

Automated Generation of Protocol Test Sequences From Formal Specifications

G. M. Lundy and C. Basaran
Department of Computer Science
U. S. Naval Postgraduate School
Monterey, CA 93943

Abstract

A program which takes as input the formal specification of a protocol using the formal model *systems of communicating machines*, and outputs a sequence of tests for an implementation of the protocol is discussed.

The protocol is specified formally as a finite state machine with local and shared variables. The test program, called TESTGEN, finds all paths which may be taken through the FSM and generates a sequence of tests to check all these paths. Certain possible error conditions or difficult to test conditions are also detected by the program, and the test designer receives a warning message.

The program is applied to a formal specification of the CSMA/CD and FDDI protocols, generating a test sequence for both of these protocols.

1. Introduction

The formal specification and analysis of communication protocols has been much studied for the past

15 years [PSTV I-XII]. A number of models have been suggested for the specification, verification and testing of protocols, and some of these have been standardized. Some of these models seem better suited for specification than others; some seem better suited for protocol verification, and the testing research community has often used still other models or variations of these models.

Systems of communicating machines [LuMi, MiLu, LuBu] is one of the formal protocol models introduced during this time period, which we believe has potential to unite these three diverse but important areas of the protocol community. The model has been used to specify and verify several communication protocols. The analysis which is

carried out with the model, called system state analysis, has been automated. A test procedure for generating tests from a formal protocol specification was given in [MiLu].

In this paper, we report that a program has been written to automate the test generating procedure, which we call TESTGEN. When combined with the earlier work, we now have the capability to take a protocol, specify it formally as a system of communicating machines, analyze the specification using the program to generate the global and/or system state analysis; finally, to generate a set of "conformance tests" to insure that an implementation of the protocol is, to some degree at least, in conformance with its specification.

A conformance test is used to ensure that the external behavior of an implementation of a protocol is equivalent to its formal specification. The implementation, for practical purposes, is considered as a black box with a finite set of inputs and outputs. The test provides a sequence of input signals, and observes the resulting outputs.

A previous study [Mil] on this issue observed gaps between the specification, the verification, and the conformance testing of network protocols. Protocol models which are designed for specification purposes usually have many powerful program language constructs, to simplify the specification, but are difficult to analyze. Protocol models designed primarily for analysis purposes, such as the CFMS model, are often too simple for the specification of modern, complex protocols.

The automation of the test sequence generation is an attempt to close the gap between specification/verification and testing of protocols. In this paper, the test generation starts from a protocol model, designed for the specification and verification of protocols. A procedure, created in [MiLu], is used for the generation of a test sequence for a protocol specified in the SCM model. This procedure and its automation as a software tool does not guarantee that all the errors or combination of errors in a protocol are found. But they do represent an attempt to exercise all parts of the protocol, providing some assurance that the implementation meets its purpose.

In the next section, we review the test procedure for the SCM model which is automated in this paper. In Section 3, we discuss the program, called TESTGEN, its inputs and outputs. In the next section we discuss its application to the FDDI protocol, after which follows the conclusion.

2. A Procedure for Generating Test Sequences

In this section a procedure and its automation are described for generating a sequence of tests for a protocol specified as a SCM model. The input is the formal protocol specification (FSM and predicate-action table) specified as a *system of communicating machines*(SCM). The output is a sequence of tests and an I/O diagram in a tabular format. The generated sequence is intended to be applied to an IUT.

The sample IUT throughout this section is the network node for CSMA/CD protocol. The test inputs (the shared and local variables that can be set in a controlled way) and the outputs (the shared and local variables can be observed for test purposes) should be identified. These inputs and outputs form the I/O for the test steps.

he test steps.

The format for each single test is

$$S_I \ i_1, i_2, \dots, i_n ; o_1, o_2, \dots, o_m \ S_E$$

S_I is the state of machine when the test begins. The i_1, i_2, \dots, i_n are the values of the input variables at the start of test

execution. The o_1, o_2, \dots, o_m are the values of the output variables after test execution. S_E is the state of the machine when the test is complete. The input and the output variables are taken from the shared and local variables of the machine. The determination of these variables is explained in the following section.

The testing procedure explained below is written in three parts: 1) preliminary steps, 2) test sequence generation, and 3) refining steps.

Preliminary Steps

1. From the machine specification diagram, mark each transition whose name appears on more than one transition. Each such instance for a given name is given a separate distinguishing label.

2. From the *predicate-action table*, note the number of clauses in each enabling predicate. Mark each clause.

3. For each shared variable x , determine if x is an input variable, an output variable, or both. For each x which is both, split x into two variables, x_i and x_o for testing purposes.

4. For each local variable l , determine if l is used as an interface to the higher layer user of this protocol. If so mark l as input, output or both. If l is both input and output, split it into two variables l_i and l_o for test purposes.

Test Sequence Generating Procedure

Initially the test sequence is empty.

1. $state \leftarrow$ initial state.

2. Let $t = (p, a)$ be an untested transition from $state$.

(a) Determine the values of the input variables which make exactly one of the untested clauses of p true. Check to see if these values allow any other transition from this state to be executed. If there is one, set additional input variables to values that insure only the transition under test is enabled. Fill these in, and mark others "DC" for "don't care."

(b) Determine and mark the expected values for the output variables; also record the expected values assumed by the local variables.

TABLE 1: PREDICATE ACTION TABLE FOR NETWORK NODES

Transition	Predicate	Action
Xmit	$msg \neq \emptyset \wedge medium = \emptyset$	$medium := msg;$ $Signal(i) := transceive$
OK	$Signal(i) = clear$	$msg := \emptyset$
coll-D	$medium = undefined$	$Signal(i) := collision$
ready	$Signal(i) = clear$	
receive	$medium.DA$	$inbuf := medium;$ $Signal(i) := transceive$

(c) Set S_I to state; determine the next state and set S_E to it.

(d) Determine if S_E is transient; if not mark it as a “stop state” and skip to (3). The state is *transient* if one of its enabling predicates is true immediately upon reaching the state. This means that it can pass on to another state immediately, without waiting for further input.

(e) Attempt to make S_E into a stop state by setting “DC” values. That is, make the DC values such that, upon reaching state S_E , none of the enabling predicates are true. If successful, go to (3).

(f) If S_E is a transient state and more than one transition leaving S_E is enabled, choose one and set inputs not yet specified (if any exist), so that only one transition leaving S_E is enabled; set $t = (p, a)$ to this transition.

3. Output this test $S_I i_1, i_2, \dots, i_n / o_1, o_2, \dots, o_m S_E$ as the next test in the test sequence.

4. Mark the clause just tested. If all clauses in transition t are now tested, mark t as tested. If all transitions are now marked as tested, exit to “refining steps.” Otherwise, continue to step (5).

5. Set state to S_E . If state is a stop state go to (2), otherwise go to step2(b).

Refining Steps

1. Construct the I/O state diagram from the test sequence.

2. Determine if the sequence are unique, so that from each state, we have a unique input output (UIO) sequence to

confirm. If not attempt to extend the sequence so that we have a unique UIO sequence from each state.

3. Check for any converging transitions. Mark these, as potential problems for testing.

The I/O diagram can be constructed from the test sequence and is a tool to help the test designer insure completeness. This finite state machine is often used as the starting point in test generation in the literature.

A UIO sequence has been defined as a sequence of inputs such that, if the input sequence is applied to the FSM when FSM is in state i , the resulting output sequence could not have been produced by the FSM when the FSM is in any other state [DSKU][SiLe]. If the sequence of tests applied to a machine implementation in a state i is a UIO sequence, and the output is expected, then we have a stronger argument that the machine was, in fact, in state i .

Example Specification: The CSMA/CD Protocol

We now give an example specification which will be used to illustrate the test procedure and the program. The CSMA/CD (carrier sense multiple access with collision detection) protocol has a formal specification as a SCM model in [LuMi3].

The specification of CSMA/CD protocol consists of the finite state machine and the local variables of the network stations (Figure 1) and the *predicate action table* for the network stations (Table 1). The shared variables, *Medium* and *Signal* and finite state machine of the controller, responsible for the control of shared variables, are shown in Figure 2. The

```

xmit | msg /= empty and medium = empty | medium := msg ; signal(i) := transceive
ok | signal(i) = clear | msg := empty
coll-D | medium = unidentif | signal(i) := collision
ready | signal(i) = clear | no
receive | medium = (x,x,i) | inbuf := medium ; signal(i) := transceive

```

Figure 1 : Predicate-Action File Input of CSMA/CD Protocol

An example line in the predicate-action input file is shown in Figure 8.

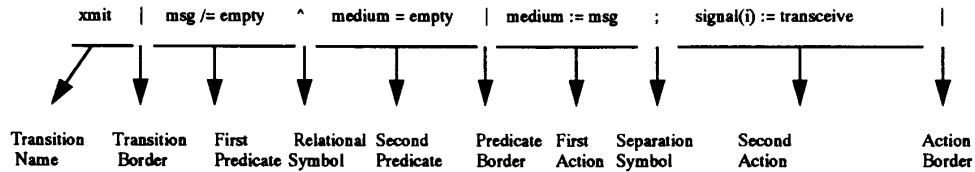


Figure 2 :Example Input line of Predicate-Action File:

predicate action table of controller is shown in Table 2. Further discussion of the protocol and its details are given in [LuMi3].

3. TESTGEN: Automatic generation of protocol test sequences

The software tool that automates the generation of test sequences is called "TESTGEN." The inputs of the program are two text files which are created and named by the user. One of these represents the FSM part of the specification , and the other represents the predicate-action table.

The input files are easily created utilizing the following procedures. Before creating the FSM input file, the user should assign a number to each transition of the FSM. This distinguishes each arc, even though they may represent the same transition name.

To create the first file, the user first specifies the initial state of the FSM as the first line in the FSM input file. Each line, thereafter, represents a transition arc and is entered in the format:

From State To State Number Assigned Transition Name

Transition arcs can be entered in any order as long as they have the previous structure.

An example FSM input file for the CSMA/CD protocol is shown in Figure 4. The "0" in the first line shows the initial state of our example CSMA/CD protocol. The second integer

is the "to state," the third integer represents the transition, and

this is followed by the transition name in text form.

```

1
0
0 1 1 receive
0 2 3 xmit
0 3 2 coll-D
1 0 4 ready
2 0 4 ok
2 3 5 coll-D
3 0 7 ready

```

Figure 4 : FSM Input File of CSMA/CD Protocol

The second input file contains *predicate action table(PAT)* of the specified protocol. This file is created in the same tabular format as the *predicate-action table*. Each column of the *PAT* is separated with vertical bar '|' with a space on each side, so that it is distinguishable from the other table entrees. The '|' delineates the borders of transition, predicate and action columns of the *PAT*. Multiple action statements should be separated with a semi-colon (;). If no action is to be taken for a transition, the keyword "no" must be entered as the action part of the input file. If a transition occurs every time we enter a state, it is indicated by putting keyword the "true" in the predicate part of the input file. An example of predicate-action input for the CSMA/CD protocol is shown in Figure 5.

If there is more than one clause in a disjunctive predicate part of a transition it is difficult to determine

which predicates need to be enabled to make a transition occur. The TESTGEN program is capable of parsing transitions in several different forms.

The TESTGEN program represents these relational clauses by putting the relational symbol between two clauses together with the values of the input variable to the output table. If the enabling predicate has more than three clauses the TESTGEN program may not correctly represent these clauses in the output test sequence. The user should control the output test sequence for these transitions.

If input variables are record structures such as *medium*, *msg*, *inbuf*; assignment or comparison of a specific fields of the record are done within parentheses and by putting "x" in the positions that is unimportant. For example, assume a variable "Z" is a record structure with three subparts a, b and c. Assignment of the value "3" to the 'a' field of Z should be in the format "Z:= (3,x,x)." This means 3 is assigned to 'a' and no changes are made to 'b' and 'c.'

The Test Sequence Generator

The algorithm of the test generator consists of two major subparts: the first part finds all possible paths and cycles in the FSM starting from the initial state. It prints the list of paths and cycles to a text output file, named by the user. It also ensures that there is a path from all cycles eventually returning to the start state. If unable to find such a path it will print out a message, warning the user of possible errors in the specification of the protocol. The pseudo-code algorithm for finding all paths and cycles of FSM is shown in [Bas]. It is based on a variation of a graph-search algorithm. Finding all possible transition sequences ensures that each instance of each transition is tested.

To trace all the possible paths which could be generated, a queue of linked lists is implemented. The trace is as follows: Starting with the initial state, all transitions are placed into the queue. The first entry is dequeued, becoming the current entry, and is used to continue the trace. The

current entry remains so until it describes a cycle back to the initial state.

All transitions out of the last node of the current path are determined, and one of them is appended to the current entry.

Any other transitions are each appended to a copy of the current path and placed at the end of the queue (*list_of_paths*). When the initial state is reached, next path in the queue becomes current path. This procedure continues until the queue is empty.

The program starts with an arc originating from the initial state. In our example CSMA/CD protocol the first arc selected is transition #1 (0 1 1 receive). It is inserted to the *list_of_paths*. Since there is more than one transition leaving the initial state, the other (0 2 3 transmit), (0 3 2 coll-D) arcs are also inserted to the *list_of_paths*. Then destination node "1" of transition #1 is found from the *list_of_transition* and since there is one transition (transition #4) leaving destination node; it is appended to the end of our path. Then transition #4 becomes current arc. Since the destination node of the transition #4 is 0 (initial state) the path is marked as processed. The current entry becomes the last arc in the next unprocessed transition sequence (transition #3). The procedure continues until all paths and cycles originating from the initial state are found. The steps of finding paths and final path list at the end of procedure *FIND_PATHS* for CSMA/CD protocol is shown in Figure 10.

Preliminaries

In our example many of our variables perform as both input and output sources. The shared variables *medium*, *Signal* and local variable *msg* are all input and output variables. The second part of the TESTGEN determines our input and output variables. If a variable is used as both an input and output variable it is marked by placing (*i*) or (*o*) next to them to indicate its current usage. The program reads the transitions, predicates and actions associated with each

Trans		input variables				output variables					
		name	S _i	medium(i)	msg(i)	signal(i)	**	inbuf	medium(o)	msg(o)	signal(o)
receive	0	(i,x,x)	DC	DC	DC	**	medium	--	--	transceive	1
ready	1	DC	DC	clear	DC	**	--	--	--	--	0
xmit	0	empty	/empty	DC	DC	**	--	msg	--	transceive	2
ok	2	DC	DC	clear	DC	**	--	--	empty	--	0
coll-D	0	undefined	DC	DC	DC	**	--	--	--	collision	3
ready	3	DC	DC	clear	DC	**	--	--	--	--	0
xmit	0	empty	/empty	DC	DC	**	--	msg	--	transceive	2
coll-D	2	undefined	DC	DC	DC	**	--	--	--	collision	3
ready	3	DC	DC	clear	DC	**	--	--	--	--	0

transition from the (predicate-action table) PAT. It then creates the test sequence table and lists all transition sequences starting from the initial state by using *list_of_paths*. It prints each transition with the expected values of any local and shared variables. It also prints the action to be taken if the predicate associated with transition is enabled.

Test Sequence Generation

The TESTGEN program begins with the first transition (#1 receive) in the path list generated by the FIND_PATHS procedure. According to the predicate action input file to enable this transition, the DA field of medium must be set to the station's address, which we assume to be i. The remaining fields of the record *medium* may be any values, and are indicated by 'x' in the output table (Figure 12). The other input variables are set to "don't care" or DC.

When the *receive* transition occurs, *signal(i)* should be set to *transceive*, and *inbuf* should contain the value which was previously in *medium*. *S_i* is set to *source state* of the current transition (in this case 0), and *S_E* to the *terminal state* (in this case 1). This completes the first test in the sequence and these values are output. The clause and transition are now marked "tested." The value of *S_i* is now set to 1, and next transition in the path is called.

The next iteration is the *ready* transition from state 1. The values selected are the second test in the output table (Figure 12). The ending state of this test is state 0 the initial state, so the path is marked as processed.

At the next iteration first transition in the next unprocessed path (*Xmit*) is chosen, followed by the *OK* transition back to state 0. The same process continues with transition *coll-D*, which takes the machine state 3, and the *ready* transition returns it to state 0. Then the *Xmit* transition is chosen a second time in the last path which takes the machine state 2; then transition *coll-D* is chosen; this takes the machine to state 3 and *ready* transition again returns it to the initial state.

The table generated by the TESTGEN program for the CSMA/CD protocol is shown in Figure 12. The table lists all nine possible transitions according to their order of occurrence. It is relatively easy to test all sequences of a transitions by simply following the order in the table.

Refinement

The first refining step calls for the construction of the I/O diagram. This diagram can be constructed from the sequence of tests generated. In this case, because there are no transient states, there are four states which correspond to the four states of the specification; and the arcs between states are the same set as in the specification. The only difference is in the labeling of the arcs; for the I/O diagram, the label on each arc is the set of values if the input and output variables, as shown in output table Figure 12.

Next we must determine if the sequence is a UIO sequence. Consider the first test in the table, the *receive* transition. If the machine is in state 0 and we apply the inputs for the first test, the outputs are the *transceive* value in *Signal(i)* and a copy of *medium* in *inbuf*. The user may

confirm that in no other state does this combination occur; so for the first state and test, we have an UIO sequence. From state 1, the *ready* transition is considered. This transition leads back to state 0; note that another *ready* transition leads from state 3 to state 0. This means that there is not a UIO sequence for states 1 and 3. This makes it difficult for the test designer to confirm these states. There is however a UIO sequence leading into these states; so the lack of a UIO sequence from these states is less disturbing.

Finally a check for converging transitions shows that there is one case of this: the *ready* transition, leading to state 0 from both states 1 and 3. The test designer must be aware of this, as a possible source of problems in the execution of tests.

4. Other Applications

TESTGEN was also applied to a formal specification of the FDDI protocol. The protocol was formally specified, including timing requirements, and verified, in [LuAk]. A detailed description of FDDI and the specification appears in [LuAk].

The protocol specification consists of the FSM description of each machine, the predicate-action table and the timer specifications. Each machine shares one variable with its upstream neighbor (called *inbuf*) and one with its downstream neighbor (called *outbuf*). (These shared variables serve as the input and output ring connections). The FSM consists of 20 states. The transition names on the transition arcs serve as a key into the PAT, which specifies the action taken when the transition is executed.

After receiving the input files, the TESTGEN program prints out all the paths in the protocol, finds all the cycles and checks them for a transition that will ultimately lead back to the initial state. The paths are depicted according to the numbers assigned by the user.

In the FDDI example, the number of paths found by the TESTGEN program is 162. There are no cycles without an outgoing transition that leads back to the initial state.

Finally, the TESTGEN program creates the testing sequence table by printing all possible transition sequences, excluding continuous cycles. The output table is 2112 lines long. Since the table generated for the FDDI protocol is much too big to show here, it is partially depicted in ref. [Bas]. Each of these 2112 output lines corresponds to a single test. The width of the table corresponds to the number of input and output variables. The input variables must be set to the values shown on the left side of the table, and the output variables are expected to take on the values shown on the right side.

The TESTGEN program can determine some conditions which make a state transient. A state is *transient* if, upon reaching that state, at least one enabling predicate for one outgoing transition is true. This means that the protocol machine may immediately pass on to another state, so that the protocol test engineer may not be able to confirm that the values of the output variables are as expected. In the FDDI example, 4 transient states are detected.

Finally, TESTGEN also detects converging transitions and prints out the list of these. A *converging transition* is one which originates from two different states, and terminates in the same state. This makes it difficult to confirm that the protocol followed the correct sequence of states. In the FDDI protocol, TESTGEN detected two converging transitions.

5. Conclusions

This paper has introduced a software tool called TESTGEN which produces a sequence of conformance tests for a protocol specified formally as a *system of communicating machines*.

The TESTGEN program takes as input a protocol specified formally as two separate text files, one containing the finite state machine part, the other containing the predicate-action table and variables. It outputs test sequences beginning from the initial state, finding all transition sequences, and generates tests for every transition on the path

back to the initial state, so long as there is such a path (when there is no path back, the user is warned).

The first purpose of the program has been to make it possible for implementors and buyers/users of protocol implementations to automatically generate a set of tests, which ideally determine if the protocol implementation meets its specification.

We have made significant progress towards meeting this goal. The TESTGEN program generates sequences of tests which will test every transition and clause if it is able, and if there are any which it is unable to test the user will be informed. The user is also warned of difficult to test conditions such as transient states and converging transitions.

A second, broader purpose of this work has been to unify the fields of protocol specification, testing and verification under a single protocol model, *systems of communicating machines*. As earlier work [LuBu] has automated the verification process (to some degree), we now have tools for specification, verification and testing in this protocol model.

Further work in this area might be to use this and the other tools to answer questions concerning protocol design, verification and testing. Protocol testing, as protocol verification, is a very difficult but critical process. If we can learn how to design protocols which can be verified and tested, we should have better functioning networks.

[Mil] Miller, Raymond E., "Protocol Verification: The first Ten Years, The Next Ten Years; Some Personal Observations," *Protocol Specification, Testing and Verification X*, North Holland, 1990.

[PSTV I-XII] *Protocol Specification, Testing and Verification*, Vols. I-XII, North-Holland.

[SiLe] Sidhu, Deepinder and Leung, Ting-kau, "Fault Coverage of Protocol Test Methods," IEEE 1988.

[Hol] Holzmann, Gerard J., *Design and Validation of Computer Protocols*, Prentice Hall Publishing Co., 1991.

[LuAk] Lundy, G. M., and Akyildiz I. F., "Specification and analysis of the FDDI MAC protocol using systems of communicating machines," *Computer Communications*, Vol. 15, No. 5, pp.286-294, June 1992.

[LuBu] Lundy G. M., and Bulbul Z. B., "Mushroom: a Program for the Automated Verification of an SCM Protocol Specification," *International Conference on Network Protocols (ICNP-93)*, San Francisco, October 19-22, 1993.

[LuMi1] Lundy, G. M., and Miller, R. E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines," *Distributed Computing*, Springer -Verlag, December 1991.

[LuMi3] Lundy, G. M., and Miller, R. E., "Analyzing a CSMA/CD Protocol Through a Systems of Communicating Machines Specification," *IEEE Transactions on Communications*, March 1993.

[MiLu] Miller R. E., and Lundy G. M., "Testing Protocol Implementations Based on a Formal Specification," *Protocol Test Systems, III*, North-Holland, 1991.

[Bas] Basaran C., *TESTGEN: Automated Generation of Test Sequence for a Formal Protocol Specification*, M.S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, March 1994.1