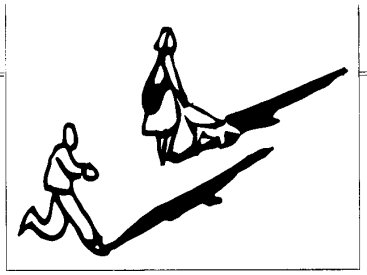


Testing Communication Protocols

KSHIRASAGAR NAIK, Concordia University, Montreal
BEHCET SARIKAYA, Bilkent University, Ankara

◆ *This model accommodates all conformance testing activities and lets you translate data and behavior into a common notation.*

Communication protocols are the rules that govern communication among components in a distributed system. Functions are usually partitioned into a hierarchical structure of protocol layers, as in the standard Open System Interconnection basic reference model.[1]

The box on pp. 30-31 is an overview of protocol design, implementation, and test concerns. Careful protocol description is important for many reasons. Early in the protocol's life cycle, descriptions give designers, who may be working on different protocol parts, a reference for cooperation and help them check the design for logical correctness. Later in the cycle, clear descriptions make it easy to check compliance across many implementations.

Informal techniques like narrative descriptions for protocol design and walkthroughs for protocol test are invaluable, but painful experience shows they are inadequate when used alone. Formal-specification methods from general software-engineering practices are also necessary.[2] These methods provide not only a more reliable way to verify the specification but also a method for partially automating protocol design, implementation, and conformance testing.

Specification languages like Lotos (Language for Temporal Ordering Systems),[3] Estelle, and SDL (Specification Description Language) have been developed for writing OSI protocols and services. Formal languages for specifying test suites include Abstract Syntax Notation-1, which describes data structures, and Tree and Tabular Combined Notation,[4] which describes test cases.

Unfortunately, these specification and test-suite languages (or notations) are incompatible. A Lotos specification's data is described in abstract data types, for example, while its behavior part is based on process algebra. TTCN is a tabular notation

0740-7459/92/0100/0027/$03.00 © IEEE

```
specification T_Protocol [U,L]no exit
  behavior TP_Simple[U,L]
  where
    process TP_Simple[U,L]: noexit :=

      U?TCONreq;L!CR (L?CC;L!DR;U!TDISind;L?DC;TP_simple[ts,ns]
                        [ ]
                        L?CC;U!TCONconf;Open[ts,ns]
                      )
      [ ]
      L?CR( L!DR;TP_simple[U,L]
            [ ]
            U!TCONind( Uclose[U,L]
                       [ ]
                       U?TCONresp;L!CC;Open[U,L]
                          )
          )
    endproc
    process Open[U,L]: noexit :=
      (U?TDATAreq;L!DT;Open[U,L]
        [ ] L?DT;U!TDATAind;Open[U,L]
        [ ] L?AK;Open[U,L]
        [ ] Uclose[U,L]
        [ ] Rclose[U,L]
        [ ] i;U!TDISind;L!DR;L?DC;Open[U,L]
        [ ] i;L!AK;Open[U,L]
      )
    endproc
    process Uclose[U,L]: noexit :=
      U?TDISreq;L!DR;L?DC;Open[U,L]
    endproc
    process Rclose[U,L]: noexit :=
      L?DR;U!TDISind;L!DC
    endproc
endspec
```

Figure 1. *A Lotos specification for a transport protocol.*

with data and behavior parts that are syntactically and semantically different from those in Lotos specifications. Thus you cannot directly compare a test case's behavior with the protocol specification to verify a test case.

We have developed a unifying, common intermediate model that lets you translate Lotos specifications and TTCN test suites into extended finite-state machines, providing a single medium for conformance testing. We chose to work with Lotos and TTCN because they are becoming increasingly important in the standardization of specifications. We chose an extended-finite-state-machine model because state machines are a proven technique for protocol modeling.

Data type declaration		
Protocol-data-unit name: CR PDU	Comments:	
Protocol-control information		
Field	Type	Comment
Called	T_suffix_type	
Calling	T_suffix_type	
Options	Options_type	
Credit	Credit_type	

(A)

Protocol-data-unit constraint		
Protocol-data-unit name: CR PDU	Constraint name C0:	
Field	Value	Comment
Called	"Server"H	
Calling	"User"H	
Options	expedited_data	
Credit	10	

(B)

Figure 2. ***(A)*** *Declaration of a protocol data unit (PDU) that is a connection request (CR) and* ***(B)*** *a declaration of a constraint on that request.*

SPECIFYING A PROTOCOL IN LOTOS

A Lotos specification consists of data and behavioral descriptions. Data is described in terms of abstract data types; behavior is described using the algebraic theory of processes, based on Milner's CCS (Calculus of Communicating Systems). Behavioral description is essentially a hierarchy of nested processes that interact using gates.

Figure 1 shows the behavior of a transport protocol (layer 4 of the OSI reference model) expressed in Lotos. For simplicity, we have omitted all data-type definitions.

Because Lotos is based on process algebra, it uses operators to express interprocess relationships. For example, a choice of the behavior expressions *B*1 and *B*2 is expressed as *B*1 [ ] *B*2. A parallel composition with no synchronization of *B*1 and *B*2 is expressed as *B*1 ||| *B*2. If *B*1 and *B*2 are two processes that interact through a list of gates *G*, the parallel composition of *B*1 and *B*2 is expressed as *B*1 |[G]| *B*2. The disabling of *B*1 by *B*2 is expressed as *B*1 [> *B*2, which means the activities of *B*2 can stop the operation of *B*1 at any time. A process's functions — the set of values it returns when it terminates — can be passed to another process using the enabling construct *B*1 >> *B*2. This means that once *B*1 terminates successfully, its functions are transferred to *B*2, and *B*2 is invoked.

Nondeterminacy is an important notion that distinguishes specification languages from programming languages. Lotos offers several facilities for specifying nondeterminate behavior. The main one is the choice ([ ]) operator, which is used with the internal event *i*.

Protocol designers can use *i* to represent many protocol features abstractly. Examples are the acknowledgment policies in a transport protocol — the reasons for sending it — and the unreliable nature of a protocol's underlying service provider. The Open process in Figure 1 consists of seven alternative behaviors. The behavior i;L!AK states that the acknowledgment policies are represented abstractly by an internal event.

Similarly, the behavior i;U!TDISind indicates that the underlying service pro-

vider cannot provide the desired service reliably all the time. Hence the transport protocol may issue a disconnect-indication event (TDISind) at any time. You can thus easily model loss, duplication, and message reordering in a specification that uses an unreliable medium.

SPECIFYING TEST SUITES IN TTCN

A TTCN specification consists of an overview, declarations, constraints, and dynamic behavior. The overview, written in English, describes the scope of the test suite. The declaration declares test-suite parameters, points of control and observation, protocol data units, abstract service primitives, and timers. Constraints let you specify values for each field of the protocol data unit and abstract service primitive.

Figure 2a shows a declaration of a protocol data unit that is a connection request. Figure 2b shows a declaration of a constraint on that request. A constraint table contains a constraint name for the abstract service primitive or protocol data unit and a list of parameter names and their values. The parameter values in a constraint on an abstract service primitive or protocol data unit are sent if the primitive or unit appears in a send event. The parameter values are the values received if the primitive or unit appears in a receive event.

The connection request in Figure 2a has four fields: Called, Calling, Options, and Credit. The constraint in Figure 2b defines the values of these four fields. If a connection request with the constraint appears in a send event in a test case's dynamic part, the test case sends a connection request that assigns the corresponding values to the parameters.

A test case's dynamic behavior specifies combinations of test-event sequences the test-suite specifier will allow. The events are combined as either sequences or sets of alternatives. A sequence of events is represented as one line after the other. Each new event is indented once from the left to represent the progression of time.

Figure 3a shows four event sequences: {*A, C, F, G*}, {*A, C, F, H*}, {*A, B, D*} , and {*A, B, E*}. However, {*A, C, B, D*} is not a correct sequence. Also, events *C* and *B* constitute a set of alternative events; either *C* or *B* occurs but both do not occur at the same time. Events *F, D,* and *E* do not form a set of alternative events because they do not have the same predecessor event; the predecessor of *F* is *C*, and the predecessor of *D* and *E* is *B*.

Figure 3b shows a test-case description structured as a tree. Test events are nodes; verdict assignments are leaves. (The box on pp. 30-31 describes how verdicts are used in test verification.) Test events with the same indentation belong to the same predecessor event. These represent the alternative events that could occur at that time. Alternative test events are specified in the order in which the tester will repeatedly attempt them until one occurs. You can trap all the undesired external input events by specifying an Otherwise event as an alternative to the desired events. In a sequence of alternative events, an Otherwise will be the last event or the event just before a Timeout.

You can also structure a test case by using test steps, as Figure 3c shows, where Step is a test step. The description of the structured test case in Figure 3c is shown as a structured tree in Figure 3d.

Every test case has an associated purpose. The correctness of test-event sequences, the correctness of test verdicts, and the purpose of a test case are closely related. In the test-case model based on extended finite-state machines, you express the test purpose using a regular expression on the interactions between the test case and the protocol specification.

To illustrate how a test case works, we chose a test case from a real test suite developed at the National Computing Center in Manchester.[5] The test case, which is shown in Figure 4, is for the basic interconnection testing of a Class 2 transport protocol. As the figure shows, the test case starts with lower tester *L* sending a connection request to an implementation. Then *L* sends a data protocol data unit if the connection request is accepted. After it receives an acknowledgment protocol data unit from the implementation, it disconnects the connection. In this example, the test purpose is denoted by {L!CR. U?TCONind. U!TCONresp. L?CC. L!DT. U?TDATAind. L?AK. L!DR. U?TDISind. L?DC}.

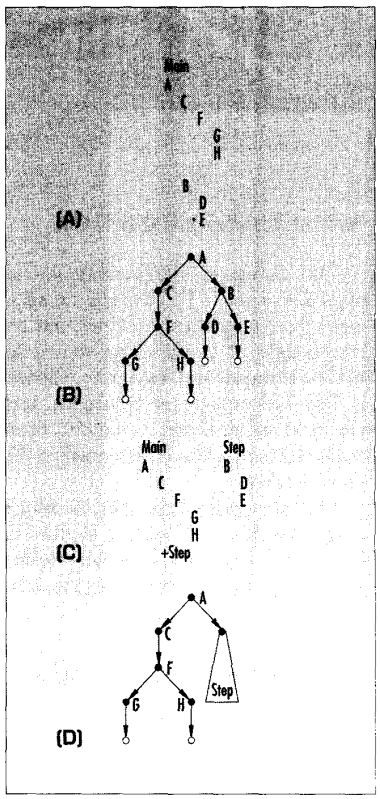


Figure 3. (A) A test case, (B) a test case in tree notation, (C) a structured test case with a test step, and (D) a structured test case in tree notation. Solid dots denote events; open dots denote verdicts.

```
L!CR
  Start(timer A)
    U?TCONind
    | U!TCONresp
    |  L?CC
    |   | Cancel(timer A)
    |   |  L!DT
    |   |    U?TDATAind
    |   |     | L?AK
    |   |     | | L!DR
    |   |     | |       U?TDISind
    |   |     | |       | L?DC
    |   |     | |       | L?OTHERWISE
                                  Fail
    |   |     | |       U?OTHERWISE
                                  Fail
    |   |     | L?OTHERWISE  Fail
    |   |     U?OTHERWISE  Fail
    |   ?Timeout(timer A)      Fail
    |   L?OTHERWISE            Fail
    L?DR                  Inconclusive
    U?OTHERWISE             Fail
```

Figure 4. A sample test case.

TRANSLATING SPECIFICATIONS TO CIM

Our common intermediate model maps Lotos and TTCN specifications to extended finite-state machines and accommodates all conformance-testing activities. Protocol and test-entity behavior is described as a set of transitions, each triggered by an input, an output, or an internal event. Data is described in abstract data types.

CIM's extended finite-state machine is a six-tuple: $M = <J, V, j_0, J_f, R, \varepsilon>$, where J is a set of states, V is a set of data declarations (variables), j_0 is the initial state, J_f is a set of final states, R is a set of transitions, and ε is a set of initial value assignments to some variables in V.

A transition of an extended finite-state machine is also a six-tuple: $r = <j, j', a, p, f, pr>$, where j is the From state of r, j' is the To state of r, a is an action (the event clause of r), p is the provided clause of r, f is the assignment clause of r, and pr is the priority number of the transition.

Figure 5 illustrates the execution semantics of a transition. The Begin ... End block in a CIM transition contains only assignment statements, no predicate or output events.

Translating a Lotos specification and a TTCN test suite involves

1. Translating the behavior part of a Lotos specification into an extended finite-state machine.
2. Translating the TTCN declarations and constraints to abstract data types.
3. Translating TTCN dynamic behavior to an extended finite-state machine.

The data part of a Lotos specification is already described in abstract data types, so it does not require translation.

Translating Lotos into an EFSM. Translating a Lotos specification into an extended fi-

PROTOCOL DESIGN, IMPLEMENTATION, AND TEST

Because protocol systems are not the same as traditional software systems, they have special design and implementation concerns. Traditional systems consist of functions that go from an initial state to a final state. Systems accept all input at the beginning of their operations and yield their output at termination.

These systems are called *transformational* because they transform an initial state to a final state. Typical examples are batch, off-line data processing, and numerical packages.

But some systems, like operating systems and process-control systems, may never terminate. These are called *reactive* systems.

The purpose of running reactive systems is not to get a final output, but rather to maintain some interaction with the system's environment. A reactive system is not restricted to accepting input on initiation and generating output on termination. Some of its input depends on intermediate output.

Thus, you cannot adequately specify reactive systems by referring only to their initial and final states. Instead you must refer to their continued behavior, which may be an infinite sequence of states and I/O events.

Communication-protocol systems are reactive systems with several unique characteristics: Each protocol input may not have a corresponding output, and one input may have many outputs.

The correctness of a protocol's output depends on the values of its preceding output. Therefore, a protocol system may be nondeterminate.

Protocol design, implementation. Protocol specification is based on the communication service to be provided and the communication service that the protocol will use. For the most part, protocol design is intuitive. Designers must also check the protocol specification to ensure that it is correct and consistent, provides the desired communication service, and offers services with acceptable efficiency.

The protocol must satisfy the rules in its specification as well as any constraints in its environment. Implementation, which can be in hardware, software, or a combination, consists of generating a concrete, executable representation from an abstract, formal description.

The process is largely manual and involves design decisions like how to support mandatory, optional, and negotiable features, including the choice of addressing mode and timer values.

The decisions made depend on performance requirements and the protocol's runtime environment.

Conformance testing. When a protocol implementation conforms with its specification, it behaves according to the specification's rules. The most feasible way to determine conformance is to test the implementation with a set of test cases.

Conformance testing is important, because it ensures that independently generated implementations of the same protocol can work with each other. Figure A shows the conformance testing activities.

Conformance testing starts with test design, in which you develop an abstract test suite. A test suite is a collection of test cases, each of which is used to test one protocol feature. Test-suite development is either manual or semiautomatic.

Manual design is the conventional way to design a test suite. The designer usually has expertise in the protocol standard and is familiar with the framework and methodology of conformance testing. But manual design is complicated and error-prone.

Test generation. Most test generators for transformational systems use symbolic evaluation to derive test data. But because a protocol system is reactive, it requires different techniques. The technique used depends on the protocol-specification model.

Ideally, test generation should be fully automatic, and much research is being done on automatic techniques. However, software testing in general and protocol testing in particular have not yet reached a state that lets you automatically generate a set of readily

nite-state machine[6] consists of

- Normalizing the Lotos specification by transforming an enable operator to a semantically equivalent parallel operator. (An interleaving parallel operator is transformed into a generalized parallel operator.) In this step, all process references are expanded by their definitions in a top-down manner until a process reference appears in its expanded form in instances of indirect recursions. All variables are uniquely renamed.
- Translating the normalized specification to an extended finite-state machine by applying a transformation rule to each

```
From j
To j'
Priority pr              /* pr is the priority of the transition. */
Event a(parameters)      /* a is an event possibly with a set of parameters. */
Provided p               /* p is a predicate. */
Begin {
      f                  /* Sequence of assignments. */
      }
End
```

Figure 5. Structure of a transition in a CIM extended finite-state machine.

usable test cases from a specification. Instead, semiautomatic test generation is used.

Test verification. A communication protocol is made up of event sequences, which appear at its service-access points. Events consist of input events to the protocol and output events that are the protocol's responses.

The test case checks whether the implementation satisfies the protocol specifications and the test purpose. It does this by applying sequences of input events that the protocol specification allows.

If the implementation behavior is allowable and the test purpose is satisfied, the test case assigns a Pass verdict. If the implementation behavior is not allowable, it assigns a Fail verdict. If the implementation behavior is allowable but the test purpose is not satisfied, it assigns an Inconclusive verdict.

Therefore, a test case's verdict of correct or incorrect conformance depends on the correctness of the input-event sequences and the expected protocol events.

To verify a test case, you compare its behavior with the protocol's behavior and determine any design errors in the test case.

Test selection and parameterization. The protocol may not support all the functions or features stated in the specification, so only a subset of the entire protocol test suite may apply. Extracting that subset is called test selection. Test parameterization is assigning values to the test-suite parameters.

Test execution. To test an implementation, you generate executable test cases from selected and parameterized test cases. The test-execution system must let a test case and an implementation communicate, provide a way to monitor test execution, and provide a mechanism to record a test case's execution history for further analysis. In this way, you can trace test cases in which the implementation fails against a formal protocol specification and more effectively diagnose the reason for failure.

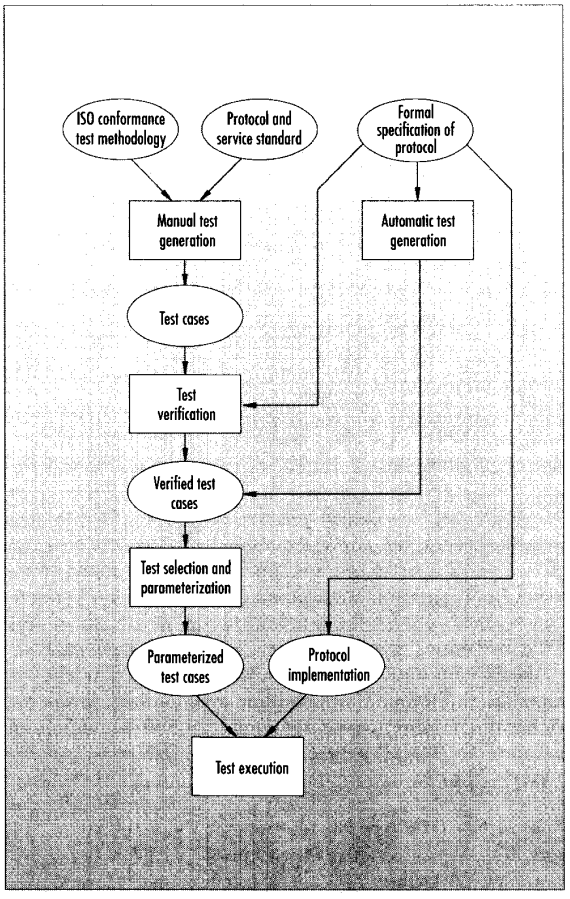


Figure A. Using formal approaches in conformance testing.

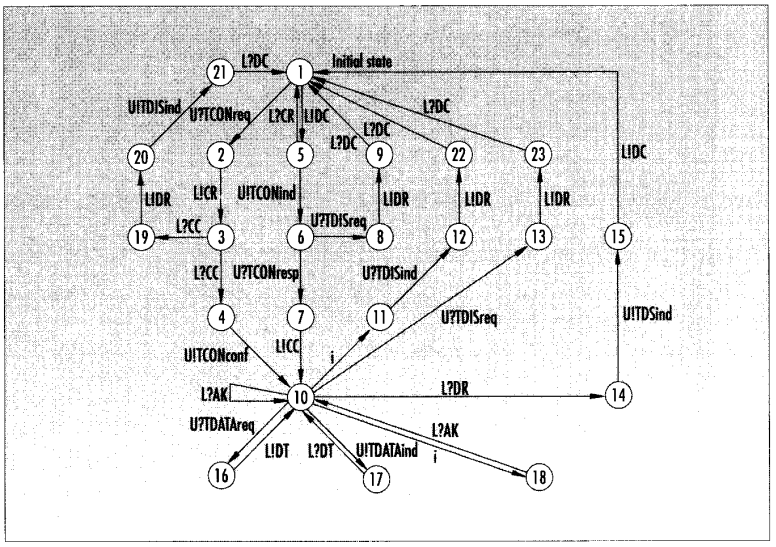


Figure 6. *The extended finite-state machine for the transport-protocol specification in Figure 1; State 1 is both the initial and final state.*

```
type CR_PDU is T_suffix_type,options_type,credit_type,pdu
  sorts CR_PDU
  opns  CR_PDU: T_suffix_sort,T_suffix_sort,options_sort,credit_sort -> pdu
        TSAP_id_called      : pdu -> T_suffix_sort
        TSAP_id_calling     : pdu -> T_suffix_sort
        options_ind         : pdu -> options_sort
        credit              : pdu -> credit_sort
eqns forall called: T_suffix_sort, calling:T_suffix_sort,
              options:options_sort, credit:credit_sort
  ofsort T_suffix_sort
    TSAP_id_called(CR_PDU(called,calling,options,credit)) = called;
    TSAP_id_calling(CR_PDU(called,calling,options,credit)) = calling;
  ofsort options_sort
    options_ind(CR_PDU(called,calling,options,credit)) = options;
  ofsort credit_sort
    credit(CR_PDU(called,calling,options,credit)) = credit;
    IsCR_PDU(CR_PDU(called,calling,options,credit)) = true;
endtype
```

Figure 7. *An abstract-data-type representation of the connection request in Figure 2a.*

Lotos operator. Conceptually, each transformation rule derives all possible sequential behavior from two operand behaviors related by the operator. You need transformation rules only for the operators not eliminated during normalization.

We have implemented this translation algorithm in Prolog. The algorithm applied to the transport protocol in Figure 1 generates the extended finite-state machine in Figure 6.

Translating TTCN data into abstract data types. TTCN uses a number of predefined data types, like integer, Boolean, bit string, hex string, octet string, and character string. Test cases have two frequently used classes of structured data types: the abstract service primitive and the protocol data unit. In any test architecture, the testers and the protocol entity communicate among themselves by exchanging abstract service primitives, some of which may contain protocol data units. In TTCN, abstract service primitives and protocol data units are in tabular form. To be able to compare the data values generated by a test case with those accepted by a protocol specification, you must translate the TTCN test case's tabular descriptions of the abstract service primitives and protocol data units to abstract data types.

Figure 7 is an example of how to translate the tabular declaration of the connection request (protocol data unit) in Figure 2a to an abstract data type.

Given a protocol data unit's individual fields, you should be able to construct it. Conversely, given the protocol data unit, you should be able to identify its type and extract its individual fields. Therefore, to translate a protocol data unit into an abstract data type, you define one constructor operator to construct the protocol data unit, one operator to identify the unit type from a constructed protocol data unit, and one selector operator to select each field in the protocol data unit.

Translating TTCN behavior to an EFSM. You can algorithmically derive an extended finite-state machine from the dynamic part of a TTCN test case.[7] Translation consists of expanding default behaviors, deriving an extended finite-state machine from the main tree and each subtree, and resolving subtree attachments by combining corresponding extended finite-state machines. Each event line in a test case is modeled as a transition in extended-finite-state-machine notation.

For example, let

L!CR_PDU [initiate == true]
(called = "him", calling = "me",
options = "nil", credit = 10)

be the first event line in a set of alternatives in a TTCN test case. Initiate is a test-suite parameter such that if its value is true, the

```
/* Let the transition occur from
   state M to N. */
From M
To   N
Priority 1
Event L!CR_PDU(called, calling,
   options, credit)
Provided [initiate == true]
Begin {
        called = "him",
        calling = "me",
        options = nil,
        credit  = 10
      }
```

Figure 8. *Transition of an event line to CIM's extended-finite-state-machine notation.*

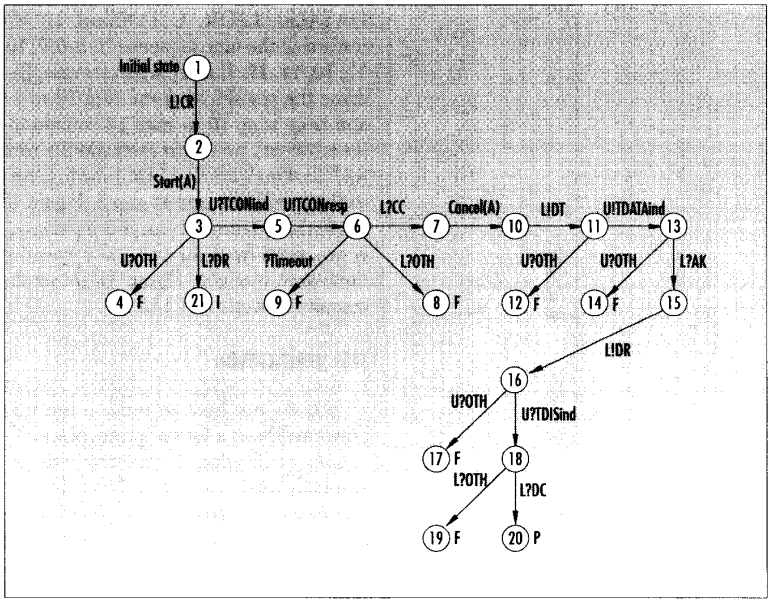


Figure 9. The extended finite-state machine for the test case in Figure 4 (generated from TTCN); OTH = Otherwise, P = Pass, F = Fail, I = Inconclusive.

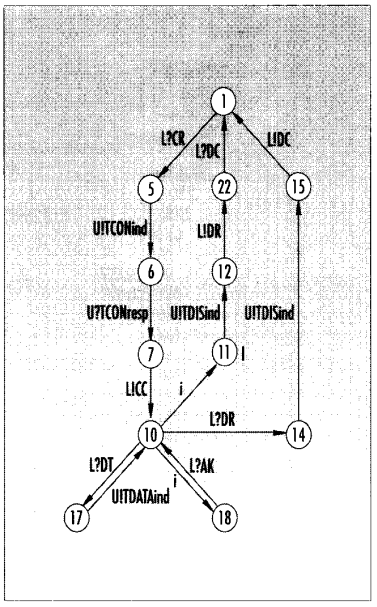


Figure 10. A test-case skeleton generated from the protocol specification.

test case establishes a connection with the implementation by sending a connection request (protocol data unit). Figure 8 shows the transition of this event line to an extended-finite-state-machine notation.

Figure 9 shows the test case in Figure 4 represented in extended-finite-state-machine notation.

TEST GENERATION

After you map a Lotos specification to an extended finite-state machine, you must derive test cases from it. The first step is to algorithmically generate the test-case control structures needed to test a desired protocol behavior. These structures are called test-case skeletons. The second step is to manually transform test-case skeletons into complete TTCN test cases. The number of test-case skeletons derived depends on the number of transitions e and the number of states n in the protocol's extended finite-state machine. The upper bound is given by the well-known cyclomatic complexity $e - n + 2$.

Generating test-case skeletons. If a protocol model is determinate, then each test case consists of a test-event sequence derived by some well-established test-generation techniques.[5] If a protocol model is nondeterminate, however, you cannot represent a test case with a pure event sequence. To correctly judge the behavior of such a protocol, a test case must also account for the expected alternative behavior.

Thus, the test-generation algorithm consists of the following steps. Input is the extended finite-state machine generated from the Lotos specification, and output is the test-case skeletons in the form of extended finite-state machines.

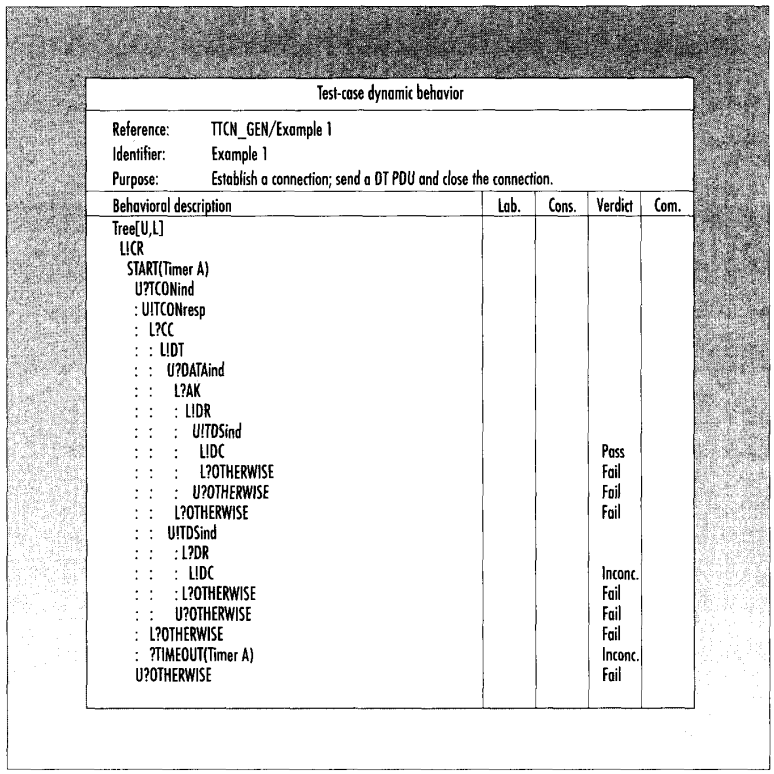
1. In the given extended finite-state machine, generate a new sequence of transitions starting and ending in the initial state. We call this sequence a partial test case.

2. In the extended finite-state machine, if there is a transition r having an internal event that is an alternative to a transition in the partial test case, then update the partial test case by adding a transition sequence such that the sequence starts with r and ends in a state belonging to the partial test case. Then repeat this step or go to step 3.

3. The partial test case is a test-case skeleton. If all test-case skeletons are generated, then stop. Otherwise go to step 1.

Transforming test-case skeletons into TTCN test cases. To derive a meaningful test case from a skeleton test case, you have to modify the skeleton test case's behavior to meet a test purpose and add two other behaviors, Timeout and Otherwise. In real-time systems, the notion of a timeout is well-known. But you cannot automatically derive timer information from a protocol specification. A test case must have an Otherwise event as an alternative to any receive event to trap unexpected events from an incorrect implementation.

When a test case is transformed into TTCN, all the input and output transitions in the test case's extended finite-state machine are transformed into output and input transitions in TTCN. An event sequence in the test case's extended finite-



Test-case dynamic behavior

Reference: TTCN_GEN/Example 1
Identifier: Example 1
Purpose: Establish a connection; send a DT PDU and close the connection.

Behavioral description	Lab.	Cons.	Verdict	Com.
Tree[U,L]				
L!CR				
START(Timer A)				
U?TCONind				
: U!TCONresp				
: L?CC				
: : L!DT				
: : U?DATAind				
: : L?AK				
: : : L!DR				
: : : U!TDSind				
: : : L!DC			Pass	
: : : L?OTHERWISE			Fail	
: : : U?OTHERWISE			Fail	
: : L?OTHERWISE			Fail	
: : U!TDSind				
: : : L?DR				
: : : L!DC			Inconc.	
: : : L?OTHERWISE			Fail	
: : U?OTHERWISE			Fail	
: L?OTHERWISE			Fail	
: ?TIMEOUT(Timer A)			Inconc.	
U?OTHERWISE			Fail	

Figure 11. A test case in TTCN.

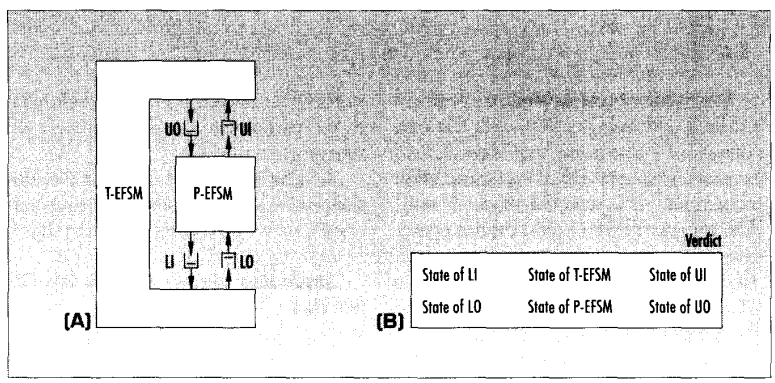


Figure 12. (A) The test-verification system for the local single-layer test architecture and (B) the representation of the global state, which is a six-tuple.

state machine is transformed into a sequence of left-indented, with each indent events, representing the progression of time. A set of alternative transitions is represented as a set of alternative event lines. Loops in the extended finite-state machine are suitably represented by goto loops in TTCN. A Pass verdict is assigned to the final event in the intended path, and Inconclusive verdicts are assigned to the other paths derived from the protocol's extended finite-state machine. Fail verdicts are assigned to Otherwise events.

A sample test-case derivation. To illustrate how to derive a test case, we apply the algorithm for test generation to the extended finite-state machine in Figure 6. As the figure shows, state 1 is the initial and final state. Applying step 1, you select the sequence {L?CR, U!TCONind, U?TCONresp, L!CC, L?DT, U!TDATAind, L?DR, U!TDISind, L!DC} traversing the state sequence {1, 5, 6, 7, 10, 17, 10, 14, 15, 1} as a partial test case. Because the transitions from state 10 to 11 and from state 10 to state 18 contain internal events, you must then add the path {i,U!TDISind, L!DR, L?DC} and {i, L!AK} to the partial test case as step 2. Figure 10 shows the skeleton test case after its behavior is generated from the protocol's extended finite-state machine. Figure 11 shows the complete test case in TTCN.

TEST VERIFICATION

You do not have to verify a test case generated from a formal protocol specification; verification is required only for manually written test cases. To verify these test cases, you analyze the global states arising from the event sequences. The global states represent all possible interactions between a test case and the protocol specification. The goal is to show that the global system's state space has no errors like unspecified receptions, blocking receptions, deadlocks, livelocks, channel-overflow errors, and synchronization errors. Also you have to establish that the verdicts attached to the final global states are correct.

You can easily generate the global state space that contains the interaction sequences between the test and protocol specifications by using traditional reachability analysis. But you cannot use this method to verify test cases. You need some way to verify the test case's dynamic properties that arise from timers, special test behavior due to Otherwise events, and information about test verdicts. To generate global behavior, we have extended the traditional reachability algorithm by adding a verdict-verification phase.

In the extended algorithm, verification consists of two distinct phases. First, the algorithm generates a global state space using extended reachability analysis. Second, it analyzes the global state space to verify test-event correctness and verifies the verdicts attached to the final global states.

The time complexity of generating the reachability graph of an extended finite-

state machine generated from a Lotos specification, called P-EFSM, with n_p states and an extended finite-state machine generated from TTCN, called T-EFSM, with n_t states is

$$O(n_p \times n_t \times c \times (E_{mp} + E_{mt}))$$

where c is all the channel capacities, E_{mp} is the maximum number of executable transitions in each P-EFSM's state, and E_{mt} is the maximum number of executable transitions in each T-EFSM's state.

Thus, the test-verification algorithm has as its input T-EFSM, P-EFSM, and the first-in, first-out channels and their capacities.

Generating a global state space. To illustrate how to generate the global state space, we chose test cases written for the local single-layer test architecture. Figure 12a shows a test-verification system, in which a test case is modeled as a T-EFSM, the protocol specification as a P-EFSM, and the upper points of control and observation (PCO (U)) by two FIFO channels, UI and UO, and the lower points of control and observation (PCO (L)) by two FIFO channels, LI and LO.

Figure 12b shows the structure of a global state, which is a six-tuple. If the T-EFSM's state has a verdict tag, the global state also gets a verdict tag. The contents of a FIFO channel represent its state.

The following algorithm generates a global state space using state perturbation. A global state is perturbed when a new global state has been generated. A new global state is generated when *one* transition is executed in any of the test-verification system's extended finite-state machines.

1. Define a set of global states G that has only the initial state g_1, which consists of the initial states of the T-EFSM and the P-EFSM and all empty channels.
2. Find g_p, a member of G that has not been marked 'perturbed." If no such state exists, then stop.
3. Compute G_p, the set of global states that can be reached by perturbing g_p using all the executable transitions in the two machines in their respective present states. If G_p is an empty set, then g_p is a final state.
4. Mark all the states in G_p, denoting

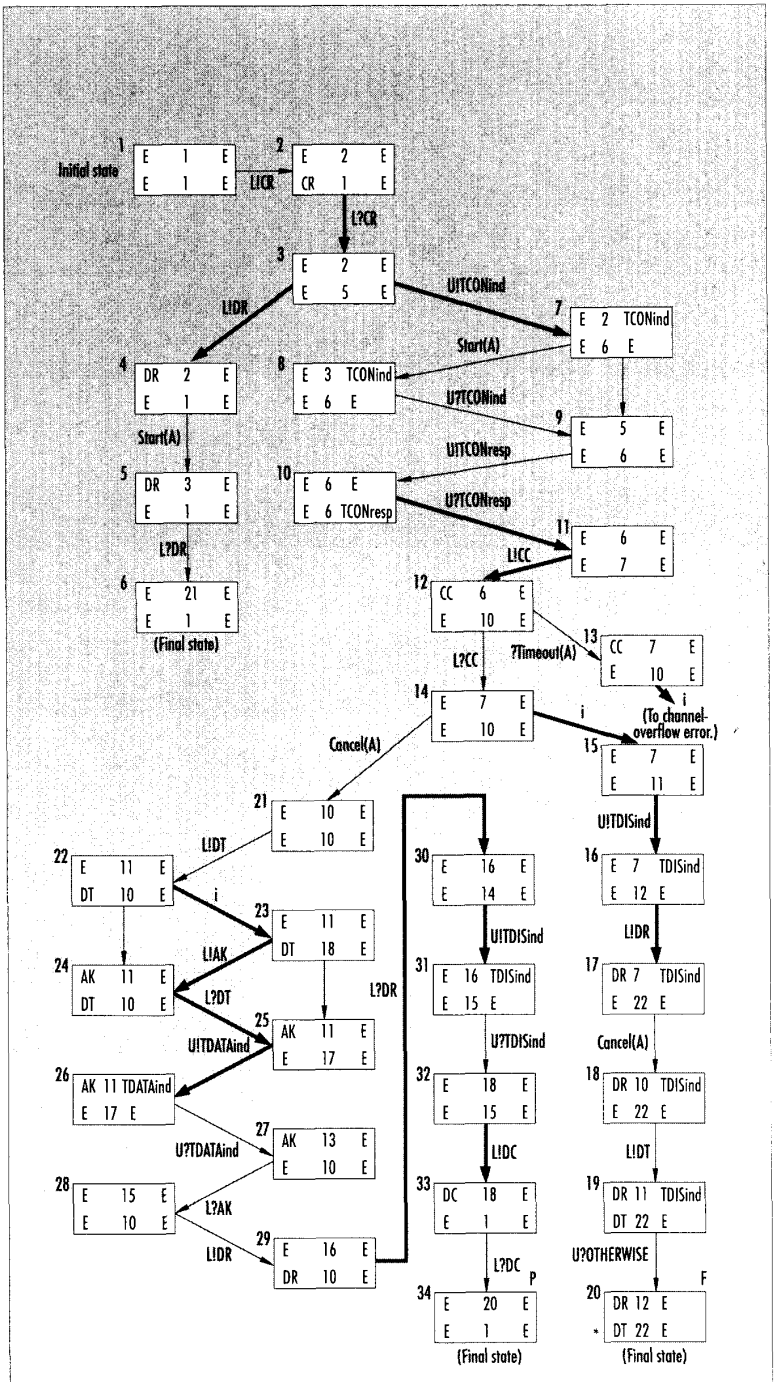
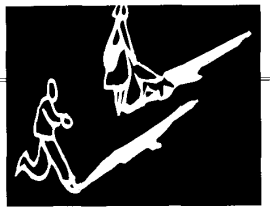


Figure 13. *Part of the global state space generated from the extended finite-state machines in Figures 6 and 9. Normal weight arrows indicate a transition in the protocol machine, while bold arrows indicate a transition in the test machine. An asterisk marks an erroneous state; E = empty channel, P = Pass, F = Fail, I = Inconclusive.*



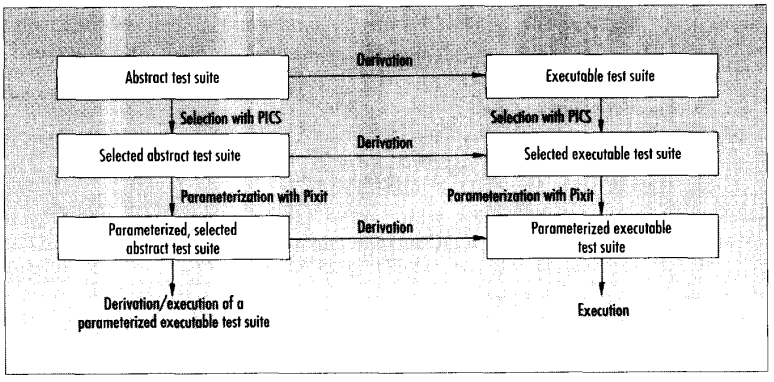


Figure 14. Steps to derive an executable test suite.

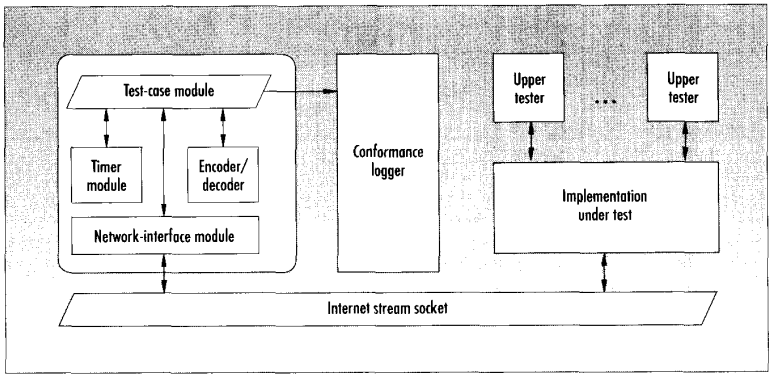


Figure 15. A test-execution system.

channel-overflow error as "perturbed," and add to G the remaining states of G_p not already in G.

5. Go to step 2.

Verifying test verdicts. Each final global state must have either a Pass (P), Fail (F), or Inconclusive (I) verdict. The box on pp. 30-31 defines the conditions for P, F, and I. Because an erroneous final state means there is a test-event error, the verdict associated with that state is not necessarily valid. However, assume that a protocol specification has only valid behavior if a P is attached to an error-free state. The verdict assignment is then correct if the event sequence leading to that state satisfies the test purpose. Conversely, if an I is attached to the error-free state, it is correct if the event sequence leading to that state does not satisfy the test purpose. No F can be attached to any error-free global state.

To illustrate verification, we verified the simplified state machine in Figure 9 against that in Figure 6. For these extended finite-state machines, we do not consider the parameters and the predicates. We assume that the capacity of all FIFO channels is two.

Figure 13 shows part of the global state space. Its analysis indicates that the test case has a few design errors. For example, state 16 represents an unspecified reception error, and state 20 represents a blocking-reception error. These errors arise because the test case does not provide a way to receive a nondeterminate disconnection-indication event TDISind. The state space also contains a few channel-overflow errors, which are due to the nondeterminate acknowledgment (protocol data unit), but they are not shown in the figure.

Because state 20, the final state, is erroneous, we do not have to verify the F attached to it. The two error-free final states in the state space are 6 and 34. Because the event sequence from the start state to the error-free state 34 satisfies the test purpose of {L!CR. U?TCONind. U!TCONresp. L?CC. L!DT. U?TDATAind. L?AK. L!DR. U?TDISind. L?DC}, the P attached to state 34 is correct. Moreover, because the event sequence leading to the error-free state 6 does not satisfy the test purpose, the I attached to state 6 is also correct.

TEST SELECTION

Figure 14 shows the possible temporal relationships among test selection, test parameterization, and derivation of executable test cases within the OSI conformance-testing framework.

In reality, a protocol specification provides several communication functions with many mandatory and negotiable optional and alternative features in terms of protocol behavior, protocol parameters, and quality of service. However, a protocol implementation need not support all the communication function types stated in the specification. The implementer provides a statement to the test laboratory, called the Protocol Implementation Conformance Statement, which identifies the features the implementation does or does not support. The test designer then uses PICS pro forma to select the test cases to be applied to the implementation under test.

TEST PARAMETERIZATION

Many protocol features are not explicitly stated in the specification, but are determined during protocol implementation. Examples of such features are addressing information, timeout intervals, and number of connections supported by a connection-oriented protocol. The implementer provides another statement, called Pixit (short for Protocol Implemen-

tation Extra Information for Testing), that tells the values of the implementation-dependent parameters. This information is then used to assign values to test-suite parameters.

TEST EXECUTION

An important step in test execution is translating abstract test cases to an executable form, an activity referred to as *derivation* in the conformance-testing standard. As Figure 14 shows, you can do derivation in several ways.

Automatic translation of TTCN test cases to an executable form is attractive — to implementers for debugging and development and to test laboratories for conformance testing — and several widely varying techniques are in use.[8]

CIM also provides a way to automatically translate TTCN test cases to an executable form because you can use it as a test-execution language. Abstract data types can be stated using Backus-Naur Form notations, so tools to manipulate abstract data types are already in use, and the operational semantics of an extended finite-state machine contains only a few very simple programming constructs. Thus, you can express a test case's extended finite-state machine using a simple BNF grammar and implement a tool for translating an extended finite-state machine to C to generate an executable version of a test case.

A conformance test system contains modules for encoding or decoding protocol data units, logging all test events, and interfacing with the implementation, possibly through the underlying service. These modules are independent of the test cases. Executable test cases, on the other hand, are the test system's core.

Figure 15 shows a prototype test-execution system for the coordinated single-layer test architecture running in the Unix environment.

Because formal specification languages are used in protocol and test specifications, you can algorithmically analyze many aspects of conformance testing without ambiguities. Although we considered Lotos as the protocol specification language and TTCN as the test specification language, the method outlined will work for other languages, like Estelle and SDL, because the algorithms to translate protocol specifications in those languages to the extended finite-state machine notation are straightforward.

More research must be done in modeling test purposes using temporal logic and parameterizing and selecting test cases using formal definitions of PICS and Pixit. To realize the potential of a conformance log in implementation debugging, we need a more precise and structured definition of the logging mechanism. ♦

ACKNOWLEDGEMENT

This research has been partially supported by the Natural Sciences and Engineering Research Council of Canada. Naik's work is financially supported by the Canadian Commonwealth Fellowship Agency.

REFERENCES

1. *ISO Open System Interconnection — Basic Reference Model*, ISO/IEC IS 7498, American Nat'l Standards Inst., New York, 1984.
2. *IEEE Software*, special issue on formal methods, Sept. 1990, pp. 7-67.
3. T. Bolognesi and E. Brinksma, "Introduction to ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, Jan. 1987, pp. 25-59.
4. OSI Conformance Testing Methodology and Framework: ISO/IEC DIS9646 parts 1 to 6, I.
5. *Abstract Test Suite for Transport Protocol Class 2*, Nat'l Computing Center Ltd., Manchester, 1988.
6. P. Tripathy and B. Sarikaya, "Test Generation from LOTOS Specifications," *IEEE Trans. Computers*, Apr. 1991.
7. K. Naik and B. Sarikaya, "An Extended Finite State Machine Model for TTCN," *Proc. Symp. Communications*, 1990, pp. 296-299.
8. S. Eswara, et al., "Towards Execution of TTCN Test Cases," *Proc. IFIP WG 6.1 Symp. Protocol Specification, Testing, and Verification*, 1990, pp. 99-112.

Kshirasagar Naik is a PhD candidate at Concordia University, Montreal, where his research interests include protocol verification and testing based on formal specifications.

Naik received a bachelor's degree from Sambalpur University, India, and a masters degree from the Indian Institute of Technology, Kharagpur — both in electrical communication engineering. He also received a masters degree in computer science from the University of Waterloo.

Behcet Sarikaya is an associate professor in the computer engineering and information sciences department at Bilkent University in Ankara, Turkey. His research interests are all aspects of conformance testing and high-speed networks.

Sarikaya received a BSEE with honors from the Middle East Technical University in Ankara, Turkey; an MSc in computer science from METU; and a PhD in computer science from McGill University, Montreal.

He is an IEEE senior member and an ACM member. He has published more than 40 papers, most on communication protocols.

Address questions about this article to Sarikaya, Dept. of Computer Engineering and Information Sciences, Bilkent University, Bilkent, Ankara 06533 Turkey; Internet sarikaya%trbilun.bitnet@ cunyvm.cuny. edu or Bitnet sarikaya@trbilun.bitnet.