Protocol Validation Tools as Test Case Generators*

Chang-Jia Wang, Liang-Seng Koh and Ming T. Liu

Department of Computer and Information Science, The Ohio State University,
2015 Neil Ave., Columbus, Ohio 43210-1277, USA

Abstract
A method is proposed to transform a test case generation problem to a protocol validation problem. Protocol validation has been studied for years and many validation tools are available. By transforming a test case generation problem into a protocol validation problem, a protocol validation tool can be used to generate test cases. The method can be implemented in a very short period of time. The complexity of the proposed method in searching for a test case is $O(n)$, where n is the number of system states in the specification.

1 Introduction

Test case generation (TCG) for *Finite State Machines (FSM)* has been studied for years, and fruitful results have been produced [1, 2, 3, 4]. The FSM is a simple model that ignores the data flow of a protocol but concentrates on the protocol's control flow. In other words, it simplifies a protocol into a state-transition machine. However, ignoring the data flow of the protocol often removes many interesting properties from the protocol. As an example, it is hard to specify a simple protocol like Go-Back-N in FSM without losing those important properties associated with the protocol's window size.

Therefore, most protocol specifications are written in more sophisticated models, such as *Extended Finite State Machine (EFSM)* [5], *Estelle* [6], *LOTOS* [7], or other programming languages. These models are called *extended models* in this paper with respect to the basic FSM model. To be more practical, a test case generation method must be able to generate test cases from extended models. However, the difficulty is that to test an extended model, one needs to test not only the control flow, but also the data flow of the protocol. Testing both control and data flows is considered a challenging problem; there-

*Research reported herein was supported by U.S. Army Research Office, under contracts No. DAAL03-91-G-0093 and No. DAAL03-92-G-0184. The views, opinions, and/or findings contained in this paper are those of the authors and should not be construed as an official Department of the Army position, policy or decision.

fore not much work has been done so far. Some references about the test case generation for the extended models can be found in [8, 9, 10, 11, 12, 13, 14].

A test case can be used for detecting a certain type of faults, which are called the *fault models* of the test case [15]. Because of the complexity involved in searching test cases for extended models, it is believed that *fault models* should be used as a guide to simplify the search [16, 17]. Traditionally, a TCG method is dedicated to generating test cases for certain fixed fault models. On the contrary, a TCG method based on fault models takes both a protocol specification and a given fault model as input and generates a test case for the fault model. An I/O sequence can be used as a test case if the specification and the fault model behave differently when applying the I/O sequence.

A fault model is usually a modification of the original protocol specification, and can be described by the same model used to define the protocol specification. For example, Figure 1 is the sender of a Go-Back-N protocol specified in EFSM with a window size equal to four (W=4). A fault model like the one drawn in Figure 2 has a window size being set to five. Note that only the action associated with the first transition is changed[†].

Generating test cases based on a given fault model has the following advantages:

1. Every test case has a well defined *test purpose*. That is, the types of errors a test case can detect is automatically provided. In addition, those who test a protocol implementation with the test case will have the confidence that the fault described by the fault model does not exist in the implementation.
2. It reduces the complexity of finding a test case. Instead of searching aimlessly, a TCG method can use a given fault model as a goal to direct the search.
3. Users can choose to test critical faults first, and test minor errors later.
4. It is more flexible than the traditional methods. The traditional TCG methods have fixed fault models embedded in them. If a critical fault does not contain in the fault models, the traditional TCG methods cannot effectively test this fault.

[†]The EFSMs used in this paper have five types of actions associated with their transitions.

1. Assignment: $W := 4$ assigns value 4 to variable W.
2. Input: $R?Ack(A)$ receives message Ack whose argument is assigned to variable A.
3. Output: $R!Msg(B[Sm], Sm)$ sends a message Msg with arguments $B[Sm]$ and Sm, where $B[Sm]$ is the Sm-th element in array B.
4. Condition: A transition with action $k <> Sm$ can only be executed when k and Sm are not equal.
5. Timeout (T/O): The transition will be executed after a certain amount of time expires.

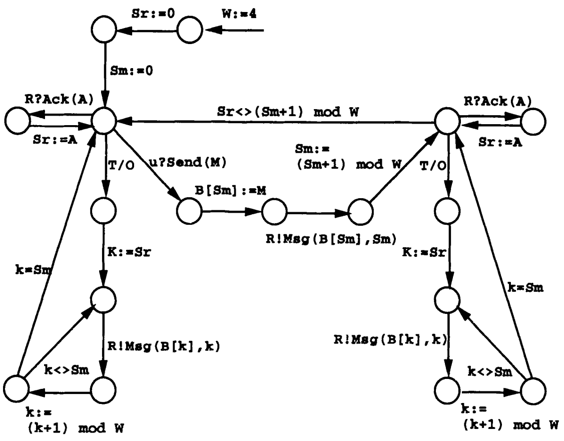

Figure 1: Go-Back-N protocol specified in EFSM

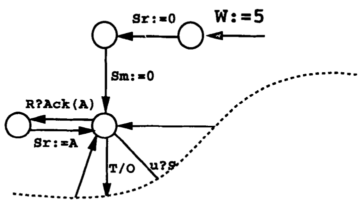

Figure 2: A fault model of Figure 1 (The rest of this figure is the same as Figure 1

In this paper, the problem of TCG based on fault models is transformed into a *protocol validation (PV)* problem. Protocol validation, which is used to detect design errors in a protocol specification, has been studied for years. While many protocol validation tools have been made available, automatic test case generators are still hard to find. Therefore, a method is proposed to transform TCG to PV so that available validation tools can be used to generate test cases. It will also be shown later that the transformation method not only can be used to generate test cases for weak conformance, but also can be used to generate test cases for strong conformance, functional testing, and multiple-module specifications.

It is well known that there is a so called *state explosion* problem for protocol validation, which means that the number of states generated by a protocol validation tool increases exponentially when the protocol specification becomes more complicated. The method proposed in this paper takes a protocol specification and a fault model as its input and generates a combined system to be fed into a protocol validation tool. It is also shown in this paper that the complexity of the combined system is $O(n)$ if the number of global states in the original protocol specification is n. Therefore, the state explosion problem will not occur as long as it will not occur in the original specification.

The organization of this paper is as follows. In Section 2, the transformation from the test case generation problem to the protocol validation problem is proposed. The same section also discusses how the transformation to weak and strong conformance, functional testing, and multiple-module protocols can be applied. In Section 3, the implementation of the test case generator is introduced, and some experimental results are discussed. Finally, the complexity of the test case generator is discussed in Section 4 and a conclusion is given in Section 5.

2 Transforming TCG to PV

2.1 The Transformation

A test case for a fault model is an I/O sequence that detects the discrepancies between the fault model and the protocol specification. That is, the test case is a possible I/O sequence for one, but not for the other. For example, an I/O sequence can be used as a test case if it can be generated from the protocol specification, but cannot be reproduced from the fault model. In other words, if a protocol implementation contains an error described by the fault model, applying the input sequence of the test case to the implementation will produce an unexpected output because the I/O sequence cannot appear in the implementation.

The task is to find an I/O sequence that can only be applied to either the protocol specification or the fault model, but not both. Let P (for *Primary*) be an automaton that describes either the protocol specification or the fault model, and let S (for *Secondary*) be an automaton that describes the other. If P and S are executed synchronously, whenever P makes a certain output, S must make the same output; otherwise, it will be a discrepancy. Similarly, if P requires a certain input, S must wait for the same input; otherwise, it will be a discrepancy as well.

The idea here is to construct a system that will deadlock when P and S take different input or output actions. The combined system can be fed into a protocol validation tool. When the validation tool signals a deadlock error, one can trace through the execution path that causes the deadlock. Then, the I/O events generated by the path can be used as a test case.

The first step is to modify automaton P. Let P' be an automaton the same as P, except for the following differences:

1. For every output action in P, make the same output. Then, wait for a **resume** signal.
2. For every input action in P, output a message that requests the input first. Then, wait for the input.
3. For every timeout in P, output a message labeled T/O. Then wait for a **resume** signal.

Next, make an automaton S' similar to S except for the following changes:

1. For every output action in S, input a message first. If the input message is the same as the message to be output, output an **ok** signal.
2. For every input action in S, input the message. Then, output an **ok** signal.
3. Remove every timeout in S.

Finally, let M (for *Monitor*) be an automaton communicating with P' and S'. M provides a variable for each parameter of every input action in P'. The variables are initialized to their lower bound values. Monitor M performs the following:

1. Increment the value of a variable by one nondeterministically, unless an upper bound is reached.
2. When an output from P' is received, pass the message to S' and wait for an **ok** signal from S'. After receiving the **ok** signal, send a **resume** signal to P', and record the output event.

Table 1: Modification made by P', S', and M

	Original	**Primary** P'	**Secondary** S'	**Monitor** M
Output	$G!msg(x_1, \cdots, x_n)$	$M!msg(x_1, \cdots, x_n)$ $M?resume$	$M?msg(p_1, \cdots, p_n)$ $p_1 = x_1$ $\vdots$ $p_n = x_n$ $M!ok$	$P'?msg(p_1, \cdots, p_n)$ $S'!msg(p_1, \cdots, p_n)$ $S'?ok$ $P'!resume$
Input	$G?msg(x_1, \cdots, x_n)$	$M!request(msg)$ $M?msg(x_1, \cdots, x_n)$	$M?msg(x_1, \cdots, x_n)$ $M!ok$	$P'?request(msg)$ $S'!msg(x_1, \cdots, x_n)$ $S'?ok$ $P'!msg(x_1, \cdots, x_n)$
Timeout	T/O	$M!T/O$ $M?resume$		$P'?T/O$ $P'!resume$

3. When P' asks for an input message, send the message to S' first. After receiving an **ok** from S', send the same message to P' and record the input event.

4. When P' outputs a `T/O` message, send **resume** signals to P', and record the timeout event.

The abovementioned modification is summarized in Table 1. Let $\mathcal{I}$ (for *Integrated*) be an integrated system that contains P', S', and M. Then, $\mathcal{I}$ will deadlock if P generates some I/O sequences that cannot be reproduced by S. This can be briefly proved as follows.

1. If S cannot generate the same message output by P, S' will stop. Without receiving any **ok** signal, M will be blocked and will not send the **resume** signal to P'. Therefore, P' will be blocked as well.

2. If S will not receive the same input as P does, S' will be blocked and no **ok** signal will be send. Therefore, M will be blocked and P' will never receive the **resume** signal. The system deadlocks.

The protocol among M, P' and S' is illustrated in Figure 3. When P' sends a message **msg** to M, M passes it to S'. If the **msg** is also a message that S' intends to output, S' will send an **ok**. Upon receiving the **ok** signal, M sends a **resume** to P' (Figure 3a). On the other hand, if P' needs to receive a certain message, it sends a request to M. Upon receiving the request, M sends the message to S' first. If an **ok** is received, M sends the requested message to P' (Figure 3b). The timeout message between P' and M' is used to record the timeout event, and will not affect the overall execution of the system.

An example of the transformation is shown in Figures 4 to 6. Figure 4 is the primary automaton P' modified from the Go-Back-N protocol specification in Figure 1. The

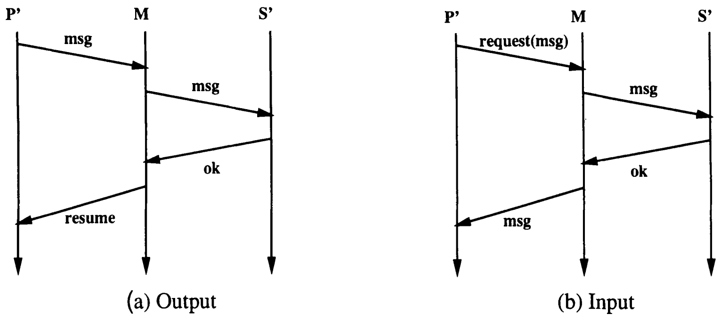

Figure 3: Communication among M, P' and S'

shaded arrows and circles are the transitions and states being changed according to the rules described in Table 1. Figure 5 is the secondary automaton S' modified from the fault model shown in Figure 2. Monitor M for Figures 4 and 5 is drawn in Figure 6. Note that all the I/O events in P' and S' are changed into communication between P' and M, and between S' and M, respectively.

When a protocol specification and its fault models are specified by a formal model (such as FSM, EFSM, Estelle, LOTOS, SDL, Petri-Net, Promela, etc.), it is trivial to transform P and S into P' and S', respectively. It is also straightforward to construct the monitor M. Automata P', S' and M can be fed into a validation tool for the formal model and a deadlock can be found if there is an I/O sequence that is possible for P but cannot be generated by S. By retracing the path leading to the deadlock, monitor M will record the I/O events along the path. The recorded I/O event sequence can then be used as a test case.

2.2 Weak and Strong Conformance

There are two types of conformance between a protocol specification and its implementation. The *weak conformance* indicates that a protocol implementation should perform every function defined in its specification. In other words, things that ought to happen should happen. The *strong conformance* imposes an additional constraint which states that a protocol implementation cannot have any behavior not defined by the protocol specification. That is, things that cannot happen must not occur.

A fault model violating the weak conformance cannot perform some functions defined by the protocol specification. In other words, there exists an I/O event sequence that

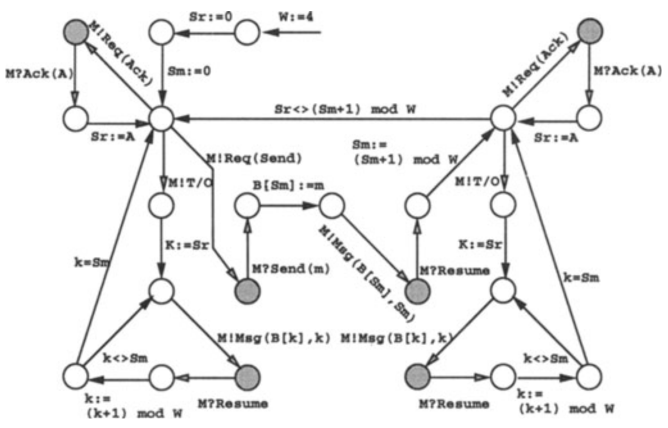

Figure 4: Modified protocol specification P'

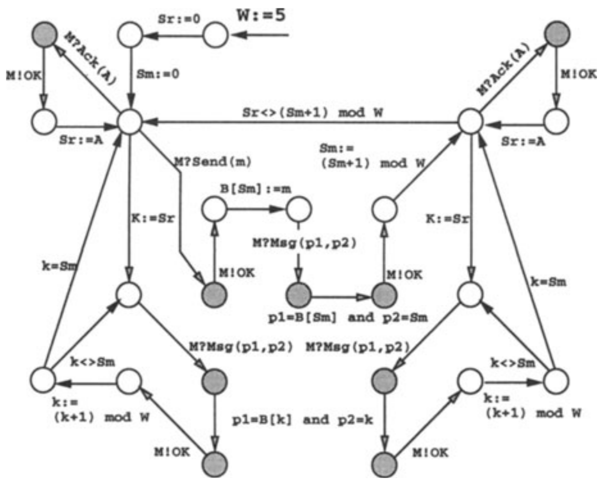

Figure 5: Modified fault model S'

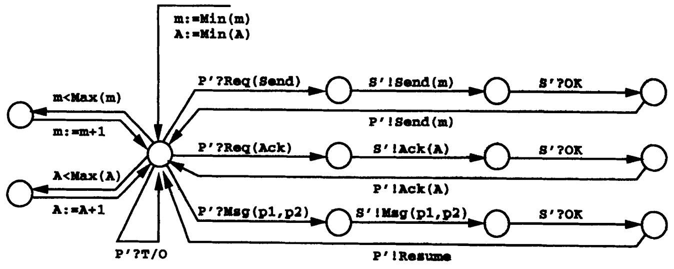

Figure 6: Monitor *M*

can be generated by the specification, but cannot happen in the fault model. To generate a test case for the fault model, one can use the protocol specification as the primary automaton P, use the fault model as the secondary automaton S, apply the transformation in Section 2.1, and feed the resulting system $\mathcal{I}$ into a protocol validation tool.

A fault model that violates the strong conformance has some execution paths that are not specified in the protocol specification. That is, some I/O sequence generated by the fault model can never happen in the specification. Thus, if one uses the fault model as the primary automaton P and uses the specification as the secondary automaton S, one can find a test case with the method described in Section 2.1. When the input of the test case is applied to a protocol implementation, and an expected output appears, the implementation must contain an error described by the fault model.

It is possible that a fault model violates both weak and strong conformance. In this case, treating the protocol specification as either the primary or the secondary will work. Sometimes, however, it is difficult to tell whether a fault model violates the weak or strong conformance. In such a case, if one method cannot find a test case, the other must be tried.

2.3 Functional Testing

It is useful to create test cases that ensure the correctness of certain behavior in the protocol specification. Such "behavior" is often called a *function* or a *requirement* of the protocol specification. For example, a requirement of the Go-Back-N protocol in Figure 1 can be "*there are no more than three messages in the channel at any given time.*"

In order to test if a protocol implementation conforms with the requirement, a fault model that violates the requirement is constructed. However, since a requirement usually is just a small portion of the original protocol specification, it will be quite troublesome to construct a fault model from the entire protocol specification. In fact, a fault model can be created by including only those transitions necessary to violate the requirement. For example, a fault model violating the requirement described in the above paragraph can be shown as an EFSM in Figure 7. In the figure, there is no restriction on how many messages can be sent to the receiver consecutively before the sender receives an acknowledgment.

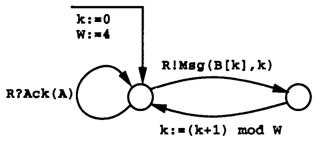

Figure 7: A fault model that sends more than three messages at a time.

The fault model in Figure 7 defines some behavior that should not exist in a correct implementation, so it violates the strong conformance requirement. Therefore, it is used as a primary automaton and the specification is used as a secondary. Since the fault model only specifies partial behavior of the entire specification, some I/O events in the specification may not exist in the fault model. Hence, a modification of the transformation described in Section 2.1 should be made as follows.

- To change P (S) into P' (S'):
    1. If an input or output action exists in both P and S, follow the procedure in Section 2.1.
    2. For every output message that exists only in P (S), send the output to M and wait for a **resume** signal from M.
    3. For every input message that exists only in P (S), send an request for the input to M and wait for the input event from M.
- To construct M:
    1. For those input or output actions that exist in both P and S, follow the procedure in Section 2.1.
    2. If an output message that exists only in P (S) is received from P' (S'), record the output event and send a **resume** signal to P' (S').
    3. If a request for input that exists only in P (S) is received from P' (S'), record the input event and send the input message to P' (S').

In other words, if an input or output message exists only in P or S, the monitor M will not try to synchronize P' and S' with the message. It will only record the I/O event and release the automaton that generates the event. For example, in Figure 1, the input event `u?Send(m)` exists only in the specification, and should be modified according to the above procedure. The resulting EFSMs P', S', and M are shown in Figures 8, 9, and 10, respectively. Note that the shaded nodes and arrows in Figures 9 and 1 show the difference between the method in this section and the one in Section 2.1.

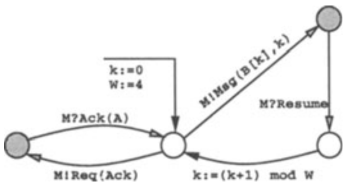

Figure 8: The primary P' from the fault model in Figure 7.

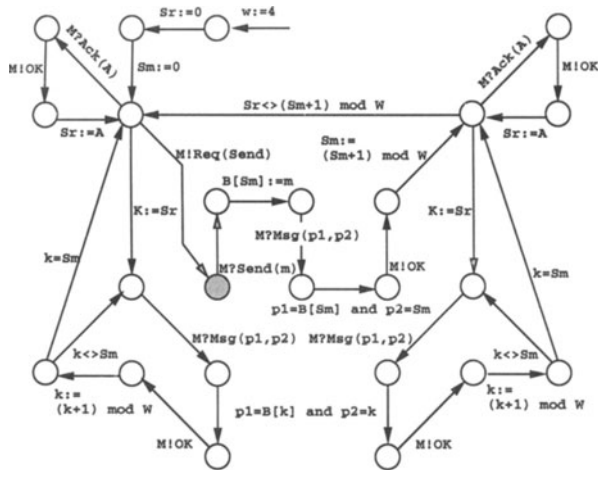

Figure 9: The secondary S' from the specification in Figure 1.

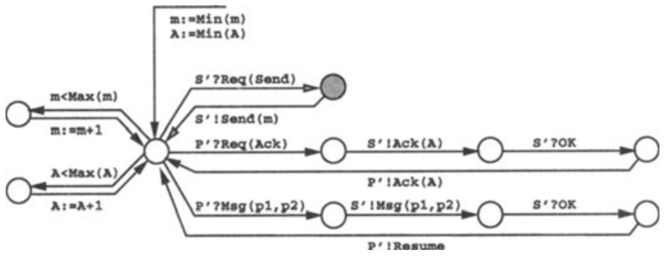

Figure 10: The monitor M from the fault model in Figure 7 and the specification in Figure 1.

2.4 Test Case Generation for Multiple Modules

Test cases can also be generated for multiple modules using this method. For example, one may want to test how a protocol containing a sender and a receiver operates when it is connected to a network. The whole system can be modeled like the one in Figure 11.

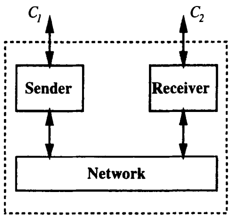

Figure 11: A multiple-module specification

A modification should be made when testing a multiple-module specification:

- Only those I/O actions that are used to communicate with the outside world should be monitored and synchronized by M.

Therefore, in Figure 11, only those actions that send or receive messages through channels C_1 or C_2 should be changed by the procedure described in Sections 2.1. Other I/O events are considered *internal* and will not be observed.

3 Implementation and Experimental Results

The major advantage of using the transformation approach is to use protocol validation tools for test case generation. A lot of efforts can be saved by using the existing programs. Since the transformation described in this paper is straightforward, the implementation is nearly trivial.

To demonstrate the idea, a protocol validation tool, called *SPIN*, implemented by Holzmann at AT&T is used as the backbone of our test case generator. SPIN uses a C- and CSP-like specification language, called *Promela*, to specify a protocol. The program to be implemented will translate a specification and a fault model written in Promela to a primary P' and a secondary S', and construct a monitor M. The program contains around 1,000 lines of *C++*, *yacc*, and *lex* codes, and is developed by a single person within one week. On a SUN SPARCstation, the program took less than 20 seconds to generate the following test case for the specification in Figure 1 and the fault model in Figure 2.

```
u?Send(1),  R!msg(1,0), u?Send(1), R!msg(1,1), u?Send(1),
R!msg(1,2), R?ack(9),   u?Send(1), R!msg(1,3), u?Send(1)
```

The meaning of the test case is as follows:

1. Input a message `Send` from channel `u`. The content of the message is `1`.
2. Expect an output, `msg(1,0)`, at channel `R`, with message `1` and sequence number `0`.
3. etc.

If constructing fault models is not desirable, a program that randomly modifies the protocol specification to create fault models is also available. The program has the following capabilities:

1. Randomly modifying the value of a constant (e.g., changing a "4" into a "5").
2. Randomly modifying variable references (e.g., changing a variable x to y).
3. Modifying the operations in an expression (e.g., changing $x + y$ into $x - y$).
4. Altering the control flow (e.g., inserting `goto` statement arbitrarily).
5. Removing a statement.
6. Combination of the above.

The fault models of the Go-Back-N protocol can automatically be generated by the program described above. The fault model generator can then be integrated with the test case generator to find test cases for random errors.

4 Performance and Complexity Considerations

In order not to confuse with the states in an EFSM, we define a *system state* that is a collection of current states of the modules and current values of the variables. For example, if the sender in Figure 11 is in state S_1, the receiver is in state R_2, the network is in state N_3, and two variables x and y in the protocol specification have values 4 and 5, respectively, then the current system state of the protocol will be $\langle S_1, R_2, N_3, x = 4, y = 5\rangle$.

As shown in Figure 3, P', S' and M are synchronized by the messages passed among them. For example, during an output event, P' sends a message to M and stops until a **resume** signal is received. Similarly, M sends a message to S' and cannot move until an **ok** signal is received. Therefore, automata P', S', and M are synchronized at the point where P issues an output. Between two input or output actions, P', S' and M can run concurrently. Since M does not have any transition to do between two I/O events, the interleaving is between P' and S'.

Since the transitions of P' and S' between two consecutive I/O events are independent, P' and S' can be executed atomically and still yield the same result as if they were executed in an interleaving way. That is, after a synchronized I/O event, P' can run first until its next I/O event, and then S' can start to run until its next I/O event. Assuming that there are n system states in the protocol specification, and m system states in the fault model, the total number of system states for $\mathcal{I}$ is $O(m + n)$, instead of $O(mn)$. Mostly, m and n are about the same, and the complexity of this method is $O(n)$.

In fact, our implementation has shown good performance, in spite of the inefficiency in using SPIN as a validation tool. To validate a Promela specification, SPIN first generates a C program. Compiling and running the program will generate a path that leads to an error. The path is retraced by SPIN, and the I/O events that lead to the deadlock will be recorded by M. It is found that compiling the C program takes most of the execution time. Therefore, if more efficient validation tools were used, the performance would have been even better.

5 Conclusion

The protocol validation problem has been studied for years and many protocol validation tools are available. This paper proposes a method to transform the test case generation problem to a protocol validation problem. Therefore, instead of developing it from scratch, a test case generator can be built upon an existing protocol validation tool.

The transformation method takes a protocol specification and a fault model as its input, and generates three automata, namely Primary P', Secondary S', and a Monitor

M. The system containing the three automata is treated as a protocol and fed into a protocol validation tool. A deadlock will occur if there is a discrepancy between the protocol specification and its fault model. By analyzing the path that leads to the deadlock and by carefully recording the I/O events, a test case can be found.

The method does not introduce extra complexity to the integrated system $\mathcal{I}$. The total number of system states explored by the protocol validation tool is in about the same order as the total number of system states in the original protocol specification. Therefore, the state explosion problem will not be as serious as a general protocol validation problem.

The method was implemented in a very short period of time. In addition, the experiment showed that the performance of the program is quite acceptable; a test case can be generated within a minute. Therefore, those who have a protocol validation tool may now use this method to transform the validation tool into a test case generator with virtually no extra cost.

References

[1] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, March 1978.

[2] G. Gönenç, "A model for the design of fault detection experiments," *IEEE Trans. on Computers*, vol. C-19, no. 6, pp. 551–558, June 1970.

[3] S. Naito, "Fault detection for sequential machines by transition tours," in *Proc. 11th IEEE Symp. on Fault Tolerant Computing*, pp. 238–243, 1981.

[4] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Computer Networks and ISDN Systems*, vol. 15, pp. 285–297, 1988.

[5] G. v. Bochmann and J. Gecsei, "A unified method for the specification and verification of protocols," in *Proc. IFIP Congress '77*, pp. 229–234, 1977.

[6] S. Budkowski and P. Dembinski, "An introduction to Estelle: A specification language for distributed systems," *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 3–23, 1987.

[7] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, pp. 25–59, 1987.

[8] E. Brinksma, "A theory for the derivation of tests," in *The Formal Description Technique LOTOS* (P. van Eijk, C.A.Vissers, and M. Diaz, eds.), pp. 235–247, Elsevier Science Publishers B.V. (North-Holland), 1989.

[9] R. Langerak, "A testing theory for LOTOS using deadlock detection," in *Proc. 10th IFIP Symp. on Protocol Specification, Testing, and Verification*, pp. 87–98, 1990.

[10] B. Sarikaya, G. v. Bochmann, and E. Cerny, "A test design methodology for protocol testing," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 5, pp. 518–531, May 1987.

[11] H. Ural, "Test sequence selection based on static data flow analysis," *Computer Communications*, vol. 10, no. 5, pp. 234–242, 1987.

[12] H. Ural and B. Yang, "A test sequence selection method for protocol testing," *IEEE Trans. on Communications*, vol. 39, no. 4, pp. 514–523, April 1991.

[13] C.-J. Wang and M. T. Liu, "Axiomatic test sequence generation for extended finite state machines," in *Proc. 12th International Conference on Distributed Computing Systems*, pp. 252–259, June 1992.

[14] C.-J. Wang and M. T. Liu, "A test suite generation method for extended finite state machines using axiomatic semantics approach," in *IFIP Trans. Protocol Specification, Testing, and Verification, XII*, pp. 29–43, North-Holland, 1992.

[15] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo, "Fault models in testing," in *Protocol Test Systems, IV*, pp. 17–30, Elsevier Science Publisher B.V. (North-Holland), 1992.

[16] C.-J. Wang and M. T. Liu, "Generating test cases for EFSM with given fault models," in *Proc. IEEE INFOCOM '93*, pp. 774–781, M•rch 1993.

[17] C.-J. Wang and M. T. Liu, "Automatic test case generation for Estelle," in *Proc. 1993 Int'l Conf. on Network Protocols*, October 1993.