Formal Methods in System Design, 2: 277–321 (1993)

Test Case Verification by Model Checking

KSHIRASAGAR NAIK
Concordia University, Dept. of Elect. and Computer Eng., 1455 de Maisonneuve West, Montreal, CANADA, H3G 1M8

BEHCET SARIKAYA
Bilkent University, Dept. of Computer Eng. and Info. Science, Bilkent, Ankara 06533, Turkey

Abstract. Verification of a test case for testing the conformance of protocol implementations against the formal description of the protocol involves verifying three aspects of the test case: expected input/output test behavior, test verdicts, and the test purpose. We model the safety and liveness properties of a test case using branching time temporal logic. There are four types of safety properties: transmission safety, reception safety, synchronization safety, and verdict safety. We model a test purpose as a liveness property and give a set of notations to formally specify a test purpose. All these properties expressed as temporal formulas are verified using model checking on an extended state machine graph representing the composed behavior of a test case and protocol specification. This methodology is shown to be effective in finding errors in manually developed conformance test suites.

Keywords: reachability analysis, extended finite-state machines, temporal logic, Estelle, TTCN, model checking, safety properties, liveness properties

1. Introduction

Communication protocols are the rules that govern communication among components in a distributed system. The four steps in a protocol development process, called *protocol engineering lifecycle*, are design of a protocol specification, validation and verification of the specification, generation of an implementation from the specification, and conformance checking of the implementation [1]. Conformance of a protocol implementation with its specification means that the implementation under test (IUT) behaves according to the rules described in its specification. In the open systems interconnection (OSI) framework [2], checking the conformance of an implementation to the corresponding protocol specification is done by *testing* the implementation using a set of test cases. The testing activity is important, because it ensures that one independently generated implementation of the same protocol can interwork with another.

In the protocol development process, the use of formal description techniques (FDTs) enables protocol designers to verify and validate specifications, and design and verify test cases. Standards organizations, such as the International Organization for Standardization (ISO) and the International Consultative Committee for Telephone and Telegraph (CCITT) have defined various FDTs such as LOTOS [3, 4], Estelle [5, 6], and SDL [7] to specify protocols; ASN.1 [8] for

defining a data transfer syntax; the Tree and Tabular Combined Notation (TTCN) [2] to specify test suites; and test architectures [2] for executing the test cases.

There are two main approaches to designing a test suite: semiautomatic and human design. In the semiautomatic approach, the formal specification of a protocol is used as a basis for generating test cases. There are a number of test design techniques [9, 10] which algorithmically generate test cases from deterministic finite state machine (FSM) models of protocols. However, generating readily usable complete test cases from other FDTs, such as LOTOS, Estelle, and SDL, is more difficult, and research until now has produced only a partial solution [11, 12]. The limited research that has been reported so far deals only with the local single-layer (LS) test architecture. Semiautomatically generating test cases for more useful test architectures such as the distributed single layer (DS), the coordinated single layer (CS), and the remote single layer (RS), requires more research to be done. Moreover, in the semiautomatic approach, no technique has yet been reported to generate classes of special test cases required to check the multiple-connection support and robustness capabilities of implementations.

Therefore, traditionally, a test suite is designed by a team of designers having expertise in protocol standards and test architectures [13]. Such a human-designed test suite has three advantages over a semiautomatically designed one. First, a test case can be designed with a specific *test purpose*. Second, test cases can be grouped into various categories, such as basic interconnection tests, capability tests, valid behavior tests, robustness/invalid behavior tests, multiple-connection support tests, etc. Finally, test cases can be manually designed for all four basic test architectures LS, DS, CS, and RS. However, the main disadvantage of human-designed test cases is that those test cases can be error-prone.

In the absence of any complete technique to generate test cases from formal description languages used in specifying protocols, a good alternative is to design test cases manually and use a methodology to verify the correctness of those test cases against the reference formal protocol specification. In this paper, we focus on developing a methodology to verify human-designed test cases.

To verify the correctness of a system, one must verify that the system satisfies its *safety* and *liveness* properties. Safety properties state that something *bad* never happens and liveness properties state that something *good* eventually does happen. In order to understand the motivations for characterizing the correctness of test cases in terms of safety and liveness, it is important to understand the difference between traditional program testing and protocol testing [1].

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 numeric computational packages. In transformational systems, a test case consists of a pair of (input, output) [14], where the input is given to the system at the

beginning of its execution and the output is the desired output of the system on its termination.

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, one cannot adequately specify reactive systems by referring only to their initial and final states. Instead one must refer to their continued behavior, which may be a very long sequence of states and input/output 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, the structure of a test case for testing communication protocols cannot be described by a simple (input, output) pair. Rather, test cases for such systems must have the following characteristics: sequences of input/output events, ability to check values of parameters in a received event, timer management facility, ability to retransmit the same event for a finite number of times if there is no response from the IUT in an expected time duration, and ability to assign a test verdict at the end of a test session.

Associated with every test case is a *test purpose*, which is a high-level description of the protocol function to be tested by the test case. If the behavior of the IUT is allowed by the protocol and the test purpose is satisfied, then the test case assigns a *Pass* verdict. If the behavior of the IUT is not allowed by the protocol, then the test case assigns a *Fail* verdict. However, if the behavior of the IUT is allowed by the protocol, but the test purpose is not satisfied, then the test case assigns an *Inconclusive* verdict. Therefore, the correctness of conformance judgment of a test case depends on the correctness of sequences of events input to the IUT, the expected protocol events stated in the test case, the verdicts assigned by the test case, and the purpose of the test case.

Some well-known examples of safety properties of concurrent systems are *partial correctness, absence of deadlock,* and *mutual exclusion*. A liveness property that has received a lot of formal treatment is *program termination*. However, program termination is not a good thing to happen to every computing system. For example, an operating system should never terminate (*crash*). For such systems, other kinds of liveness properties are important, for example: Each request for service will eventually be answered, a process will eventually enter its critical section, etc. The nature of safety and liveness properties of a system depends on the nature of the computing system. Therefore, to verify the correctness of test cases, one must define safety and liveness properties applicable to test systems. One contribution of this paper is to define safety and liveness properties relevant to test systems.

An outline of the test verification methodology presented in this paper is as follows. Because protocols and test cases are generally specified using different FDTs, it is essential to represent them in a common notation for the purpose of being able to obtain their combined behavior. Thus, we define a kind of extended finite state machine (EFSM) to which a variety of protocol and test specification languages can be translated. Since a test architecture plays an important role (in the form of defining the logical and distributed interconnection among the entities in a test system) in the design of a test case, we define the notion of a test verification system based on a test architecture and generate the global behavior of the test verification system. That is, we consider test architectural issues [15] in the verification process. To verify the correctness of test cases, we express the test case properties in terms of formulas in branching time temporal logic using the well-known notions of *safety* and *liveness*. We define four types of safety properties and one type of liveness property. These properties are then verified, using a model-checking approach, on the global behavior of the test verification system.

The paper is organized as follows. In section 2, we present the EFSM models of a test case and a protocol specification, a uniform way of representing exchanged data, and a brief introduction to branching time temporal logic. In section 3 the test case verification system is defined, an overview of model-checking–based test case verification methodology is given, followed by step 1 of the methodology, the global state space generation. Step 2, the generation of the model is discussed in section 4. Step 3, formulation of the test case properties is discussed in section 5 where a notation for test purpose representation is developed. The last step of the methodology, the verification of test case properties on the model is presented in section 6. A detailed example of test case verification is given in section 7. In section 8, we summarize all the reported research on test case verification. Some concluding remarks are stated in section 9.

2. EFSM models and temporal logic

In general, test cases and protocols are specified in different formal description techniques which use different notations to define data types and different behavioral semantics of their operations. For example, according to ISO's standardization framework, test cases are specified in TTCN, whereas a protocol can be specified in LOTOS, Estelle, or SDL.

Estelle is based on a finite state machine model, which is extended by Pascal data types, expressions, and statements. The Estelle specification of a protocol may consist of a large number of interconnected modules which communicate among themselves through (FIFO) channels. LOTOS, a process algebraic specification language, is a combination of Milner's calculus of communicating systems (CCS) formalism for behavior description and abstract data types (ADTs) for

data description. A set of *composition rules* is used to derive larger specifications from the primitive notions of *events* and *processes*. SDL, like Estelle, is also based on an extended finite state machine model. It is largely oriented toward a graphical representation. Abstract data types are used to define data in an SDL specification. In the TTCN test specification language, *constrained events* and *subtrees* constitute the building blocks in the design of the behavior part of a test case. Data in a test case are described using both a *tabular* notation and the abstract syntax notation 1 (ASN.1) [8].

It is not possible to compare the behavior of a test case and the behavior of a protocol specified in different FDTs and using different data definition techniques. Moreover, with such differences, it is not possible to get a combined behavior representing all interaction sequences between a test case and a protocol specification. Therefore, we introduce the notion of a common intermediate model, which is a kind of extended finite-state machine (EFSM), to which protocols and test cases specified in different FDTs can be translated.

In this section, we define an EFSM for modeling protocol and test specifications, define a common notation for events exchanged among the entities in a test architecture, and present a brief overview of branching time temporal logic.

2.1. *EFSM models*

In this section, we describe the EFSM models of test cases and protocol specifications. A communicating EFSM is an 8-tuple, $F = < S, V, R, s_{init}, Z, h_0, I_c, O_c >$, where S is a tagged set of states such that a tag is an element of the verdict set {Pass, Fail, Inconclusive, Null}; V is a finite set of variables; R is a finite set of possible transitions between states; s_{init} is the initial state; $Z \subseteq S$ is the set of final states; h_0 is the initial assignments to the variables in V in the form of $v_i \leftarrow val_i$ for some $v_i \in V$; I_c is a set of input channels from which F receives messages; and O_c is a set of output channels to which F sends messages to communicate with other EFSMs.

A transition in an EFSM is a 6-tuple, $r =< s, s', a, e, h, n >$, where s is the *from* state; s' is the *to* state; a is an *action* or *event* clause causing the transition to fire; e is the *enabling* predicate; h is a set of assignments of values to the variables in V; and n is the priority number of the transition. The priority number is used to model the priority of execution in the case of several alternative transitions from the same state.

An event in a transition can be one of {input, output, internal}, where the input and output events are known as external events and occur at some well-defined interaction points through which EFSMs communicate. An external event is characterized by three parameters: *interaction point, direction* ("?" denotes an input and "!" denotes an output), and the value (message) passed in the event. No channel or direction is associated with an internal event.

We use function notations to access the parameters of a transition. For

example, $dir(E)$ returns the direction field in the event E; $pco(E)$ returns the interaction point of the event E; and $from(r)$ and $to(r)$ return the *from* and *to* states of transition r, respectively. The notation $ps(X)$ denotes the present state of an EFSM M. The function $ext(E)$ ($int(E)$) returns a *true* value if E is an external (internal) event. To extract the first message in a channel Q, we use the notation $head(Q)$. The function *verdict(state)* returns the verdict tag of the *state*. We also denote the enabling condition e of a transition r by e_r and the set of transitions R of an EFSM M by R_F.

2.1.1. EFSM model of a specification. Communication protocols can be specified using specification languages such as LOTOS, Estelle, and SDL. Any specification in one of these languages can be algorithmically mapped to the EFSM model. In this paper, we only describe a mapping for Estelle [6].

The transformation procedure assumes single-module normalized Estelle transitions where the **Begin ... End** block may contain a sequence of assignments and a sequence of output events [16] such as in:

```
From ACSE_IDLE
To   AWAIT_AARE_APDU
when A.A_ASCreq
Provided Event.Mode = MODE_Supported
Begin
  P_CONreq.User_data.Protocol_version := 1;
  P_CONreq.Session_Con_Id := Event.Session_Con_Id;
  Output P.P_CONreq;
End,
```

where the **From** and **To** clauses represent the current and the next state of the transition; **When** clause represents the input event *A_ASCreq* at the interaction point *A*; **Provided** clause represents the enabling predicate; and **Output** statement represents the output event *P_CONreq* at the interaction point *P*.

The normalized transitions with no **Output** statements transform directly to the EFSM transitions. Two or more transitions are generated from a normalized transition with **Output** statements by separating the input and output events and creating new states between **From** and **To** of the Estelle transition. **Spontaneous** transitions, i.e., the transitions with no **When** clause are represented in the EFSM model by transitions whose action clause is set to *internal*. In this case if the transition has a priority, it is kept in the priority clause of the EFSM transition.

Applying the above mapping process to the example transition we generate two EFSM transitions:

```
<ACSE_IDLE, X, A?ASCreq, [Event.Appl_Mode = MODE_Supported],
 {P_CONreq.User_Data.Protocol_Version := 1,
  P_CONreq.Session_Con_Id := Event.Session_Con-Id}, 1>

<X, AWAIT_AARE_APDU, P!PCONreq, True, {}, 1>
```

The initial assignment h_0 of S-EFSM is obtained from the initial transition of the Estelle specification. The set of final states Z is usually equal to s_{init}, but in some cases may include other states. I_c/O_c are FIFO queues denoting the external input/output interaction points of the Estelle specification, respectively.

The process of mapping from normalized Estelle to EFSM is of linear complexity since in many cases transitions map one-to-one. However, if there are n output events in a normalized transition, then we generate $(n + 1)$ transitions, one for the input event and one for each output event.

2.1.2. EFSM model of a test case. Test cases are specified in a test specification language called the Tree and Tabular Combined Notation (TTCN). A TTCN specification contains four parts: *overview, declarations, constraints,* and *dynamic behavior.* The overview part specifies the name of the test suite and contains information about the hierarchical test suite structure. The declaration part is used to declare test suite parameters, points of control and observation (PCO), protocol data units (PDU), abstract service primitives (ASP), and timers.

The constraint section allows one to specify values for each field of the ASPs and PDUs. A constraint table contains a constraint name for the ASP (PDU) and a list of parameter names and their values. The parameter values in an ASP (PDU) constraint are sent if the ASP (PDU) appears in a send event and they shall be the values received if the ASP (PDU) appears in a receive event, i.e., values of parameters in a send event constraint are used to assign values to the corresponding parameters, whereas values of parameters in a receive event constraint are used to check whether the received values of parameters are equal to the respective values in the constraint.

The dynamic behavior table for a test case contains the specification of the combinations of sequences of test events that are deemed possible by the test suite specifier. The events are combined in two ways: sequences and sets of alternatives. A sequence of events is represented one line after the other, each new event being indented once from left to right, as time is assumed to progress. Test events at the same level of indention and belonging to the same predecessor event represent the possible alternative events which occur at that time. Alternative test events are specified in the order in which the tester shall repeatedly attempt them until one occurs. All the undesired external input events can be trapped by specifying an OTHERWISE event as an alternative to the desired events. Therefore, in a sequence of alternative events, an OTHERWISE event is the last event or an event just before a TIMEOUT. Design of a test case can be modularized by using *subtrees* and *default trees.*

It is possible to algorithmically map a TTCN test case to an EFSM [17]. The translation process consists of three steps. In the first step constraints are processed and default behaviors are expanded. A send event constraint is translated to a set of assignments and a receive event constraint is translated to a conjunction of predicates. In the second step, an EFSM is derived from the main tree and for each of the subtrees. In the third step, subtree attachments

are resolved by combining the corresponding EFSMs. Each event line in a test case is modeled as a transition. For example, consider the TTCN event line:

$$L!P\_CONreq[x = 2](a := 1)P\_CON\_base$$

where ! represents a send event *P_CONreq* at the PCO L; $x = 2$ is a predicate; $a := 1$ is an assignment; and *P_CON_base* is a constraint reference. Constraint table that defines *P_CON_base* is processed first to obtain a sequence of assignments $f1$ to be placed in the assignment clause of the EFSM transition. Then the predicate is placed in the enabling clause, the assignment $a := 1$ is added to $f1$, and finally the event clause is set to *L!P_CONreq*. Assuming that the transition occurs from state $S1$ to state $S2$ then the verdict tags of both the states are set to *Null*. If this is the first event in a set of alternatives, then the priority is 1. The corresponding transition in EFSM notation then becomes:

$$< \texttt{S1}, \texttt{S2}, \texttt{L!P\_CONreq}, [\texttt{x} = 2], \texttt{f1}, 1 >$$

The initial assignment h_0 of T-EFSM is obtained from the initial values provided in the declarations section of the TTCN test case specification. Initial values of ASPs/PDUs are obtained from the base constraints declared in the constraints section. The set of final states Z is derived from the **To** states of the last event in a sequence. I_c/O_c are the PCOs. Since PCOs are bidirectional $I_c = O_c$. Consecutive events in a set of alternatives are assigned consecutive priority numbers starting with 1. The higher the priority number is, the lower the execution priority.

2.1.3. Representation of data structures. In the layered OSI communication architecture [18], two protocol entities communicate through the exchange of *events*, called abstract service primitives (ASP) and protocol data units (PDU), at the service boundary between them. If data in two communicating entities are defined using different data definition techniques, it is not possible to interpret a received event in a communicating entity. It is rather natural to use different data definition techniques while specifying communication protocols in different FDTs, a test case in TTCN, and a test management protocol in a semiformal manner. The specification languages LOTOS, Estelle, SDL, and TTCN use abstract data types, Pascal data types, abstract data types, and a tabular notation in addition to ASN.1 to define data and events, respectively.

Therefore, it is important that the syntax of and the naming conventions used in the events are interpreted in a unified manner. For this purpose, we use a notation called input/output diagram (IOD) to model events exchanged between two communicating entities. The concept of an I/O diagram was first introduced by Jackson in the context of structured programming [19].

The IOD notation is selected as a means of representing a communication event because of its ability to represent a variety of attributes of the parameters in the event, such as *tree structure* of ASP/PDU parameters representing *composite*

Figure 1 (a) Structure of an internal node and (b) structure of a leaf node.

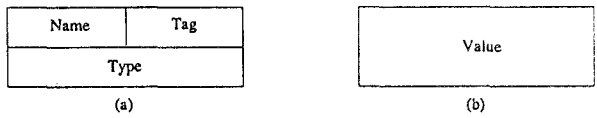

Figure 1. (a) Structure of an internal node; and (b) structure of a leaf node.

data types, grouping of parameters using *sequence* and *set* semantics, *choice* of a parameter among a set of alternatives, *repetitive* nature of the same parameter, *optional/mandatory* presence of some parameters in an event, and *default* values of some parameters.

To represent a communication event as an IOD, we define two primitive building blocks and a notation to combine them to describe a complete ASP/PDU. An IOD takes a tree structure using two types of nodes: *internal* and *leaf*. An internal node, shown in figure 1, contains three fields: *name, type,* and *tag*. The name and type fields represent the name and type of a parameter field. The tag represents a data value's attribute, whose possible values are {optional, mandatory, default, choice, set, sequence}. A leaf node has only one field to contain the value of a type stated in its parent node. Only the leaf nodes in a tree structure contain actual values, whereas the internal nodes are used to build composite data types. For example, an internal node in an ASP may contain a composite type representing a PDU.

An IOD is a tree structure representing an ASP or a PDU. Since only ASPs and PDUs are exchanged among the entities in a test system and since these structures can be specified using different techniques, IOD becomes a common representation. Mapping of IOD from or to individual data representations need only be done at the time ASP/PDUs are placed into channels.

An event parameter can be either a *primitive* type or a *composite* type. Examples of a primitive type are integer type, boolean type, bit string type, etc. A composite type may contain more than one primitive type or a combination of primitive and composite types.

Abstract data types in LOTOS and SDL, record types in Pascal, and tabular notations and ASN.1 in TTCN are used to define events. It is possible to represent all those data types as I/O diagrams [20].

2.2. *Branching time temporal logic*

Temporal logic has been of particular interest to the designers of both hardware and software specifications for more than a decade in the form of verifying

some well-defined properties of specifications [21, 22]. With the widespread use of communication protocols and the subsequent global effort in standardizing formal specifications of protocols, temporal logic has also been used in verifying protocol specifications [23, 24].

Temporal logics are extensions of the propositional logic. A temporal formula is constructed using propositional variables, the conventional logical operators, and a set of temporal operators. The set of temporal operators in a temporal logic is defined based on the structure of time on which the temporal formalism is based. There are two main classes of temporal logic: *linear time* and *branching time* [25]. The linear time temporal logic considers time to be a linear sequence and the branching time approach adopts a tree structured time allowing some instants to have more than a single successor instant. Both types of temporal logic are used in the verification of communication protocols [23, 24, 26].

Whether to use the linear time logic or the branching time logic is pragmatically based on the types of systems and properties one wishes to formalize and study [25]. Since the global behavior of a test system containing multiple nondeterministic protocol and test entities takes a tree structure rather than a sequential structure, we use branching time temporal logic for studying the properties of a test system. In the following, we present the syntax and semantics of branching time logic.

The advantages of using temporal logic to express test case properties are that it allows us to verify test case properties in terms of the well-known notions of *safety* and *liveness* and to express the validity of Pass test verdicts with the satisfaction of test purposes.

Let AP be a set of atomic propositions. BTL formulas are obtained by using the following two rules.

1. Every atomic proposition $p \in AP$ is a BTL formula.

2. If f and g are BTL formulas, then so are $\neg f$, $f \wedge g$, $f \vee g$, $A[f\,U\,g]$, $E[f\,U\,g]$.

The symbols $\neg$, $\wedge$, and $\vee$ have their usual meanings. U is the *until* operator; the formula $A[f\,U\,g]$ ($E[f\,U\,g]$) intuitively means that for every computation path (for some computation path) there exists an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix.

The semantics of a BTL formula are defined with respect to a labeled state transition graph. Formally, a BTL structure is a 5-tuple: $M =< S, V, R, P_r, s_{init} >$ where S is a finite set of states; V is a finite set of variables; R is a finite set of possible transitions between states; $(P_r : S \rightarrow 2^{AP})$ assigns to each state the set of atomic propositions that hold in that state; and $s_{init} \in S$ is the initial state.

A *path* is a sequence of states $(s_0, s_1, s_2, \ldots)$ such that $\forall i[(s_i, s_{i+1}) \in R]$. The state s_0 need not be the initial state of a BTL structure. We use the standard notation to express truth in a structure: $(M, s_0 \models f)$ means that the temporal

formula f holds at state s_0 in structure M. When the structure M is understood, we simply write $s_0 \models f$. The relation $\models$ is defined inductively as follows:

1. $s_0 \models p$ iff $p \in P_r(s_0)$.
2. $s_0 \models \neg f$ iff $not(s_0 \models f)$.
3. $s_0 \models f \wedge g$ iff $s_0 \models f$ and $s_0 \models g$.
4. $s_0 \models f \vee g$ iff $s_0 \models f$ or $s_0 \models g$.
5. $s_0 \models A[f\,U\,g]$ iff for all paths $(s_0, s_1, \ldots)$ starting with s_0,

$$\exists i[(i \geq 0) \wedge (s_i \models g) \wedge (\forall_j[0 \leq j \leq i \rightarrow (s_j \models f)])].$$

6. $s_0 \models E[f\,U\,g]$ iff for some path $(s_0, s_1, \ldots)$ starting with s_0,

$$\exists i[(i \geq 0) \wedge (s_i \models g) \wedge (\forall_j[0 \leq j \leq i \rightarrow (s_j \models f)])].$$

The following abbreviations are also used in writing BTL formulas:

- $AF(f) \equiv A[True\,U\,f]$ means that f holds in the future along every path from s_0; that is f is *inevitable*.
- $EF(f) \equiv E[True\,U\,f]$ means that there is some path from s_0 that leads to a state at which f holds; that is, f *potentially* holds.
- $EG(f) \equiv \neg AF(\neg f)$ means that there is some path from s_0 on which f holds at every state.
- $AG(f) \equiv \neg EF(\neg f)$ means that f holds at every state on every path from s_0; that is f holds *globally*.
- $(f_1 \mapsto f_2) \equiv AG(f_1 \rightarrow AF(f_2))$ (read "f_1 leads to f_2") means that for any time at which f_1 is true, f_2 must be true then or at some later time.

3. Test verification system state space

Since every test case is designed in the context of a specific test architecture, in this section we explain the notion of a test architecture in layered-protocol testing. We define a generic test verification system applicable to all test architectures. Then we give an outline of the test case verification methodology followed by a reachability analysis–based algorithm to generate a global state space from a test verification system.

3.1. *Test architectures*

According to the OSI reference model, a protocol entity communicates by exchanging abstract service primitives (ASPs) with the layers immediately above and below it in order to provide services to the layer above it using the services provided by the layer below it. Thus, a protocol test system contains two test entities, a lower tester (LT) and an upper tester (UT). The LT controls and observes the ASPs at the lower service boundary by monitoring the ASPs at the lower point of control and observation (PCO). Similarly, the UT controls and observes the ASPs at the upper service boundary by monitoring the ASPs at the upper point of control and observation.

Generally, the LT functions as the master tester and the UT functions as a test responder. In practice, there are some variations to the above descriptions of testing activities. The lower service boundary of an IUT may not be accessible to the LT, in which case the LT controls and observes those events at a point away from the actual lower boundary of the IUT through the use of an underlying service provider. Moreover, the LT may control and observe the events at the upper service boundary of the IUT in an indirect manner by controlling the activities of the UT through a specialized test management protocol.

Thus, the number of PCOs, the proximity of the PCOs to the IUT, and the interconnection mechanism among the entities in a protocol test system give rise to the notion of *basic test architectures* [2]. In the OSI testing framework, there are four basic test architectures: local single layer (LS), coordinated single layer (CS), distributed single layer (DS), and remote single layer (RS). From the point of error-detection capabilities, these test architectures have been compared in [15].

In the CS architecture, the LT communicates with the IUT through the service provider while controlling and observing the IUT's behavior at the lower PCO (L) as shown in figure 2. The IUT communicates with the service provider through the interaction point N and with the UT through the interaction point U. The LT part of a CS architecture–based test case is written in TTCN and a standardized test management protocol (TMP) is used as the UT whose behavior is deterministically controlled by the LT through the use of command and reply test management protocol data units (TMPDU). A TMP is described in detail in section 7.

3.2. *Test verification system*

Replacing the IUT by the EFSM model of the corresponding protocol specification, the test entities by their respective EFSM representations, the service provider by its EFSM representation, and representing each point of interaction between two communicating entities by two FIFO channels, we derive a *test verification system* as defined below.

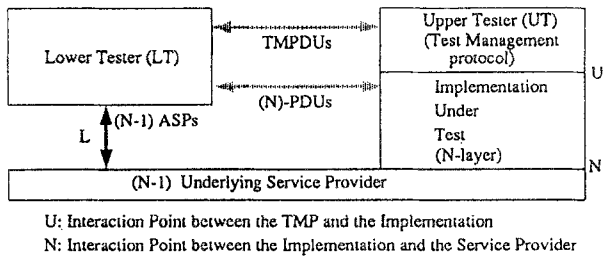

Figure 2. Coordinated single-layer (CS) test architecture.

Definition 1. A *test verification system* (TVS) is defined to be a 5-tuple, $TVS = < \Sigma, \Omega, P, \Psi, C >$, where Σ is an EFSM corresponding to the lower tester (LT-EFSM); Ω is an EFSM corresponding to the underlying service provider (USP-EFSM); P is an EFSM corresponding to the protocol specification (S-EFSM); Ψ is an EFSM corresponding to the upper tester (UT-EFSM); and C is a set of *channel functions* defining the interconnection among Σ, Ω, P, and Ψ.

A channel function *channel(EFSM1, EFSM2)* denotes that EFSM1 outputs messages to the *channel* which are received by *EFSM2*. In general, we denote a test EFSM by T-EFSM while referring to either the lower tester or the upper tester EFSM. Corresponding to a test architecture, we derive a test verification system as follows.

1. Replace the implementation under test (IUT) module by the EFSM representation of the corresponding protocol specification.
2. Replace the LT module by the EFSM representation of the lower tester part of the test case.
3. For a given test architecture, replace the UT module in the following manner.
   a. For LS and DS test architecture, replace the UT module by the EFSM representation of the TTCN specification of the upper tester part of the test case.
   b. For CS architecture, replace the UT module by the EFSM representation of the test management protocol (TMP) specification.
   c. For RS architecture, the behavior of the UT module is dynamically generated during the model generation process. Initially, the UT-EFSM consists of only one state s_0 and one transition $r = < s_0, s_0, U?OTH, true, \{\}, 1 >$. Implicit send events in conjunction with the behavior of the S-EFSM are used to dynamically update the UT-EFSM while generating a global state space.

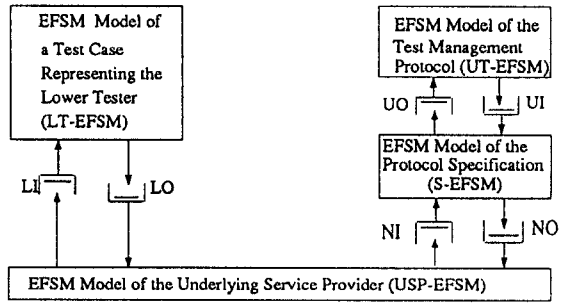

Figure 3. The test verification system for the CS architecture.

4. Replace the underlying service provider module by its EFSM representation, that is, by its input/output behavior.
5. Replace each interaction point and point of control and observation (PCO) between two modules in the test architecture by two unidirectional FIFO channels.

In general, we denote a test EFSM by the notation T-EFSM while we refer to any test EFSM including the service provider EFSM. The test verification system for the CS architecture in figure 2 is shown in figure 3. The LT-EFSM is obtained from a TTCN test case specification as explained in section 2.2. The UT-EFSM is derived from a TTCN test case specification of the upper tester in case of LS and DS architectures and the specification of a test management protocol in case of CS architecture. In case of the RS test architecture, the EFSM model of the UT is dynamically generated in an incremental manner [20]. The S-EFSM is obtained from the formal specification of the protocol written in one of the FDTs: LOTOS, Estelle, and SDL. An outline of a methodology to translate an Estelle specification to an EFSM is discussed in section 2.1 and a detailed methodology can be found in [20]. Protocols specified in LOTOS and SDL can also be translated to their corresponding EFSM models using the methodologies in [27]. The USP-EFSM is obtained from the service specification of the layer providing services to the protocol layer under test.

3.3. Outline of the methodology

Temporal logic can be used to model test case properties in terms of safety and liveness properties. Two approaches exist for verifying temporal formulas: *theorem proving* and *model checking*. Theorem proving involves logical deduction and requires the protocol and test case specifications to be expressed as temporal formulas. Model checking, on the other hand, involves verifying the properties

on a state space. We take the model checking approach because:

1. Formal specifications in one of FDTs and TTCN of protocols and test cases are of common use.
2. These specifications can algorithmically be mapped to EFSM models.
3. Using a modified version of a traditional reachability analysis algorithm, a global state space can be generated.

Therefore, the model-checking approach to test case verification consists of the following four steps:

1. Derive a global state space by combining the behavior of all the entities in the TVS.
2. Associate a set of atomic propositions to each global state.
3. Express the safety and liveness properties of a test case as temporal formulas.
4. Use a model checker to verify whether the global state space derived in step 1 is a model for the test case properties in step 3.

Step 1 is covered below in section 3.4; steps 2–4 are detailed in sections 4, 5, and 6, respectively.

3.4. State space generation

We generate a global state space representing the combined behavior of a test verification system by using a reachability analysis algorithm. The reachability analysis algorithm is based on *perturbing* a *global state* using all the *executable transitions* in the component machines' present states [28].

Definition 2. The global state s of a test verification system $TVS = < \Sigma, \Omega, P, \Psi, C >$ is defined as a 6-tuple $< \Sigma_s, \Omega_s, P_s, \Psi_s, C_s, \Pi \cup v >$, where Σ_s, Ω_s, P_s, and Ψ_s represent the present states of Σ, Ω, P, and Ψ, respectively; C_s is a set of states consisting of the present states of each channel in C; Π is the set containing values of all the variables of the EFSMs in the TVS; and v is a verdict variable assumed to be unique.

In a global state space, the states are connected by global transitions. A global transition in the transition space R is a 6-tuple defined in section 2. The initial global state s_0 is defined as follows:

$$< s_{init}(\Sigma),\ s_{init}(\Omega),\ s_{init}(P),\ s_{init}(\Psi),\ C_{empty},\ init(\Pi) \cup v = Null >,$$

where $s_{init}(\Sigma)$ is the initial state of Σ; $s_{init}(\Omega)$ is the initial state of Ω; $s_{init}(P)$ is the initial state of the protocol specification entity P; $s_{init}(\Psi)$ is the initial state of Ψ; C_{empty} denotes all the channels in C to be empty; and $init(\Pi) = h_0(\Sigma) \cup h_0(\Omega) \cup h_0(P) \cup h_0(\Psi)$, where the function h_0 denotes initial assignments to the variables in the corresponding EFSM. Notationally, the present state of an EFSM, M, is denoted by the function notation $ps(M)$.

In the following, we define the sets of enabled transitions, executable transitions, pending transitions, and must transitions.

Definition 3. The set of executable transitions $XT(s)$ occurring in the present state $s = < \Sigma_s, \Omega_s, P_s, \Psi_s, C_s, \Pi \cup v >$ in $TVS = < \Sigma, \Omega, P, \Psi, C >$ is defined as the set of all transitions, in the EFSMs Σ, Ω, P, and Ψ whose enabling conditions evaluate to true, that is,

$$XT(s) = \{r | r \in \{R_\Sigma \cup R_\Omega \cup R_P \cup R_\Psi\} \wedge from(r) \in \{\Sigma_s, \Omega_s, P_s, \Psi_s\} \wedge e_r = true\}.$$

Since the evaluation of e_r involves accessing the channel contents and taking the priority number of a transition in a set of alternative transitions in an EFSM, $XT(s)$ is computed in the following manner.

$$XT(s) = Exec(\Sigma, s) \cup Exec(\Psi, s) \cup Exec(P, s) \cup Exec(\Omega, s),$$

where the procedure $Exec$ is given below.

procedure $Exec(M, s)\{/*$ M is an EFSM $\in \{\Sigma, \Omega, P, \Psi\}$ with all its transitions and $s = < \Sigma_s, \Omega_s, P_s, \Psi_s, C_s, \Pi \cup v > */$
$/*$ TXT is a temporary variable that holds the set of executable transitions in M corresponding to s. $*/$
$TXT(s) := \phi, R = \{r \mid from(r) = ps(M)\}$, init_priority := 0,
$Flag(c_j)$:= False $\forall c_j \in C$
While $R \neq \phi$ begin {
  init_priority := init_priority +1
  for $r \in R|$ (priority(r) = init_priority) do {
    if $((int(E) \wedge eval(s, \phi, e))$ then
      $TXT(s) := TXT(s) \cup \{r\}$, $R := R - \{r\}$
    if $(ext(E) \wedge (dir(E) =!) \wedge eval(s, \phi, e))$ then
      $TXT(s) := TXT(s) \cup \{r\}$, $R := R - \{r\}$
    if $(ext(E) \wedge (dir(E) =?) \wedge (msg(E) = head(channel(E))) \wedge$
      $(flag(channel(E) = False) \wedge eval(s, head(channel(E)), e)))$
    then
      $\{TXT(s) := TXT(s) \cup \{r\}, R := R - \{r\},$
        $Flag(channel(E)) := True\}$
    if $(ext(E) \wedge (dir(E) =?) \wedge (msg(E) = OTH) \wedge$
      $(content(channel(E)) \neq \phi) \wedge (flag(channel(E) = False) \wedge$

```
            eval(s, head(channel(E)), e))) then
          {TXT(s) := TXT(s) ∪ {r}, R := R − {r},
            Flag(channel(E)) := True}
        }/* for-loop */
    }/* while-loop */
Return(TXT)
}
```

Here TTCN *TIMEOUT* events translated as *timeout* transitions in T-EFSMs are treated qualitatively by assuming that a timeout can occur any time without referring to its quantitative value. The advantage of such a treatment is that all possible effects of the timeout transitions can be studied. Also TTCN *OTHERWISE* events translated as *OTH* transitions in T-EFSMs are selected only if none of the events of higher-priority alternatives match with the first event in the channel.

3.4.1. Predicate evaluation. Here the procedure *eval* used in computing XT will be defined. This procedure decides on the truth value of the enabling predicate e of a transition in S-EFSM or one of T-EFSMs given a global state s and the IOD at the head of the channel.

Assume that e is expressed in conjunctive normal form such that each operand of a logical operator in e is either a boolean variable or an expression of the form op_1 *relop* op_2, which we call an elementary predicate, and *relop* is a relational operator. Next it is assumed that IOD has nonsymbolic values assigned at all leaf nodes. These assumptions are essential for the procedure *eval* to always return a true or false value.

procedure $eval(s, IOD, e)$
{/* IOD is a composite tree-structured data representing an ASP in a FIFO channel or ϕ for an internal transition */

1. Execute step 2 for each elementary predicate in e. If any of the returned values are false, then exit the procedure with a false value. If all the returned values are true, then exit the procedure with a true value.

2. Evaluation of a relational operator consists of the following three cases which return a boolean result:

   a. *op*1 **relop** *op*2, where both *op*1 and *op*2 are local variables: Evaluate *op*1 **relop** *op*2 and return the boolean result.

   b. *op*1 **relop** *op*2, where *op*1 is a field of the IOD and *op*2 is a constant: Traverse the tree-structured data IOD and extract the value of *op*1, evaluate "*op*1 **relop** *op*2", and return the boolean result.

   c. *op*1 **relop** *op*2, where *op*1 is a field of the IOD and *op*2 is a local variable: Traverse the tree-structured data IOD and extract the value of

$op1$. Since $op2$ can be an expression in the local variables, first compute the value of $op2$, then compute "$op1$ **relop** $op2$", and return the boolean result.}

3.4.2. State perturbation. The idea of state perturbation [28] is central to generating the global state space of a system containing a set of communicating modules. The reachability analysis algorithm [28] generates the global state space of a protocol system consisting of deterministic FSM entities. The reachability analysis algorithm presented in this paper generates the global state space of a system modeled as a set of communicating nondeterministic EFSMs. In addition, the algorithm takes into account the semantics of OTHERWISE events used in a test case [2].

Given the present global state and a transition, the perturbation function computes the next global state. If the transition is an implicit send in an RS architecture, then the perturbation function also updates the UT-EFSM by calling the *UT_gen* procedure. In the following definition, the procedure *map_to_IOD*(*Event*) is assumed to transform the *Event* of type ASP or PDU to an IOD.

Definition 4. The perturbation of a global state $s = < \Sigma_s, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi \cup v >$ by an executable transition $r \in XT(s)$ in $TVS = < \Sigma, \Omega, P, \Psi, C >$, denoted by the function $pert(s, r)$, is defined as the process of obtaining a new state s_n of the TVS by executing r in s. Notationally, s_n $pert(s, r)$.

The perturbation function $pert(s, r)$ is given below.

procedure $pert(s, r)\{$
/* $s =< \Sigma_s, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi \cup v >$ and
$r =< From, To, E, e, h, \text{n} >$ */
if $((From = \Sigma_s) \wedge int(E))$ then
$s_n :=< To, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi' \cup v' >$, where
$\Pi' := h(\Pi)$ and $v' := new\_verdict(v, verdict(To))$.
if $((From = \Sigma_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i))) \vee$
$((msg(E) = OTH) \wedge (c_i \neq \phi))$, then
$s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, tail(c_i), \ldots, c_n\}, \Pi' \cup v' >$.
if $((From = \Sigma_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$ then
$s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, c_{i\_new}, \ldots, c_n\}, \Pi' \cup v' >$, where
$c_{i\_new} := append(c_i, map\_to\_IOD(msg(E)))$.
if $((From = \Sigma_s) \wedge (dir(E) = !) \wedge (channel(E) = IUT))$ then
$s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi' \cup v' >$, s.t. $c_i(\Psi, P) \in C$
Call UT_$gen(s, r)$.
if $((From = P_s) \wedge int(E))$, then
$s_n := < \Sigma_s, \Omega_s, To, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi' \cup v' >$.
if $((From = P_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i)))$,
then $s_n := < \Sigma_s, \Omega_s, To, \Psi_s, \{c_1, \ldots, tail(c_i), \ldots, c_n\}, \Pi' \cup v' >$.

if $((From = P_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$, then
$s_n := < \Sigma_s, \Omega_s, To, \Psi_s, \{c_1, \ldots, c_{i\_new}, \ldots, c_n\}, \Pi' \cup v' >$, where
$c_{i\_new} := append(c_i, map\_to\_IOD(msg(E)))$.
if $((From = \Omega_s) \wedge int(E))$, then
$s_n := < \Sigma_s, To, P_s, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi' \cup v' >$.
if $((From = \Omega_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i)))$,
then $s_n := < \Sigma_s, To, P_s, \Psi_s, \{c_1, \ldots, tail(c_i), \ldots, c_n\}, \Pi' \cup v' >$.
if $((From = \Omega_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$, then
$s_n := < \Sigma_s, To, P_s, \Psi_s, \{c_1, \ldots, c_{i\_new}, \ldots, c_n\}, \Pi' \cup v' >$, where
$c_{i\_new} := append(c_i, msg(E))$.
if $((From = \Psi_s) \wedge int(E))$, then
$s_n := < \Sigma_s, \Omega_s, P_s, To, \{c_1, \ldots, c_i, \ldots, \ldots, c_n\}, \Pi' \cup v' >$, where
$\Pi' := h(\Pi)$ and $v' = new\_verdict(v, verdict(To))$.
if $(((From = \Psi_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i))) \vee$
$((msg(E) = OTH) \wedge (c_i \neq \phi)))$, then
$s_n := < \Sigma_s, \Omega_s, P_s, To, \{c_1, \ldots, tail(c_i), \ldots, c_n\}, \Pi' \cup v' >$
if $((From = \Psi_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$, then
$s_n := < \Sigma_s, \Omega_s, P_s, To, \{c_1, \ldots, c_{i\_new}, \ldots, c_n\}, \Pi' \cup v' >$, where
$c_{i\_new} := append(c_i, map\_to\_IOD(msg(E)))$.
return (s_n))
}

In the TTCN specification of a test case, a test designer may associate intermediate test verdicts with expected receive events from an implementation and a final test verdict may be assigned on termination of a test case behavior. Depending on the expected responses of an implementation, different verdicts—pass, fail, or inconclusive—may be assigned to receive events in a sequence of test behavior. The resultant test verdict at any point in a sequence of test behavior depends on the previous verdict and the current verdict. Therefore, to compute a resultant verdict in a state given the previous verdict and the present verdict, in the following we define a procedure *new_verdict*(*ov*, *pv*), where *ov* is the old verdict or the previous verdict and *pv* is the present verdict.

procedure *new_verdict*(*ov*, *pv*) {
if (*ov* == "none" then return(*pv*)
else{
if (*ov* == Fail) or (*pv* == Fail) then return(Fail)
else if (*ov* == Pass) and (*pv* == Inconclusive) then return(Inconclusive)
else if (*ov* == Inconc.) and (*pv* == Inconc.) then return(Inconclusive)
else if (*ov* == Pass) and (*pv* == Pass) then return(Pass)
}}

The RS test architecture does not have an explicitly defined upper tester. However, while executing an RS architecture–based test case, it is required to specify some behavior at the upper service boundary of the IUT. Therefore,

during the global state space generation process, we dynamically generate the desired behavior of the upper tester in an incremental manner by calling the following *UT_gen*( ) procedure.

procedure $UT\_gen(s, r_1)\{$
Let $\Psi =< S, \{\}, V, R, a_0, \{a_j\}, h.C_1, C_O >$,
$s =< \Sigma_s, \Omega_s, P_s, \Psi_s, \{c_1, \ldots, c_i, \ldots, c_n\}, \Pi \cup v >$, and
$r_1 =< \Sigma_s, s_1, E_1, e_1, h_1, n_1 >$

1. Explore all paths from P_s until a transition $r_3 =< s_3, s_3', E_3, e_3, h_3, n_3 >$ is encountered such that the event E_3 matches E_1.
   Let $r_2 =< s_2, s_2', E_2, e_2, h_2, n_2 >$ be a transition on the path from P_3 to s_3 such that $((dir(E_2) =?) \wedge (channel(E_2) = channel(\Psi, P)))$.
2. Generate:
   $E_n = channel(E_2)!event(E_2)$
   h_n = a set of assignments to the field of E_n such that e_2 is satisfied and all tansitions up to and including r_3 can be fired.
3. Update the UT behaviour by creating a state a_{j+1} and two transitions
   $r' =< a_j, a_{j+1}, E_n, true, h_n, 1 >$, $r'' =< a_{j+1}, a_{j+1}, OTH, true, \phi, 1 >$
   $\Psi =< S \cup \{a_{j+1}\}, \phi, V \cup var(h_n), R \cup \{r', r''\}, a_0, \{a_{j+1}\}, h, C_I, C_O >$

}

In the following, we present a state space generation algorithm based on the traditional reachability analysis algorithm extended to EFSMs [28]. Our algorithm handles some special characteristics of test specifications such as nondeterminism, OTHERWISE events, and verdict computation.

3.4.3. Channel capacities. In a reachability analysis–based validation system, where communication paths between state machines are modeled as FIFO channels or queues, it is important to analyze the effects of channel capacities on the validation process. A channel capacity can be either bounded or unbounded. The existence of system states in which the channel bounds are exceeded may indicate that the validation is incomplete [29]. However, the validation problem becomes unsolvable if channel capacities are assumed to be unbounded [16]. In a physical implementation, all channels must be bounded. Therefore, while analyzing a test system, we assume that all channels are bounded. This assumption precludes any possibility of the global state space being infinite. For the model checking method to work, it is essential to have a finite global state space.

With bounded channel capacities, there are possibilities of channel overflows. In the global state generation algorithm, channel overflows are handled by not perturbing a state using a transition that causes a channel overflow. This method of handling channel overflows is similar to the one in [29].

In the following algorithm, the predicate $channeloverflow(s)$ in a state s is evaluated as follows. Every global state contains the present contents of each

channel. To evaluate the predicate $channeloverflow(s)$, the number of messages in each channel is compared with the bounded capacity of the channel. If the number of messages in the channel exceeds the channel capacity, then the predicate evaluates to true.

Algorithm 1

Input: a set of EFSMs, a set of communication channels, and the capacities of the channels.

Output: a global state space.

S1. Define a set of global states S and a set of global transitions R. Initially, S contains only the initial global state s_{init} and $R = \phi$.

S2. Find a member $s \in S$ of the set of global states whose perturbations have not been determined. If no such member exists, then terminate.

S3. Calculate the set of executable transitions $XT(s)$ in state s using definition 4.

S4. Compute S_p, a set of global states by perturbing s. Initially $S_p = \phi$.

$\forall_r =< From, To, E, e, h, n > \in XT(s)$, *do*
{ $S_p = S_p \cup \{s'\}$, where $s' = pert(s, r)$;
$P_r(s) = \phi$; / * $P_r(s)$ is the set of predicates begin true in state s. * /
$R = R \cup \{< s, s', E, e, h, n >\}$;
}

S5. If S_p is an empty set, report s as a terminal state in the global state space.

S6. $\forall s \in S_p$ do {
if $channeloverflow(s)$ then mark s "perturbed" and $S = S \cup \{s\}$
else if $s \notin S$ then mark s "unperturbed" and $S = S \cup \{s\}$
}

S7. Go to step *S2*.

4. Model generation

A model for a TVS is generated from the global state space of the TVS by associating a set of atomic propositions to each state. Therefore, we first identify the predicate types and then present an algorithm to systematically associate a set of predicates with each global state.

4.1. *Atomic propositions*

In the following, we identify five types of predicates—*state predicates, variable predicates, event predicates, PCO predicates*, and *verdict predicates*—to be associated with the states in the global state space of a test verification system. Identification of the predicates types is guided by the test properties to be verified.

1. **State predicates:** The state predicate INIT is associated with the initial state

2. **Variable predicates:** These are assertions about the values of the variables in the structure. These assertions arise from the enabling conditions of the transitions of all the TVS entities.

3. **Event predicates:** These characterize the possibility or the actual execution of specified events. There are two types of event predicates: AT and AFTER used with different parameters. The predicate *AT(Treceive(Channel, Event))* is true in a state s if there is a test case transition such that the *Event* is received from the *Channel*. Similarly, *AT(Sreceive(Channel, Event))* is true in a state s if there is a protocol transition such that the *Event* is received from the *Channel*. *AFTER(Treceive(Channel, Event))* is true in a state s' in the BTL structure if there is a transition to the state s' such that the *Event* is received from the *Channel* as a result of firing the transition. A similar explanation holds for *AFTER(Sreceive(Channel, Event))*.

4. **PCO predicates:** The direction of events in a TTCN test case is with respect to the points of control and observation (PCO). We define a set of assertions about the PCOs and the input/output directions of events occurring at the PCOs: LOWER, UPPER, INPUT, OUTPUT, LOWER_OUTPUT, UPPER_OUTPUT, INTERNAL, and NULL. The predicate LOWER (UPPER) is true in a state if a transition fires in the state with an external event occurring at the lower (upper) PCO. The predicate INPUT (OUTPUT) is true in a state if a transition fires in the state with an external input (output) occurring at one of the two PCOs. If the external event is output at the lower (upper) PCO then LOWER(UPPER)_OUTPUT is true. If an internal event occurs in a state, then INTERNAL is true. If a transition containing neither an external nor an internal transition, but containing some assignment functions occurs in a state, then the predicate NULL is true in that state.

5. **Verdict predicate:** This is an assertion about the test verdict and is one of the following three: (Verdict == P), (Verdict == I), and (Verdict == F).

The first three classes of predicates are common to all communication systems and have been found to be useful in verifying communication protocols [26]. The last two classes of predicates are specific to protocol test systems. The first

four classes of predicates are used in specifying safety properties and the verdict predicate is used in specifying liveness property of a test case.

4.2. *Algorithm*

Algorithm 2

Input: the state set S and the transition set R of the global state space.
Output: predicated S.

S1. Initialization: $\forall s \in S$, $Pr(s) = \phi$, where Pr assigns a set of predicates to s.

S2. $\forall r =< s, s', a, e, h, n > \in R$ do $Pr(s) = Pr(s) \cup Pper(r)$
$Pr(s') = Pr(s') \cup Pgen(r)$

The functions $Pper$ and $Pgen$ are defined to compute the set of atomic predicates evaluated to true in a global state. The function $Pper(r)$ associates a set of predicates with a state s when s is perturbed by the executable transition r. Similarly, the function $Pgen(r)$ associates a set of atomic predicates with the state s' when s' is generated by perturbing the state s using the transition r.

```
Pper(r){/ * r =< s, s', E, e, h, m > * /
  if (ext(E) ∧ (dir(E) ==!)) then
     { if (pco(E) == L) then temp = {LOWER_OUTPUT, LOWER};
       else if (pco(E) == U) then temp = {UPPER_OUTPUT, UPPER};
     }
  else if (ext(E) ∧ (dir(E) ==?)) then
     { if (pco(E) == L) then temp = {LOWER};
       else if (pco(E) == U) then temp = {UPPER};
     }
  else if (E == i) then temp = {INTERNAL};
  else temp = {NULL};
  if (ext(E) and r is a test transition), then
     { if (dir(E) ==?) then temp = temp ∪ {AT(Trecieve(pco(E), message(E)))};
       else if (dir(E) ==!) then temp = temp ∪ {AT(Tsend(pco(E), message(E)))};
     }
  else if (ext(E) and r is a protocol specification transition), then
     { if (dir(E) ==?) then temp = temp ∪ {AT(Sreceive(pco(E), message(E)))};
       else if (dir(E) ==!) then temp = temp ∪ {AT(Ssend(pco(E), message(E)))};
     }
  return (temp ∪ {e})
}
Pgen(r){/ * r =< s, s', E, e, h, m > * /
  if (ext(E) and r is a test transition), then
     { if (dir(E) ==?) then temp = {AFTER (Treceive(pco(E), message(E)))};
       else if (dir(E) ==!) then temp = {AFTER(Tsend(pco(E), message(E)))};
```

```
    }
  else if (ext(E) and r is a protocol specification transition), then
    { if (dir(E) ==?) then temp = {AFTER (Sreceive(pco(E), message(E)))};
      else if (dir(E) ==!) then temp = {AFTER(Ssend(pco(E), message(E)))};
    }
  else temp= {};
  return (temp)
}
```

5. Safety and liveness properties

5.1. *Safety properties*

Based on the idea that nothing bad happens during a testing process, we classify the safety properties of a test case into four distinct types: *transmission safety, reception safety, synchronization safety*, and *verdict safety*. Each type of safety property is defined below.

Transmission safety: There are two transmission safety properties corresponding to transmission of events by the test case and transmission of events by the protocol.

1. $INIT \models (AFTER(Tsend(Q, E)) \mapsto AFTER(Sreceive(Q, E)))$ and
2. $INIT \models (AFTER(Ssend(Q, E)) \mapsto AFTER(Treceive(Q, E)))$.

In the above formulas, Q is any communication channel between the test system and the protocol specification; and E stands for an event. Intuitively, the first property states that every event sent by the test case must eventually be accepted by the protocol specification. That means the test case does not generate any event that is unacceptable to the protocol. The second property states that every event generated by the protocol specification during the testing process must eventually be accepted by the test case, i.e., the test case is ready to receive any event generated by the protocol. Satisfaction of these two properties ensures that there is no blocking reception error in the test case [30].

Reception safety: There are two reception safety properties corresponding to reception of events by the test case and reception of events by the protocol.

1. $INIT \models (AT(Treceive(Q_s, E_s)) \mapsto AFTER(Treceive(Q_i, E_i)) \vee AFTER(Tinternal))$ and
2. $INIT \models (AT(Sreceive(Q_s, E_s)) \mapsto AFTER(Sreceive(Q_i, E_i)) \vee AFTER(Pinternal))$.

In the above formulas, $AT(Treceive(Q_s, E_s)) \equiv AT(Treceive(Q_1, E_1)) \vee \vdots \vee AT(Treceive(Q_n, E_n))$.

The predicate $AT(Sreceive(Q_s, E_s))$ is defined in a similar way.

Intuitively, the first (second) reception property states that when control reaches a state in the test case (protocol specification) where there is a set of alternative receive events, one receive event must be enabled or an internal event must occur in that state. Satisfaction of these two properties ensures that the test case is not deadlocked with the protocol specification. The internal event may be due to a timeout.

Synchronization safety: This safety property is a special characteristic of protocol testing and is not found in conventional program testing. The issue of synchronization in a test case, which is an event timing problem, was first studied in [31] using deterministic FSM models of a protocol specification and a test case. This problem arises when the test system interacts with the protocol through two PCOs. Conceptually, a test case faces a synchronization problem at one of the PCOs if an output test event is preceded by a sequence of events consisting of internal protocol events and/or a test event occurring at the other PCO. We express the synchronization safety properties using the $until(U)$ operator as follows.

1. For every state s in the BTL structure,
   $s \models \neg E[f_1 U f_2]$, where $f_1 \equiv UPPER \vee INTERNAL \vee NULL$
   $f_2 \equiv LOWER\_OUTPUT$ and

2. For every state s in the BTL structure,
   $s \models \neg E[f_1 U f_2]$, where $f_1 \equiv LOWER \vee INTERNAL \vee NULL$
   $f_2 \equiv UPPER\_OUTPUT$.

The first (second) synchronization safety property is for the lower (upper) PCO.

Verdict safety: Intuitively, during the testing process a test case must not assign a *Fail* verdict to any behavior allowed by the protocol specification. Therefore, a BTL structure representing the composed behavior of a test and a protocol specification must not contain any state with the (Verdict == Fail) predicate true. Therefore, the verdict safety property is formulated as follows.

$$INIT \models AG(\neg(Verdict == Fail))$$

5.2. *Liveness property*

While testing a protocol implementation using a test case, if the implementation fulfills the *test purpose*, the test case assigns a *Pass* verdict. Therefore, the test case behavior satisfying the test purpose must end with a Pass verdict. Satisfaction of the test purpose is a good thing that must happen in a test case. Thus, a test case that has the liveness property means the test behavior satisfies the test

purpose and eventually assigns a Pass verdict. The liveness property is formally stated as follows.

$INIT \models (f_1 \mapsto (Verdict == Pass))$, where f_1 is a temporal formula representing the test purpose. In the following, we present a notation for representing test purposes f_1.

5.2.1. A notation for test purposes. *5.2.1.1. Primitive test purposes.* We identify the following five primitive test purposes from which more complex test purposes can be derived. The test purposes are expressed as temporal formulas using the $\mapsto$ operator defined in section 2.

1. **Direct response**

$$(p_s \wedge AFTER(Tsend(Q_i, E_i)) \wedge (t = T)) \mapsto (p_r \wedge AFTER(Treceive(Q_j, E_j)) \wedge (T < t \leq T + T_0)$$

This purpose states that if the test system sends an event E_i to the protocol through the channel Q_i at time T with the predicate p_s true, then it receives an event E_j from the protocol through the channel Q_j during an interval of length T_0 such that the predicate p_r is true.

2. **Timed response**

$$(p_s \wedge (t = T)) \mapsto (p_r \wedge AFTER(Treceive(Q_j, E_j)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the predicate p_s is true in the test system at time T and the test system waits without doing anything, then it receives an event E_j from the protocol through the channel Q_j during an interval of length T_0 such that the predicate p_r is true. This test purpose can be used to specify a conformance requirement in which the protocol outputs an event after a timeout.

3. **No response to external input**

$$(p_s \wedge AFTER(Tsend(Q_i, E_i)) \wedge (t = T)) \mapsto (p_s \wedge \neg AFTER(Treceive(ANY\_channel, ANY\_event)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the test system sends an event E_i to the protocol through the channel Q_i at time T with the predicate p_s true and waits an interval T_0, then no event is received from the protocol specification through any of the channels during the same period. This purpose is useful to model a conformance requirement in which the protocol ignores an invalid/inopportune event and does not output any event.

4. **No response in an interval**

$$(p_s \wedge (t = T)) \mapsto (p_s \wedge \neg AFTER(Trecieve(ANY, ANY)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the predicate p_s is true in the test system at time T and the test system waits for an interval T_0, then no event is received from the protocol specification through any of the channels during the same period. This purpose is useful to model a conformance requirement in which a timer does not prematurely expire in the protocol.

5. **Eventual response**

$$(p_s \wedge (t = T)) \mapsto (p_r \wedge AFTER(Treceive(Q_i, E_i)) \wedge (T < t < \infty))$$

Intuitively, this purpose states that if the predicate p_s is true in the test system at time T and the test system waits indefinitely without doing anything, then it eventually receives an event E_i from the protocol through the channel Q_i such that the predicate p_r is true. This test purpose can be used to specify a conformance requirement in which the protocol nondeterministically outputs an event. Though this purpose looks like a special case of the *timed response* test purpose, conceptually they are different.

5.2.1.2. Composing test purposes. We define SEQ, a *sequential* operator for composing primitive test purposes into larger test purposes using the following syntax where f_1 and f_2 are two test purposes.

$s_0 \models (f_1\ SEQ\ f_2)$ iff $s_0 \models f_1$ and $\forall s_i \in last(f_1) \exists s_j \in \text{reachable}(s_i)$ s.t. $s_j \models f_2$. Intuitively, $(f_1\ SEQ\ f_2)$ means f_2 is satisfied after f_1.

The function $reachable(s)$ returns a set of states that can be reached from s.

The function $last(f)$ is the set of all the terminating states of the finite paths from s_0, over which f is evaluated. In the expression $s_0 \models f$, the formula f is evaluated over a set of paths starting with s_0.

For example, referring to figure 4, let a formula f be evaluated over all the finite paths whose extents are indicated by the bold states. Then $last(f) = \{s_3, s_5, s_6, s_9, s_{11}\}$.

6. Property Verification

Safety and liveness properties of a test case are verified by evaluating, also called *model checking*, the corresponding temporal formulas. The formulas used to express the test case properties contain expressions of the form: $E[f_1\ U\ f_2], f_1 \mapsto$

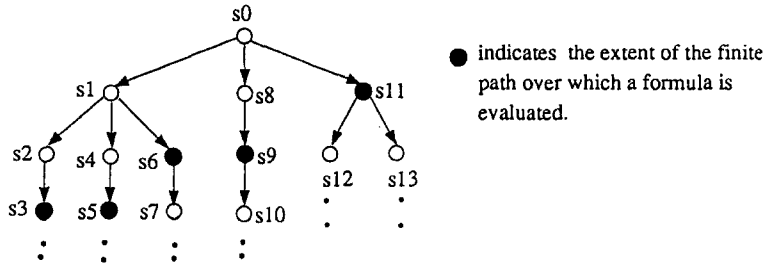

Figure 4. An example to compute $last(f)$

f_2, and $f_1\ SEQ\ f_2$. In the following, we present algorithms for each of these expressions.

In general, f_1 and f_2 can be any temporal formulas with arbitrary nesting of subformulas and temporal operators. However, in the context of this paper in expressing the safety properties of test cases, only the logical operators $\neg$, $\vee$, and $\wedge$ are used. No temporal operators such as the *until* operator are needed.

6.1. *Evaluation of SEQ*

Algorithm 3

Input: A BTL structure, s, and $f = (f_1\ SEQ\ f_2)$.
Output: True or False.

S1. **begin**
  If $\neg(s \models f_1)$ then go to step S4
  else **begin**
    Compute $S_l = last(f_1)$;
    If $S_l = \phi$ then go to Stop/fail else go to step S2
  end
S2. If $S_l = \phi$ then go to S5
  else for any $s_i \in S_l$ **begin**
    compute $S_r = reachable(s_i)$;
    $S_l \leftarrow (S_l - \{s_i\})$;
    Go to step S3
  end
S3. If $S_r = \phi$ then go to S4
  else for $s_j \in S_r$ **begin**
    If $s_j \models f_2$ then go to step S2
    else **begin** $S_r \leftarrow S_r - \{s_j\}$; go to step S3 **end**
  end

S4. *Result* := *False*; Stop.
S5. *Result* := *True*; Stop.

6.2. *Model-checking algorithm*

Using global state space S and its transitions R derived from algorithm 1 and predicated in algorithm 2 and safety and liveness properties stated in section 5, we are in a position to apply the model-checking algorithm of [23]. The following algorithm uses algorithms 3 and 4 and the algorithm for evaluating the *until* operator given in [23] to verify test cases.

Algorithm 4

Input: BTL structure M and a test case property as a temporal formula f.
Output: Correctness of f.

1. Obtain the total number of subformulas in f denoted by $length(f)$.
2. Build two arrays $nf[1 : length(f)]$ and $sf[1 : length(f)]$ where $nf[i]$ is the ith subformula of f, $sf[i]$ is the list of the numbers assigned to the immediate subformulas of the ith formula. Essentially these two arrays maintain the tree structure describing the formula f.
3. Define a bit array $L[s]$ of size $length(f)$ for each global state s such that $L[s][i]$ is set to true if the subformula $nf[i]$ holds in s.
4. /* Successively apply the state labeling algorithm label_graph to f. */
   for f_i := $length(f)$ **step** - 1 **until** 1 **do**
   label_graph($nf[f_i]$)

Procedure *label_graph*(f)
begin
{main operator of f is AU}
{execute the corresponding procedure in [8]}
{main operator of f is EU}
{execute the corresponding procedure in [8]}
{main operator of f is AG}
{execute the corresponding procedure in [8]}
{main operator of f is AF}
{execute the corresponding procedure in [8]}
{main operator of f is SEQ}
execute Algorithm 3 for f
end

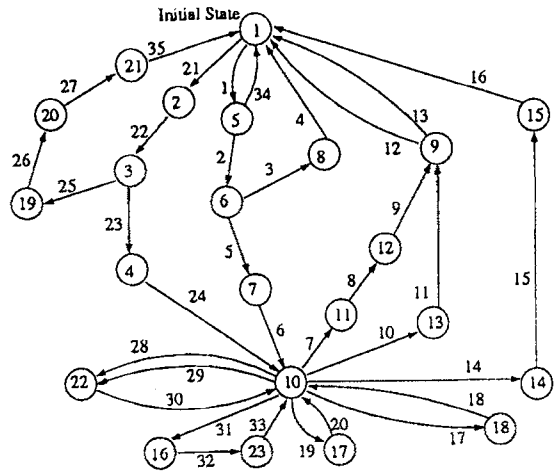

Figure 5. Class 2 transport protocol specification.

In the above procedure, we do not need a procedure to evaluate the operator $\mapsto$, since $(f_1 \mapsto f_2) \equiv AG(f_1 \rightarrow AF(f_2))$.

7. Example

In this section, we present an example of verifying a test case chosen from a CS architecture–based test suite [13] designed to test a class 2 transport protocol implementation. The CS architecture and its corresponding test verification system are shown in figures 2 and 3, respectively.

7.1. Class 2 transport protocol specification

A nondeterministic EFSM model of a class 2 transport protocol specification [32] is shown in figure 5. State 1 is both the initial and final state of the EFSM and represents a *closed connection*. State 10 corresponds to an *open connection* state. On one hand, if the connection establishment procedure is initiated by the user of the transport entity, then the EFSM moves from state 1 to state 10 through the state sequence {1, 2, 3, 4, 10}. In this case, if the peer entity refuses to establish a connection, then the EFSM goes back to the initial state through the sequence {1, 2, 3, 19, 20, 21, 1}. On the other hand, if the connection is established by the peer entity, then the EFSM moves from state 1 to state 10 through the sequence {1, 5, 6, 7, 10}. In this case, a request for connection establishment can be refused by the protocol EFSM such that the EFSM goes back to the

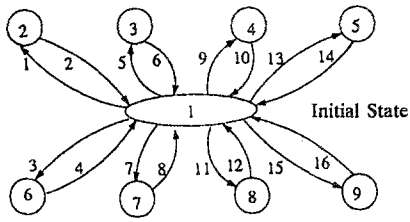

Figure 6. Underlying service provider of the transport protocol.

initial state through the state sequence {1, 5, 1}. However, if the request for connection establishment is refused by the user of the protocol entity, then the protocol EFSM goes back to the initial state through the sequence {1, 5, 6, 8, 1}. There are two internal transitions in the EFSM. The first internal transition from state 10 to 11 models the effect of the environment on the protocol specification leading to a disconnection of the transport connection along the state sequence {10, 11, 12, 9, 1}. The second internal transition from state 10 to 18 models the acknowledgment (AK) transmission policies including timeouts. The transitions of the EFSM shown in figure 5 are given in Appendix A.

7.2. Service provider

The transport protocol, in order to provide the desired service to its user, uses the services of a network layer. That is, the network layer acts as the underlying service provider in the CS test architecture. A simplified EFSM view of the service provider is shown in figure 6. The transitions of the EFSM are given in Appendix B

7.3. Test management protocol (TMP)

A TMP is required to function as a test responder (upper tester) while testing an implementation in the CS architecture. In this section, we explain a TMP [13] corresponding to a transport protocol. The TMP contains three types of internal variables: *counts, models*, and *stored items*.

There are 38 count variables, C1–C38, which monitor the traffic of transport service primitives across the interface between the TMP and the transport protocol entity. Counts are assigned to each category of service primitive in each direction across the interface. Data and expedited data octets received by the TMP entity are also counted. The behavior of the TMP is controlled by the lower tester through a set of 25 mode parameters, M1–M25. Mode parameters are used to define the series of actions which make up the response to received events

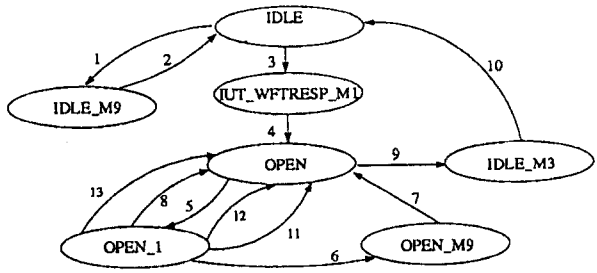

Figure 7. Test management protocol.

or for generating data to be sent. The 28 stored items S1–S28 consist of additional variables, which include the last received parameters from incoming service primitives and the values supplied as parameters to outgoing primitives. The TMP's internal variables can be controlled and monitored by the lower tester through the 23 test management protocol data units (TMPDUs), which can be classified into two groups: *command* and *reply*. The lower tester can issue a command to the TMP to take some action by sending a command TMPDU and the TMP can send a reply to the lower tester through a reply TMPDU.

We show the EFSM description of the part of the TMP that is used in the verification of a test case in figure 7. The transitions of the TMP EFSM are given in Appendix C.

7.4. Test case

For verification purposes we select a test case from a human-designed CS architecture–based test suite [13]. The main dynamic behavior of the test case is shown in figure 8. For the sake of clarity, the subtrees of the test case, identified by a "+" sign in front of them, are not shown here. The functionality of the test case consists of two distinct phases: *preamble* and *test body*. The preamble part consists of five steps. In the first step, the test case establishes a transport connection between the LT and the TMP. In the second step, the LT sends a command TMPDU (TMPDU1) directing the TMP to set the default values of its counters, mode parameters, and stored items. In the third step, the LT sends a command TMPDU (TMPDU4) directing the TMP to set the value of the stored item S8, which is used as the "User_data" parameter in a T-CONNECT response primitive. The "User_data" parameter in a T-CONNECT response primitive becomes the "User_data" parameter in a connect confirm transport protocol data unit (CC TPDU). In the fourth step, the LT waits for acknowledgments of the previously sent two DT TPDUs containing the TMPDUs from the implementation. In the fifth step, the LT disconnects the transport

Test Case Dynamic Behavior

Identifier: ABCT2URA00
Group:
Purpose: LT sends CR, receives CC with VAL octets of user data.
Default: Def1
Comments :

Nr.	Lab.	Behavior Description	Constraint	Verdict	Comment
		Test (VAL)			
		preamble			
		+ LT_con			Step1: Establish a transport connection.
		L!TMP1()			Step2: Send TMPDU1 in a DT TPDU.
		L!TMP4 (S8 := VAL)			Step3: Send TMPDU4 in a DT TPDU.
		+ Wait_for_ak			Step4: Wait for acknowledgements.
		+ LT_dis			Step5: Release the transport connection.
		body			
		+ preamble			
		L!CR	CR1		Initiate a connection establishment.
		Start (A, no_response)			Start a no_response timer.
		L?CC [user_data = VAL]	CC1	Pass	Connection is established.
		| Cancel (A)			
		| + PO1/Postamble			Release the connection.
		|			
		?Timeout (A)		Fail	Implementation not responding

Comments: This test relies on the ability to control user data.

Figure 8. Single connection test case.

connection. The objective of the preamble is to set S8 to a known value VAL.

In the test body, the LT initiates the establishment of a transport connection with the TMP by sending a connection request (CR) TPDU and waiting for a CC TPDU. If the LT receives a CC TPDU with VAL as the "User_data" parameter, then the LT assigns a Pass test verdict and releases the transport connection. Otherwise, if the LT receives a CC with the "User_data" value different from VAL, or it receives a different TPDU, or a timeout occurs, then the LT assigns a Fail test verdict and terminates its operations. The transitions of the above test case are given in Appendix D.

7.5. *Model generation*

We generate the global state space of the test verification system, shown in figure 3, by using the LT-EFSM in figure 9, the USP-EFSM in figure 6, the S-EFSM in figure 5, and the TMP-EFSM in figure 7. The global state space contains 110 states and 109 transitions. We show the entire global state space in EFSM notation in [20] and only partially in graphical form in figure 10.

The sets of executable transitions in the global states $g16$ and $g33$ are given by $XT(g16) = \{LT\_5, LT\_12\}$ (Appendix D) and $XT(g33) = \{SPEC\_7, SPEC\_17\}$ (Appendix A.) Global states $g16$ and $g17$ have been obtained by perturbing state $g16$ using the transitions LT_12 and LT_6 in $XT(g16)$, respectively. Similarly,

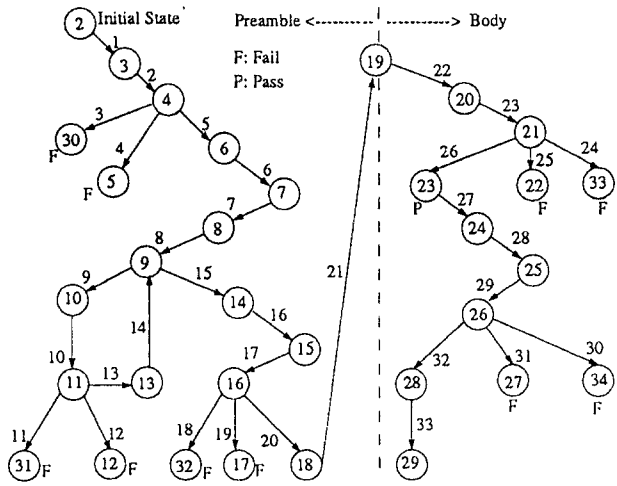

Figure 9. EFSM representation of the test case in figure 8.

states $g34$ and $g87$ have been obtained by perturbing state $g33$ using the transitions SPEC_17 and SPEC_7 in $XT(g33)$, respectively.

7.6. *Verification of safety and liveness properties*

The global state space of the TVS, shown in figure 10, contains one initial state and 10 final states. Therefore, there are 10 paths from the initial state to the final states. Let us denote a path by a function $path(g_i, g_j)$, where g_i and g_j are the initial and a final states, respectively. The 10 paths in the global state space are denoted by $path(g_1, g_6)$, $path(g_1, g_{96})$, $path(g_1, g_{97})$, $path(g_1, g_{40})$, $path(g_1, g_{46})$, $path(g_1, g_{102})$, $path(g_1, g_{110})$, $path(g_1, g_{60})$, $path(g_1, g_{78})$, and $path(g_1, g_{86})$.

7.6.1. Safety properties. In the given test case, there are a few transmission safety errors. Let us consider the property

$$INIT \models AFTER(Ssend(N, NDTreq(CC))) \mapsto AFTER(Treceive(L, NDTind(CC))).$$

The predicate AFTER(Ssend(N, NDTreq(CC))) holds in the global state g_{14}, but the predicate AFTER(Treceive(L, NDTind(CC))) holds only in state g_{17}. Therefore, the property $AFTER(Ssend(N, NDTreq(CC))) \mapsto AFTER(Treceive(L, NDTind(CC)))$ holds on all the paths except $path(g_1, g_6)$. That is, the safety property does not hold on all possible executions of the test system. This safety error arises because of a timeout in the test case as explained in the

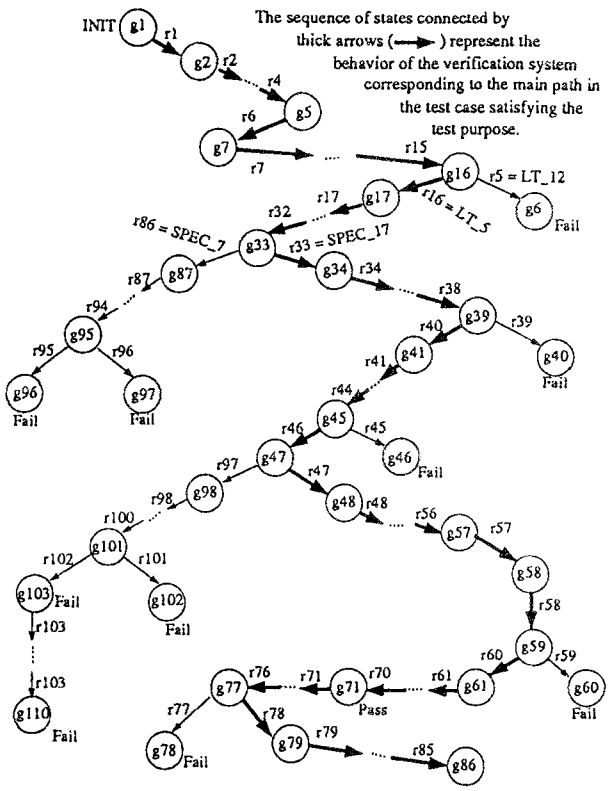

Figure 10. Global state space.

following.

From the global state g_{16}, there are two possible transitions, $r5$ and $r16$, which are derived from the LT_4 and LT_5 transitions in the test case EFSM. Transition LT_4 represents a timeout event as an alternative to transition LT_5, which is an L?NDTind(CC) event. If the timeout occurs before the test case receives the NDTind(CC) event from PCO L, then the safety error would occur.

Consider another transmission safety property

$$INIT \models AFTER(Ssend(N, NDTreq(AK))) \mapsto AFTER(Treceive(L, NDTind(AK))).$$

The predicate $AFTER(Ssend(N, NDTreq(AK)))$ holds in state g_{35}, but the predicate Treceive(L, NDTind(AK)) holds only in state g_{41}. Therefore, the formula

$$AFTER(Ssend(N, NDTreq(AK))) \mapsto$$
$$AFTER(Treceive(L, NDTind(AK)))$$

does not hold on the path $path(g_1, g_{40})$ leading to a transmission safety error in the test system. This error is also due to the timeout in the test case represented by the LT_12 transition.

The other transmission safety properties, which are not satisfied due to the test case timeout transitions LT_19, LT_25, and LT_31, are

$$INIT \models (AFTER(Tsend(L, NDTreq(DR))) \mapsto AFTER(Sreceive(N, NDTind(DR)))),$$
$$INIT \models (AFTER(Tsend(L, NDTreq(CR))) \mapsto AFTER(Sreceive(N, NDTind(CR)))), \text{ and}$$

$$INIT \models (AFTER(Tsend(L, NDTreq(DC))) \mapsto AFTER(Sreceive(N, NDTind(DC)))),$$

respectively. It is observed that if the timeouts in the test case are appropriately tuned, then all the above transmission safety properties are satisfied. That is, in a qualitative sense, if the timeouts are well tuned, then the global transitions $r5$, $r39$, $r45$, $r59$, and $r77$ would be absent from the global state space and the safety properties would hold in the remainder of the state space.

Two other transmission safety properties, which do not hold in the test system, are:

$$INIT \models (AFTER(Ssend(N, NDTreq(DR))) \mapsto AFTER(Treceive(L, NDTind(DR)))) \text{ and}$$
$$INIT \models (AFTER(Ssend(N, NDTreq(AK))) \mapsto AFTER(Treceive(L, NDTind(AK)))).$$

The predicate $AFTER(Ssend(N, NDTreq(DR)))$ holds in the state g_{89}, but the predicate does not hold in any of the states following g_{89}. To find out the cause of the error, we analyze the test system in the following.

The test system arrives at global state g_{89} from state g_{33} due to the global transition sequence $\{r_{86}, r_{87}, r_{88}\}$ derived from the transition sequence $\{SPEC\_7, SPEC\_8, SPEC\_9\}$ in the protocol specification EFSM. The sequence $\{SPEC\_7, SPEC\_8, SPEC\_9\}$ means the specification nondeterministically releases an open connection by sending a TDISind primitive to its service user and a DR TPDU to its peer entity, which is the test case EFSM in this situation. The safety error

denotes a design error in the test case in the sense that the test case is not ready to receive the DR TPDU from the protocol specification.

Similarly, the second transmission safety error is due to a design error in the test case in the sense that after the establishment of a transport connection, the test case is not ready to receive spontaneous acknowledgment (AK) TPDUs from the protocol specification.

7.6.2. Liveness property. To verify the liveness property of the test case, we first express the test purpose as a temporal formula. As a part of the test case, the test purpose is specified in a natural language as follows:

1. *LT sends CR.*
2. *LT receives CC with VAL octets of user data.*

In the following, corresponding to each part of the test purpose, we state a temporal formula:

1. $AFTER(Tsend(L, NDTreq(CR)))$.
2. $(AFTER(Treceive(L, NDTind(CC)))$ and $(NDTind.CC.User\_data = VAL))$.

We compose these basic test purposes using the SEQ operator to give rise to a formula for the entire test purpose as follows:

$$f_1 = (AFTER(Tsend(L, NDTreq(CR))))SEQ\ (AFTER(Treceive(L, NDTind(CC))) \wedge (NDTind.CC.User\_data = VAL))).$$

Then, the liveness property of the test case is stated as $INIT \models (f_1 \mapsto (verdict = Pass))$.

The liveness property is satisfied by the sequence of global states denoted by the $path(g1, g86)$. The predicate $(AFTER(Tsend(L, NDTreq(CR))))$ holds in state $g58$ and the predicates $(AFTER(Treceive(L, NDTind(CC))))$ and $(NDTind.CC.User\_data = VAL)$ hold in a subsequent state $g71$ where the predicate $(verdict = Pass)$ also holds.

All other paths do no satisfy the test purpose. The paths reachable due to the timeout transitions ($r5$, $r39$, $r45$, $r59$, and $r77$) can be avoided from the global states by appropriately turning the timeout intervals. However, the test purpose is not satisfied if the implementation behaves nondeterministically.

8. Related research

The topic of test case verification is a relatively new research area. There are very few published works in the direction of verifying single connection test cases, that is, test cases with no parallelism [33–35].

The first verification approach [33] consists of two steps. First, a TTCN test case is translated into a LOTOS specification. Second, a test and trace analysis tool, based on a LOTOS interpreter [36], takes the LOTOS description of the test case and that of the protocol specification as inputs and computes their parallel composition by tracing all the executable paths in the two specifications. The concerns expressed about the tool are its high run-time space requirement and long verification time. Another disadvantage of the technique is that it is applicable only to test cases designed to run in the local single-layer test architecture.

The second verification approach [34] consists of three steps. First, a TTCN test case and the formal description of a protocol are translated to a common EFSM notation. Second, a global state space representing all the possible interaction sequences between the test case and the protocol specification is obtained using a reachability analysis algorithm. Third, the global state space is analyzed to detect errors such as unspecified receptions, deadlocks, synchronization errors, etc. [30]. This approach does not take into account many test system attributes such as test architectures, test management protocols, test purpose, etc.

In the third verification approach [35], a tool interprets TTCN test cases and compares the defined behavior with the behavior produced by a formal description technique (FDT) simulator. In this approach, there is no formal model of the verification process, rather the verification process has been treated in an informal and intuitive manner.

9. Conclusions

We presented a test case verification methodology consisting of four steps: generating a state space by combining the behaviors of the test case and protocol specification EFSMs, associating atomic propositions to these states, expressing the safety and liveness properties of a test case as branching time temporal logic formulas, and finally verifying the test case properties on the state space using a model-checking algorithm. The complexity of the model-checking process is linear in the size of the state space.

We identified four types of test safety properties: transmission safety, reception safety, synchronization safety, and verdict safety. The first two types of safety properties are due to the communication functions of a test case and are similar to the safety properties of any data transmission protocol. The last two safety properties are unique to test systems. The synchronization safety is due to the fact that output events of the lower and upper testers must satisfy some timing constraints. The verdict safety property ensures that the test case does not assign a Fail verdict to an implementation behavior accepted by the protocol specification. Since the assignment of a Pass verdict is directly coupled with the satisfaction of the test purpose, a notation was presented to formally specify test purposes. The liveness property of a test case is derived using the test purpose

and the Pass verdict.

We applied the verification methodology to a test case in a real test suite developed for the transport protocol. Our methodology led to the detection of a few errors in safety properties. Those errors were due to the nondeterministic output acknowledgment PDUs and nondeterministic disconnection of transport connections.

In our technique, so far only a sequential composition of primitive test purposes was sufficient to model the test cases of the test suites we have studied. Investigation of other composition operators, especially parallel composition operators would be useful in the context of concurrent TTCN, which is being defined by the standardization organizations. We assumed that fixed values are assigned to the PDU/ASPs exchanged during global state space computation. Consideration of symbolic values in the input/output structures is another extension of our methodology.

In this verification methodology, we generate the global behavior of a test verification system using a reachability analysis technique. State explosion is a limitation of any verification technique involving rechability analysis. In the context of protocol validation, a number of techniques have been proposed to contain the state explosion [37]. The state explosion problem arising while generating the global behavior representing all possible interaction sequences between a test case and a protocol specification is analyzed as follows. A protocol specification is assumed to provide several protocol functions and, generally, one test case is designed to test one protocol function. That is, due to the execution of one test case, only a small part of a large protocol specification is activated. Therefore, the process of generating the global behavior representing all the interaction sequences of a test case represented by a small EFSM and a protocol specification denoted by a large EFSM would not result in state space explosion.

The liveness property is expressed using a sequential combination of a few basic test purposes which may contain predicates in terms of real-time. Further research is needed to incorporate the notion of time in protocol and test case models similar to the timed-transition systems in [38].

Acknowledgments

The authors are grateful to the reviewers for careful reading and constructive criticism.

Appendix 1: Transitions of the EFSM shown in figure 5

```
The following variables are updated during the example state
exploration process.
opt := ""       /* negotiation option */
```

```
PRseq :=0       /*receive sequence number */
PRcredit :=2    /* receive credit */
SPEC_1:     <1, 5, N?NDTind (CR), T,
             {opt :=CR.Exp_option, PScredit :=CR.credit}, 1>
SPEC_2:     <5, 6, U!TCONind(CR.called_addr,
             CR.calling_addr, opt, CR.Qos, CR.User_data),
             T, {}, 1>
SPEC_3:     <6, 8, U?TDISreq, T, {}, 1>
SPEC_4:     <8, 1, N!NDTreq(DR(TDISreq.Reason,
             IDISreq.User_data)), T, {}, 1>
SPEC_5      <6, 7, U?TCONresp, [TCONresp.Exp_option<=opt],
             {opt:TCONresp.Exp_option, PRseq:=PSseq:=0}, 1>
SPEC_6:     <7, 10, N!NDTreq(CC(TCONresp.Calling_addr, TCONresp.Qos,
              opt, PRcredit, TCONresp.User_data), T, {}, 1>
SPEC_7:     <10, 11, i, T, {}, 1>
SPEC_8:     <11, 12, U!TDISind (Reason, User_data),
             T, {User_data:NULL}, 1>
SPEC_9:     <12, 9, N!NDTreq(DR(Reason, User_data)),
             T, {User_data:=NULL}, 1>
SPEC_10:    <10, 13, U?TDISreq, T, {}, 1>
SPEC_11:    <13, 9, N!NDTreq(DR(TDISreq.Reason,
             TDISreq.User_data)), T, {}, 1>
SPEC_12:    <9, 1, N?NDTind(DC), T, {}, 1>
SPEC_13:    <9, 1, N?NDTind(DR), T, {}, 1>
SPEC_14:    <10, 14, N?NDTind(DR), T, {}, 1>
SPEC_15:    <14, 15, U!TDISind (DR, Reason, DR.User_data), T, {}, 1>
SPEC_16:    <15, 1, N!NDTreq (DC), T, {}, 1>
SPEC_17:    <10, 18, i, T, {}, 1>
SPEC_18:    <18, 10, N!NDTreq (AK (PRcredit)),
             T, {}, 1>
SPEC_19:    <10, 17, N?NDTind(DT),
             [PRcredit <> 0 & DT.Seq=PRseq, {}, 1>
SPEC_20:    <17, 10, U!TDATAubd (DT.User_data, DT.EOT), T,
              {PRseq:=(PRseq+1)mod 128, PRcredit:PRcredit-1}, 1>
SPEC_21:    <1, 2, U?TCONreq, T, {opt:=TCONreq.proposed_options}, 1>
SPEC_22:    <2, 3, N!NDTreq(CR(TCONreq.called_addr,
             TCONreq.Calling_addr, opt, PRcredit)),
             T, {}, 1>
SPEC_23:    <3, 4, N?NDTind(CC), [CC.Exp_option <=opt],
             {opt:CC.Exp_option, PRseq:=0,
              S_credit:=CC.credit_value}, 1>
SPEC_24:    <4, 10, U!TCONcon(CC.Calling_addr,
             CC.Exp_option, CC.Qos, CCUser_data),
             T, {}, 1>
SPEC_25:    <3, 19, N?NDTind(CC),
             [not(CC.Exp_option <=opt)], T, {}, 1>
SPEC_26:    <19, 20, N!NDTreq(DR(procedure_error,
             User_data)), T, {User_data:=NULL}, 1>
SPEC_27:    <20, 21, U!TDISind (procedure_error,
             User_data), T, {User_data:=NULL}, 1>
SPEC_28:    <10, 22, N?NDTind(AK),
             [TSseq < AK.expected_send_sequence],
```

```
            {new_credit:=AK.credit_value +
            AK.expected_send_sequence - (TSseq + 128)}, 1>
SPEC_29:    <10, 22, N?NDTind(AK),
            [PSseq>=AK.expected_send_sequence],
            {new_credit:AK.credit_value +
            AK.expected_send_sequence - PSseq}, 1>
SPEC_30:    <22, 10, i, T, {PScredit:=new_credit}, 1>
SPEC_31:    <10, 16, U?TDATAreq,
              [PScredit > 0], {PScredit:PScredit - 1}, 1>
SPEC_32:    <16, 23, n!ndtREQ(dt(tsREQ,
            tdaataREQ.ts_USER_DATA, TDATAreq.EOT)),
            T, {}, 1>
SPEC_33:    <23, 10, I, t,
            {PSseq:=(PSseq+1) mod 128}, 1>
SPEC_34:    <5, 1, N!NDTreq (DR(Reason, User_data)),
            [opt not supported], {User_data:Null}, 2>
SPEC_35:    <21, 1, N?NDTind(DC), T, 1>
```

Appendix 2: Transitions of the EFSM given in figure 6

```
USP_1:  <1, 2, L?NCONreq(x), T, {}, 1>,    USP_2:  <2, 1, N!NCONind(x), T, {}, 1>
USP_3:  <1, 6, N?NCONresp(x), T, {}, 1>,   USP_4:  <6, 1, L!NCONind(x), T, {}, 1>
USP_5:  <1, 3, L?NCONresp(x), T, {}, 1>,   USP_6:  <3, 1, N!NCONconf(x), T, {}, 1>
USP_7:  <1, 7, N?NCONresp(x), T, {}, 1>,   USP_8:  <7, 1, L!NCONconf(x), T, {}, 1>
USP_9:  <1, 4, L?NDTreq(x), T, {}, 1>,     USP_10:  <4, 1, N!NDTind(x), T, {}, 1>
USP_11:  <1, 8, N?NDTreq(x), T, {}, 1>,    USP_12:  <8, 1 L!NDTind(x), T, {}, 1>
USP_13:  <1, 5, L?NDISreq(x), T, {}, 1>,   USP_14:  <5, 1, N!NDISind(x), T, {}, 1>
USP_15:  <1, 9, N?NDISreq(x), T, {}, 1>,   USP_16:  <9, 1, L!NDISind(x), T, {}, 1>
```

Appendix 3: Transition of the EFSM shown in figure 7

```
TMP_1:    <IDLE, IDLE_M9, Internal_START, [T],
           {C1:=0, C2:=0, ..., C38:0, , M1:=A4, M2:=a0, ...,
            M10:=A0, M11:=A5, M12:=A5, M13:=A0, ..., M25:=A0,
            S1:=.., S2:=.., , , .S5:="1", S6:="you",
            S7:="No", S8:="any data", .., S28}, 1>
TMP_2:    <IDLE_M9, IDLE, Null, [M9=A0], {}, 1>
TMP_3:    <IDLE, IUT_WFTRESP_M1, U?TCONind, [T],
           {S16:=TCONind.Called_address,
            S17:=TCONind.Calling_address,
            S18:=TCONind.Exp._data_option,
            S19:=TCONind.Qos,
            S20:=TCONind.TSuser_data, inc(C1, C19), 1>
TMP_4:    <IUT_WFTRESP_M1, OPEN,
           U!TCONresp (S5, S6, S7, S8), [M1=A4],
           {inc (C24, C37)}, 1>
TMP_5:    <OPEN, OPEN_1, U?TDTind, [P_TMPDU],
           {inc (C4)per octet, inc (C5, C19)}1>
```

```
TMP_6:    OPEN_1, OPEN_M9, Null, [TMPDU1 in TDTind],
          {M1:=TDTind.  TMPDU_M1
           M2:=TDTind.TMPDU_M2, ...,
           M25:=TDTind.TMPDU_M25}, 1>
TMP_7:    OPEN M9, OPEN, Null, [M9=A0], {}, 1>
TMP_8:    OPEN_1, OPEN, Null, [TMPDU4 in TDTind],
          {S5:TDTind.TMPDU4.S5, S6:TDTind.  TMPDU4.S6,
           S7:=TDTind.  TMPDU4.S7, S8:=TDTind.TMPDU4.S8}, 4>
TMP_9:    <OPEN, IDLE_M3, U?TDISind, [T],
          {S14:TDISind.  Reason,
           S15:=TDISind.TSuser_data, inc(C3, C10)}, 1>
TMP_10:   <IDLE_M3, IDLE, Null, [M3=A0], |, 1>
TMP_11:   <OPEN_1, OPEN, Null, [TMPDU3 in TDTind],
          {S1:=TDTind.  TMPDU3.S1}, 3>
TMP_12:   <OPEN_1, OPEN, Null, [TMPDU5 in TDTind],
          {S9:=TDTind.  TMPDU5.S9,
           S10:=TDTind.  TMPDU5.S10,
           S11:=TDTind.  TMPDU5.S11
           S12:=TDTind.  TMPDU5.S12
           S13:=TDTind.  TMPDU5.S13}, 5>
TMP_13:   <OPEN_1, OPEN, U!TDTreq(TSuser_data), [TMPDU8 in TDTind],
          {TSuser_data:=HERALD||"8"||"25"||...
           ...||m25||TRAILER}, 8>
```

Appendix 4: Transitions of the EFSM shown in figure 9.

```
Following are the verdict TAGS of the states:
verdict (5) =verdict (12) = verdict(17) = verdict(22) =Fail
verdict (30) =verdict (31) = verdict(32) = Fail
verdict (33) =verdict (34) =Fail verdict(23) = Pass
Initialization:

VAL :="test_data"
/* Parameters of a TCONresp primitive.*/

TS5 :=1              /*Quality of services*/
TS6:= "you           /*Called_address*/
TS7 := No            /*Expedited data option*/
TS8 := VAL           /*User_data*/
Called_address := "you"
Calling_address := "me"
Exp_option :="No"
Qos := "1"
User_data0 := "any_data"
Seeqrecak := 0
Seqsendt := 0

Transitions of the LT EFSM :

LT_1:  <2, 3, L!NDTreq(CR(Called_addr, calling_addr,
        Exp_option, Qos, User_data0)), T, {}, 1>
```

```
LT_2:  <3, 4, Start (A, no_response), T, {}, 1>
LT_3:  <4, 5, L?OTH (A), T, {}, 3>
LT_4:  <4, 5, ?Timeout (A), T, {}, 2>
LT_5   <4, 6, L?NDTind (CC), T, {}, 1>
LT_6:  <6, 7, Cancel (A), T, {}, 1>
LT_7:  <7, 8, L!NDTreq (DT(User_data1, EOT)),
        {M1:=A4, M2:=A0, ..., M10:=A00, M11:=A5, M12:=A5,
         M13:=A0, ..., M25:=A0,
         User_data1:=TMP1(M1, M2..., M25),
         EOT:=True, seqsendt:=0}, 1>
LT_8:  <8, 9, L!NDTreq (DT(User_data2, EOT)),
        {User_data2:=TMP4(TS5, TS6, TS7, TS8),
         EOT:=True, seqsendt:=1}, 1>
LT_9:  <9, 10, Null, [seqrecak < seqsendt], {}, 1>
LT_10: <10, 11, Start (B, wait_ak), T, {}, 1>
LT_11: <11, 31, L?OTH, T, {}, 3>
LT_12: <11, 12, ?Timeout (B), T, {}, 2>
LT_13: <11, 13, L?NDTind(AK), T, {seqrecak:=AK.seqno}, 1>
LT_14: <13, 9, Cancel (B), T, {}, 1>
LT_15: <9, 14, Null, [seqrecak >= seqsendt], {}, 1>
LT_16: <14, 15, L!NDTreq(DR(Reason, User_data3)), T,
        {Reason:="normal_disconnect"' User_data3:=Null}, 1>
LT_17: <15, 16, Start (A, wait_dc), T, {}, 1>
LT_18: <16, 32, L?OTH, T, {}, 3>
LT_19: <16, 17, ?Timeout (A), T, {}, 2>
LT_20: <16, 18, L?NDTind (DC), T, {}, 1>
LT_21: <18, 19, Cancel (A), T, {}, 1>
LT_22: <19, 20, L!NDTreq(CR(Called_addr, calling_addr,
        Exp_option, Qos, User_data0)), T, {}, 1>
LT_23:  <20, 21, Start (A, no_response), T, {}, 1>
LT_24: <21, 33, L?OTH (A), T, {}, 3>
LT_25: <21, 22, ?Timeout (A), T, {}, 2>
LT_26: <21, 23, L?NDTind(CC), [CC.User_data=VAL], {}, 1>
LT_27: <23, 24, Cancel (A), T, {}, 1>
LT_28: <24, 25, L!NDTreq(DR(Reason, User_data3)), T,
        {Reason:="normal_disconnect"' User_data3:=Null}, 3>
LT_29: <25, 26, Start (A, wait_dc), T, {}, 1>
LT_30: <26, 34, L?OTH, T, {}, 3>
LT_31: <26, 27, ?Timeout (A), T, {}, 2>
LT_32: <26, 28, L?NDTind (DC), T, {}, 1>
LT_33: <28, 29, Cancel (A), T, {}, 1>
```

References

[1] K. Naik and B. Sarikaya Testing communication protocols. *IEEE Software*, 27–37, 1992

[2] ISO/IEC 9646: Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework, 1991.

[3] ISO/IEC IS8807: LOTOS, a formal description technique based on the temporal ordering of observable behavior, ISO/TC97/SC21/WG1-FDT/SC-C, June 1988.

[4] T. Bolognesi and E. Brinksma. Introduction to ISO specification language LOTOS. *Computer*

Networks and ISDN Systems, 25–59 1987.

[5] ISO/IEC IS9074: Estelle – A formal description technique based on an extended state transition model, ISO/TC97/SC21/WG1, 1987.

[6] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14: 3–23, 1987.

[7] CCITT, Specification and Description Language SDL, Recommendation Z.100, 1992.

[8] ISO/IEC 8824: Profile of abstract syntax notation one, IS8824, 1987.

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

[10] A.T. Dahbura, K.K. Sabnani, and M.U. Uyar. Formal methods for generating protocol conformance test sequences *Proceedings of the IEEE*, 78 (8): 1317–1326, 1990.

[11] B. Sarikaya, G.v. Bochmann, E. Cerny. A test design methodology for protocol testing. *IEEE Trans. on Software Eng.*, 13(5): 518–426, 1989.

[12] P. Tripathy and B. Sarikaya. Test case generation from LOTOS specification. *IEEE Trans. on Computers*, 40: 543–552, 1991.

[13] Abstract test suite for transport protocol class 2. The National Computing Centre Limited, Manchester, UK, 1988.

[14] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *IEEE Trans. on Sofware Eng.*, SE-1 (2): 20–37, 1975.

[15] B. Sarikaya. Conformance testing: Architectures and test sequences. *Computer Networks and ISDN Systems*, 17: 111–126, 1989.

[16] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM* 30, (2): 323–342, 1983.

[17] K. Naik and B. Sarikaya. An extended finite state machine model for TTCN. Proc. of the 15th. Biennial Symposium on Communications, Kingston, Ontario, 1990, pp. 296–299.

[18] Information Processing Systems – Open System Interconnection – Basic Reference Model, ISO 7498, 1984.

[19] M. Jackson. *System development*. Prentice Hall, 1983.

[20] K. Naik. *Verification of test cases for protocol conformance testing*. Ph.D. theseis, Concordia University, Montreal, 1992.

[21] G. v. Bochmann. Hardware specification with temporal logic: An example. *IEEE Trans. on Computers*, C–31: 223–231, 1982.

[22] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5, (2): 190–222, 1983.

[23] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8 (2), 224–263, 1986.

[24] K. Sabnani. An algorithm technique for protocol verification. *IEEE Transaction on Comm*, COM–36 (8): 924–931, 1988.

[25] M. Ben–Ari. A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20: 207–225, 1983.

[26] J.C. Fernandez, J.L. Richier, and J. Voiron. Verification of protocol specifications using the CESAR system. IFIP PSTV V, 1985.

[27] P. Tripathy. *A Unified Model for test generation for communication protocols*. Ph.D. thesis, Concordia University, Montreal, 1992.

[28] C.H. West. General techniques for communication protocol validation. *IBM Journal of Res. and Development*, 22 (4): 393–404, 1978.

[29] J. Rubin and C.H. West. An improved protocol validation technique. *Computer Networks*, 6: 65–73, 1982.

[30] P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan, and D. Brand. Towards analyzing and synthesizing protocols. *IEEE Trans. on Comm.*, COM–28 (4): 651–661, 1980.

[31] B.Sarikaya and G. v. Bochmann. Synchronization and specification issues in protocol testing. *IEEE Trans.* on Comm. COM–32 (4): 389–395, 1984.

[32] G. v. Bochmann. Specification of a simplified transport protocol using different formal description techniques. *Computer and Networks and ISDN Systems*, 18: 335–377, 1990.

[33] M. Dubuc and G. v. Bochmann. Translation from TTCN to LOTOS and verification of Test Cases. *FORTE'90*, Madrid, 1990.

[34] K.Naik and B. Sarikaya. Verification of protocol conformance test cases using reachability analysis. *The Journal of Systems Sofware*, 19: 41–57, 1992.

[35] U. Bar and J.M. Schneider. Automated validation of TTCN test suites. IFIP PSTV XII, Orlando, FL, pp. 279–295, 1992.

[36] L. Logrippo, et. al. An interpreter for LOTOS, A specification language for distributed systems. *Software Practice and Experience*, 18: 365–385, 1988.

[37] F.J. Lin, P.M. Chu, and M.T. Liu. Protocol verification using reachability analysis: The state space explosion problem and relief strategies. SIGCOMM'87, Stowe, Vermont, 126–135, 1987.

[38] J.S. Ostroff. Deciding properties of timed transition models. *IEEE Trans. on Parallel and Distributed Systems* 1 (2): 170–183, 1990.