A NEW APPROACH TO TEST SEQUENCE DERIVATION BASED ON EXTERNAL BEHAVIOUR EXPRESSION (EBE)

by

Jianping Wu and Samuel T. Chanson

Technical Report 89-3

January 1989

A NEW APPROACH TO TEST SEQUENCE DERIVATION BASED ON EXTERNAL BEHAVIOUR EXPRESSION (EBE)†

by

Jianping Wu†† and **Samuel T. Chanson**

Technical Report 89-3

January 1989

Department of Computer Science
University of British Columbia
Vancouver, B.C.,
Canada V6T 1W5

---

† The work has been supported in part by the Natural Sciences and Engineering Research Council of Canada, and IDACOM Electronics Ltd. of Edmonton, Canada.

†† On leave from the Department of Computer Science, Tsinghua University, Beijing, China.

ABSTRACT

This paper presents a new approach to test sequence derivation from formal protocol specifications for protocol conformance testing. The approach is based on a model of External Behaviour Expression (EBE) which specifies only the external behaviour of a protocol in terms of the input/output sequences and their logical (function and predicate) relations, and can be obtained from formal protocol specifications in either Estelle or LOTOS. A basic test derivation theory is defined for the purpose of formalizing test sequence derivation strategies. Based on the EBE of a protocol, a test sequence derivation method is proposed to identify associations between inputs and outputs through the interaction paths and their I/O subpaths. Generic Test Cases generated from these I/O subpaths are based on specific testing purposes. Abstract Test Cases are selected in terms of particular test methods and additional requirements. Comparison to other existing schemes shows the method proposed here is simple and concise, and the resulting set of test sequences is complete and effective. It is our belief that this approach to test sequence derivation can provide the basis of a formalized framework for protocol conformance testing.

1 Introduction

One of challenging and important problems in the field of protocol testing is the derivation of test sequences from formal protocol specifications for the purpose of testing implementations for conformance to their specifications. Based on the various test architectures for OSI conformance testing which have been proposed [1,2], an implementation under test (IUT) is tested locally or remotely as a black box by observing its external behaviour to the stimuli provided by test sequences. A test sequence is a sequence of inputs and expected outputs for the IUT. The inputs are applied to the IUT, and outputs from the IUT are compared to those expected in the test sequences. If the outputs match the expected ones, then the IUT is said to conform to the formal protocol specification the IUT implemented; otherwise, the IUT contains one or more faults. The test sequences test the data flow as well as the control flow of an IUT. Exhaustive testing (i.e., testing for all possible input and/or output sequences) is practically infeasible because the input domain is often infinite. Consequently, a great deal of attention has been given to deriving test sequences which are minimal but provide good coverage.

Most of the existing test sequence derivation methods are based on some formal models such as the Finite State Machine (FSM), Extended Finite State Machine (EFSM), or state transition systems (e.g. [8,9,10,11,12,13,14,15]). Some of them assume that the formal protocol specification is given in a particular Formal Description Technique (FDT) such as Estelle [10] or LOTOS [11,12,13]. Generally, these methods do not take into account the Protocol Data Unit (PDU) and service primitive parameters, and only derive test sequences to test the control flow portion of an IUT.

Recently, some test sequence derivation methods taking into account the PDU and service primitive parameters have been proposed [4,5,6,7]. They assume that protocol specifications are given in Normal Form Specification (NFS) which is a variation of Estelle with a single module. The first method [4] applies the principles of functional program testing to generate test sequences. However, it requires considerable effort to identify functions and their relationships for non-trivial protocols. The second method [5,6] is based on the principles of data flow analysis techniques and generates a set of test sequences to cover all definition and usage pairs satisfying certain constraints; it is less comprehensive than the third method [7] which uses a structural test sequence selection method and seems to be better in path coverage. However, the scheme presented in [7] has difficulty distinguishing between feasible and infeasible paths.

All three methods mentioned above make use of the internal structure and variables related to protocol implementation that are contained in the formal protocol specifications. In our opinion, this is unnecessary and complicates the procedure for test sequence derivation. As well, the test sequences obtained are generally less effective in terms of coverage and some protocol behaviour could be covered more than once.

We also observe that few researchers have considered the selection of test cases on the basis of different test methods, or studied the relationships between test sequence derivation and the PICS (Protocol Implementation Conformance Statement) and the PIXIT (Protocol Implementation Extra Information for Testing).

This paper presents a new approach to test sequence derivation from formal protocol

specifications. The approach is based on an External Behaviour Expression (EBE) which specifies only the external behaviour of a protocol in order to derive test sequences, and can be obtained from formal protocol specifications in either Estelle or LOTOS. In EBE, the data flow portion and the control flow portion of a protocol are represented by externally observable input/output sequences and their logical (function and predicate) relations. A basic test derivation theory is given for the purpose of formally defining the test derivation strategies. Our test derivation strategy is defined in three steps: *Test Generation* for generic test cases, *Test Selection* for abstract test cases, and *Test Choice* for executable test cases. A test generation and selection method is proposed to identify associations between inputs and outputs in the EBE of a protocol through interaction paths and their I/O subpaths. Generic test cases obtained from these I/O subpaths are based on specific testing purposes of a protocol. Abstract test cases are selected in terms of a particular test method and additional requirements specified in the PICS and the PIXIT.

Comparing with other methods, our test sequence generation and selection method is quite simple and more concise. As well, the resulting set of test sequences is complete and effective. It is our belief that this approach to test sequence derivation can provide the basis of a formalized framework for protocol conformance testing.

The rest of the paper is organized as follows. Section 2 gives a formal definition of EBE and illustrates it by an example. Section 3 presents the basic test derivation theory, and the details of the proposed test sequence generation and selection method. Comparisons of the proposed method with some existing methods such as [4,5,6,7] are presented in Section 4 using the ISO Class 0 Transport protocol. Finally, Section 5 concludes the paper.

2 External Behaviour Expression

Estelle (Extended State Transition Model) and LOTOS (Language of Temporal Ordering Specification) are two FDT's developed by ISO (International Organization for Standardization) for the formal specification of open distributed systems, in particular the services and protocols of the layers of the Open Systems Interconnection (OSI) architecture defined by ISO. Estelle may be used to specify a protocol in terms of externally observable behaviour as well as internal actions of possibly more than one module [17]. LOTOS may be used to specify a protocol in terms of the temporal relation among the interactions that constitute the externally observable behaviour of the protocol and their internal modules [18]. In theory, either Estelle or LOTOS may be used to specify only the external behaviour of a protocol. However, applications of these FDT's to ISO protocols have always included internal structures, variables and actions of the protocols [24, 25].

In order to obtain a formal specification which describes only externally observable behaviour of a protocol for the purpose of test sequence derivation and which is also FDT independent, we propose and describe a new formal specification model called External Behaviour Expression (**EBE**) as follows.

2.1 Basic definition of EBE

The External Behaviour Expression (EBE) models the externally observable behaviour of a system in terms of possible sequences of interactions exchanged between the

system and its external environment, and possible logical relations among elements (input and/or output primitives and their parameters) of these sequences.

Definition 2.1

*An External Behaviour Expression (**EBE**) is a quadruple* $EBE = \langle S, S_0, T, R \rangle$, *where:*

- S *is a set of external finite states of the system;*
- S_0 *is the initial external state of the system;*
- T *is a set of transitions of external states; and*
- R *is a set of logic relations of transitions.*

Definition 2.2

A system described by the EBE has four types of external states:

1) ***equivalent state*** S_e *— states with identical succeeding behaviour;*

2) ***terminal state*** S_t *— a state in which there is no transition to other external states;*

3) ***nested state*** S_n *— a state in which there are some nested subEBE's; and*

4) ***common state*** S_c *— a state which is not equivalent, terminal or nested.*

For the case of equivalent states, the succeeding behaviour for only one of the states is specified in EBE. The external terminal states S_t do not have any succeeding behaviour, but a special action @ will allow a system to return to its initial state. The hierarchical and parallel compositions within a system may be mirrored externally in terms of the external nested states.

Definition 2.3

A transition of external states is the interactions exchanged between the system and its external environment in terms of input and/or output primitives and their parameters. The general form of a transition is given by $T_{ij} = (I, O)$, *where:*

1) I *is a set of input primitives from the external environment, and each input primitive is denoted by:* $I_p\ (X_{p1}, \ldots, X_{pn})$, *where "*$I_p$*" is the input primitive identifier, and* $X_{p1}, \ldots, X_{pn}\ (n \geq 0)$ *are parameters of the input primitive* I_p.

2) O *is a set of output primitives to the external environment of the system, and each output primitive is denoted by:* $O_q\ (Y_{q1}, \ldots, Y_{qm})$, *where "*$O_q$*" is the output primitive identifier, and* $Y_{q1}, \ldots, Y_{qm}\ (m \geq 0)$ *are parameters of the output primitive* O_q.

3) The absence of an input primitive or an output primitive is denoted by " — ". Thus, a transition can be in one of three forms:

a. $T_{ij} = (I, O)$;

b. $T_{ij} = (—, O)$; *and*

c. $T_{ij} = (I, —)$.

Definition 2.4

The set of logical relations of a transition $R_{ij} = (S_i, S_j, F, P)$ *holds if and only if there exists a transition* T_{ij} *from state* S_i *to state* S_j.

1) F *is a set of function relations of a transition. The output primitive parameter* $\{Y_{qp}\}$ *of a transition will be produced if and only if there exists a group of elements* Z *which satisfies a function of* $\{Y_{qp}\} = F(Z)$, *where* $q = 1, \ldots, k$ *and* $p = 1, \ldots, m$..

2) P *is a set of predicate relations of a transition. The transition will happen if and only if there exists a group of elements* Z *which satisfies the property* $P(Z)$.

3) Z *refers to those elements (usually input primitives or parameters) which occur in this transition and/or the preceding transitions, and may include those mentioned in the PICS and the PIXIT.*

The basic definitions of EBE are given above. There are two ways to describe the formal EBE model. One way to describe this model is by a directed graph which we call Behaviour Tree oriented EBE (EBE-BT), the another method is by a set of syntax rules and its operational semantics, called Normal Form oriented EBE (EBE-NF).

2.2 Behaviour Tree Oriented EBE

The Behaviour Tree Oriented EBE describes a system in terms of a directed graph with tree structure. In the EBE-BT, tree nodes represent externally observable states of a system (i.e., the set S). In particular, the tree root of EBE-BT is the initial state S_0. Tree branches linking tree nodes represent transitions among the external states of the EBE (i.e., the set T). Logical relations associated with a transition of the EBE may be described in terms of additional specifications (i.e., the set R). Thus it can be seen that there are clear mappings between the basic definitions of EBE and the EBE-BT.

2.3 Normal Form Oriented EBE

The Normal Form Oriented EBE is another form of describing a system using the EBE. The typical structure of system specification and process definition in the EBE-NF is shown below:

```
SPECIFICATION system_name [input/output primitive and their
                           parameter list]
    system implementation statement list
    type definition
BEHAVIOUR
    external behaviour expression
WHERE
    type definition
    process definition
ENDSPEC
```

```
process process_name [input/output primitive and their
                      parameter  list] :=
    external behaviour expression
WHERE
    type definition
    process definition
ENDPROC
```

In EBE-NF, an essential component of a system specification or a process definition is its external behaviour expression. The process definition may be used to describe hierarchical or parallel behaviour of a system. An external behaviour expression is built by applying syntax rules of the EBE-NF which is simpler than the FDT's such as Estelle or LOTOS because it describes externally observable behaviour only.

Table 1. Syntax rules of EBE-NF

Name	Notation
Inaction:	**STOP**
Transition:	S_i [$T_{ij(k)}$ | $R_{ij(k)}$] * S_j
Transition Choice:	S_i [$T_{ij1(k)}$ | $R_{ij1(k)}$] * S_{j1}
	+ ...
	+ [$T_{ijn(k)}$ | $R_{ijn(k)}$] * S_{jn}
Process Instantiation:	**P** (e_1,...,e_n)
Termination:	**EXIT**

The complete list of basic syntax rules of the EBE-NF is given in Table 1. In the table, symbols "**S**i", "**S**j", "**S**jq" are external states of a system; symbol "**T**ij(k)" represents the *k*th transition from **S**i to **S**j; and symbol "**R**ij(k)" stands for the logical relations associated with Tij(k). Basically, the EBE-NF includes three kinds of operators, i.e., nullary operators (**Stop** and **EXIT**), sequence operator "*", and parallel operator "+". A deterministic transition is built by using the sequence operator "*" between a state **S**i with **T**ij(k) | **R**ij(k) and another state **S**j, where "|" represents a condition relationship between the transition **T** and the logical relation **R**. Nondeterministic transition (i.e., conditional on the inputs) is built by using parallel operator "+" among multiple possible sequence transitions. Only one of such possible sequence transitions can be enabled because of the assumption that a system responses to external events in sequence. A process instantiation **P** (e_1, ... ,e_n) is formed by a process identifier "**P**" with an associated list (e_1, ... ,e_n) of input and/or output primitives and their parameters, where $\{e_i\} \subseteq \mathbf{T}$.

The operational semantics of the EBE-NF is given in Table 2 and provides a means to derive the actions that a system or a process may perform from the external behaviour expression itself. By applying axioms and inference rules of the operational semantics we can derive an action tree, also called a transition tree. The transition tree obtained from EBE-NF is just another form of specifying EBE which we have called EBE-BT. Thus there is a simple mapping between the two forms of the EBE model.

Table 2. Operational semantics of EBE-NF

Notation	Meaning
STOP	none
EXIT	**EXIT** — @ → **STOP** { @ is the termination action }
Si [Tij(k) | Rij(k)] * Sj: Si [Tij1(k) | Rij1(k)] * Sj1 + ... + [Tijn(k) | Rijn(k)] * Sjn:	Si — [Tij(k) | Rij(k)] → Sj Si — [Tijq(k) | Rijq(k)] → Sjq { n ≥q ≥ 1 } implies Si — [Tij1(k) | Rij1(k)] + ... + [Tijn(k) | Rijn(k)] → Sjq { n ≥ q ≥ 1 }
P (e1,...,en):	If " process **P** (e1,...en) := **EBE** endproc " is a process definition then **P** (e1,...,en) — [t ∈ T' | r ∈ R'] → S' { S' is a state set of P. T' is a transition set of P and R' is a relation set of P }

2.4 EBE of a Protocol

The EBE of a protocol may be produced directly from the protocol document in English. However, this paper will focus on derivation from FDT's such as Estelle or LOTOS.

The formal structures of Estelle are very close to those of EBE. The tree structure of the EBE associated with states and transitions can be formally obtained from the finite state machines in Estelle. Logical relations of the EBE can be formally produced by searching the operation part associated with each transition of Estelle. Thus, formal protocol specification in Estelle can be directly transformed into EBE-BT by using formalized algorithms. The operational semantics of LOTOS provides a means to derive a transition tree from its behaviour expression. This transition tree has the same structure as EBE-BT. Logical relations of the EBE can also be obtained by examining the description of data structures and value expressions in LOTOS. Therefore, formal protocol specification in LOTOS can best be transformed into EBE-BT. However some formalized algorithms may also be used to transform specifications in LOTOS into EBE-NF directly. Details of obtaining an EBE from formal protocol specifications in Estelle and LOTOS are given in [19].

An EBE for the ISO Class 0 Transport Protocol is given in the Appendix. Its EBE-BT is shown in Appendix A and its EBE-NF is given in Appendix B.

3 Test Sequence Derivation Strategy

In this section, we present a basic test derivation theory for protocol conformance testing and outline the steps of the test derivation strategy. A test generation and selection method is proposed to generate generic test cases from formal protocol specification, and select abstract test cases based on particular abstract test methods.

3.1 Basic test derivation theory for protocol conformance testing

As protocol testing is a special case of software testing, much can be learned from the theory of software testing. The following test derivation theory for protocol conformance testing has incorporated some concepts from software testing (see for example [20] and [21]), and is based on the concepts for protocol conformance testing [1].

Definition 3.1

PS is a generic specification of a protocol in terms of the externally observable behaviour with the input sequence domain D and the intended output PS(D) on D. PI is a conceptual conformance implementation of PS if there exists a finite test sequence set TS ⊆ D such that

$$PI(x) = PS(x) \text{ for all } x \in TS \Rightarrow PI(x) = PS(x) \text{ for all } x \in D.$$

Note that since *PS* is a generic specification, in the absence of implementation specific information such as those contained in PICS and PIXIT, it is nondeterministic and so are *PI* and *TS*. In other words, since there are implementation variables whose values are unspecified in *PS* and *PI*, some outputs of *TS* are nondeterministic.

Definition 3.2

PI_i is a particular real implementation of PS based on information specified in some given implementation statements. The protocol specification that PI_i implements is PS_i and its associated input sequence domain is D_i. PI_i is a conformance implementation of PS_i if there exists a TS_i, where $TS_i \subseteq TS$ and $TS_i \subseteq D_i$, such that

$$PI_i(x) = PS_i(x) \text{ for all } x \in TS_i \Rightarrow PI_i(x) = PS_i(x) \text{ for all } x \in D_i.$$

PI_i is deterministic as a real implementation must resolve any nondeterministic specification based on the PICS and the PIXIT. Thus PS_i and TS_i are also deterministic. Test sequence TS_i with the property in definition 3.2 is known as a reliable test sequence set. In other words, TS_i is reliable for PI_i if TS_i reveals that PI_i is incorrect whenever PI_i contains a conformance error (i.e., $PI_i(x) \neq PS_i(x)$ for some $x \in TS_i$). The rest of this section is concerned with the derivation of test sequences from formal protocol specifications.

Definition 3.3

Test derivation strategies TDS are procedures for generating test sequence set TS_i from formal protocol specification PS. A test derivation strategy is reliable for a protocol implementation PI_i if it produces a reliable test sequence set TS_i for PI_i.

The test sequence derivation strategies for protocol conformance testing have also

been called test generation in [10,14,15], test selection in [6,7] and test design in [3,4]. In [1,2], test sequences for protocol conformance testing are known as *Test Suites*. Test suites have a hierarchical structure in which the basic unit is the *Test Case*. Each test case has a narrowly defined purpose. Three kinds of test cases are used in protocol conformance testing, i.e., *Generic Test Cases*, *Abstract Test Cases* and *Executable Test Cases*. Their detailed definitions can be found in [2].

Definition 3.4

*Let **PS** be the formal specification of a protocol. We subdivide the test derivation strategy **TDS** for protocol conformance testing into three steps:*

*(1) Test Generation **TDSG** by which generic test cases **TS** are generated from **PS**, i.e.,*

$$(\exists\ TDSG)\ (TDSG(PS) = TS \subseteq D)\ ;$$

*(2) Test Selection **TDSS** by which abstract test cases TS_i are selected from **TS** on the basis of some additional statements AS_a, i.e.,*

$$(\exists\ TDSS)\ (TDSS(TS,\ AS_a) = TS_i \subseteq TS)\ ;$$

*(3) Test Choice **TDSC** by which executable test cases TS_e are chosen from TS_i on the basis of some additional statements AS_e, i.e.,*

$$(\exists\ TDSC)\ (TDSC(TS_i, AS_e) = TS_e \subseteq TS_i)\ .$$

From the results of [21], it can be shown that TS, TS_i and TS_e exist for a protocol implementation. The problem to address next is how to formulate ***TDSG***, ***TDSS*** and ***TDSC*** for TS, TS_i and TS_e respectively. In this paper, we focus our attention on an approach to test sequence generation and selection for abstract test cases from formal protocol specifications. The approach assumes that the protocol specification is given in EBE (EBE-BT or EBE-NF) only. First, generic test cases are generated from the EBE by using the test generation method presented in the next subsection. Given a particular abstract test method, abstract test cases can then be selected from the generic test cases.

3.2 Test generation for generic test cases

The key idea in the method proposed here is the concept that all associations between each output primitive and those input primitives (as well as their parameters) that influence them are examined during testing. The same is done also for each output primitive parameter. All of these associations can be obtained through the interaction paths and their I/O subpaths (defined below) of a protocol which can easily be identified from the EBE. The I/O subpath is the basic unit of test cases. On the basis of a specific protocol testing purpose, a set of I/O subpaths are identified and grouped into a test group (following ISO terminology). Test cases are completed by the addition of some statements such as those producing test verdicts. The following subsections describe the details of the method.

3.2.1 Identifying all interaction paths and their I/O subpaths

Definition 3.5 (interaction path)

*An interaction path **IP** is the externally observable track on which a sequence of interactions between the protocol and its external environment occurs, starting from the initial external state S_0 of the protocol and ending in the same state. Any interaction path **IP** is different from another one, i.e.,*

$$(\forall\ IP_i)\ (\forall\ IP_j)\ (IP_i \neq IP_j).$$

Definition 3.6 (I/O subpath)

*An I/O subpath **SIP** is the externally observable subtrack (e_1, ..., e_k), where*

(1) e_1 is an input primitive I_p with its parameters X_{pq} and ek is an output primitive O_q with its parameters Y_{qp};

(2) ek is influenced by some logical relations which satisfy

*a) property **P(Z)** ; and/or*

*b) a set of functions $\{Y_{qp}\}$ = **F(Z)**.*

Definition 3.7

*Each interaction path **IP** includes one or more independent I/O subpath **SIP**'s. Each I/O subpath **SIP** can belong to one or more different interaction path **IP**'s.*

If a protocol specification is given in EBE-BT or EBE-NF, the following algorithm is used to identify all interaction paths.

Algorithm A (Identifying all interaction paths)

Input : EBE of a protocol.

Output: $IP_x, x = 1, 2, \ldots$ (The set of interaction paths).

Initialization: x := 0, all states and all transitions are set to "unmarked".

Step 1. Let x := x + 1, i:= 0, and $S_0 \rightarrow IP_x$ (" $\rightarrow$ " means " appended to ").

Step 2. Find an "unmarked" $T_{ij(k)}$ with minimum j and k (i.e., the lowest numbered transition from state i to the lowest numbered state j). If none exists, go to Step 6; otherwise, go to Step 3.

Step 3. If j = 0, then the end of this interaction path has been reached, $T_{ij(k)} \rightarrow IP_x$, $S0 \rightarrow IP_x$, $T_{ij(k)}$ is set to "marked", and go to Step 5. If j $\neq$ 0, then $T_{ij(k)} \rightarrow IP_x$, $S_j \rightarrow IP_x$, and go to Step 4.

Step 4. Find an "unmarked" $T_{ij(k)}$ with minimum j and k. If found, then go to Step 3; If not, check the status of S_i. If S_i is "unmarked", then a transition loop has occurred in this

interaction path. The state S_i is set to "marked" and this interaction path is marked as a "loop transition"; go to Step 5.

Step 5: Traverse this IP_x backward. For each $T_{ij(k)}$ of the IP_x, if S_j is "marked" or if j=0 then set $T_{ij(k)}$ to "marked". If a S_i does not have any "unmarked" $T_{ij(k)}$, then S_i is set to "marked". An elementary interaction path has been obtained, and go to Step 1 for the next interaction path.

Step 6: For each IP with transition loops at any state along the path, a sequence of transitions @ from the last state on the path going back to S_0 is added. Then all interaction path IP_x's obtained are output.

The application of Algorithm A to the EBE of the ISO Class 0 Transport Protocol is shown in Table 3. There are two kinds of interaction paths — those with and those without transition loops from a state to itself. In the case of two or more transition loops in the same state, some paths of the transition loops may be inexecutable. Also the length of an interaction path may be infinite if it contains transition loops. These problems must be dealt with by the test derivation strategies.

Table 3. All interaction paths for ISO Class 0 Transport Protocol

All interaction paths:

IP1: [S0,T00(1),S0]
IP2: [S0,T00(2),S0]
IP3: [S0,T01,S1,T13,S3,T30,S0]
IP4: [S0,T01,S1,T14,S4,T40(1),S0]
IP5: [S0,T01,S1,T14,S4,T40(2),S0]
IP6: [S0,T01,S1,T14,S4,T40(3),S0]
IP7: [S0,T01,S1,T14,S4, |T44(1),S4|, | T44(2),S4|, | T44(3),S4|, | T44(4),S4|, T40(1),S0]*
IP8: [S0,T01,S1,T14,S4, |T44(1),S4|, | T44(2),S4|, | T44(3),S4|, | T44(4),S4|, T40(2),S0]*
IP9: [S0,T01,S1,T14,S4, |T44(1),S4|, | T44(2),S4|, | T44(3),S4|, | T44(4),S4|, T40(3),S0]*
IP10: [S0,T02,S2,T24,S4, |T44(1),S4|, | T44(2),S4|, | T44(3),S4|, | T44(4),S4|, T40(1),S0]*
IP11: [S0,T02,S2,T24,S4, |T44(1),S4|, | T44(2),S4|, | T44(3),S4|, | T44(4),S4|, T40(2),S0]*
IP12: [S0,T02,S2,T24,S4, |T44(1),S4|, | T44(2),S4|, | T44(3),S4|, | T44(4),S4|, T40(3),S0]*
IP13: [S0,T02,S2,T20,S0]
IP14: [S0,T02,S2,T25,S0,T50,S0]

where the symbol * denote the interaction paths with transition loops ; the symbol | ... | denote a transition loop.

Algorithm B (Identifying all I/O subpaths)

Input : All of the IP_x's and the EBE of a protocol;

Output: SIP_y, $y = 1, 2, \ldots$ (The set of I/O subpaths).

Step 1. Traverse each interaction path backward .

For the output primitive in each transition, an I/O subpath can be obtained in the interaction path which tests the correctness of this output primitive. The I/O subpath ends at the output primitive and starts from:

1) the earliest preceding transition if the output primitive is influenced by the input primitive and/or its parameters in this preceding transition;

2) the input primitive in the same transition if the output primitive is influenced by this input primitive and/or its parameters, or by parameters defined in implementation statements such as PICS and PIXIT.

If the output primitive is influenced by both the above conditions, then condition 1) takes precedence.

This procedure is repeated for each output primitive parameter in every transition in the interaction path.

Step 2. Those I/O subpaths containing states with executable transition loops can generate more I/O subpaths which contain alternately zero or one traversal of each executable loop for the purpose of avoiding infinite testing paths.

Step 3. Combines those I/O subpaths which are identical even though they have different testing purposes into a feasible I/O subpath. Thus each I/O subpath will have one or more testing purposes.

The I/O subpaths obtained by the application of Algorithm B to the interaction paths and the EBE of ISO Class 0 Transport Protocol are shown in Table 4.

3.2.2 Combining I/O subpaths based on the specific testing purpose

I/O subpaths are selected on the basis of the specific testing purposes in order to combine them into some groups of test cases called *Test Group* (following ISO terminology). This is quite easy to do because each I/O subpath has been defined for one or more particular testing purposes.

For an EBE with nested structure, a *nested test group* will be formed. In general, test groups may be nested to an arbitrary depth. A nested test group may be associated with a testing purpose for parallel compositions (e.g., multiple connection testing).

3.2.3 Completion and notation of generic test cases

The above test cases are completed by the addition of some statements (e.g., testing verdicts of "pass", "fail", and "inconclusive"). Also, some additional primitives and their parameters may be considered for the purpose of testing some defensive behaviour not characterized in the formal protocol specifications. The generic test cases of a protocol are given in a standardized test notation such as the Tree and Tabular Combined Notation (TTCN).

3.3 Test selection for abstract test cases

Generic test cases are used as the common basis for selecting the corresponding abstract test cases for different abstract test methods [2]. The main factors that influence this selection is the abstract test method to be used and its testing environment. In addition, the specifications of a preamble and a postamble are also included in each abstract test case.

Table 4. *All I/O Subpaths* for ISO Class 0 Transition Protocol

I/O Subpaths:
SIP1: [TCREQ / TDIND]
SIP2: [TCREQ / CR]
SIP3: [DR / NDREQ]
SIP4: [DR / NDREQ, — / TDIND]
SIP5: [CC / TCCON]
SIP6: [TDREQ / NDREQ]
SIP7: [NDIND / TDIND]
SIP8: [NRIND / TDIND]
SIP9: [CR / TCIND]
SIP10: [CR / DR]
SIP11: [TDREQ / DR]
SIP12: [CR / TCIND, TDREQ / DR]
SIP13: [TCRES / CC]
SIP14: [CR / TCIND, TCRES / CC]
SIP15: [TCRES / DR]
SIP16: [CR / TCIND, TCRES / DR, — / TDIND]
SIP17: [TCRES / DR, — / TDIND]
SIP18 [TDATR / —, — / DT]
SIP19: [DT / —, — / TDATI]
SIP20: [TCREQ / CR, CC / TCCON, TDATR / DT1, — / DT2, ... , —/ DTn]
SIP21: [TCREQ / CR, CC / TCCON, DT1 / — , DT2 / — , ... , DTn / TDATI]
SIP22: [CR / TCIND, TCRES / CC, TDATR / DT1, — / DT2, ... , — / DTn]
SIP23: [CR / TCIND, TCRES / CC, DT1 / — , DT2 / — , ... , DTn / TDATI]
SIP24: [TDATR / —, — / DT, DT / —, — / TDATI]
SIP25: [DT / —, — / TDATI, TDATR / —, — / DT]
SIP26: [TDATR / —, DT / —, — / DT, — / TDATI]
SIP27: [TDATR / —, DT / —, — / TDATI, — / DT]
SIP28: [DT / —, TDATR / —, — / DT, — / TDATI]
SIP29: [DT / —, TDATR / —, — / TDATI, — / DT]

Different parts of an I/O subpath in the generic test cases are identified on the basis of different abstract test methods. Thus, for the Remote Single-layer Test Method, the PDU's input and output for an I/O subpath, but not the interlayer service primitives, will be selected. For the Distributed Single-layer Test Method, the PDU's input and output as well as service primitives to the higher layer will be selected. In the ISO Transport Class 0 example (Table 4), [DR / TDIND] shall be selected from the SIP4 [DR / NDREQ, — / TDIND] I/O subpath for the Distributed Single-layer Test Method; [CR / CC] and [CR / DR] shall be selected from the SIP14 [CR / TCIND, TCRES / CC] and SIP12 [CR / TCIND, TCREQ / DR] I/O subpaths respectively for the Remote Single-layer Test Method.

Abstract test cases are also specified using the TTCN notation.

4 Comparison with other Test Derivation Methods

Comparison of the various test derivation methods is difficult because there is no good evaluation standard available for this purpose. An attempt has been made in [7] to compare their test selection method, namely selecting test sequences to cover *all simple OI-paths* (Output and Input paths), with a method to cover *all DU-paths* (Definition and Usage paths) [5, 6], and with an approach to combined *flow coverage/parameter variation* (*FCPV*) [4]. Some interesting results have been obtained from this work which uses the ISO Class 0 Transport Protocol as the basis of comparison. In this section, we compare our test sequence derivation method, namely to cover *all I/O subpaths,* with the method to cover *all simple OI-paths*, and with the combined *FCPV* method. The method to cover *all DU-paths* is not used in our comparison, because [7] has shown that *all simple OI-path* coverage is much more comprehensive than *all DU-paths* coverage; both methods were proposed by the same researchers.

The comparison is based on the same protocol, the ISO Class 0 Transport Protocol. We only deal with the feasible paths. The paths produced by applying these three methods are given in Table 4, Table 5, and Table 6 .

Our comparison examines test completeness and effectiveness based on externally observable behaviour, namely *combined primitive and parameter coverage*. The coverage checks the correctness of control flow concerned with input and/or output primitives as well as data flow concerned with input and/or output primitive parameters.

4.1 Comparison with all simple OI-paths

By comparing Tables 4 and 5, we can see that the method of *all I/O subpaths* is more complete and effective than that of *all simple OI-paths*. In the latter method, some duplicate paths and unnecessary paths can be found. For instance, OI3 is a duplicate of OI4, and OI6 a duplicate of OI7. Moreover, as a result of OI3 being a duplicate of OI4, additional duplications occur between OI13 and OI14, and between OI21 and OI22. OI11, OI12, OI16 and OI17 are unnecessary due to the definition of internal variables. In the results of *all I/O subpaths* method, SIP19, SIP20, SIP21 and SIP22 are used to test the fragmentation and reassembly functions of data transfer; these paths are missing in the results obtained by the *all simple OI-paths* method.

It is easy to see that the *all simple OI-paths* method generates many more paths than the *all I/O subpaths* method. Many of the paths resulting from the former method are not necessary and the rest are covered by the *all I/O subpaths* method in less paths.

4.2 Comparison with Flow Coverage/ Parameter Variation

It is not as straightforward to compare *FCPV* method with *all I/O subpaths* method. However, we are still able to discover differences between them on the basis of the differences in the generation procedures.

The *FCPV* method assumes the decompositions of the NFS in terms of the subtours for the control graph (CG) and data flow functions (function for short) for the data flow graph (DFG). These functions are tested by parameter variations and covering all the control paths that exist in the specification. For each function, the subtours selected are those subtours of

the CG which include Normal Formal Transitions (NFT's) in the set of labels of the function block. These subtours are longer than necessary. Thus, *FCPV* coverage for testing a function is lower in effectiveness than *all I/O subpaths* which only includes paths necessary for testing the function. The subtours of the *FCPV* method are more like the interaction paths of our methods. In addition, an unbounded number of tests for subtours sb4 and sb5 is clearly unacceptable.

Table 5. *All Simple OI-Paths* for ISO Class 0 Transition Protocol

Simple OI-Paths:	
OI1:	[tcreq/cr]
OI2:	[tcreq/tdind]
OI3:	[cr/tcind]
OI4:	[cr/tcind]
OI5:	[cr/dr]
OI6:	[cc/tccon]
OI7:	[cc/tccon]
OI8:	[dr/ndreq,tdind]
OI9:	[dr/ndreq,tdind]
OI10:	[tcres/cc]
OI11:	[(t3); tcres/cc]
OI12:	[(t4); tcres/cc]
OI13:	[cr/tcind; tcres/cc]
OI14:	[cr/tcind; tcres/cc]
OI15:	[tcres/dr,tdind]
OI16:	[(t3); tcres/dr,tdind]
OI17:	[(t4); tcres/dr,tdind]
OI18:	[cr/tcind; tcres/dr,tdind]
OI19:	[cr/tcind; tcres/dr,tdind]
OI20:	[tdreq/dr]
OI21:	[cr/tcind; tdreq/dr]
OI22:	[cr/tcind; tdreq/dr]
OI23-OI58:	[tdatr/(out.buff); CD1; tdatr/(out.buff) ; CD2; (out.buff)/dt; CD3 (out.buff)/dt]
OI59-OI94:	[tdatr/(out.buff); CD1; (out.buff)/dt; CD2; tdatr/(out.buff); CD3; (out.buff)/dt]
OI95-OI130:	[dt/(in.buff); AB1; dt/(in.buff); AB2; (in.buff)/tdati; AB3; (in.buff)/tdati]
OI131-OI166:	[dt/(in.buff); AB1; (in.buff)/tdati; AB2; dt/(in.buff); AB3; (in.buff)/tdati]
OI167:	[tdreq/ndreq]
OI168:	[nrind/tdind]
where	
the concatenation of AB1; AB2; AB3 = (A; A; B; B) | (A; B; A; B), and	
the concatenation of CD1; CD2; CD3 = (C; C; D; D) | (C; D; C; D), and	
A denotes the string: tdatr/(in.buff);	
B denotes the string: (out.buff)/dt;	
C denotes the string: dt/(in.buff);	
D denotes the string: (in.buff.)/tdatr;	
AB1, AB2, AB3, CD1, CD2, CD3 can be empty strings.	

Table 6. Coverage of the *FCPV* Method for ISO Class 0 Transition Protocol

Subtours:

sb1. (t2 | t5)
sb2. t1; (t8 | t9)
sb3. (t3 | t4); (t11 | t12)
sb4. t1; (t6 | t7); (t13 | t14 | t15 | t16) *; (t17 | t18 | t19)
sb5. (t3 | t4); t10; (t13 | t14 | t15 | t16) *; (t17 | t18 | t19)

where the symbols ; , | , and * denote the operations catenation, set union, and iteration, respectively.

Block	SIL(block)	Subtours of the block
Connection Referencing:	t1,t3,t4,t10	sb2,sb3,sb4,sb5
Addressing:	t1,t3,t4	sb2,sb3,sb4,sb5
Disconnection:	t2,t5,t8,t9,t11,t12,t17,t18,t19	sb1,sb2,sb3,sb4,sb
Quality of Service:	t1,t3,t4,t6,t7	sb2,sb3,sb4,sb5
Data Transfer:	t6,t7,t10,t13,t14,t15,t16	sb4,sb5

4.3 Comparisons of test derivation complexity

Because the definitions and operations of internal variables are involved, the *FCPV* method requires considerable effort to identify functions and their relationships, and is the most complex of the three methods. *All simple OI-paths* method assumes that a protocol specification is given in NFS. A flowgraph modeling both the control flow and data flow is constructed by identifying all associations between definition and usage of each variable employed in the specification which includes the internal structures and variables concerned with implementation. Based on this information, associations between each output and those inputs that influence the output are identified. Test sequences are selected to cover each of the associations at least once. This method appears to be less complex than *FCPV* method.

Our test sequence derivation strategy is based on the EBE of a protocol which only describes externally observable behaviour through the external input/output actions and the logical relations associated with these actions. Thus unlike the other two methods, no work need be done involving internal implementation details. All interaction paths and their I/O subpaths can be identified easily by two formalized algorithms. Each I/O subpath is defined for one or more testing purposes. By combining and completing these I/O subpaths, generic test cases of the protocol can be obtained. The method appears to be the simplest of the three methods.

5 Conclusions

We have presented a framework for a new approach to test sequence derivation from formal protocol specifications. This approach is based on the EBE of a protocol which can be obtained from formal protocol specifications in either Estelle or LOTOS. A basic test

derivation theory for protocol conformance testing has been defined and forms the basis of the test sequence derivation strategy. The strategy proposed uses the I/O subpaths in the EBE as the basic units of a test case. Algorithms have been outlined to identify all interactional paths and their I/O subpaths associated with specific testing purposes. By grouping and completing these I/O subpaths, generic test cases of the protocol can be obtained. Abstract test cases are selected from the generic test cases on the basis of the abstract test method specified and its testing environment.

Compared to the other test derivation methods proposed recently, this approach has the following advantages:

(1) In the EBE, a protocol implementation under test is considered a black box and its internal structure need not be known. The EBE of a protocol includes all interaction paths and their I/O subpaths, which represent both the control flow and the data flow, and can be used to generate test sequences directly. Thus, the test sequence derivation strategy based on the EBE is simple and concise, and test sequences derived from it will be more complete and effective.

(2) The EBE of a protocol can be obtained from either Estelle or LOTOS specification, while the formal models used by other methods can only be transformed from one FDT such as Estelle or LOTOS.

(3) In our approach, both the Generic Test Cases and the Abstract Test Cases are produced . Other test generation methods have not addressed this issue.

More research is needed for a tool that implements this method. It may also be interesting to investigate whether the methodology is applicable to the area of function software testing.

This approach is being validated in the UBC/IDACOM joint project for OSI conformance testing. We believe that this approach to test sequence derivation can provide the basis of a formalized framework for protocol conformance testing.

References

[1] D. Rayner, "Standardizing conformance testing for OSI", Proc. of 5th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Toulouse-Moisssac, France, June 1985.

[2] ISO/TC97/SC21, "OSI conformance testing methodology and framework - Part 1: General Concepts", ISO 2nd DP 9646-1 revised text, Vancouver, December 1987.

[3] B. Sarikaya and G.v. Bochmann, "Synchronization and specification issues in protocol testing", IEEE Trans. on Communication, Vol.32, No.4, 1984, pp.389-395.

[4] B. Sarikaya, G.v. Bochmann, and E. Cerny, "A test design methodology for protocol testing", Proc. of 18th Hawaii ICSS, Jan 1985, Vol.2, pp.710-721.

[5] H. Ural, "A test derivation method for protocol conformance testing", Proc. of 7th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Zurich, Switzerland, May, 1987.

[6] H. Ural, "Test sequence selection based on static data flow analysis", Computer Communication, Vol.10, No.5, 1987, pp.234-242.

[7] H. Ural, B. Yang, and R.L. Probert, "A test selection method for protocol specified in Estelle", Technical Report: TR-88-18, Dept. of Computer Science, University of Ottawa, June 1988.

[8] H.J. Burkhardt, H. Eckert, and A. Giessler, "Testing of protocol implementations - a systematic approach to derivation of test sequence from global protocol specifications", Proc. of 5th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Toulouse-Moissac, France, June 1985.

[9] B. Kanungo, L. Lamont, R.L. Probert, and H. Ural, "A Useful FSM representation for test suite design and development", Proc. of 6th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Montreal, Quebec, Canada, June 1986.

[10] J.P. Favreau and R.J. Linn, "Automatic generation of test scenarios from protocol specifications written in Estelle", Proc. of 6th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Montreal, Quebec, Canada, June 1986.

[11] J. de Meer, "Derivation and Validation of test scenarios based on the formal specification language LOTOS", Proc. of 6th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Montreal, Quebec, Canada, June 1986.

[12] E. Brinksma, G. Scollo, and C. Steenbergen, "Lotos specification, their Implementation and their ·tests", Proc. of 6th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Montreal, Quebec, Canada, June 1986.

[13] E. Brinksma, "A theory for the derivation of tests", Proc. of 8th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification , Atlantic City, New Jersey, USA, June 1988.

[14] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours", Proc. of 8th IFIP/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, Atlantic City, New Jersey, USA, June 1988.

[15] K. Sabnani and A. Dahbura, "A protocol test generation procedure", Computer Networks and ISDN Systems, Vol.15, No.4, 1988, pp.285-297.

[16] A.M. Davis, "A comparison of techniques for the specification of external system behavior", Communication of the ACM, Vol.31, No.9, 1988.

[17] S. Budkowski and P. Dembinski, "An introduction to Estelle: a specification language for distributed systems", Computer Network and ISDN Systems, Vol.14, No.1, pp3-23, 1987.

[18] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS", Computer Network and ISDN Systems, Vol.14, No.1, pp25-59, 1987.

[19] J.P. Wu and S.T. Chanson, "External Behaviour Expression (EBE) and its Derivation

from formal protocol specifications", to appear.

[20] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Trans. on Software Engineering, Vol. SE-1, No.2, June 1975.

[21] W.E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Trans. on Software Engineering, Vol. SE-2, No.3, June 1976.

[22] L.J. White, "Software Testing and Verification", Advances in Computers, Vol.26, pp335-391, 1987.

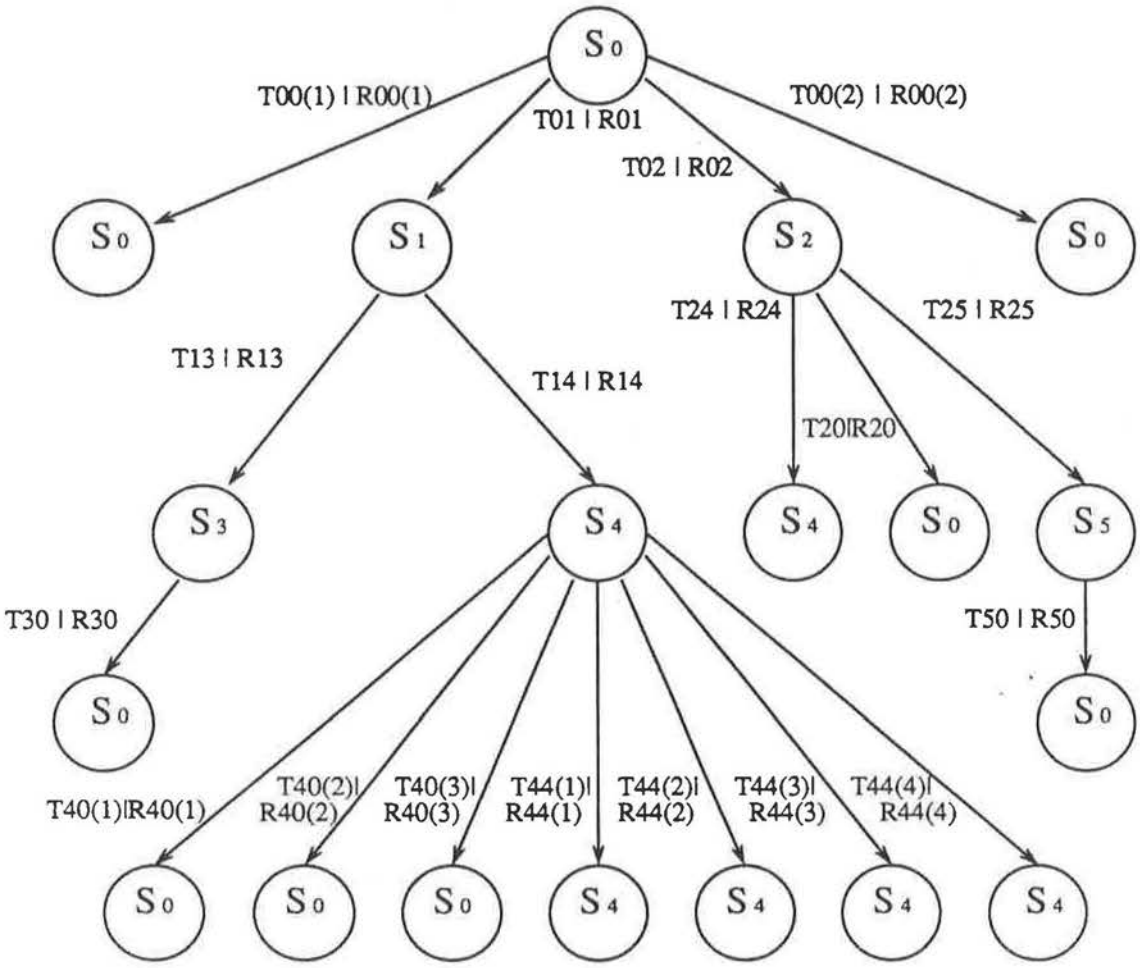
[23] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Trans. on Software Engineering, Vol. SE-6, No.3, May 1980.

[24] ISO/TC97/SC6 N4394, "Formal Description of ISO 8073 (Transport Protocol) in Estelle", Information Processing Systems, Open System Interconnection, 1986.

[25] ISO/TC97/SC6 N4396, "Formal Specification in LOTOS of ISO 8073 (Transport Protocol)", Information Processing Systems, Open System Interconnection, 1986.

Appendix A

EBE-BT for ISO Class 0 Transport Protocol



Abbreviation:

T00(1) = TCREQ / TDIND;
T00(2) = CR / DR;
T01 = TCREQ / CR;
T02 = CR / TCIND;
T13 = DR / NDREQ;
T14 = CC / TCCON;
T20 = TDREQ / DR;
T24 = TCRES / CC;
T25 = TCRES / DR;
T30 = — / TDIND;
T40(1) = TDREQ / NDREQ;
T40(2) = NDIND / TDIND;
T40(3) = NRIND / TDIND;
T44(1) = TDATR / — ;
T44(2) = — / DT;
T44(3) = DT / — ;
T44(4) = — / TDATI;
T50 = — / TDIND.

Appendix B

EBE-NF for ISO Class 0 Transport Protocol

```
SPECIFICATION Transport_class0 [ input/output primitive and their parameter list ]
   local.refer := ... ;                { statements of transport class0 implementation }
   tpdu.size := ... ;
   qts.estmate := ... ;
   tcreq.qts.req.ok := ... ;
   cr.option.ok := ... ;
   user.init.disc.reason := ... ;
   cr.option.normal := ... ;
BEHAVIOUR
   S0 [ TCREQ(to.t.addr, from.t.addr, qts.req), TDIND(ts.disc.reason, ts.user.reason) |
        function:  TDIND.ts.disc.reason = ... ;
                   TDIND.ts.user.reason = ... ,
        predicate: TCREQ.qts.req <> tcreq.qts.req.ok ] * S0          { T00(1) | R00(1) }
    + [ TCREQ(to.t.addr, from.t.addr, qts.req), CR(source.ref, option, calling.addr,
                                                   called.addr, max.tpdu.size) |
        function:  CR.source.ref = local.ref;
                   CR.opiton = ' norma l ' ;
                   CR.calling.addr = TCREQ.from.t.addr;
                   CR.called.addr = TCREQ.to.t.addr;
                   CR.max.tpdu.size = tpdu.size;
        predicate: TCREQ.qts.req = tcreq.qts.req.ok ] * S1           { T01 | R01 }
    + [ CR(source.ref, option, calling.t.addr, called.t.addr, max.tpdu.size),
        DR(dest.refer, disconnect.reason) |
        function:  DR.dest.refer = CR.source.ref;
                   CR.disconnect.reason = ... ,
        predicate: CR.option <> cr.option.ok ] * S0                  { T00(2) | R00(2) }
    + [ CR(source.ref, option, calling.t.addr, called.t.addr, max.tpdu.size),
        TCIND((to.t.addr, from.t.addr, qts.pro) |
        function:  TCIND.to.t.addr = CR.called.t.add;
                   TCIND.from.t.addr = CR.calling.t.addr;
                   TCIND.qts.pro = qts.estimate,
        predicate: CR.option = cr.option.ok ] * S2;                  { T02 | R02 }

   S1 [ DR(disconnect.reason, add.clear.reason), NDREQ(disc.reason) |
        function:  NDREQ.disc.reason = DR.disconnect.reason,
        predicate: True ] * S3                                       { T13 | R13 }
    + [ CC(max.tpdu.size), TCCON(qts.res) |
        function:  TCCON.qts.res = qts.estmate,
        predicate: true ] * S4;                                      { T14 | R14 }

   S2 [ TDREQ(ts.user.reason), DR(dest.refer, disconnect.reason, add.clear.reason) |
        function:  DR.dest.refer = CR.source.ref { T02 = CR, TCIND};
                   DR.disconnect.reason = ... ;
                   DR.add.clear.reason = TDREQ.ts.user.reason,
        predicate: true ] * S0                                       { T20 | R20 }
    + [ TCRES(qts.req),
```

```
    CC(dest.refer, source.ref, calling.t.addr, called.t.addr, max.tpdu.size) |
    function:  CC.source.ref = local.refer;
               CC.dest.refer = CR.source.ref { T02 = CR, TCIND };
               CC.calling.t.addr = CR.calling.t.addr { T02 = CR, TCIND };
               CC.called.t.addr = CR.called.t.addr { T02 = CR, TCIND };
                if CR.max.tpdu.size <> nil then
                   CC.max.tpdu.size = CR.max.tpdu.size { T02 = CR, TCIND }
               else CC.max.tpdu.size = tpdu.size,
    predicate: TCRES.qts.req <= qts.estimate ] * S4                    { T24 | R24 }
 + [ TCRES(qts.req), DR(dest.refer, disconnect.reason, add.clear.reason) |
    function:  DR.dest.refer = CR.source.ref { T02 = CR, TCIND };
               DR.disconnect.reason = ... ;
               DR.add.claer.reason = ... ,
    predicate: TCRES.qts.req > qts.estimate ] *  S5;                   { T25 | R25 }

S3 [ —, TDIND(ts.disc.reason) |
    function:  TDIND.ts.disc.reason = DR.disconnect.reason { T13 = DR, NDREQ }
               if DR.disconnect.reason = 'user.init.disc.reason' then
                  TDIND.ts.user.reason = DR.add.clear.reason
               else TDIND.ts.user.reason = nil { T13 = DR, NDREQ },
    predicate: true ] * S0;                                            { T30 | R30 }

S4 [ TDREQ(ts.user.reason), NDREQ(disc.reason) |
    function:  NDREQ.disc.reason = TDREQ.ts.user.reason
    predicate: true ] * S0                                             { T40(1) | R40(1) }
 + [ NDIND(), TDIND(ts.disc.reason) |
    function:  TDIND.ts.disc.reason = ... ,
    predicate: true ] * S0                                             { T40(2) | R40(2) }
 + [ NRIND(), TDIND(ts.disc.reason) |
    function:  TDIND.ts.disc.reason = ... ,
    predicate: true ] * S0                                             { T40(3) | R40(3)
 + [ TDATR(tsdu.fragment), — |
    function:  nil,
    predicate: true ] * S4                                             { T44(1) | T44(1) }
 + [ —, DT(user.data) |
    function:  DT.user.data = TDATR.tsdu.fragment.user.data { T44(1) = TDATR, — }
    predicate: true ] * S4                                             { T44(2) | T44(2) }
 + [ DT(user.data), — |
   [ function:  nil,
    predicate: true ] * S4                                             { T44(3) | R44(3) }
 + [ —, TDATI(tsdu.fragment) |
    function:  TDATI.tsdu.fragment = DT.user.data { T44(3) = DT, — }
    predicate: true ] * S4;                                            { T44(4) | R44(4) }

S5 [ —, TDIND(ts.disc.reason) |
     function:  TDIND.ts.disc.reason = ... ,
     predicate: true ] * S0;                                           { T50 | R50 }

ENDSPEC
```