# A Test Generation Tool for Specifications in the Form of State Machines[1]

Q. M. Tan, A. Petrenko and G. v. Bochmann
Department d'IRO, Université de Montreál
C.P. 6128, Succ. Centre-Ville, Montreál, P.Q. H3C 3J7, Canada
E-mail:tanq@iro.umontreal.ca Fax:(514)343-5834

**ABSTRACT:** This paper describes a software tool, TAG (Test Automatic Generation), that automatically generates test cases for an FSM specification. It implements the so-called transition identification approach for test derivation, and may output test cases in the form of an SDL skeleton. The description focuses on the functions of the tool and the methods implemented in the tool, especially, the heuristic solution to the minimization of state identification sequences.

## 1 Introduction

The finite state machines (FSMs) have been an important model in certain high integrity software developments; especially, it has been extensively used in the testing phases of system developments, for example, conformance testing of communication protocols.

Communication protocols are the rules that govern the communication between the different components within a distributed system. In order to organize the complexity of these rules, they are usually partitioned into a hierarchy of several layers, as exemplified by the 7-layer OSI model. Several formal description techniques have also been established for formal specifications of protocols, amongst which SDL [1] and Estelle [6] are based on the FSM model. A protocol specification generally can lead to several implementations in software and/or hardware, and it is very important to test an implementation against the specification in order to assure the compatibility with other implementations of the same protocol. This is called protocol conformance testing.

Quite a number of methods have been proposed in the literature for generating tests from specifications given in the form of FSMs, and several of them have been implemented as automated software tools [7, 13, 5, 15, 17, 19, 8, 14, 16, 11, 12]. Most of the methods not only check each transition in an FSM specification at least once, which corresponds to the branch coverage criteria often used in software testing, but also verify the tail state of the transition to obtain high fault coverage and to guarantee conformance in the context of a more general fault model. The tail states are generally verified by the state identification techniques [7, 15, 19, 8, 14, 11, 12]. Without state identification, test cases do not guarantee to detect the fault that the machine enters a different state than specified.

Most of the existing protocols are not completely specified, in the sense that in real communication systems, not all the sequences of interactions are foreseen. However,

[1]This work was largely funded by EICON Technologies in the context of a CANARIE project.

most of the existing test derivation methods for protocols, especially the existing tools, are limited to completely specified specifications. Even though one may impose some implicit definition for "undefined" transitions in a partially specified FSM, in many cases "undefined" means "don't care" or "forbidden" [4], and it is not necessary or not feasible to transform a partially specified FSM to a completely specified FSM for test derivation. Based on the method given in [12], we developed a software tool TAG (Test Automatic Generation) working directly for deterministic, partially specified FSM specifications.

Using this tool, a complete test suite that guarantees full fault coverage, or a set of test cases that cover a given test purpose, may be derived. The test output may be in TTCN-like mnemonic format or in the form of an SDL skeleton. The inclusion of the state identification in test cases is optional. A simple and readable language is supplied to describe an FSM specification, as explained in Section 2. A heuristic method for obtaining minimal state identification sequences has been deveopled and is described in Section 3. This method reduces the size of the obtained test suite. In Section 4, we discuss the results of some experiments and applications of our tool.

## 2 Using the tool

### 2.1 An Example

TAG implements the so-called transition identification approach for test derivation from an FSM. In particular, to achieve a particular test purpose which is a certain transition to be tested, the following steps have to be performed:

- bring the FSM from its initial state to the starting state of the transition under test using the shortest input sequence possible (called a preamble of the test case);
- execute the transition and check the observed output;
- check a tail state of the transition by observing its reaction to a pre-selected set of state identification sequences, which can verify the correctness of the tail state (a test body to achieve the test purpose);
- apply an input sequence to return to the initial state of the FSM (a postamble of the test case). The user may specify a so-called homing sequence which is expected to take the FSM from any state back to the initial state.

The set of all preambles is called a state cover; the set of sequences used to execute all specified transitions is called a transition cover.

State identification sequences are input sequences which distinguish states by their output reactions. Some FSMs may have undistinguishable states for which there exists no sequence which tells them apart. If this is the case for the given FSM then the tool still produces test cases, however, certain transfer faults in implementations might not be detected. A reduced or minimal machine [12] has no undistinguishable states, and the tool produces a test suite for such a machine with the guaranteed coverage of all output and transfer faults within the specified number of states.

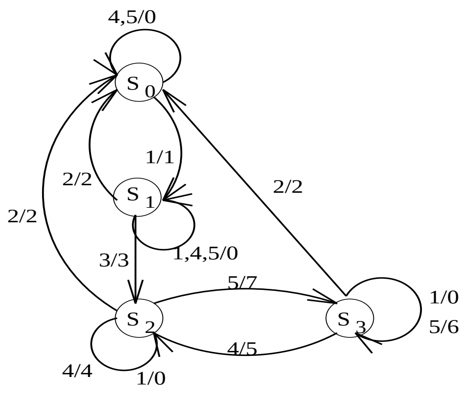

Figure 1: The INRES responder

The tool implements the so-called HSI method [12, 14] which is similar to the widely used W-method [7] in which a characterization set is used for state identification. However, the HSI-method uses a tuple of subsets of a W set for the identification of each state and can be applied to partially specified FSMs.

To illustrate the process of test derivation based on this method, consider the INRES protocol [9]. This simple protocol has been widely used in a number of publications, so we omit here its detailed explanation. The behavior of the responder part of this protocol can be specified by an FSM shown Figure 1. The input alphabet is : 1- CR; 2- IDISreq; 3- ICONrsp; 4- DT0; 5- DT1. The output alphabet is 0- NULL; 1- ICONind, 2- DR; 3- CC; 4- ACK0; 5- ACK0,IDATind; 6- ACK1; 7-ACK1,IDATind. The states are: $S_0$-Closed (the initial state); $S_1$- Opening; $S_2$- Waiting_DT0; $S_3$- Waiting_DT1. The FSM is deterministic and partially specified.

The final test suite and intermediate results are shown in the following table. Note that no postambles are included in the test cases of the following test suite.

| State | $S_0$ | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|
| State Identifier | 41 | 41 | 4 | 4 |
| Preamble (State cover) | $\varepsilon$ | 1 | 13 | 135 |
| Transition Cover | 1,4,5, | 11,12,13,14,15 | 131,132,134,135 | 1351,1352,1354,1355 |
| Postamble | $\varepsilon$ | 2 | 2 | 2 |
| Test Suite | { 41, 441, 541, 1241, 13241, 135241, 141, 1141, 1441, 1541, 134, 1314, 13544, 13514, 13554 } | | | |

The tool supports the following two modes of test derivation:

1) Complete test derivation, when a test suite has to test all transitions specified in the given FSM.

2) Selective test derivation, when a test case has to test a single transition given as a test purpose.

For selective test derivation, the test purpose is given by a state and input. For example, in order to test the transition starting at $S_1$ and ending at $S_2$ with input 3 and output 3, state $S_1$ and input 3 are given as the test purpose. Test case 134 will be derived, where the first 1 is preamble, the second 3 is to test the given transition, and the last 4 is used to identify the state $S_2$. For this test case a postamble 2 is also given to lead the FSM back to the initial state.

## 2.2 Functions Provided by the Tool

Before starting the tool, the user must have a text file containing the FSM specification with suffix ".fsm". This file can be produced by transforming an SDL specification through the tool FEX [2], or directly by using a text editor. The FSM specification is required to be deterministic and initially connected, but it may be partially or completely specified.

First the text file is loaded and compiled. Two files containing the symbol table (with suffix ".tbl") and the FSM structure (with suffix ".cpl") are created after compiling. Then the FSM specification is analysed, the tool displays the related information, such as whether or not it is initially connected, and whether or not it has undistinguishable states, equivalent states, etc.

If there are non-deterministic transitions in a certain state in a given specification, one among these non-deterministic transitions is kept in the compiled FSM and the others are ignored. Test derivation for an FSM with undistinguishable states is also possible, though some faults in these states might not be detected. In these two cases the tool will prompt a warning message.

One may choose *complete* or *selective* test derivation. In selective test derivation, one must first give the test purpose, which specifies one transition in the FSM. One test case is produced for the test purpose, and consists of a preamble, a transition under test, an optional state identification sequence and a postamble.

When using complete test derivation, a complete test suite is generated, and each test case consists of an integrated part composed of a preamble, a transition under test and an optinal state identification, and a postamble part. No test purpose is given.

The test output is written in a text file in one of the following two formats.

- Mnemonic format (with suffix ".mnc")
- SDL skeleton (with suffix ".sdl")

The state table of the complied FSM specification and intermediate results of test derivation, including preambles, state identification sequences and postambles, can be displayed in a numeric form. The mapping between mnemonic names in the specification and numeric codes in the state table is also displayed.

A test case in the form of an SDL skeleton for the INRES responder is given in Appendix 2. The test case is used to verify that the responder accepts a service ICONrsp (3) in the Opening state ($S_1$).

## 2.3 Required Input: the FSM specification

In order to derive test cases, the tool requires the user to prepare previously a script file which defines an FSM specification by giving the names of states, inputs and outputs as well as the transitions.

An FSM description consists of six parts: (1) the state definitions, (2) the input definitions, (3) the output definitions, (4) the transition definitions, (5) the variable declaration and (6) the homing sequence definition; parts (5) and (6) are optional. At the end of a script a keyword "end;" should be put. The following is a script file for the INRES responder.

```
/* This an FSM script for the INRES responder */
Variables:
 v Integer;
STATES:
 Closed;
 Opening;
 Wait_DT0;
 Wait_DT1;
INPUTS:
 CR :PDU;
 IDISreq;
 ICONrsp;
 DT(v=0) :PDU;
 DT(v=1) :PDU;
OUTPUTS:
 ICONind;
 DR :PDU;
 CC :PDU;
 ACK(v=0) :PDU
 ACK(v=1) :PDU;
 IDATind;
 comb1:  ACK(v=0):PDU,IDATind;
 comb2:  ACK(v=1):PDU,IDATind;
TRANSTIONS:
 Closed ?CR !ICONind >Opening;
 Closed ?DT(v=0) !NULL >Closed;
 Closed ?DT(v=1) !NULL >Closed;
 Opening ?ICONrsp !CC >Wait_DT0;
 Opening ?IDISreq !DR >Closed;
 Opening ?CR !NULL >Opening;
 Opening ?DT(v=0) !NULL >Opening;
 Opening ?DT(v=1) !NULL >Opening;
 Wait_DT0 ?IDISreq !DR >Closed;
 Wait_DT0 ?DT(v=0) !ACK(v=0) >Wait_DT0;
 Wait_DT0 ?DT(v=1) !comb2 >Wait_DT1;
 Wait_DT0 ?CR !NULL >Wait_DT0;
 Wait_DT1 ?IDISreq !DR >Closed;
 Wait_DT1 ?DT(v=0) !comb1 >Wait_DT0;
 Wait_DT1 ?DT(v=1) !ACK(v=1) >Wait_DT1;
 Wait_DT1 ?CR !NULL >Wait_DT1;
End;
```

The state definitions are a list of state names; the input definitions are a list of input names; and the output definitions are a list of output names. A short name, such as “S0”, “X1”,“idle”,“Y5”, etc., may be put in the front of a name and separated by a ‘:’. In the above script, “comb1” and “comb2” are two short names. The shortnames are used in the transition definitions and test outputs as a short hand notation.

Two different inputs or outputs may have the same name but are distinguished by different parameters. For example in the case of X.25, a n_connect_ack.rsp call with the parameter 1 accepts an incoming connection, while with the parameter 0, it refuses the connection request. The two input names may be defined as:

n_connect_ack.rsp(RC=1);
n_connect_ack.rsp(RC=0);

where RC is an integer variable, which may take different values. A list of parameters separated by ‘,’ may occur in an input or output. The variables in parameters may be the names which are declared by the variable definition or any C expression.

The key word “pdu” is used to indicate that an input or output is a protocol data unit (PDU). Without this tag, the input or output is supposed to be an abstract service primitive.

Several outputs may be combined into a single output. For such a combined output, a short name is necessary. In the test output that is in the form of a SDL skeleton, the service primitives and PDUs in a combined output may occur in arbitrary oder, since they occurs at different interaction points, For example, for output `ACK(v=0):PDU,IDATind`, it is not determined whether PDU `ACK` or service primitive `IDATind` occurs first. However, if output is `OUT1,OUT2,OUT3:PDU`, then `OUT2` must follow `OUT1`, because these primitives occur at the same interaction point.

The transition definitions define the FSM state table itself by a list of transition specifications. Each transition specification consists of a current state name, input name, output name and next state name, introduced by ‘?’, ‘!’ and ‘¿’, respectively. These names may be a short name or a full name as defined earlier. Every transition may be followed by a set of the comments related to the transition, included between ‘’ and ‘’. The coments will be inserted in the corresponding places in a derived test case in the form of an SDL skeleton for the purpose of helping the user to select appropriate parameters.

The variable definitions define the variables in parameters. The variable types are “Integer”, “Charstring”, “Octetstring” and “Boolean”. The charstring type is any sequence of printable ASCII characters enclosed between ‘"’, such as "*abc*"; the octetstring type is any sequence of ASCII codes enclosed between ‘′’, such as ′\23\34\128′; the boolean type is “true” and “false”.

The user may use keyword “homing” to give a sequence of input names as a homing sequence of the FSM specification, that is, it leads the FSM from any state to the initial state. The names in the homing sequence may be undefined input names. The principle is that TAG adds a postamble in a test case, if there is a postamble for a tail state, the postamble is used; otherwise, if the homing sequence is given, the homing sequence is used. If there is no postamble and no homing sequence is given, no postamble is included in the test case.

The formal grammar of the FSM script language is given in Appendix 1.

## 3 Test Derivation Methods

### 3.1 Preambles, Postambles and Transition Cover

To obtain preambles, a tree with the initial state as its root is constructed such that the tail states of the outgoing transitions from the state corresponding to a current node, if they have not become nodes of the tree, are added to the tree as sons of the current node. All nodes in this tree must become a current node once and only once in the order that they enter this tree. If there exist states that are not in this tree, then the FSM is not initially connected; the path from the root to a given node is a preamble for the corresponding state.

A similar procedure is also used to obtain a postamble from a given state, in which a tree with this state as its root is constructed. Once the initial state has been added to the tree, the procedure stops, and the path from the root to the last added node is a postamble from the given state. If all the nodes in this tree are tried but the initial state is not added, then there is no postamble from the given state.

For complete test derivation, for each state, the transition cover can be obtained by appending each outgoing transition from this state to its preamble. For selective test derivation, a transition is given by the user as the test purpose.

### 3.2 State Identification Sequences

Given a deterministic, partial specified FSM specification $\mathsf{S}$, the *characterization set* [7, 12] $W$ is a set of input sequences such that for any two distinguishable states of $\mathsf{S}$, there exists a sequence in $W$ such that it can be accepted by both these two states and produce different outputs. It is usually not necessary to use the whole $W$ for state identification; subsets of this set, called *harmonized state identification sets* [12], can used for state identification.

The harmonized state identification sets (HSI sets) are a tuple of sets $\{D_0, D_1, \ldots, D_{n-1}\}$, where $D_i$ is a set of prefixes of sequences in $W$ and $n$ is the number of states of $\mathsf{S}$. For any two distinguishable states $S_i$ and $S_j$ of $\mathsf{S}$, there exists a sequence $\sigma$ that is a prefix of both $\sigma_i \in D_i$ and $\sigma_j \in D_j$ such that $\sigma$ can be accepted by these two states and produces different outputs.

By adapting the FSM minimization algorithm given in [10], we can obtain a characterization set for $\mathsf{S}$ in polynomial time. Further from this set, according to the above definition, we can easily obtain HSI sets for $\mathsf{S}$ in polynomial time.

However, the state identification sequences obtained in the above way are not optimal. For example, a characterization set $\{4, 1\}$ can be obtained for the specification given in Figure 1, and further from this set, the HSI sets are $\{\{4, 1\}, \{4, 1\}, \{4, 1\}, \{4, 1\}\}$. Obviously, if the sequences are used for state identification, by comparison with the example in Section 1, a test suite of nearby the double size will be derived.

For an FSM specification, we say that a characterization set $W$ is optimal if, first of all, the number of sequences in $W$ is minimal, and secondly the sum of their lengths is minimal. Similarly, we say that a tuple of HSI sets is optimal if for the union of HSI sets the number of sequences is minimal and secondly the sum of their lengths is minimal. (Note that the union of all HSI sets for an FSM is also a characterization for this FSM.)

To obtain a minimal characterization set for a given FSM, as well as minimal harmonized state identification sets, may be an NP-hard problem [3]. A heuristic solution to the minimization of a characterization set is tried by transforming this problem to the so-called *set cover* problem in [3]. However, this method requires that a characterization set $W$ is given previously and only if there exists an optimal characterization set which is contained in $W$, can a near optimal solution be obtained. For example, if the characterization set $\{4, 1\}$ for the FSM in Figure 1 is given, no better solution but this set itself can be obtained using the method in [3]. However, as we have seen, $\{41\}$ is a better characterization set for the FSM in Figure 1 because it has only one sequence while the set $\{4, 1\}$ has two sequences. It is not easy to determine whether a given characterization set contains an optimal one for a given FSM unless this set contains all sequences of length less than the number of states.

The tool TAG uses also another heuristic solution in the following three steps to attempt to obtain minimal HSI sets. In the first step $A_1$, a set $P_0$ of all the state pairs that are distinguishable for the given FSM is produced. In the second step $A_2$, with $P_0$, a characterization set $W$ is obtained by forming a search tree from the input alphabet to distinguish the state pairs in $P_0$, such that an optimal solution could be contained in $W$ with great possibility. In the third step $A_3$, from this $W$, some HSI set $D_i$ is selected for each state $i$, such that the number of distinguishable state pairs and the length of a sequence are traded off.

The algorithm $A_1$ is obtained by adapting the FSM minimization algorithm given in [10]. This algorithm is also used in the FSM script compilation to tell the user which states are indistinguishable and equivalent.

The algorithm $A_2$ first forms the root as the current node to probe. In probing the current node $t_i$, for each input word, a son is built if it can lead to a better solution than other built nodes. If it is estimated that a subtree with a minimal number of branches and with a minimal average lenghth that distinguishes a maximal number of state pairs could be formed by probing an unprobed son of $t_i$, then this son is selected for a recurrent probe. The selection and probe continue until the state pairs that are to be distinguished by probing $t_i$ has been covered or there are no unprobed sons for $t_i$. The tree built by the algorithm forms the resulting $W$.

Note that the goal of this algorithm is to find a characterization set $W$ that contains optimal HSI sets. Thus the probe of the current node is not affected by any previous probes. (Note that the sequences returned by a previous probe may distinguish some state pairs that are to be distinguished by probing the current node.) This strategy will lead to a bigger $W$, but the probability that an optimal solution is contained is increased, too.

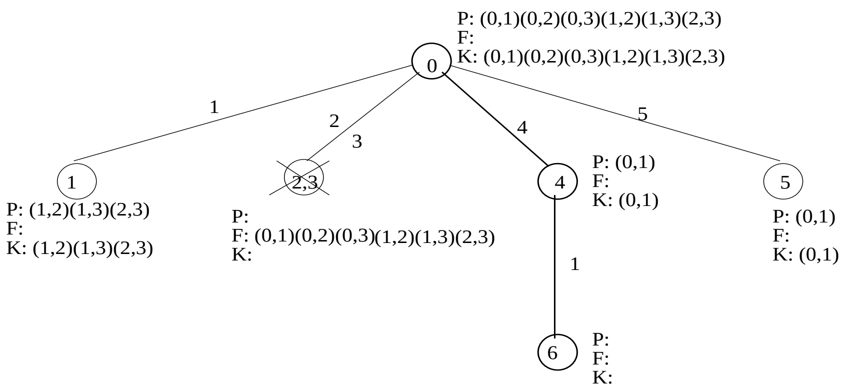
P: (0,1)(0,2)(0,3)(1,2)(1,3)(2,3)
F:
K: (0,1)(0,2)(0,3)(1,2)(1,3)(2,3)
0
1
2
3
4
5
1
2,3
4
5
6
P: (1,2)(1,3)(2,3)
F:
K: (1,2)(1,3)(2,3)
P:
F: (0,1)(0,2)(0,3)(1,2)(1,3)(2,3)
K:
P: (0,1)
F:
K: (0,1)
P: (0,1)
F:
K: (0,1)
P:
F:
K:

Figure 2: Apply the algorithm $A_2$ to the FSM in Figure 1

The algorithm $A_3$ selects the HSI set $D_i$ for each state $i$ of the given FSM from the $W$ obtained in the previous step. The algorithm starts with $P_0$ as the set of remaining state pairs. First a sequence $\sigma_k$ in $W$ is selected such that the weighted sum of its length and the number of the state pairs that it can not distinguish in the left state pairs is minimal. After $\sigma_k$ is chosen, for each remaining state pair $(l, m)$ that can be distinguished by $\sigma_k$, find a prefix of $\sigma_k$ such that $(l, m)$ is distinguished. Then the prefix is put into the HSI sets $D_l$ and $D_m$. The above procedure is repeated until no state pairs remain.

For the FSM in Figure 1, the result $P_0$ of the algorithm $A_1$ is the set of all possible state pairs. The characterization set $W$ from the algorithm $A_2$ is $\{41\}$. From this set, the HSI sets $\{\{41\}, \{41\}, \{4\}, \{4\}\}$ are produced by applying the algorithm $A_3$. The Figure 2 shows the search tree that is formed by applying the algorithm $A_2$ to the INRES specification given in Figure 1. In the figure, $P_i$ is the set of the given state pairs that have not been distinguished from the root to the node $i$, $F_i$ is the set of the state pairs that can not be distinguished from the root to $i$, and $K_i$ is the set of the current state pairs that are to be distinguished by probing $i$.

## 4 Experiments and Applications

Near 50 typical or randomly generated FSMs with a number of states between 4–8 have been tried by the tool. We find that the optimal harmonized state identification sets can be derived by the tool from about 87% of the FSMs, and the number of sequences in the union of harmonized state identification sets is an average 1.4 less than that of the characterization set obtained by applying the adapted FSM minimization algorithm given in [10], where no sequences that are prefixes of other sequences are computed.

The following table shows the CPU times taken by the tool for the derivation of state identification sequences for several FSMs of different sizes. In the table, the CUP time is an average of the CPU times for 10 FSMs of the same size that are produced randomly. The FSMs have the number of inputs equal to the number of states and the number of outputs equal to half the number of states. The experiment is done on a SUN SPARC Station 2.

| State number | 10 | 25 | 50 | 75 | 100 | 150 | 200 |
|---|---|---|---|---|---|---|---|
| Transition number | 100 | 625 | 2500 | 5625 | 10000 | 23500 | 40000 |
| CPU time (sec.) | 0.11 | 0.40 | 2.96 | 11.73 | 40.87 | 197.06 | 760.89 |

The above table shows that the CPU time grows polynomially with the size of the FSM.

The tool was also used to derive test cases for an X.25 packet level protocol, which is specified as an FSM with 18 states, 28 inputs, 32 outputs and total 471 transitions. The number of test cases in the complete test suite derived by the tool is 687, an average of 1.4 test cases per transition.

The tool was also used in an experiment on optimal testability design [20]. This experiment tries to find an optimal test suite for a given partially specified FSM through arbitrarily augmenting the "don't care" transitions (total 1,185,192 possibilities).

## 5 Conclusion

In this paper, we have presented the functions of the test generation tool TAG and the methods used in this tool. In particular, we have proposed a heuristic solution to derive near-optimal harmonized state identification sets as well as characterization set from an FSM. The performance of the tool is demonstrated by several experiments and applications.

The tool only requires that the given FSM is deterministic and initially connected. If there are undistinguishable states in the FSM, test derivation is still possible. In this case, certain transition faults in implementations might not be detected by the derived test cases. Complete or selective test derivation is provided, and the inclusion of the state identification sequences in test cases is optional. Test cases may be output in mnemonic format or in the form of an SDL skeleton.

The test cases produced by the tool do not include test parameters. The test parameters should be added by the user. A tool for test parameter generation and selection has also been implemented to assist the addition of the test parameters [2].

Other applications using the tool are the derivation of test cases for labeled transition systems through transforming them into corresponding trace FSMs [18] and the derivation of test cases for non-deterministic FSMs [11]. We have noted that a non-deterministic FSM can be viewed as an LTS in a sense that a pair of input and output of a transition in the FSM is treated as a label, and moreover any LTS can be transformed into a deterministic trace FSM for testing trace-equivalence [18]. Thus the test suite for the trace FSM could lead to a test suite for the given non-deterministic LTS.

## References

[1] F. Belina and D. Hogrefe. The CCITT–specification and description language SDL. *Computer Networks and ISDN Systems*, 16(4):331–341, 1989.

[2] O Bellal, G. v. Bochmann, A. Petrenko, and Q. M. Tan. Tool chain for test case derivation from sdl specification. Technical report, Dept. of I.R.O., University of Montreal – Hewlett Packard PTC Project, 1995.

[3] P. J. Bernhard. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology*, 3(3):201–220, 1994.

[4] G. v. Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *Proceeding of the ACM 1994 International Symposium on Software Testing and Analysis*, pages 109–124, Seattle USA, 1994.

[5] G. v. Bochmann and B. Sarikaya. Some experience with sequence generation for protocols. In *IFIP Protocol Specification, Testing, and Verification II*. North-Holland, 1982.

[6] S. Budkowski and P. Dembinski. Introduction to ESTELLE: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[7] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[8] S. Fujiwara et al. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, SE-17(6):591–603, 1991.

[9] D Hogrefe. OSI formal specification case study: the inres protocol and service. Technical report, University of Berne, 1991.

[10] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, New Jersey, 1991.

[11] G. Luo, G. v. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, SE-20(2):149–162, 1994.

[12] G. Luo, A. Petrenko, and G. v. Bochmann. Selecting test sequences for partially-specified nondeterministic finite machines. In *IFIP 7th International Workshop on Protocol Test Systems*, pages 91–106, Japan, 1994.

[13] S. Naito and M. Tsunonyama. Fault–detection for sequential machines by transitions tours. In *Proceedinds of IEEE Fault Tolerant Computing Conference*, pages 238–243, 1981.

[14] A. Petrenko. Checking experiments with protocol machines. In *IFIP 4th International Workshop on Protocol Test Systems*, pages 83–94. North-Holland, 1991.

[15] K. Sabnani and A. T. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[16] Y. N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple UIO sequences. *IEEE Transactions on Communications*, 40(8):1282–1293, 1992.

[17] D. P. Sidhu and T. K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, SE-15(4):413–426, 1989.

[18] Q. M. Tan, A. Petrenko, and G. v. Bochmann. Modeling basic LOTOS by FSMs for conformance testing. In *IFIP Protocol Specification, Testing, and Verification XIIII*, Poland, 1995.

[19] S. T. Vuong and et al. The UIOv-method for protocol test sequence generation. In *IFIP 2th International Workshop on Protocol Test Systems*, pages 203–225. North-Holland, 1990.

[20] N. Yevtushenko, A. Petrenko, R. Dssouli, K. Karoui, and S. Prokopenko. On the design for testability of communication protocols. Technical Report 971, Dept.of I.R.O., University of Montreal, 1995.

## Appendix 1: The formal grammar of an FSM specification

```
<FSM_script>        ::=  [<variable_defs>]<state_defs><input_defs><output_defs>
                         [<homing_def>]<transition_defs> end';'
<variable_defs>     ::=  variables':'<variable_def>';'{<variable_def>';'}
<state_defs>        ::=  states':'<state_def>';'{<state_def>';'}
<input_defs>        ::=  inputs':'<input_def>';'{<input_def>';'}
<output_defs>       ::=  outputs':'<output_def>';'{<output_def>';'}
<transition_defs>   ::=  transitions':'<transition_def>';'{<transition_def>';'}
<homing_def>        ::=  <input_name>{','<input_name>}';'
<variable_def>      ::=  <variable_name> Integer|Charstring|Octetstring |Boolean';'
<state_def>         ::=  [<short_name>':']<state_name>';'
<input_def>         ::=  [<short_name>':']<input_name>';'
<output_def>        ::=  [<short_name>':']<output_name>';'|<short_name>':'
                         <output_names>';'
<transition_def>    ::=  s_name>'?'<i_name>'!'<o_name>'>'<s_name>';'
<state_name>        ::=  <identifier>
<input_name>        ::=  <identifier>['('<parameters>')']
<output_name>       ::=  <identifier>['('<parameters>')']
<output_names>      ::=  <output_name>','<output_name>{','<output_name>}
<short_name>        ::=  <identifier>
<s_name>            ::=  <short_name> | <state_name>
<i_name>            ::=  <short_name> | <input_name>
<o_name>            ::=  <short_name> | <output_name>
<parameters>        ::=  <parameter>{‘,‘<parameter>}
<parameter>         ::=  <v_name> | <v_name>'='<value>
<variable_name>     ::=  <identifier>
<v_name>            ::=  <variable_name> | <C expression>
<value>             ::=  <integer>|<charstring>|<octstring>|true|false
```

## Appendix 2: Example of test case in the form of an SDL skeleton

```
/* Verify that an IUT accept ICONrsp in state Opening */
PROCEDURE preamble_to_state_Opening;
START;
        TASK pdu!packet := CR;
        /* USER: fill the PDU parameters */
        OUTPUT l_data_out(pdu);
        NEXTSTATE wait_ICONind;
```

```
STATE wait_ICONind;
        INPUT ICONind(/* USER: fill parameters */)
        /* USER: check the input parameters */
        RETURN;
        INPUT *;
        MACRO fail('ICONind expected');
        RETURN;
ENDPROCEDURE preamble_to_state_Opening;
PROCEDURE transition_under_test_in_Opening_on_ICONrsp;
START;
        /* USER: fill the output parameters */
        OUTPUT ICONrsp(/* USER: fill parameters */);
        NEXTSTATE wait_CC;
STATE wait_CC;
        INPUT l_data_in(pdu);
        DECISION pdu!packet;
        (CC):   /* USER: check the PDU parameters */
                RETURN;
        ELSE:   MACRO fail('CC expected');
                RETURN;
        ENDDECISION;
        INPUT *;
        MACRO fail('l_data_in with CC expected');
        RETURN;
ENDPROCEDURE transition_under_test_in_Opening_on_ICONrsp;
PROCEDURE identifing_state_Wait_DT0;
START;
        TASK pdu!packet := DT;
        TASK pdu!v := 0;
        OUTPUT l_data_out(pdu);
        NEXTSTATE wait_ACK;
STATE wait_ACK;
        INPUT l_data_in(pdu);
        DECISION pdu!packet;
        (ACK):  /* USER: check the PDU parameters */
                RETURN;
        ELSE:   MACRO fail('ACK expected');
                RETURN;
        ENDDECISION;
        INPUT *;
        MACRO fail('l_data_in with ACK expected');
        RETURN;
ENDPROCEDURE identifing_state_Wait_DT0;
PROCEDURE postamble_from_state_Wait_DT0;
START;
        /* USER: fill the output parameters */
        OUTPUT IDISreq(/* USER: fill parameters */);
        NEXTSTATE wait_DR;
```

```
STATE wait_DR;
        INPUT *;
        RETURN;
ENDPROCEDURE postamble_from_state_Wait_DTO;
/* Test purpose: verify the transition in state Opening on input ICONrsp */
PROCEDURE TestCase_in_Opening_on_ICONrsp;
START;
        MACRO init_test_case;
        CALL preamble_to_state_Opening;
        MACRO exception;
        CALL transition_under_test_in_Opening_on_ICONrsp;
        MACRO exception;
        CALL identifing_state_Wait_DTO,
        MACRO exception;
        MACRO pass( );
        CALL postamble_from_state_Wait_DTO;
exception_label:
        RETURN;
ENDPROCEDURE TestCase_in_Opening_on_ICONrsp;
START;
        Call /* USER: initialize IUT */;
        MACRO exception;
        Call TestCase_in_Opening_on_ICONrsp;
exception_label:
        Call /* USER: clear up IUT */;
        STOP;
```