. For example, the third sequence of  $T_{EBCM}^{n,1}$  traverses 6 nodes, whereas the second sequence of  $T_{EBCM}^{n,2}$  traverses 13 nodes. We can also calculate the average cumulative number of nodes covered by a set of test sequences. For example,

<sup>7</sup> Due to the space limit, we only present part of the data in the tabulated format. The average cumulative node coverage versus the number of test sequences is presented in Fig. 14 for each process.





Part (a) The original hot spot grst33 has been covered



Part (b) The new hot spot has a smaller weight of 3

Fig. 12. Partial display of the updated EBCM process after acceptance of the test sequence from Part (b) of Fig. 10.

on average there are 19  $((20 + 20 + 17)/3)$ , 22.67  $((23 + 23 + 22)/3)$  and 25  $((24 + 26 + 25)/3)$  nodes covered by the first, the first two, and the first three sequences of  $T_{EBCM}^{n,1}$ ,  $T_{EBCM}^{n,2}$  and  $T_{EBCM}^{n,3}$ . These numbers are then divided by the number of nodes of the EBCM process to get the average cumulative percentages of node coverage. These percentages are plotted against the number of test sequences as shown in Fig. 14.

The results of the node coverage-based test generation are summarized in Fig. 14. Parts (a), (b), (c), (d) and (e) give the average cumulative node coverage (in percentage) versus the number of test sequences for IBCM, EBCM, ccproc, l2adapter and controller processes. These curves show how each additional test sequence increases the node coverage. Each curve has a steeper slope in the beginning, implying that node coverage increases in a more effective way with respect to the first few test sequences. We also note that coverage of more than 50% of the nodes was achieved

after only two test sequences. This clearly shows that our test generation method can generate test sequences to effectively increase the all-node coverage.

Let  $\alpha$  be the number of test sequences considered (in the order they are generated) and  $R_{\alpha}^n$  be the ratio of the average cumulative number of nodes traversed by the first  $\alpha$  test sequences in  $T_p^{n,1}$ ,  $T_p^{n,2}$  and  $T_p^{n,3}$  (the all-node adequate test sets generated for process  $P$  by testers 1, 2 and 3, respectively) to the average cumulative number of nodes covered by the same  $\alpha$  sequences. The second average number is calculated before the curves in Fig. 14 are plotted, whereas the first average number can be calculated from the data in columns 5–7 in Table 2. For example, with respect to the EBCM process the average number of nodes traversed by the first two sequences of  $T_{EBCM}^{n,1}$ ,  $T_{EBCM}^{n,2}$  and  $T_{EBCM}^{n,3}$  is  $(20 + 13 + 20 + 13 + 17 + 17)/3 = 33.33$ . A smaller value of  $R_{\alpha}^n$  indicates most nodes in the corresponding test sequences are not previously covered; thus,

Active	Delete	Test Name	Covered Nodes	Covered Edges
<input checked="" type="checkbox"/>	<input type="checkbox"/>	Test 1	20/42	19/121
<input checked="" type="checkbox"/>	<input type="checkbox"/>	Test 2	13/42	12/121
Cumulative:			23/42	24/121

Inputs/Conditions/Tasks	States/Transitions
---transition tr166	start tr166 E_IDLE
Input	E_IDLE tr220 8
Conditions	8 tr172 Decision_79
Tasks	Decision_79 tr142 1
	1 tr133 2
	2 tr127 grst17
---transition tr220	grst17 tr126 33

Fig. 13. Current test set display window after accepting test sequences from Part (b) of Figs. 10 and 12.

Table 1

Number of nodes and edges for each SDL process.

	Number of nodes	Number of edges
IGCS/IBCM	43	152
IGCS/EBCM	42	121
Bluetooth/ccproc	27	61
Bluetooth/l2adapter	18	68
Railway/controller	20	137

covering them effectively increases the overall node coverage. This means test sequences with lower values of  $R_{\alpha}^n$  are of great significance from the point of view of coverage improvement. An observation is that  $R_{\alpha}^n$  is always 1.00 when  $\alpha$  equals 1. This is because in the beginning, none of the nodes are covered. Hence, the number of nodes covered equals the number of nodes traversed.

From Table 3, we observe that if we consider the first two sequences generated (i.e.,  $\alpha$  equals 2) for the EBCM process, the ratio is 1.47 (33.33/22.67), meaning that on average for every 1.47 nodes traversed, a node that was not previously covered will be covered. Similarly, for the ccproc process when the first three sequences are considered, on average one additional uncovered node will be covered for every 1.29 nodes traversed. All the values of  $R_{\alpha}^n$  in Table 3 are small, which reinforces our prior observation that the first few test sequences improve the node coverage in a very effective way.

#### 4.2. Edge coverage

Three all-edge adequate test sets were generated for each SDL process. The procedure was similar to that of node coverage, except

Table 2

The first three test sequences of the EBCM process for the node coverage.

	Cumulative number of nodes covered			Number of nodes traversed by each test sequence		
	$T_{EBCM}^{n,1}$	$T_{EBCM}^{n,2}$	$T_{EBCM}^{n,3}$	$T_{EBCM}^{n,1}$	$T_{EBCM}^{n,2}$	$T_{EBCM}^{n,3}$
1st test sequence	20	20	17	20	20	17
2nd test sequence	23	23	22	13	13	17
3rd test sequence	24	26	25	6	14	13

that we selected test sequences to cover the edge with the highest weight, rather than the highest weighted node. Let  $T_{EBCM}^{e,1}$ ,  $T_{EBCM}^{e,2}$  and  $T_{EBCM}^{e,3}$  be the all-edge adequate test sets for the EBCM process generated by testers 1, 2 and 3, respectively. Table 4 lists the first five sequences of these test sets where each column has the same meaning as in Table 2 except that node is replaced by edge. Similarly, the average cumulative number of edges covered by a set of test sequences can be calculated by using the data in columns 2–4. For example, on average there are 25, 37, 40, 44.33 and 49.33 edges covered by the first one, two, three, four and five sequences of  $T_{EBCM}^{e,1}$ ,  $T_{EBCM}^{e,2}$  and  $T_{EBCM}^{e,3}$ . The average cumulative edge coverage versus the number of test sequences is summarized in Fig. 15. As shown, each additional generated test sequence increases the edge coverage, with the first few sequences making the most effective contribution. These data suggest that our test generation method can also generate test sequences to effectively increase the edge coverage.

Following the same convention as the node coverage, we calculate  $R_{\beta}^e$  as the ratio of the average cumulative number of edges traversed by the first  $\beta$  test sequences of the corresponding three all-edge adequate test sets to the average cumulative number of edges covered by the same  $\beta$  sequences. For example, from Table 5 we observe that if we consider the first three sequences generated (i.e.,  $\beta$  equals 3) for the EBCM process, the ratio is 1.27, meaning that on average for every 1.27 edges traversed, an edge that was not previously covered will be covered. In summary, a smaller value of  $R_{\beta}^e$  indicates most edges in the corresponding test sequences were not previously covered; thus, covering them effectively increases the overall edge coverage. This means test sequences with lower values of  $R_{\beta}^e$  are of great significance for effective coverage

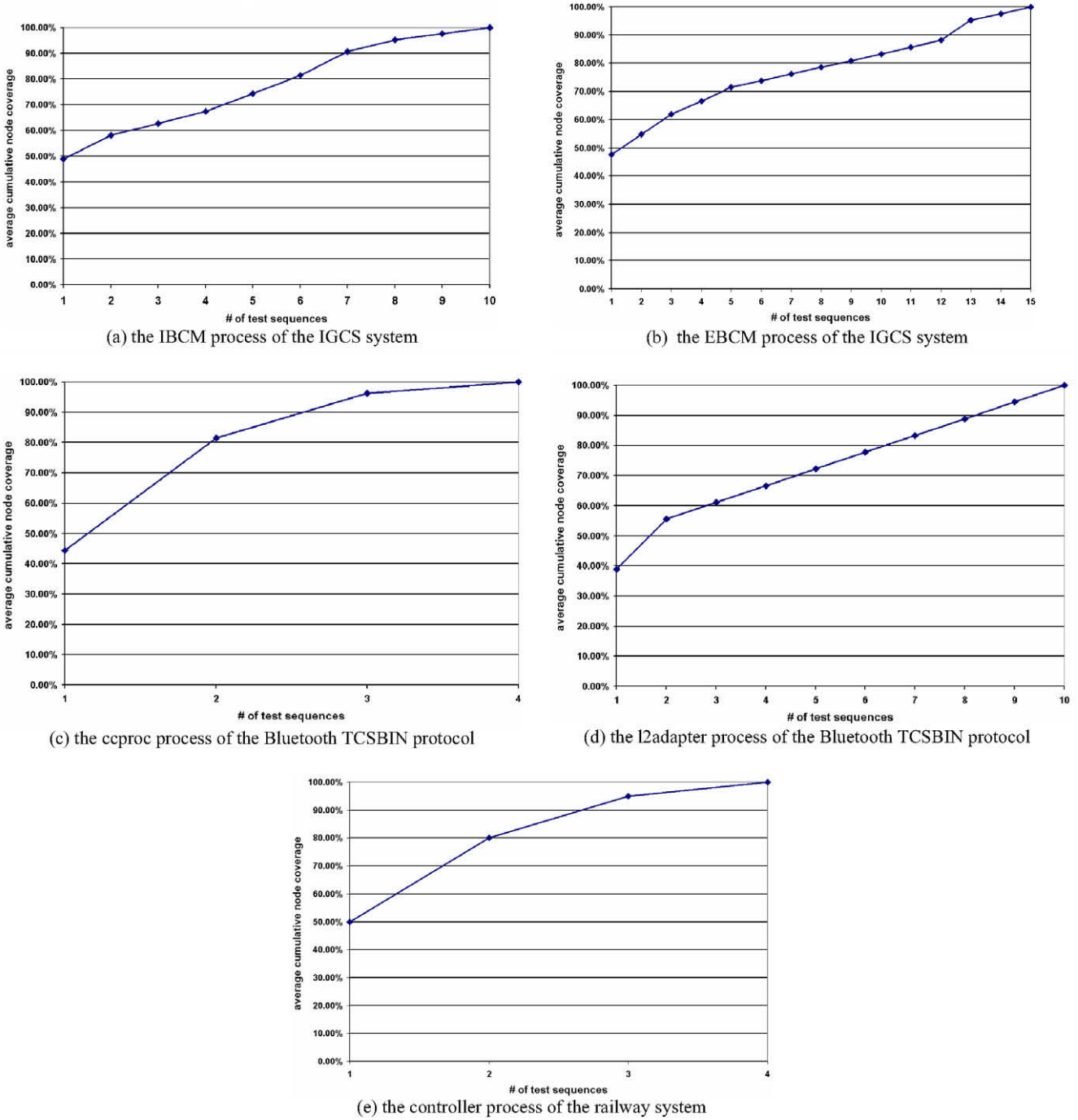


Fig. 14. The average cumulative node coverage versus the number of test sequences.

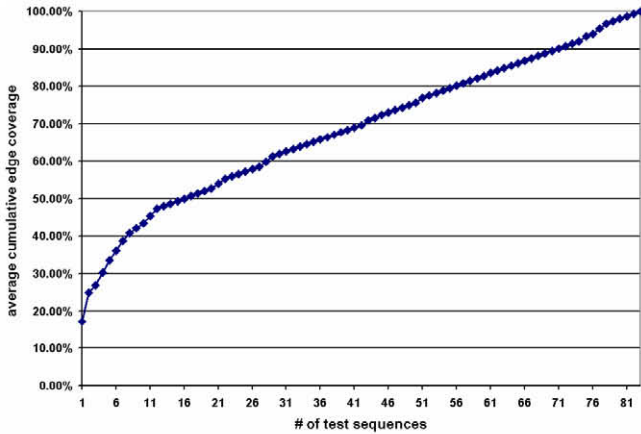
Table 3

Node coverage effectiveness of the first three test sequences generated.

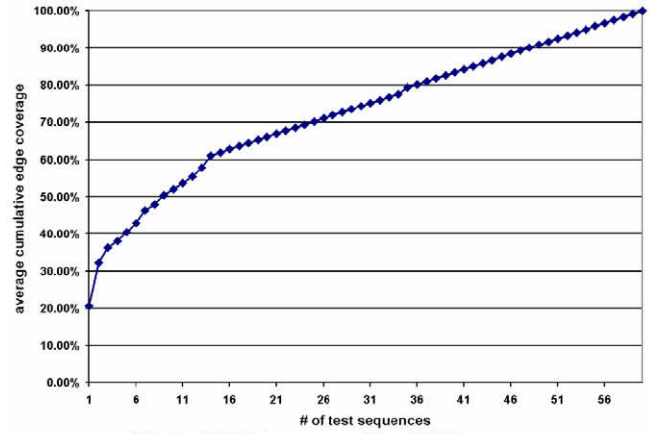
Process name	$\alpha$	$R_{\alpha}^n$	Process name	$\alpha$	$R_{\alpha}^n$
IGCS/IBCM	1	1.00	Bluetooth/ccproc	1	1.00
	2	1.57		2	1.16
	3	1.93		3	1.29
IGCS/EBCM	1	1.00	Bluetooth/l2adapter	1	1.00
	2	1.47		2	1.30
	3	1.77		3	1.55
Railway/controller	1	1.00			
	2	1.37			
	3	1.50			

**Table 4**  
The first five test sequences of the EBCM process for the edge coverage.

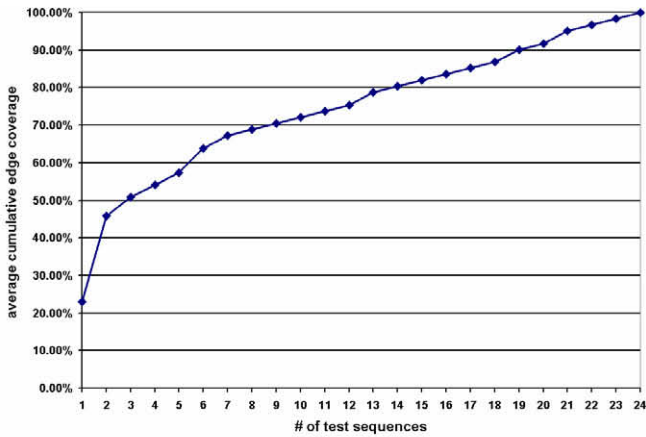
	Cumulative number of edges covered			Number of edges traversed by each test sequence		
	$T_{EBCM}^{e.1}$	$T_{EBCM}^{e.2}$	$T_{EBCM}^{e.3}$	$T_{EBCM}^{e.1}$	$T_{EBCM}^{e.2}$	$T_{EBCM}^{e.3}$
1st test sequence	23	26	26	23	26	26
2nd test sequence	35	38	38	17	20	17
3rd test sequence	38	41	41	13	16	13
4th test sequence	42	45	46	15	18	18
5th test sequence	47	50	51	15	18	15



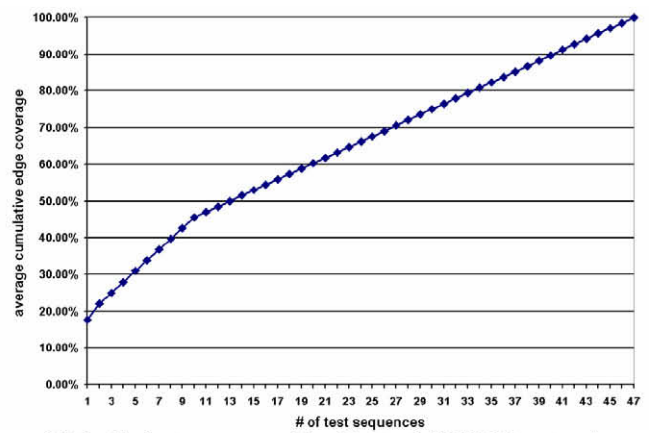
(a) the IBCM process of the IGCS system



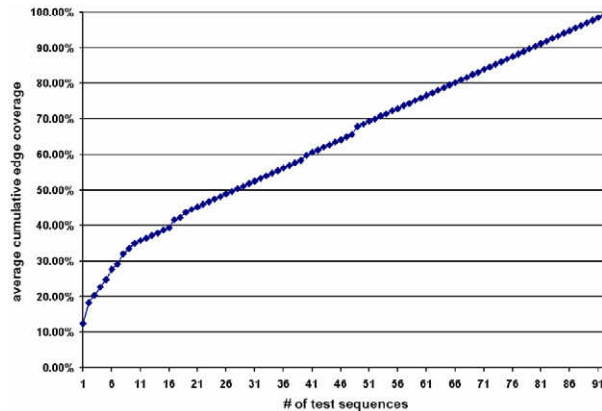
(b) the EBCM process of the IGCS system



(c) the ccproc process of the Bluetooth TCSBIN protocol



(d) the l2adapter process of the Bluetooth TCSBIN protocol



(e) the controller process of the railway system

**Fig. 15.** The average cumulative edge coverage versus the number of test sequences.

**Table 5**

Edge coverage effectiveness of the first five test sequences generated.

Process name	$\beta$	$R_{\beta}^e$	Process name	$\beta$	$R_{\beta}^e$
IGCS/IBCM	1	1.00	Bluetooth/ccproc	1	1.00
	2	1.16		2	1.06
	3	1.43		3	1.21
	4	1.67		4	1.31
	5	1.82		5	1.46
IGCS/EBCM	1	1.00	Bluetooth/l2adapter	1	1.00
	2	1.10		2	1.40
	3	1.27		3	1.55
	4	1.45		4	1.70
	5	1.62		5	1.76
Railway/controller	1	1.00			
	2	1.08			
	3	1.18			
	4	1.29			
	5	1.40			

**Table 6**

Node (edge) coverage effectiveness for the entire all-node (all-edge) adequate test.

Process	$R_{\text{adequate}}^n$	$R_{\text{adequate}}^e$
IGCS/IBCM	3.84	7.70
IGCS/EBCM	5.24	5.40
Bluetooth/ccproc	1.93	4.50
Bluetooth/l2adapter	1.58	3.69
Railway/controller	2.50	3.06

improvement. Our observation that the first few test sequences improve the edge coverage effectively is further supported by the small values of  $R_{\beta}^e$  in Table 5.

### 4.3. Discussion

In this section, we discuss some important aspects related to our test generation method.

- To better support the claim that our method can generate test sequences to effectively improve the node (edge) coverage, not only do we need to look at the curves obtained by plotting the average cumulative node (edge) coverage versus the number of test sequences (see Figs. 14 and 15), but we also need to calculate  $R_{\alpha}^n$  and  $R_{\beta}^e$  to examine the ratio of the average cumulative number of nodes (edges) traversed to the average cumulative number of nodes (edges) covered by the first  $\alpha$  test sequences (see Tables 3 and 5 which give these ratios for the first three and five sequences, respectively).

- Table 6 gives the ratios for the entire all-node and all-edge adequate test sets. More specifically,  $R_{\text{adequate}}^n$  ( $R_{\text{adequate}}^e$ ) is calculated as the ratio of the average cumulative number of nodes (edges) traversed by all the test sequences in the all-node (all-edge) adequate test set of an SDL process to the number of nodes (edges) of that process. For example, to generate an all-node adequate test set for the l2adapter process, on average we only need to traverse 1.58 nodes in order to cover an uncovered node. This is very effective. For other processes, this ratio ranges from 1.93 to 3.84 (except for EBCM which has a ratio of 5.24), which indicates our test generation is still effective. The ratios for edges are relatively higher (ranging from 3.06 to 5.40 except for IBCM which has a ratio of 7.70). Nevertheless, considering how difficult and frustrating it can be to generate additional test sequences to cover the last few remaining uncovered edges for achieving the 100% adequacy, these ratios suggest that our test generation method is effective for coverage improvement.

- The actual values of these ratios depend on many factors, including the structure of the EFSM (derived from an SDL process), the number of nodes (edges), etc. It is more difficult to cover the remaining uncovered nodes (edges) of an EFSM when the coverage is already high. This is especially true for those with a more complicated structure, such as a larger number of loops or decision branches, and/or more nodes (edges). Consequently, they have a larger  $R_{\text{adequate}}^n$  ( $R_{\text{adequate}}^e$ ). Stated differently, although test sequences generated in the beginning increase the coverage effectively, those generated at the end become less effective. The latter may include many nodes (edges) which are already covered and only increase the coverage by a relatively small number such as one or two nodes (edges).

- We observe that after the initial significant improvement, the edge coverage increases at a slower and steadier rate. This is especially true for the test sequences generated towards the end to cover the remaining uncovered edges. In many cases, each of these test sequences only increases the edge coverage by one additional edge. This trend is not so obvious for the node coverage.

- Although there may be more than one hot spot at each step of the test generation and selection of a different hot spot may affect the subsequent test generation, our experimental data show that for any of the five SDL processes we studied, the difference in the number of test sequences of the three all-node (or all-edge) adequate test sets is very small. One explanation is that we use a *conservative* approach as discussed in Section 2 to calculate the weight of each uncovered node. In this way, we can guarantee that whatever hot spot (assuming with a weight of  $\omega$ ) is selected and no matter which test sequence is generated to cover the selected hot spot, *at least*  $\omega$  uncovered nodes will be covered if the selected hot spot is covered.

- Identifying “infeasible paths” is always a challenge for any model-based test generation methods except for those on the reachability graphs where every path is feasible. A high number of infeasible paths can significantly reduce the performance and the applicability of test generation methods such as the one proposed in [8]. In this paper, we solve the *infeasibility* problem using a two-step approach. First, a greedy approach is used to backtrack a test sequence that covers a selected hot spot. Then, the set of constraints in the test sequence is examined to detect any contradiction during the forward validation. We provide the capability to export the test sequence constraints to a format readable by a constraint solver presented in [25]. These constraints can also be easily reformatted and exported to other constraint solvers. The only reason why we still need some manual intervention in our studies is because the constraint solver in [25] is not yet robust enough. Such intervention can be minimized (or ideally eliminated) if a more

powerful constraint solver is used to replace the one in [25]. We think this is possible as our test generation method only requires the constraint solver to solve one particular set of constraints at a time. That is, instead of using a constraint solver to determine whether there is one *feasible* path from the start node to a selected hot spot, TGSP only asks the solver to identify whether a given test sequence is feasible. The latter is a much easier problem than the former. However, developing a constraint solver is outside the scope of the current paper.

- We would like to emphasize that the effectiveness of our test generation method varies for different SDL systems. The constraint solver used for the forward validation also has an impact.
- Although we propose a test generation method aiming at the white box coverage-based testing of EFSMs, it does not imply that we should ignore other conformance testing methods. To the contrary, we believe all of them are complementary.
- We claim that our test generation method leads to a test set that achieves high coverage with fewer test sequences, which is cheaper and easier to execute and maintain. Note that this advantage is with respect to the execution of the test sequences against the SDL specification in an appropriate software simulation environment, such as Telelogic Tau [20]. In some systems, the execution of certain test sequences may imply significant overhead, such as the need to repeatedly reset multiple distributed and heterogeneous components (databases, file systems, etc.) to their initial state. For such a system, the execution cost of a test set cannot be judged solely on the number of test sequences.

## 5. Related studies

A study reported in [19] generates FSMs from an SDL specification through a tool FEX [2] or directly by using a text editor. While the manual generation might be a suitable choice for simple SDL specifications, it is difficult for large complex systems such as IGCS and Bluetooth/TCSBIN used in our studies. The tool FEX is an FSM extractor which extracts an FSM from an SDL specification for a partial view of its behavior. This implies the FSM extracted by using FEX is an “approximating machine” as it only approximates the behavior of an SDL specification. During the extraction, FEX uses an approach to partially unfold variables of enumerated types and incorporates the enabling conditions as part of the corresponding FSM inputs. Although FEX was originally built to extract an FSM from an SDL specification, with modifications it can also be used to extract the corresponding EFSM [3]. When compared with our TGSP tool, FEX suffers from two major drawbacks. First, the capability of handling complex SDL systems is limited. Second, the EFSMs are not in a standard format. Additional transformers are needed to transform the FEX-specific outputs into other formats such as the normal form [17]. This reduces the compatibility and interoperability of FEX with other applications. In particular, the EFSMs from FEX can only be directly used by the specific test generation method described in [3], which will be discussed later. TGSP has no such problem as it can represent EFSMs in the XML format which can be easily converted into different formats using a standard XML parser.

Following is a brief overview of some representative studies related to the subject of this paper. There are two major differences between these studies and our method. First, none of them is focused on how *new* test sequences should be generated to effectively increase the coverage. Second, many of them are focused on the conformance testing on the system implemented based on the specification (i.e., testing the consistency between the specification and the implementation) with an assumption that the underlying specification is correct.

Automated test generation based on an FSM (Finite State Machines) has been previously explored and many techniques

exist such as the W-method [5], the partial W-method (also known as the Wp-method) [6] as an improvement over the W-method in terms of the size of the test set generated while retaining the fault detection ability, and the distinguished sequence method or the D, W, UIO, UIOV, and HIS-methods [10,11]. These techniques can be used to satisfy various coverage criteria, the most popular of which are state coverage, branch coverage, or fault coverage. Petrenko and Yevtushenko [16] developed a conformance test generation approach which checks an implementation FSM against a design-level FSM that may be non-deterministic or have behavior not completely defined in all cases.

Bourhfir et al. [3] proposed a test selection technique for communication protocols represented by an SDL specification. For each individual EFSM from an SDL process, they computed a so-called partial product, which is similar to a reachability graph. Test cases for each process were then generated based on its partial product, and the associated input/output sequences for each test case were suitable for verifying the conformance of an implementation against the SDL specification. Test generation on the partial product was guided by the UIO (Unique Input Output) and all-def-use coverage criteria.

Maloku and Frey-Pucko [14] proposed a test generation technique for EFSM-based specifications. They constructed flowgraphs which represent the changes and associations of inputs, variables, and outputs throughout the specification. Test selection was guided by these associations and the all-uses coverage criterion. A simulation process was used to verify the feasibility and executability of each generated test sequence.

Several techniques have been developed for generating tests for SDL specifications based on control flow or data flow-based criteria [4,12,13,21]. However, these tests are ultimately targeted at the system implementation rather than the specification. The purpose is to create a test set that verifies conformance of the implementation’s behavior against the control flow and data dependencies defined in the SDL specification, with the underlying assumption that the specification is error-free. Our approach goes beyond this, emphasizing a coverage-based white box testing approach with the intention of checking the correctness of the SDL specification itself.

The Tau suite from Telelogic [20] is a commercial software tool with test generation capabilities for SDL specifications. However, the tests are generated from a black box perspective, and the testing process is based on simulation with an emphasis on the correct functionality of the system rather than white box coverage measurement. Our test generation technique takes the alternate approach, relying on such coverage measurement to determine which parts of the system have already been executed and to identify which parts should be targeted by future tests.

Wong and Lei reported a reachability graph-based test sequence generation for concurrent programs [23]. The major difference between that study and this one is that every node in a reachability graph is a reachable state and every path is a feasible test execution sequence. As a result, no forward validation is required in [23] for any test sequence generated by the backtracking. This is not the case for EFSM-based testing generation with respect to each SDL process.

## 6. Conclusions and future research

In this paper, we have presented a test generation method for SDL processes which can satisfy white box-based coverage criteria such as all-node or all-edge for the purposes of validating the specification itself. Our generated test set is also suitable for conformance testing between the specification and an implementation. Our tool, TGSP, reformats EFSMs from an SDL specification, identifies “hot spot” nodes or edges which should be prioritized for

testing to effectively increase coverage, and supports automatic generation of test sequences to cover a selected node or edge. Case studies based on five SDL processes (from the IGCS (intelligent gateway call server) system developed by Telcordia Technologies, the Bluetooth TCSBIN (Telephony Control Specification – Binary) protocol by Motorola, and a railway system from the SDL Forum website) show that all-node or all-edge coverage can be achieved in an effective way using our method. The results show that the first few test sequences generated by our method increase the coverage much more effectively than the rest of the sequences. Also, for each additional test sequence generated, the ratio of the cumulative number of nodes (edges) traversed to the cumulative number of nodes covered is in general small. This is critical for situations in which testing resources are limited, as it allows high coverage to be achieved with the execution of fewer test sequences.

Our next step is to identify an appropriate constraint solver to reduce the human intervention in determining the test sequence feasibility. Furthermore, we intend to expand the set of coverage criteria available to guide the test generation process. Since our EFSM model retains information about variable definitions and uses within the SDL specification, the application of dataflow coverage criteria will be explored. Also, new coverage criteria focused on message communications between different EFSMs are to be developed. Finally, while the advantage of using a smaller set of test sequences is obvious in terms of test management, output verification, etc., the fault detection effectiveness of our approach has yet to be determined. A future study will apply our test generation method to the testing environment used by the engineers at Motorola and assess its suitability for locating defects in their SDL specifications.

## References

- [1] H. Agrawal, Dominators, super blocks, and program coverage, in: Proceedings of the 21st Symposium on Principles of Programming Languages, 1994, pp. 25–34.
- [2] G.v. Bochmann, A. Petrenko, O. Bellal, S. Maguiraga, Automating the process of test derivation from SDL specifications, in: Proceedings of the 8th SDL Forum, 1997, pp. 261–276.
- [3] C. Bourhfir, E. Aboulhamid, R. Dssouli, N. Rico, A test case generation approach for conformance testing of SDL systems, *Computer Communications* 24 (3–4) (2001) 319–333.
- [4] L. Bromstrup, D. Hogrefe, TESDL: experience with generating test cases from SDL specifications, in: Proceedings of the 3rd SDL Forum, 1989, pp. 267–279.
- [5] T.S. Chow, Testing software design modeled by finite state machines, *IEEE Transactions on Software Engineering* SE-4 (3) (1978) 178–187.
- [6] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, *IEEE Transactions on Software Engineering* 17 (6) (1991) 591–603.
- [7] E.R. Ganser, S.C. North, An open graph visualization system and its applications to software engineering, *Software – Practice and Experience* 30 (11) (2000) 1203–1233.
- [8] S.D. Gouraud, AuGuStE: A Tool for Statistical Testing Experimental Results, Rapport de Recherche No. 1400, CNRS and Université Paris XI, 2005.
- [9] Graphviz – Graph Visualization Software from AT&T, <<http://www.graphviz.org/>>.
- [10] D. Lee, M. Yannakakis, Testing finite-state machines: state identification and verification, *IEEE Transactions on Computers* 43 (3) (1994) 306–320.
- [11] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines – a survey, *Proceedings of the IEEE* 84 (8) (1996) 1090–1123.
- [12] G. Luo, A. Das, G. Bochmann, Software test based on SDL specification with save, *IEEE Transactions on Software Engineering* 20 (1) (1994) 72–87.
- [13] F. Kristoffersen, L. Verhaard, M. Zeeberg, Test derivation for SDL based on ACTs, in: Proceedings of the FORTE'92, 1992, pp. 373–388.
- [14] N. Maloku, M. Frey-Pucko, SDL-based feasible test generation for communication protocols, in: Proceedings of the International Conference on Trends in Communication, 2001, pp. 536–539.
- [15] T.J. Parr, R.W. Quong, ANTLR: a predicated-LL(k) parser generator, *Software – Practice and Experience* 25 (7) (1995) 789–810.
- [16] A. Petrenko, N. Yevtushenko, Conformance tests as checking experiments for partial nondeterministic FSM, in: Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software, 2005.
- [17] B. Sarikaya, G.v. Bochmann, Obtaining normal form specifications for protocols, in: Proceedings of the Computer Network Usage: Recent Experiences, 1985, pp. 601–613.
- [18] SDL Forum Society, <<http://www.sdl-forum.org/>>.
- [19] Q.M. Tan, A. Petrenko, G.v. Bochmann, A test generation tool for specifications in the form of state machines, in: Proceedings of the International Communications Conference, 1996, pp. 225–229.
- [20] Telelogic Tau, Telelogic Inc., Malmö, Sweden, 2007.
- [21] H. Ural, K. Saleh, A. Williams, Test generation based on control and data dependencies within system specifications in SDL, *Computer Communications* 23 (7) (2000) 609–627.
- [22] W.E. Wong, K. Cooper, AGES: automatic generation of EFSMs from SDL specifications, in: Proceedings of the 10th ISSAT International Conference on Reliability and Quality in Design, 2004.
- [23] W.E. Wong, Y. Lei, Reachability graph-based test sequence generation for concurrent programs, *International Journal of Software Engineering and Knowledge Engineering* 18 (6) (2008) 803–822.
- [24] W.E. Wong, T. Sugeta, J.J. Li, J. Maldonado, Coverage testing software architectural design in SDL, *Journal of Computer Networks* 42 (3) (2003) 359–374.
- [25] J. Zhang, X. Wang, A constraint solver and its application to path feasibility analysis, *International Journal of Software Engineering and Knowledge Engineering* 11 (2) (2001) 139–156.