DETERMINISTIC EXECUTION TESTING OF FSM-BASED PROTOCOLS

Darrell Hubbard

Darrell Hubbard *is a member of technical staff in the Networking Standards Group at AT&T Bell Laboratories in Holmdel, New Jersey. He develops international conformance testing and high-speed networking standards. His research interests include protocol specification, verification, and validation. He has a B.S. in computer science and a B.A. in business administration from North Carolina Wesleyan College and an M.S. in computer science from North Carolina State University. He joined AT&T in 1988.*

This paper discusses how the deterministic execution testing approach can be applied to solve the synchronization and nondeterminism problems in protocol testing. Deterministic execution testing is achieved by incorporating test coordination procedures to control the execution of the test system and to control specific nondeterministic behavior in the implementation under test. The deterministic execution testing approach is applied to the Open Systems Interface Class 0 Transport Layer protocol.

Introduction

There has been an extensive amount of research to enable distributed computer systems and networks to communicate with each other. The method in which communication networks exchange data is through commonly used protocols. A large number of these protocols are standardized by the International Organization for Standardization (ISO) and the International Telegraph and Telephone Consultative Committee (CCITT). From these standard protocol specifications, vendors must provide an implementation of the protocol for their communication systems. To ensure that different protocol implementations can communicate, testing is needed to ensure that their behavior adheres to the requirements of the reference specification. However, testing may be problematic because of the difficulty in synchronizing a system under test. This paper describes a solution to these synchronization problems based on deterministic execution testing.

The paper provides an overview of finite state machine (FSM) based protocol testing and discusses the synchronization problems associated with protocol testing. It presents solutions to the deterministic execution testing problem for general communication software that uses `SEND` and `RECEIVE` constructs. After suggesting a general solution to the deterministic execution testing problem for communication software with `SEND` and `RECEIVE`, the paper provides a solution to the deterministic execution testing problem for FSM-based communication protocols. A final section summarizes the paper and suggests issues for further research.

Panel 1. Acronyms and Abbreviations in This Paper

CCITT	International Telegraph and Telephone Consultative Committee
ISO	International Organization for Standardization
IUT	implementation under test
FIFO	first in, first out
FSM	finite state machine
ID	identifier
MISR	mailbox-indexed SR
OSI	Open System Interconnection
R	responder
SR	SEND-RECEIVE
SRM	SEND-RECEIVE MAILBOX
T	tester

FSM-Based Protocol Testing

The FSM model can be used to describe a protocol specification or implementation. The sections below define the FSM model, explain how several processes represented as FSMs can communicate or exchange messages in the protocol testing environment, and indicate the synchronization problems exhibited in the testing environment.

The Finite State Machine Model. The finite state machine model[1] is a formal notation for describing a protocol or the individual states that comprise a protocol. The FSM model can be represented by the 5-tuple (S, I, O, N, A), where S is a finite set of all states in the machine, I is a finite set of all possible inputs, O is a finite set of all possible outputs, N is the state transition function, $S \times I \rightarrow S$, which indicates the next state of the machine as a result of some input transition, and A is the action to be performed and output message to be generated upon some input transition. Since the FSM may receive messages from more than one process, the input/output messages are in the form of $P.M$, where P denotes the source or destination process of the message and M is the content of the message. If the content of the message can uniquely identify its source or destination, then P can be omitted from the message content. Only one transition is assumed to exist in the FSM on any input.

A Model for Testing FSM-Based Protocols. A widely used model for testing FSM-based protocols is shown in Figure 1. This model consists of the tester (T), responder (R), and the implementation under test (IUT). The responder acts as a service user of the IUT. In this model, the IUT is assumed to be based on an FSM specification. As shown in Figure 1, the interactions between T, R, and IUT occur by exchanging messages through first-in, first-out (FIFO) mailboxes.

One mailbox is between T and the IUT and the other is between the IUT and R. The purpose of the mailboxes is for the IUT to SEND/RECEIVE messages to/from T and R. T or R may send a message to the IUT through the appropriate mailbox. The IUT requests to receive input from both mailboxes, M_{tp} and M_{rp}. If no message is available, the IUT waits for a message to arrive. If only one mailbox has a message available, it is accepted for processing. If there is not an input transition available for that message, the IUT will traverse to an error state. If both mailboxes have an input message available, one of them will be chosen nondeterministically. If neither of the available messages will initiate a valid transition, the IUT will traverse to an error state. Upon receiving a message from T or R, the IUT may send an output message to T and/or R.

Synchronization Problems with the T/R/IUT Model. In Reference 2, a definition of the synchronization problem is provided. This definition states that a synchronization problem exists if T (or R) does not take part in a transition and if the next transition requires that it send a message to the IUT. However, this definition is too general because it does not specify how the IUT accepts messages as they arrive at its input queues. Thus, different synchronization problems can exist, depending on the IUT's algorithm for accepting input messages at its input queues. A new definition of the synchronization problem, based on the T/R/IUT model defined above in "The Finite State Machine Model," is: When the IUT requests to receive an input message, a synchronization problem exists (1) if only one message is available and it is not the expected message according to the test sequence or (2) if a message is available at both mailboxes and the message nondeterministically selected is not the expected input message according to the test sequence.

An exchange of information between a pair of processes (i.e., T and IUT or R and IUT) is called a rendezvous. An ordered sequence of rendezvous between various pairs of processes is called a synchronization sequence (syn-sequence). Nondeterministic behavior in communication protocols causes unpredictable syn-sequences during repeated executions of the IUT with the same test sequence. This may yield different results during each execution. Deterministic execution testing eliminates this nondeterministic behavior and reproduces the intended syn-sequence during repeated executions of the IUT with the same test sequence.

Approaches to Deterministic Execution Testing. There are several approaches that can be used to solve the deterministic execution testing problem for protocols. One approach is by mailbox modification. The T/R/IUT model has two mailboxes. If the mailboxes are modified to control the ordering of `SEND`s and `RECEIVE`s, a `SEND-RECEIVE` (SR) sequence could be applied to each mailbox in order to deterministically test the protocol.

Another approach to solving the problem is to use only test sequences that do not have a synchronization problem. This is the approach described in Reference 2. An advantage of this approach is that deterministic execution testing can be achieved without using test coordination procedures.

A third approach is program transformation. The purpose of program transformation is to control the execution of a concurrent program by inserting additional synchronization statements in the concurrent program. This approach requires transformation of the IUT. The solutions presented in the following sections eliminate the synchronization problems defined under "Synchronization Problems with the T/R/IUT Model" for any test sequence using the third approach.

Synchronization by Program Transformation

This section provides a solution to the synchronization problem with `SEND/RECEIVE` for general communication software with multiple processes and mailboxes which communicate in any interleaved order, as

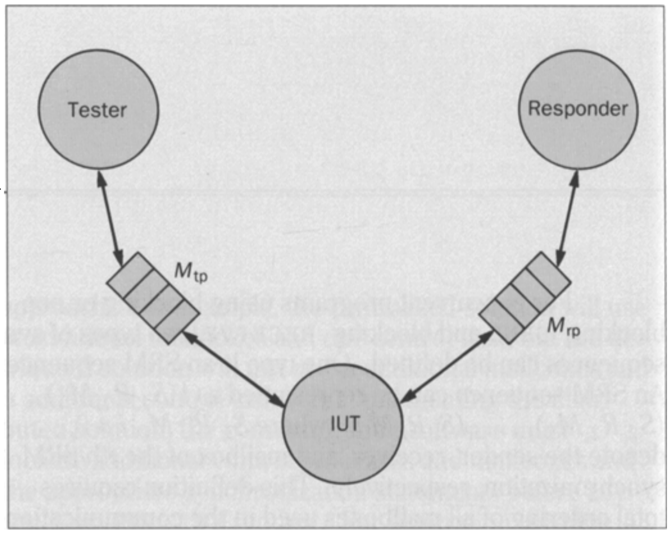

Figure 1. T/R/IUT model.

described in Reference 3. In a later section, program transformation is applied to the specific problem of protocol testing.

Syn-Sequence Definition with Mailbox Naming. When a syn-sequence is defined, the syntax of the constructs must first be analyzed. In the approach using mailbox-naming, the source/destination is a mailbox to which one or more processes send messages and from which one or more processes receive messages. The syntax of mailbox naming can be generically defined as follows:

```
SEND    message  TO   destination;
RECEIVE message  FROM source;
```

The `SEND` and `RECEIVE` can be blocking or nonblocking. A process that executes a blocking `SEND` is suspended until a corresponding `RECEIVE` is issued from another process, unless a corresponding blocking `RECEIVE` has already been issued. A nonblocking `SEND` is never suspended. A process that executes a blocking `RECEIVE` is suspended until a corresponding blocking `SEND` is issued from another process, unless a corresponding blocking `SEND` has already been issued. A nonblocking `RECEIVE` is never suspended. When the message sent by a `SEND` is received by a `RECEIVE` through a mailbox, this `SEND` and `RECEIVE` are said to have a synchronization through this mailbox, referred to as a `SEND-RECEIVE-MAILBOX` synchronization (SRM sequence).[3]

For concurrent programs using blocking or non-blocking SEND and blocking RECEIVE, two types of syn-sequences can be defined. One type is an SRM sequence. An SRM sequence can be represented as $\{(S_1, R_1, M_1), (S_2, R_2, M_2), \ldots, (S_i, R_i, M_i)\}$, where S_i, R_i, M_i, $i > 0$, denote the sender, receiver, and mailbox of the ith SRM synchronization, respectively. This definition requires total ordering of all mailboxes used in the communication software. However, to increase the amount of concurrency, a SEND/RECEIVE (S/R) sequence can be derived individually for each mailbox. This can be represented as $\{(S_1, R_1), (S_2, R_2), \ldots, (S_i, R_i)\}$, where S_i and R_i, $i > 0$, denote the sender and receiver of the ith SRM synchronization through M. Thus, if a process uses several mailboxes, an execution of P can be characterized by the following sequences: SR sequence through M_1, SR sequence through $M_2, \ldots$, SR sequence through M_n. This ordering is referred to as a mailbox-indexed SR (MISR) sequence. In the MISR sequence there is no total ordering of rendezvous between all processes, but, for each mailbox, there is an order in which rendezvous must occur. These syn-sequence definitions were used to solve reproducible testing problems for SEND and RECEIVE between processes.

Synchronization Procedures. From the syn-sequence definitions above, two solutions to the deterministic execution testing problem are developed in this section by using the program transformation approach. The first solution is a centralized solution, and the second solution is a distributed solution.

A centralized solution. Given W, a feasible SRM sequence of P, represented as $\{(S_1, R_1, M_1), (S_2, R_2, M_2), \ldots, (S_n, R_n, M_n)\}$ and a concurrent program using blocking SEND and RECEIVE, replaying of the SRM sequence can be achieved by creating a new process called the control task and by inserting additional synchronization statements in the sending and receiving processes. The purpose of the control task is to control the execution of the SENDs and RECEIVEs in each process of P to the order specified in the SRM sequence. This control is essential because a mailbox can be sent and can distribute messages from one or more processes in an interleaved order. If this ordering is not controlled, synchronizations between processes through the mailbox cannot be controlled. This is an essential task for replaying an execution. For every process there is a unique mailbox that allows communication between the process and the control task. This communication between a process and control task exists because of the additional control statements inserted in program P. The program transformation is completed as follows. Any process which desires to SEND or RECEIVE a message must first receive permission from the control task by inserting

```
(a) RECEIVE permission FROM PERMIT_i;
```

immediately before the SEND/RECEIVE statement, where i is the process identifier (ID) of the requesting process and PERMIT_i is the mailbox through which the process and control task rendezvous. Any receiving process must also insert

```
(b) SEND completion TO PERMIT_i;
```

immediately after the RECEIVE statement. The purpose of statement (a) is to suspend the requesting process until the process is the intended process to complete the next rendezvous in the SRM sequence. Therefore, the process remains blocked until it is given permission to SEND or RECEIVE a message to/from the mailbox. The purpose of statement (b) is to inform the control task that the ith rendezvous is complete and that permission can be granted to the next sending and receiving rendezvous pair.

For every SEND/RECEIVE pair, the control task executes the following statements:

```
(a) SEND    permission TO   PERMIT_Si;
(b) SEND    permission TO   PERMIT_Ri;
(c) RECEIVE completion FROM PERMIT_Ri;
```

where PERMIT_Si and PERMIT_Ri are the mailboxes through which the sending process and the receiving process rendezvous with the control task, respectively.

Statements (a) and (b) allow the next sender and receiver to rendezvous, and statement (c) forces the control task to wait until the rendezvous is complete before giving permission to the next pair of processes to synchronize.

The above solution will not work for nonblocking SEND. Consider the following two consecutive SEND and RECEIVE statements in the control task:

```
(a) SEND    permission TO   PERMIT_Rj;
(b) RECEIVE completion FROM PERMIT_Rj;
```

where R_j is the synchronization mailbox between the control task and the requesting process. Because the SEND is nonblocking, the RECEIVE may synchronize with the SEND, since they share the same mailbox. The RECEIVE, however, is supposed to synchronize with the requesting process after the requested rendezvous has been completed. This problem is solved by using two PERMIT mailboxes instead of one. The first PERMIT mailbox is for the permission to start a synchronization, and the second is for completion of a synchronization. Therefore, the control task is modified to execute

```
(a) SEND    permission TO   PERMIT_RPj;
(b) RECEIVE completion FROM PERMIT_RCj;
```

where RP_j is the permission mailbox and RC_j is the completion mailbox for the receiving process P_j.

A distributed solution. In the centralized solution, the program P uses only one control task for controlling the synchronizations in m mailboxes, but in the distributed solution, P uses m control tasks for controlling the synchronizations in m mailboxes. In this distributed environment, the mailbox-indexed SR sequence (MISR sequence) should be used for controlling the synchronizations between processes. Each mailbox has a control process (controller) that enforces the synchronizations between processes. The advantage of this approach is that multiple control tasks increase the amount of concurrent processing in P. However, there is a greater amount of overhead involved in using this approach. For example, the centralized solution will use m additional mailboxes and one control task, but the distributed solution will use $m * n$ additional mailboxes and n additional control tasks. In implementing the distributed solution, the communication software must include additional control processes and mailboxes and the appropriate synchronization statements before any SEND or RECEIVE request. For illustration, assume there are n mailboxes which include an MISR sequence called W. For each W_j, $0 < j \le n$, there is a CONTROL_j which enforces the synchronization sequences in W_j. The CONTROL_j, $0 < j \le n$, communicates with each process P_i, $i > 0$, in P through mailbox PERMIT_i_j, where i is the process identifier and j is the control task identifier. The transforming of P is done as follows: for every SEND/RECEIVE through mailbox M_j, $0 < j \le n$, in process P_i, insert

```
RECEIVE permission FROM PERMIT_i_j;
```

immediately before the SEND/RECEIVE. This statement will suspend process P_i until P_i is permitted to SEND/RECEIVE a message from mailbox M_j. For each RECEIVE, insert

```
SEND completion TO PERMIT_i_j;
```

immediately after the RECEIVE statement. This statement notifies the CONTROL_j process of the completion of the rendezvous through mailbox M_j. For every SEND/RECEIVE pair, the process CONTROL_j issues the following statements:

```
(a) SEND    permission TO   PERMIT_Si_j;
(b) SEND    permission TO   PERMIT_Ri_j;
(c) RECEIVE completion FROM PERMIT_Ri_j;
```

Statement (a) gives permission to the process of the ith synchronization to SEND a message to mailbox M_j. Statement (b) gives permission to the process of the ith synchronization to RECEIVE a message from mailbox M_j. Statement (c) forces the CONTROL_j task to wait for completion of the rendezvous before giving permission to the next two processes to synchronize through

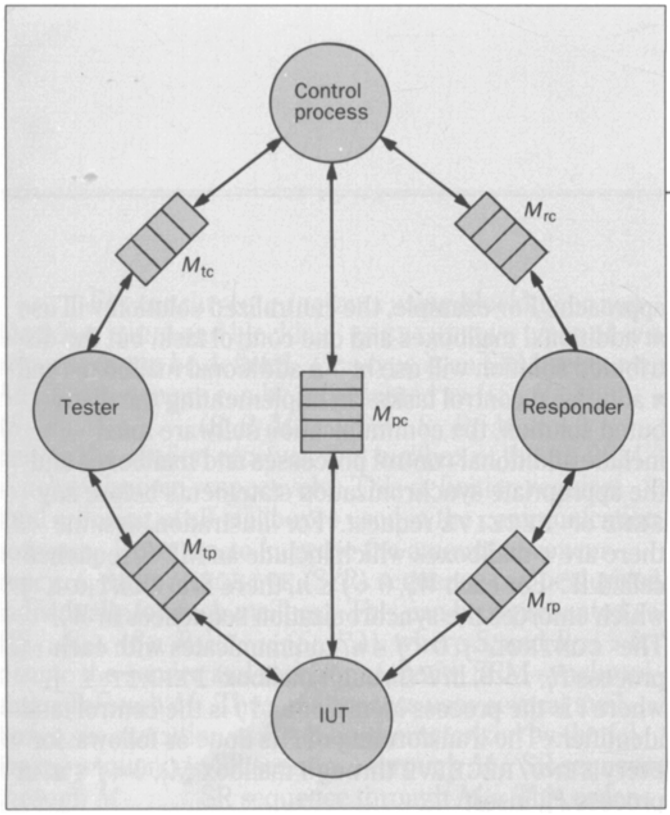

Figure 2. T/R/C/IUT model.

mailbox M_j. This solution also requires two additional mailboxes for each receiving process to solve the problem with nonblocking `SEND` as discussed in the centralized solution.

Testing FSM-Based Protocols with the T/R/IUT Model

This section presents a solution to the synchronization problems mentioned under "Synchronization Problems with the T/R/IUT Model." Even though the centralized solution for solving synchronization problems for general communication software can be used to solve the synchronization problems in the T/R/IUT model, the solution can be simplified because there is only one mailbox between any two pairs of processes (i.e., T and IUT or R and IUT) and the IUT is always the receiver of any input sent by T or R.

Synchronization Procedures. The solution presented in this section is a variation of the centralized solution using the program transformation method. It is assumed that T and R use blocking `RECEIVE`s. However, since the message control process controls when T and R can send an input message to the IUT, T and R are not dependent on the output messages sent to them by the IUT. Therefore, T and R can be simplified to become test drivers that send input messages to the IUT only on receiving permission from the message control process. All output messages generated by the IUT can be examined after the testing process for conformance to the specification.

The message controller. In solving the deterministic execution testing problem, another process, called the *message controller,* is needed. The message control process is responsible for controlling the sending of input messages to the IUT from the tester T and the responder R. Controlling when T and R can send a message to the IUT is done by forcing them to first receive the input data from the message control process. Once the input data are received from the message control process, then T/R can send the data to the IUT. The IUT must also send a completion message to the control process after the completion of the rendezvous between T/R. This signal informs the message control process to send the next test data to the appropriate process. Controlling the order in which the IUT receives the input sequences from T and R is done indirectly. Because of the process interactions between T, R, and the control process, only one input message can be outstanding for the IUT at any given time. Therefore, when the IUT executes a nondeterministic statement, such as `RECEIVE ANY`, only one message will be available, which is the intended message from the message control process. Thus, the message control process uses the following logic for every transition in the syn-sequence:

```
SEND permission TO Si;
RECEIVE completion FROM IUT;
```

where S_i indicates either T or R.

Insertion of control statements. The insertion of control statements in the tester, responder, and IUT is similar to the program transformation approach for `SEND` and RECEIVE. The tester and responder must insert

```
RECEIVE permission FROM CONTROLLER;
```

before every message to be sent to the protocol. The protocol does not have to insert any synchronization statements before it attempts to RECEIVE a message from the tester or responder. The reason is that control of when the tester or responder sends a message to the protocol causes the input messages to be in the correct order when requested by the protocol. Therefore, there is no need to control when the messages are received from the protocol's mailbox. However, the IUT must insert

```
SEND    completion TO   CONTROLLER;
```

after it receives a message from its mailbox. The purpose of this statement is to inform the control process that a rendezvous has occurred so that permission can be granted to the next sending process in the synchronization sequence. Figure 2 illustrates the newly proposed testing model.

Since these solutions require transformation of the IUT, this approach may be suitable only for in-house testing. In order to eliminate the communication link between the IUT and the control process, the COMPLETION message needs to be sent by the tester or responder. This can be done by forcing the tester and responder to insert

(a) RECEIVE permission FROM CONTROLLER;

before every message to be sent to the IUT and to insert

(b) RECEIVE completion FROM IUT;
(c) SEND completion TO CONTROLLER;

whenever an output message is to be received from the IUT. Statement (b) ensures that the IUT has completed its transition, and statement (c) informs the control process to begin the next transition. If the output message from the IUT is null, then statement (b) could be replaced with a timeout operation.

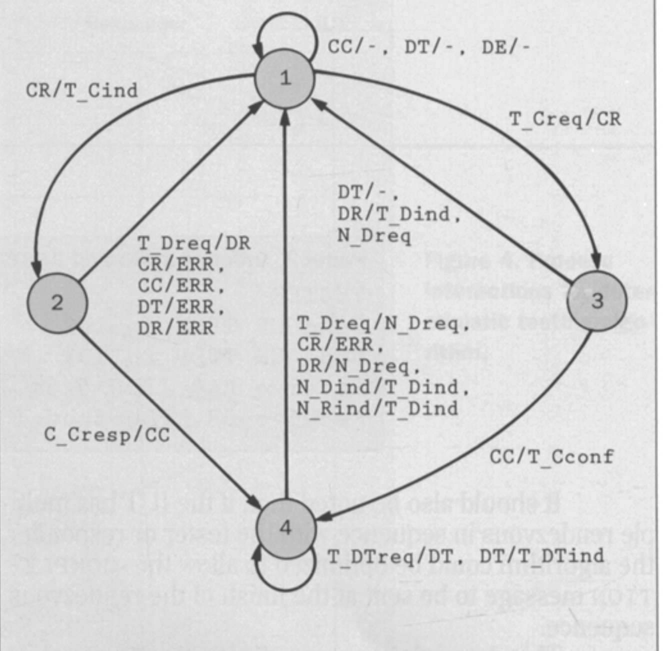

Figure 3. OSI Class 0 Transport Layer protocol.

Notation for input events

From tester

CR	Connect_Request PDU
CR	Connect_Request PDU
CC	Connect_Confirm PDU
DT	Data_Request PDU
DR	Disconnect_Request PDU
N_Dind	Network_Disconnect_Indication
N_Rind	Network_Reset_Indication

From responder

T_Creq	T_Connect_Request
T_Dreq	T_Disconnect_Request
T_Cresp	T_Connect_Response
T_DTreq	T_Data_Request

Notation for output events

To tester

ERR	Error PDU
N_Dreq	Network_Disconnect_Request
CC	Connect_Confirm PDU
DT	Data_Request PDU
DR	Disconnect_Request PDU
N_Dind	N_Disconnect_Indication
N_Rind	N_Reset_Indication

To responder

T_Cind	T_Connect_Indication
T_Dind	T_Disconnect_Indication
T_Cconf	T_Connect_Confirm
T_DTind	T_Data_Indication

Panel 2. Order of SEND and RECEIVE Events

```
TESTER                                   TEST RESPONDER
Send CR TPDU to IUT                      Receive T_CIND ASP from IUT
Send CR TPDU to IUT                      Send T_CREQ ASP
Receive ERR TPDU from IUT
Receive CR TPDU from IUT
```

It should also be noted that, if the IUT has multiple rendezvous in sequence with the tester or responder, the algorithm could be optimized to allow the COMPLETION message to be sent at the finish of the rendezvous sequence.

This deterministic execution testing approach is used to produce a selected synchronization sequence for testing communication protocols. However, this approach may also be useful for debugging protocols. By continuously replaying the syn-sequence, the deterministic execution testing algorithm will force the syn-sequence to be exercised repeatedly, which would allow for careful analysis of the execution history of the implementation under test.

Examples Using Transport Layer Protocol. The deterministic execution testing algorithms for communication protocols can be illustrated using the OSI transport layer class 0 protocol. The state diagram and the acceptable input and output messages are shown in Figure 3. From Figure 3, the following input test sequence is selected: T.CR, R.T_DREQ, T.CC, T.DT, T.DR, T.CR, T.CR, R.T_CREQ, T.DT, T.CR, R.T_CRESP, R.T_DTREQ, T.DT, R.T_DREQ, T.CR, T.CC, T.CR, T.DT, T.CR, T.DR, R.T_CREQ, T.DR, R.T_CREQ, T.CC, T.CR, T.CR, R.T_CRESP, T.DR, T.CR, R.T_CRESP, T.N_RIND, T.CR, R.T_CRESP, T.N_DIND.

This input sequence is provided to the control task. The output messages are not a mandatory input to the control process if the testers are simplified to become test drivers and the output messages are analyzed after the execution of the test sequence. Since the IUT is the receiver of each input message, an SR sequence for each input message is derived from the input message followed by IUT. From the above input sequence, an SR sequence would be constructed as (T.CR, IUT.T_Cind), (R.T_DREQ, IUT.DR),..., (T.N_DIND, IUT.T_Dind).

The 7th and 20th transitions in the above syn-sequence may create synchronization problems because of timing constraints. During the 6th transition the tester sends a Connect_Request (CR) protocol data unit (PDU) while the protocol is in state 1, and the responder waits for a T_connect_indication (T_Cind) abstract service primitive. Once the responder receives the T_Cind from the protocol, it is free to send a T_Connect_Request (T_Creq) to the IUT while the protocol is in state 2. However, if the message T_Creq arrives at the IUT earlier than the CR from the tester, then the selected syn-sequence will not be exercised as desired. A similar problem occurs at the 20th transition. Before the 20th transition starts, the protocol is in state 2. According to the SR sequence, the IUT is waiting for a Disconnect_Request PDU from the tester. It is possible that a Creq message from the responder arrives at the IUT earlier than a DR from the tester. However, these possible synchronization problems are solved by the deterministic execution testing algorithm because only one message is available to the IUT at any given time. This is enforced because of the PERMISSION and COMPLETION control statements inserted between each transition.

To illustrate, let's examine the first synchronization problem starting at the 6th transition and ending

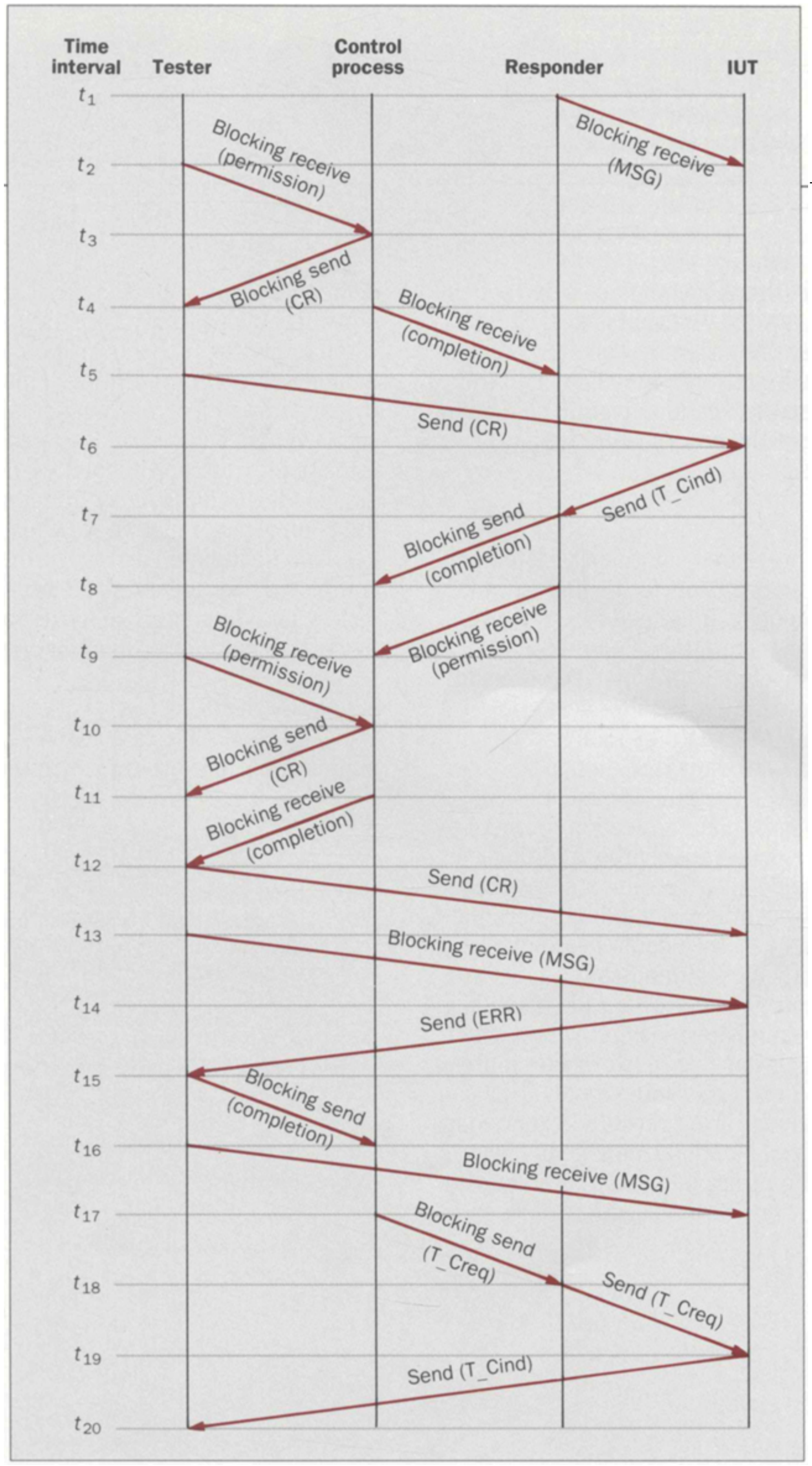

Figure 4. Process interactions for deterministic testing algorithm.

after the 8th transition in the SR sequence. The sequential order of `SEND` and `RECEIVE` events for each tester during these transitions is as shown in Panel 2. Assuming that the IUT cannot directly communicate with the controller, the deterministic execution testing algorithm described above will force the process interactions in Figure 4.

Conclusion

This paper has presented solutions to deterministic execution testing problems for testing communication software and protocols. It defined the synchronization problems in testing communication protocols modeled as finite state machines, provided solutions for deterministic execution testing of communication software with `SEND` and `RECEIVE` by mailbox modification and by program transformation, and described solutions to the deterministic execution testing problems for FSM-based communication protocols. The solutions were developed with C programming language. The source code and experimental results for the transport layer protocol are presented in Reference 4.

However, further research could be done to improve these deterministic execution testing tools to solve additional nondeterministic problems in protocol specifications. Nondeterministic behavior occurs when a protocol implementation is capable of exercising more than one transition upon receiving some input. This can result in different behaviors during repeated excursions of the same test sequence. Nondeterministic behavior also occurs when transitions are initiated upon receiving no input. This can occur on internal, spontaneous, or delay transitions. Since internal and spontaneous transitions may occur at any time, the control task must know when to expect such a transition and when to detect the completion of the transition. This could be accomplished by expanding the syn-sequence definition to include such events. Delay transitions are "time-dependent" transitions. If the protocol does not receive a message in X seconds, then the delay transition is executed. Again, the control task must know when a delay transition needs to be initiated and when the delay transition has completed. By solving these nondeterministic problems, testing and debugging of more protocol implementations by deterministic execution testing can be easily achieved.

Acknowledgment

I would like to thank K.-C. Tai for his advice and guidance in the work on which this paper is based.

References

1. A. A. S. Danthine, "Protocol Representation with Finite-State Models," *IEEE Transactions on Communications*, Vol. COM-28, No. 4, April 1980, pp. 632-642.
2. B. Sarikaya and G. V. Bochmann, "Synchronization and Specification Issues in Protocol Testing," *IEEE Transactions on Communications*, Vol. COM-32, No. 4, April 1984, pp. 389-395.
3. K.-C. Tai and S. Ahuja, "Reproducible Testing of Communication Software," *Compsac Proceedings*, Tokyo, October 1987, pp. 331-337.
4. Darrell Hubbard, *Deterministic Execution Testing of Communication Protocols*, M.S. thesis, North Carolina State University, Raleigh, May 1988.

(Manuscript received July 5, 1989)