A Test Design Methodology for Protocol Testing

BEHÇET SARIKAYA, MEMBER, IEEE, GREGOR v. BOCHMANN, SENIOR MEMBER, IEEE, AND EDUARD CERNY, MEMBER, IEEE

Abstract—Communication protocol testing can be done with a test architecture consisting of remote Lower Tester and local Upper Tester processes. For real protocols, tests can be designed based on the formal specification of the protocol which uses an extended finite state machine model. The specification is transformed into a simpler form consisting of normal form transitions. It can then be modeled by a control and a data flow graph. The graphs are decomposed into subtours and data flow functions, respectively. Tests are designed by considering parameter variations of the input primitives of each data flow function and determining the expected outputs. The methodology gives complete test coverage of all data flow functions and control paths in the specification. Functional fault models are proposed for functions that are not formally specified.

Index Terms—Extended finite state automata, fault models, formal specification, normal form transitions, symbolic execution, test sequences.

I. Introduction

WIDE range use of public data networks linking computers and terminals makes it possible to interconnect heterogeneous systems for the purpose of distributed applications. Interconnection of "open" systems is done by implementing standard protocols as defined by ISO standards. As the number of implementations of higher-level protocols in line with the OSI Reference Model [4] increases, it must be verified that the implementations adhere to the protocol/service specification. NPL in England [11] and Project RHIN in France [1] have done pioneering work in this respect. The verification of adherence is also called assessment. In general it is based on testing.

Reference [12] proposes an architecture (known as distributed single layer test architecture) to be used in testing protocol implementations of level N in the OSI Reference Model (see Fig. 1). The protocol upper layer interface is assumed to be accessible through a user task which provides stimuli to the implementation under test (IUT in short). This task is called the upper tester (UT in short). The major part of the architecture (also called lower tester or LT in short) resides remotely in another computer. The LT and the UT stimulate the IUT with a given sequence of input interactions and observe the resulting output from the IUT.

```
                          Test                      ---------------
                      Coordination--------------->| Upper Tester |
                       Procedures                   ---------------
                           |                               A
---------------            |                               |
| Lower Tester |<-----------'                              V
---------------                                      -----| |------
       A                                             | (N)-Entity |
       |                                             | under test |
       V                                             -----| |------
-----| |---------------------------------------------------| |------
|                     (N-1)-Service Provider                       |
|                                                                  |
```

Fig. 1. An architecture for testing protocol implementations.

We consider an IUT as a black box and assume that a formal specification of the protocol which defines the required behaviour of an IUT in a concise and precise manner is available. A particular formal specification language Estelle [5] for communication protocols and services is used throughout the paper; however, the test design method described in the paper can be adapted to other specification languages.

Functional program testing views a program as an integrated collection of functions and selects test data so as to verify that the program correctly implements these functions. This method is known to give best results for discovering errors [6]. In this paper we show that functional program testing can be applied to protocols.

The paper proceeds as follows. Section II introduces the Estelle specification language and its underlying model. In Section III we simplify Estelle specifications and obtain an equivalent form which can be modeled with control and data flow graphs. Determination of protocol functions is the subject of Section IV. Section V details the test methodology. A real protocol is used as an example to illustrate the methodology throughout the paper. Finally, Section VI gives some conclusions.

II. Estelle Specification Language

To meet the goals of open system interconnection (OSI), formal description techniques (FDT) have been developed to provide unambiguous, clear, and concise specification of communication protocols and services [17]. Important specification concepts are modules and channels. A module is a unit of specification whose behavior may be defined using an FDT language, or it may be defined in terms of a structure of interconnected submodules (stepwise refinement). Modules (and/or submodules) interact with each other through channels. FDT's

Manuscript received June 29, 1984; revised January 31, 1985. This work was supported in part by the National Science and Engineering Research Council and Department of Communications, Canada.

B. Sarikaya is with the Department of Electrical Engineering, Concordia University, 1455 de Maisonneuve W. #915, Montreal, P.Q. H3G 1M8, Canada.

G. v. Bochmann and E. Cerny are with the Department of Computer Science and Operations Research, University of Montreal, C.P. 6128, Montreal, P.Q. H3C 3J7, Canada.

IEEE Log Number 8613882.

0098-5589/87/0500-0518$01.00 © 1987 IEEE

differ in their underlying model and the specification language. We describe here the FDT called Estelle.

A. The Model

The underlying model of Estelle is an *extended finite-state machine* (EFSM). The state space of a module is determined by a set of variables; a state is determined by the values assumed by each of these variables. One of these variables is a distinguished variable called the "STATE"; it represents the state of a finite-state machine (FSM). It is also called the "major state" to distinguish it from the other variables which are called "context variables." STATE is often used to encode the status of a connection, e.g., closed, opening, etc., while the context variables are often used to store sequence numbers, quality of service, data, and the like.

Transitions are specified from a major state to a major state. These transitions may depend on predicates on the context variables, and they may depend on an input. Transitions that do not depend on any input are called *spontaneous transitions*. Associated with each transition is an *operation* to be executed as part of the transition. It may change the values of the context variables, and it may initiate output interactions with the environment of the module. The operation is assumed to be atomic.

The EFSM model allows the specifications to be nondeterministic in the sense that for a given state and input interaction, more than one enabling predicate may be true and thus several different transitions may be possible. If one or more transitions are enabled, then exactly one of these will be nondeterministically chosen for execution.

B. The Language

To specify transitions and the operations associated with them, a language that is based mainly on Pascal was developed. A specification comprises three major parts: the channel type definitions, the module definitions, and the system structure definition.

The type of interactions that may occur over the channels and their parameters are defined in the channel type definitions. The interaction primitives received from/sent to the peer entities are called protocol data units (PDU). The module definitions specify the actual transitions and their operations. The transitions of a module are defined by the specification of a number of transition types (or transitions in short). The enabling part of a transition type and its operation are specified using the Estelle clauses explained below (a complete description of Estelle can be found in [5]):

The *FROM* clause defines the present major state and has the form:

FROM state-list

where state-list is a list of major states.

The *WHEN* clause defines the input interaction and has the form:

WHEN interaction-reference

```
FROM idle
TO wait_for_T_connect_resp
WHEN mapping.CR(credit,source_reference,dest_reference,variable_part)
PROVIDED (/ Transport Entity able to provide the quality of service asked for/)
BEGIN
  remote_reference:=source_reference;
  if variable_part.max_TPDU_size <> undefined
  then
    TPDU_size := variable_part.max_TPDU_size
  else
    TPDU_size := 128;
  remote_address := variable_part.calling_T_address;
  called_address := ...;
  calling_address := ...;
  QOTS_estimate := ...;
  output TSAP.T_CONNECT_ind
       (TCEPI,called_address,calling_address, QOTS_estimate, normal, ...);
END;
```

Fig. 2. A transition type in Estelle.

where interaction-reference has the form:

AP.I(formal-parameter-list)

and AP stands for an access point identifier and I is the interaction primitive which may be followed by its parameters.

The *PROVIDED* clause defines an enabling predicate which must be satisfied when the transition is executed. Its syntax is defined as:

PROVIDED boolean-expression

The operation of a transition type is specified in two parts, a *TO* clause and an action. The TO clause defines the next major state after the execution of the transition. The syntax of this clause is:

TO to-list

where to-list is a state identifier or SAME making the next major state equal to the present major state. An action to be performed when the transition is executed is specified in a *BEGIN* block. Pascal executable statements such as assignment statements IF, WHILE, CASE statements and the like can be used. The generation of output interactions is specified using an OUTPUT statement which has the form:

OUTPUT AP.I(actual-parameter-list)

An example transition type appears in Fig. 2. It describes the module's behavior upon the reception of an interaction primitive called "Connect Request" (CR) and is extracted from a formal specification [9] of the transport protocol which represents level 4 in the OSI reference model.

C. Incomplete Specifications

Estelle identifiers may be defined as a "· · ·" and the "· · ·" may be used as a type, constant or expression to indicate that the specifier is leaving the interpretation to the implementor, as shown in Fig. 2. Often this is accompanied by a comment of the form:

(/binding comment/)

to guide the implementor in his choice. We call specifications containing "· · ·" or binding comments *incomplete specifications*.

III. Analyzing Specifications

This section deals with analysis and transformations of a protocol specification which allows the decomposition of the specification into its "functions," which is further explained in the following section. This functional analysis is the basis for the testing methodology described in Section V.

A specification can be modeled using graphs, one representing the flow of control and the other the flow of data. Functions of the protocol can be identified using these graphs. In order to simplify the determination of the control and data flow graphs of a formal specification given in Estelle, it is convenient to first transform the specification into an equivalent form containing only the so-called "normal form transitions" (NFT). NFT's do not use certain Estelle language constructs which would make the determination of the control and data flow graphs more complicated.

A. *Transformations*

We apply the transformations described below in order to avoid the following Estelle constructs:

- Major state lists in FROM and TO clauses,
- Conditional IF and CASE statements,
- Procedure/function calls.

Major state lists such as:

[AKWAIT, OPEN, OPEN__WFEA]

are eliminated from the FROM and TO clauses by generating more than one NFT corresponding to each possible state value (state values of AKWAIT, OPEN and OPEN__WFEA in the above example).

To remove conditional statements and local procedure/function calls we adopt the techniques from *symbolic execution* of sequential programs [3]. The idea is to create a new transition for every distinct path in the BEGIN block and to modify the PROVIDED clause to reflect the conditions for taking these paths. Local procedure/function calls in the BEGIN block are translated by symbolically executing the local procedure/function body. IF and CASE statements are removed by generating an NFT for each path they define. For example, the IF statement in Fig. 2 may be removed by writing the following two NFTs which are equivalent to the transition of Fig. 2:

```
FROM idle
TO wait__for__T__CONNECT__resp
WHEN mapping.CR(credit,source__reference,dest__reference,variable__part)
PROVIDED (/Transport Entity able to provide the quality of service asked for/)
    and variable__part.max__TPDU__size < > undefined
BEGIN
  remote__reference := source__reference;
  TPDU__size := variable__part.max__TPDU__size;
  . . .
FROM idle
TO wait__for__T__CONNECT__resp
WHEN mapping.CR(credit,source__reference,dest__reference,variable__part)
PROVIDED (/Transport Entity able to provide the quality of service asked for/)
    and variable__part.max__TPDU__size = undefined
BEGIN
  remote__reference := source__reference;
  TPDU__size := 128;
  . . .
```

We assume that the local procedures are not recursive and the specification does not contain any loops with variable bounds. These transformations are applied to all transition types of all the modules. In addition, modules may be combined by making textual substitutions. A method of combining Estelle modules is explained in [15].

Applying these transformations to the Estelle specification of [9] we obtain an equivalent *normal form specification* which appears in Appendix A. The NFT's of this specification are identified as P1 through P19 for further reference.

B. *Flow Graphs*

We distinguish two types of flow in a normal form specification:

- *Flow of control* which reflects the changes of the value of the major state variable, and
- *Flow of data* which reflects how the input primitive parameters determine the values of the context variables and they in turn determine the parameter values of the output primitives.

1) Control Graph: The control graph (CG) is easily constructed from the FROM/TO clauses of the NFT's by drawing an arc from the present state to the next state. The arcs are identified by the corresponding labels of the NFTs. These labels are short-hand notations for:

input/output

as used in the FSM's. The CG for the specification in the Appendix A appears in Fig. 3.

Sequences of NFT's in correct state order may be obtained from the CG. Those which start and end in the initial state are called *subtours*. Subtours of the CG in Fig. 3 are listed in Table I. In communication protocols, certain sequences of NFT's represent distinct *control phases* such as connection establishment, data transfer, connection freeing, etc. Each subtour contains some of them.

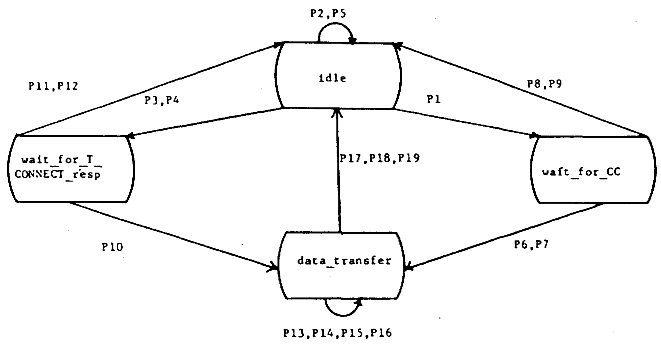


Fig. 3. A control graph for the class 0 TP.

TABLE I
SUBTOURS OF THE CLASS 0 TP

$P1(P6 + P7)(P13 + P14 + P15 + P16)^*(P17 + P18 + P19)$
$(P3 + P4)P10(P13 + P14 + P15 + P16)^*(P17 + P18 + P19)$
$P1(P8 + P9)$
$(P3 + P4)(P11 + P12)$
$P2 + P5$

For instance, in Table I, the following control phases can be identified for each of the five subtours in top-to-bottom order:

- user initiated connection establishment, data transfer, freeing,
- peer initiated connection establishment, data transfer, freeing,
- call refusal by peer,
- call refusal by user,
- call refusal by protocol entity.

C. Data Flow Graph

A data flow graph (DFG) models the flow of information in a normal form specification, excluding major state changes. Four types of nodes are used in a DFG: I-nodes to represent input primitive parameters, D-nodes to represent context variables and constants, O-nodes to represent output primitive parameters, and F-nodes to represent data operations (functions). Assuming that the information flows from top to bottom, I-nodes are drawn on top, D- and F-nodes in the middle, and O-nodes on bottom. Arcs are used to represent the flow as derived from the BEGIN block of the NFT's. Each arc of the DFG is labeled with the identifier of the NFT to which it belongs. The enabling conditions of the transitions are not reflected in the DFG, however, they are considered in the test sequence design (see also Section V).

Assignment statements in NFT's are modeled by drawing an arc from the node representing the right hand side (of type D, F, or I) to the node representing the left hand side (of type D or O). We furthermore define 3 types of F-nodes. *Type 1* F-nodes are used to represent incompletely specified function calls (see Section II-C). Consider for example the assignment statement in P15 of Appendix A:

in__buffer.append(DT.user__data)

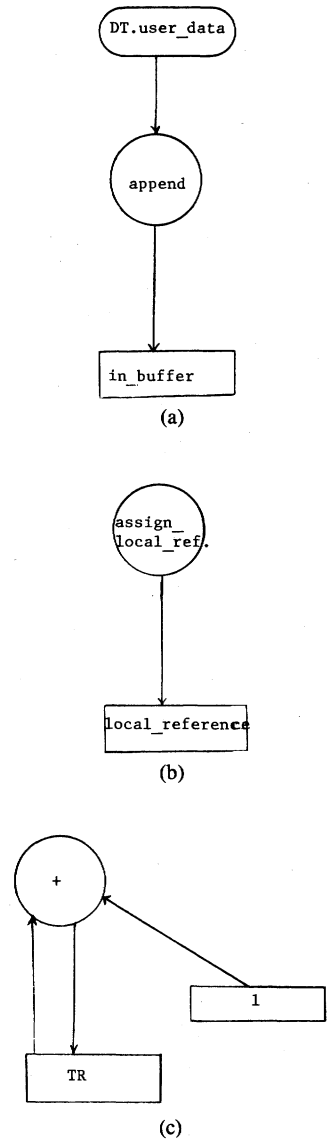


Fig. 4. Example DFG's.

where in__buffer is a context variable (in this case an abstract data type), DT is an input primitive with user__data being one of its parameters and append is a local function (an operation on abstract data type) whose body is left unspecified. A DFG for this statement is given in Fig. 4(a). *Type 2* F-nodes represent assignment statements whose right hand side is of the form:

local__reference := · · ·

(taken from P1 of Appendix A) where local__reference is a context variable. The corresponding DFG appears in Fig. 4. *Type 3* F-nodes are used to represent assignment statements containing arithmetic or Boolean expressions. For example the assignment statement:

TR := TR + 1

where TR is a context variable is transformed as shown in Fig. 4(c).

The output statement in the BEGIN block of an NFT is modeled by creating an O-node and treating the parameter list as assignments to the parameters of the primitive.

A complete DFG for the specification in Appendix A appears in Appendix B.

IV. Protocol Functions

A data flow graph, as described above, can algorithmically be partitioned into smaller blocks, and then these blocks can be combined into larger blocks representing various protocol functions. These resulting blocks give structural information about the protocol functions, and are used for designing tests of the corresponding function, as explained in Section V.

A. Decomposition of the DFG

A DFG of even a simple protocol (e.g., Appendix A) can be quite complicated, as can be observed in Appendix B. This is because each I- or O-node may contain complex parameters. Similarly, D-nodes may be of record type where each component takes part in a different function of the protocol. We define a *block Bi* as a collection of nodes and the associated arcs. The sets of these nodes belonging to the block are identified as: set of I-nodes SIN(*Bi*), set of F-nodes SFN(*Bi*), set of D-nodes SDN(*Bi*), and set of O-nodes SON(*Bi*).

A DFG can be partitioned into n disjoint blocks $B1$, $B2, \cdots, Bn$, each representing flows over a D-node, or directly to an O-node in cases where the O-node is assigned by an I-node or an F-node. The following algorithm finds such a partition, the inclusion of a node in one of the blocks is guided by the flow of data. A block in the resulting partition includes the nodes participating in a distinct flow of data from I- to D- nodes and finally to O-node(s). A node A is said to *feed* node B if there exists an outgoing arc from A to B.

Algorithm 4.1: Input: DFG *Output*: The sets SIN, SFN, SDN, and SON for each block.

- All variable D-nodes in the DFG are processed by creating a block for the D-node, or including it in one of the blocks already created if it feeds another D or a shared O-node. All I-, F- and D-nodes feeding the D-node are included in the sets SIN, SFN and SDN, respectively. O-nodes that are assigned by the D-nodes are included in the SON.
- For all O-nodes in the SON of the block, the I-, F- and D-nodes feeding the O-node are included into its SIN, SFN and SDN, respectively.
- F-nodes are processed as follows:

If the D-nodes feeding the F-node are already included in the same block, the F-node is added to the SFN and the O-nodes that are assigned by the F-node are included in the SON of the block. In all other cases the F-node is added to the SFN of the block created by the O-nodes assigned by the F-node.

- When input primitive parameters are directly assigned to output primitive parameters, a block is created to include only I- and O-nodes.
- Blocks containing only F- and O-nodes and (possibly) constant D-nodes are created by constant assignments to the O-nodes or through F-nodes which are not included in any of the blocks created before.

An application of the algorithm to the DFG in Appendix B creates the blocks shown by dotted lines.

B. Functional Partitioning of the DFG

The level of block refinement obtained from Algorithm 4.1 is not appropriate for testing purposes, since a very high number of blocks is usually obtained and relatively complex concepts such as quality of service provided, addressing, etc., that are generally specified using several D-nodes create several blocks. These blocks, however, should be treated together as a unique function of the protocol. Therefore, several of these elementary blocks may be combined to form what we call *functional blocks*. Interaction with the test designer is usually necessary to identify the elementary blocks which should be merged. Let SIL (SOL) be the set of labels associated with the input (output) arcs of a node. Then the SIL(B) (SOL(B)) of a block B is the set of labels formed by the union of the SIL (SOL) of the block's D-nodes (O-nodes if B contains no D-nodes; however, in that case SOL(B) is empty).

Block Merging Procedure: Considering the types of the nodes and SIL and SOL of the blocks and nodes, less refined partitions can be obtained by iterative application of the following steps (possibly through interaction with the test designer). The application terminates when a partition is obtained in which blocks can not be further combined.

Step 1: Two blocks *Bi* and *Bj* are combined if SON(*Bi*) and SON(*Bj*) contain parameter(s) of the same type.

This step combines the blocks in which the same parameters but of different output primitives are assigned, since O-nodes of the same type are considered as being part of the same protocol function.

Step 2: Independent blocks (blocks with no incoming arcs from other blocks) *Bi* and *Bj* are combined if the types of all the nodes in SIN(*Bi*) are also contained in SON(*Bj*).

Here, two blocks are combined if one block contains in its O-nodes all the I-nodes of the other block, since the two blocks generally represent the same protocol functions, but in different control phases.

Step 3: Let *Bi* and *Bj* be independent blocks. *Bi* and *Bj* can be combined if SON(*Bi*) and SON(*Bj*) contain different but "related" parameters of the same primitive, and

$$\mathrm{SIL}(Bi) \supseteq \mathrm{SIL}(Bj) \text{ holds.}$$

Which parameters of a primitive are related is determined by the test designer based on the type and use of the parameters.

Since some output primitives can have more than one parameter of the same function, Step 3 is used to combine blocks that assign similar parameters of a given primitive in the same NFT's.

Step 4: The blocks that contain only O- and F-nodes with F-nodes having incoming arcs from D-nodes of different blocks are combined with one of the blocks that contain the D-nodes.

This step is used to combine the blocks of some of the data transfer primitive parameters (O-nodes) with the blocks of the input and output buffers (D-nodes).

Step 5: Blocks Bi and Bj are combined if SDN(Bi) and SDN(Bj) contain related D-nodes (variable or constant) that are used to specify different features of a relatively complex concept (such as quality of service, addressing, etc.), and

$$\mathrm{SIL}(Bi) \supseteq \mathrm{SIL}(Bj) \text{ holds.}$$

Which D-nodes are related must be determined by the test designer, considering the D-nodes used to specify the complex concept in question.

Step 5 is a generalization of Step 3 to D-nodes. It may be used to combine those blocks that contain different parameters (O-nodes) assigned in the same NFT's by related D-nodes.

Step 6: Let Di be a D-node with SOL(Di) = ϕ, i.e., Di is internal not assigning any other node (usually used in enabling predicates). An independent block Bi containing an internal Di is combined with another block Bj if

$$\mathrm{SIL}(Di) \subseteq \mathrm{SIL}(Bj).$$

The block Bi is chosen such that the D-nodes of the block and the internal D-node are assigned in the same NFT's, except possibly for the initialization of the internal D-node.

C. An Example

When Step 1 of the block merging procedure is applied to the DFG in Appendix B, the block containing "source_ref" of the DR primitive is combined with the block containing "local_ref", and the block containing "disconnect_reason" of DR is combined with the block containing "disc_reason". Also, the block of "called_address" is combined with the block of "calling_address".

Step 2 combines the block of "local_ref" with the block of "remote_ref". Similarly, the block of "additional_clear_reason" is combined with the block of "user_reason".

In Step 3, the blocks containing the O-nodes of "additional_clear_reason" and "disconnect_reason" parameters of the DR are combined.

Steps 4 and 6 do not apply to Appendix B example, but are useful in more complex protocol such as the transport protocol class 2 [8].

In Step 5, the blocks containing the O-nodes of "TPDU_size", "QOTS_estimate", "max_TPDU_size", "class_0" and "normal" are combined to obtain a block associated with quality of service (qos). Similarly, the blocks containing "calling_T_addr", "called_T_addr", "remote_address", "calling_address", and "called_address" are combined giving a block associated with the addressing function.

The resulting partition blocks are delimited in Appendix B using dashed lines.

D. Data Flow Functions

Every DFG block resulting from the merging procedure is considered as a data flow function (DFF). Usually, they coincide with specific protocol functions such as connection referencing, endpoint identification, quality of service, user-to-peer or peer-to-user data transfer, etc.

The partition in Appendix B reveals seven data flow functions for the protocol in Appendix A. These functions and the control phases in which they occur are the following:

{ *connection referencing*, *transport user EPI*, *quality of service*, *addressing* }	{ *connection establishment protocol phase* }
{ *user-to-peer data transfer*, *peer-to-user data transfer* }	{ *data transfer phase* }
{ *disconnection* }	{ *connection freeing phase* }

1) Spontaneous Transitions: Through the transition labels on the arcs, a DFG shows the allowed order of execution of self loop NFT's in the CG. For instance, in the block of user-to-peer data transfer of Appendix C, we can see that the NFT labeled P14 (spontaneous) must follow the NFT labeled P13 (WHEN transition). Depending on the length of the data placed in "out_buffer", P14 may be executed one or more times. If the buffer is empty, P14 can no longer be executed. A similar execution order applies to P15 and P16.

E. Data Flow Dependencies

The blocks that have incoming arc(s) from other blocks are called *data dependent* blocks. All other blocks are independent. For example, user-to-peer data transfer block in Appendix B is data dependent on the quality of service block.

Data dependent blocks are usually created when D-nodes assigned in one control phase (dependency causing D-nodes) are used in the operations of the NFT's in other control phases.

V. Test Methodology

Test sequences for protocols can be generated using various test methods designed for FSM's (see for example [13]). However, these techniques ignore the data flow in a protocol. The methodology to be described handles the control as well as the data flow.

We assume the existence of the graph models of normal form transitions (NFT) and the decompositions of the graphs, in terms of the subtours for the control graph (CG) and data flow functions (or functions in short) for the data flow graph (DFG), as described in the preceding sections.

The methodology draws ideas from other test sequence selection methods such as FSM, software, and hardware testing. The resulting test sequences are based on the subtours of the CG, i.e., the transition tours of a FSM [13]. Parameter values of input primitives are selected as in software testing [10], [7]. For incompletely specified functions, we propose the use of fault models as in microprocessor testing [16] (see Section V-E). Therefore the test of a function of the protocol involves the following steps:

A subtour of the CG (giving a sequence of transitions to be applied to the IUT) is selected. From the DFG, the parameters of the input primitives that belong to the function are determined. These parameters are enumerated while all others are fixed to certain values.

The next step is the determination of the output primitives and parameters corresponding to the subtour selected and the function to be tested since they are the only way of observing the effects of parameter variations. The expected values of the remaining parameters (belonging to other functions) are determined from the fixed values assigned to the input primitive parameters.

Note that the enabling conditions of NFT's in the subtour selected must be satisfied in order to be able to execute the subtour. This in turn means that the individual elementary expressions in the NFT's' PROVIDED clauses have to be satisfied.

The proposed test methodology covers all the control paths in the specification and verifies the data flow in each function.

The methodology is now detailed in the following subsections and illustrated by the test design for a single block of the protocol of Appendix A.

A. Selecting a Subtour

The *subtours of a block* are those subtours of the CG which include NFT's in the SIL of the block. For example, peer-to-user data transfer block in Appendix B has the first two subtours of Table I in its set. A given block should be tested using all the subtours of the block. In cases where the same set of arcs is covered by more than one subtour, the subtour initiating the connection from the Lower Tester is selected.

B. Parameter Types and Enumerations

As in software testing, different types of inputs must be varied in the tests. An input variable may be of *enumeration type* or have a *continuous domain* (integer, array of octets, etc.) [7].

The input variables of enumeration type can be tested exhaustively, i.e., data generated for each possible value, while for the variables of continuous domain exhaustive test data generation is not possible. The I-nodes of enumeration type in the DFG of Appendix B are:

options of T_CONNECT_req and T_CONNECT_resp,
class, options, max_TPDU_size of CR and CC, and
disconnect_reason of DR.

I-nodes of continuous type can be divided into the following five groups:

Parametric I-nodes: The values of these I-nodes are determined by each particular implementation and their values are then fixed. For instance, the I-nodes corresponding to the addresses of the user and peer entities belong to this group. The parametric I-nodes of Appendix B are:

calling_T_addr, called_T_addr of CR and CC,
to_T_addr, from_T_addr of T_CONNECT_req.

Reference Value I-nodes: For instance the I-nodes used as source and destination reference values for the connections can be selected arbitrarily, but must be nonzero. The methods of test data selection for software testing [7] can be applied here. Three specific values are selected: the two end points and some interior point of the domain. The following I-nodes of Appendix B are in this group:

source_ref, dest_ref of CR, CC and DR.

Large Integers: The I-nodes that are integers consisting of one or more octets belong to this group. Test data can be selected in the same manner as for reference value I-nodes. Appendix B contains the following I-nodes in this group:

QTS_req of T_CONNECT_req and T_CONNECT_resp.

User Data: The data exchanged between the communicating parties are usually specified as a record containing the length and data (the contents) fields. Due to the importance of error-free user data transmission we suggest the following enumeration scheme:

Length is enumerated exhaustively, starting with the smallest value, i.e., 1 byte. At the same time the contents are varied systematically. An algorithm implementing this scheme is given in Section V-E, it also verifies the correct delivery of every data octet. The following I-nodes in Appendix B are in this group:

length and data of TSDU_fragment of T_DATA_req,
length and data of user_data of DT.

End Point Identifiers (EPI): The interaction with the user of a protocol takes place over an (N)-service access point [4]. An interaction parameter is used to identify the connection end point which the interaction refers to. This parameter is called EPI and its value is locally decided. EPI's are verified in multiple connection tests, since different values are used for different parallel connections.

The following I-nodes of Appendix B are in this group:

TCEPI of T_CONNECT_req and T_CONNECT_resp.

C. Data Flow Considerations

The flow over all variable D-nodes of a block (i.e., the assigning arcs) should be tested by properly selected subtours. In cases where a D-node is assigned by both an I- and an F-node, two tests are designed: One of the tests involves parameter variations for the I-node, and the other tests the variation of values assigned by the F-node.

The F-nodes of a block are treated depending on their type:

F-nodes of Type 1: The test designer determines the values returned depending on the inputs given, usually by consulting the protocol standard. The F-nodes of "get_next_fragment" and "append" in Appendix B are of Type 1.

F-nodes of Type 2: F-nodes which assign implementation dependent values to D- or O-nodes are observed through the O-nodes and verified by consulting the standard for the allowed values. An F-node may also initialize a D-node. In this case, the transitions that occur after the initializing transition are used to observe the value of the D-node. In Appendix B, the F-node of "assign_local_ref" assigns implementation dependent values to "local_ref".

F-nodes of Type 3: Test design for blocks containing these nodes is based on the types of the D-nodes assigned and the set of arcs relating these D-nodes. Appendix B contains no such nodes.

D. Multiple Connection Tests

Handling multiple parallel connections is an important aspect of protocol implementations. Usually, D-nodes representing the connection array sizes define separate blocks in a decomposed DFG, e.g., the TCEP block of Appendix B.

The enumeration of the D-nodes representing the connection array sizes is done to determine the maximum number of parallel connections supported. Testing all other blocks using parallel connections may not be practical due to the increased number of tests. Instead, only the blocks which may be shared by multiple connections (blocks containing data buffers, etc.) are tested. The choice is left on the test designer, unless an exact specification of these shared parts is provided.

E. Test Design for a Block of The Example Protocol

A complete test design for Appendix A is given in [14]. Results of the application of the tests to two implementations are reported in [2]. Here we apply the test design methodology to the peer-to-user data transfer block of Appendix B.

The second subtour in Table I is selected, since its NFT's (P15 and P16) cover the block and the connection establishment is initiated by the LT (see Section I). The I-node of the block is the "user_data" parameter of the DT primitive. It is a continuous domain I-node of type "user data". As discussed in Section V-B, the length and contents of this parameter can be varied using the following procedure:

```
procedure prepare_data(var user_data:string of octets;
        var start_value:octet;
        var remaining_length:pos_integer;
        current_TPDU_size:pos_integer);
var i,ml:pos_integer;
begin
  ml:=min(current_TPDU_size-data_header,remaining_length);
  user_data.length:=ml;
  remaining_length:=remaining_length-ml;
  for i:=1 to ml do
  begin
    user_data.contents[i]:=start_value;
    start_value:=(start_value+1) mod 256
  end;
end prepare_data;
```

Connection establishment sets the variable "TPDU_size" which limits the length of the "user_data" of the DT. The I-node "max_TPDU_size" of the CR PDU is of enumeration type. Therefore, the test for the peer-to-user data transfer block should be repeated for every TPDU_size supported by the IUT.

Predicates of the NFT's of the block contain binding comments on *flow control* such as:

(/flow control from the Network layer is ready/).

Since flow control is not formally specified, a fault model may be used to test this function: If there is an error in flow control, it is assumed that the implementation will either stop or deliver the data to the UT in wrong order or with losses. To test for this fault, the LT sends DT PDUs to the IUT independently of the responses received, creating a continuous flow of data. The UT checks the delivered data for any errors, and sends then a report to the LT.

The peer-to-user data transfer block contains 3 F-nodes:

"append" and "get_next_fragment" of type 1, and "clear" of type 2. The F-nodes of type 1 are operations on "in_buffer" which is an abstract data type. These operations can be observed from the O-nodes that they assign directly (get_next_fragment) or indirectly (append). The F-node of "clear" initializes "in_buffer".

The data transfer phase is terminated when TSDU size reaches a predetermined value and the connection is freed. Since the subtour contains an uncontrollable input, i.e., "Network_reset" (see P19 in Appendix A). the data transfer phase can be terminated unexpectedly. In this case the LT repeats the test after resynchronizing with the UT.

VI. Conclusions

An approach to testing protocol implementations was introduced. It is based on a formal specification of the protocol. Control and data flow graphs for the protocol are obtained from the simplified form of the specification and the graphs are decomposed to obtain various functions of the protocol. These functions are tested by parameter variations and by stimulating all the control paths that exist in the specification. A simple protocol was chosen as an example to illustrate the methodology. Detailed test design for one of the functions of the protocol was given. A more complex example of the class 2 TP was treated in [14].

The flow graphs of the protocol specification are also helpful in protocol design validation. Syntactic and semantic errors in the specification can be detected during the construction of these graphs [15].

In order to derive tests for complex protocols, there is a need for at least partially automating the different steps of the methodology. A prototype implementation is described in [18]. More research is needed for automatically generating the tests from the flow graphs. It may also be interesting to investigate whether the methodology is applicable in other areas of software development.

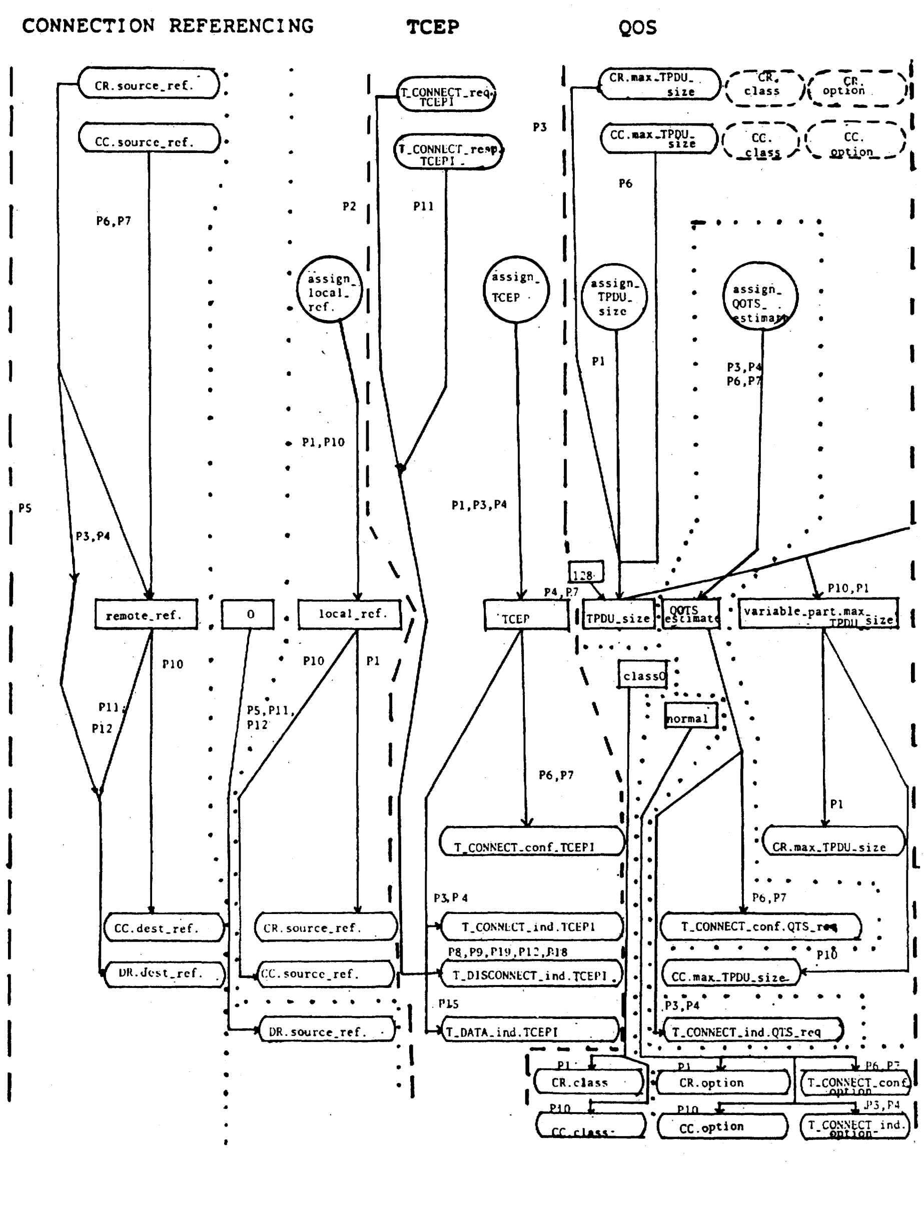
Appendix A
Normal Form Transitions of the Class 0 TP

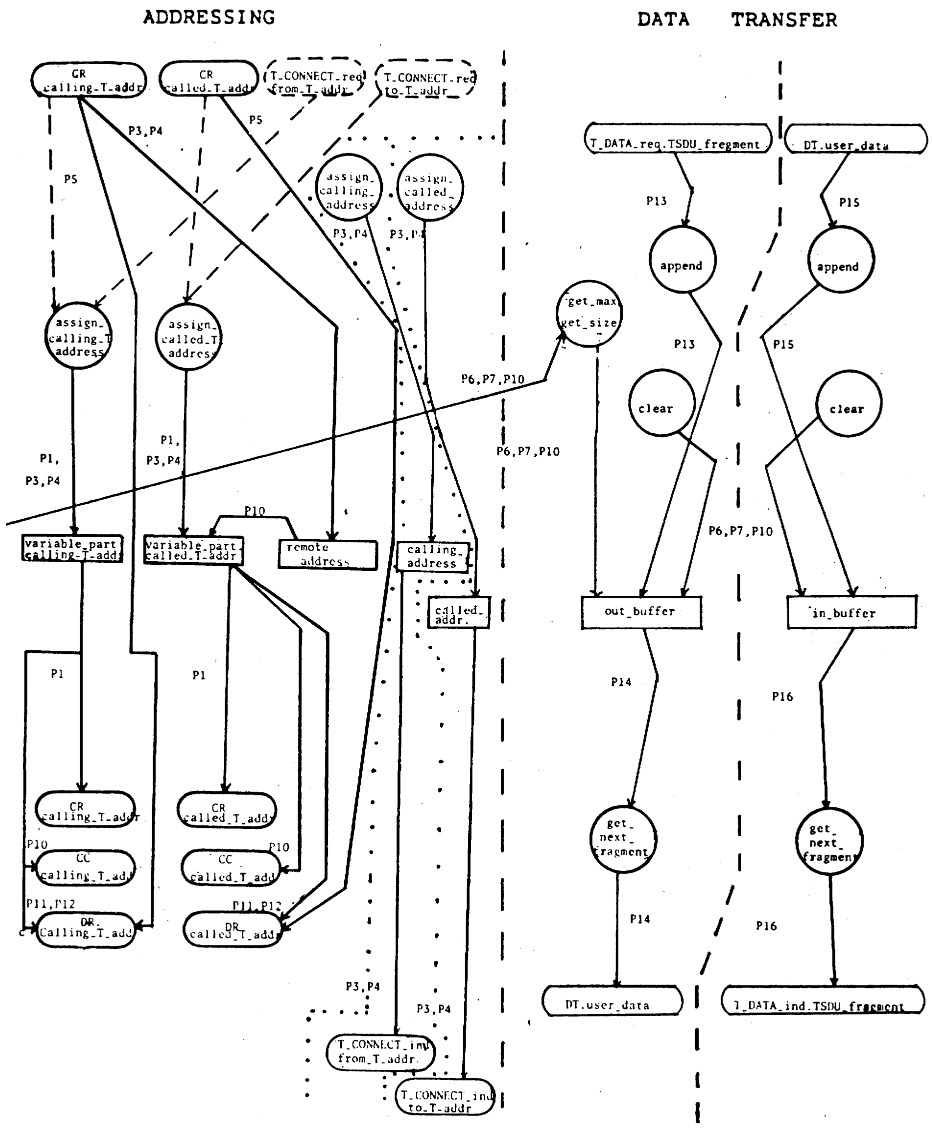
```
(* Data Definitions are as in [9] *)

WHEN TSAP.T_CONNECT_req
FROM idle
PROVIDED (/Transport entity able to provide the quality of service asked for/)
TO wait_for_CC
P1 :BEGIN
   local_reference:=...;
   TPDU_size:=...;
   variable_part_to_send:=...;
   output CR(0,local_reference,class_0,normal,variable_part_to_send);
END;

WHEN TSAP.T_CONNECT_req
FROM idle
PROVIDED (/Transport entity not able to provide the quality of service asked for/)
TO idle
P2 :BEGIN
   output T_DISCONNECT_ind(TCEPI,inability_to_provide_the_quality);
END;

WHEN mapping.CR
FROM idle
PROVIDED variable_part.max_TPDU_size <> undefined and
        (/able to provide the quality of service/)
TO wait_for_T_CONNECT_resp
P3 :BEGIN
   remote_reference:=source_reference;
   TPDU_size:=variable_part.max_TPDU_size;
   remote_address:=variable_part.calling_T_address;
   TCEP:=...;
   called_adress:=...;
   calling_address:=...;
   output T_CONNECT_ind(TCEP,called_address,calling_address,QOTS_estimate,normal);
END;

WHEN mapping.CR
FROM idle
PROVIDED variable_part.max_TPDU_size = undefined and
        (/able to provide the quality of service/)
TO wait_for_T_CONNECT_resp
P4 :BEGIN
   remote_reference:=source_reference;
   TPDU_size:=128;
   remote_address:=variable_part.calling_T_address;
   TCEP:=...;
   called_address:=...;
   calling_address:=...;
   output T_CONNECT_ind(TCEP,called_address,calling_address,QOTS_estimate,normal);
END;

WHEN mapping.CR
FROM idle
PROVIDED (/not able to provide the QOS/)
TO idle
P5 :BEGIN
   variable_part_to_send.additional_clear_reason:= ...;
   output DR(source_reference,0,connection_negotiation_failed,variable_part_to_send);
END;
```

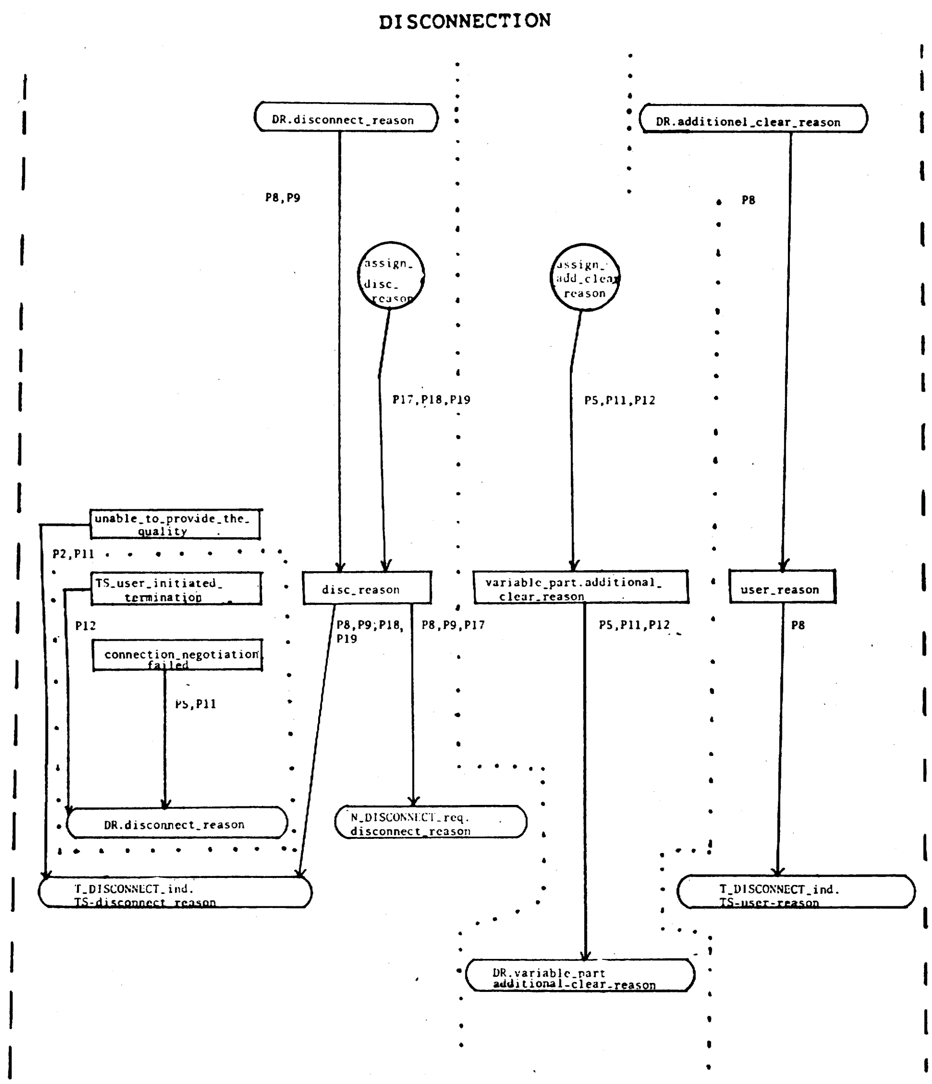
```
WHEN mapping.CC
FROM wait_for_CC
PROVIDED variable_part.max_TPDU_size<>undefined
TO data_transfer
P6 :BEGIN
   remote_reference:=source_reference;
   TPDU_size:=variable_part.max_TPDU_size;
   QOTS_estimate:=...;
   output T_CONNECT_conf(TCEP,QOTS_estimate,normal);
   in_buffer.clear;
   out_buffer.clear;
   out_buffer.set_max_get_size(TPDU_size);
END;

WHEN mapping.CC
FROM wait_for_CC
PROVIDED variable_part.max_TPDU_size=undefined
TO data_transfer
P7 :BEGIN
   remote_reference:=source_reference;
   TPDU_size:=...;
   QOTS_estimate:=...;
   output T_CONNECT_conf(TCEP,QOTS_estimate,normal);
   in_buffer.clear;
   out_buffer.clear;
   out_buffer.set_max_get_size(TPDU_size);
END;

WHEN mapping.DR
FROM wait_for_CC
PROVIDED disconnect_reason=TS_user_initiated_termination
TO idle
P8 :BEGIN
   disc_reason:=disconnect_reason;
   user_reason:=variable_part.additional_clear_reason;
   output N_DISCONNECT_req(...,disc_reason);
   output T_DISCONNECT_ind(TCEP,disc_reason,user_reason);
END;

WHEN mapping.DR
FROM wait_for_CC
PROVIDED disc_reason<>TS_user_initiated_termination
TO idle
P9 :BEGIN
   disc_reason:=disconnect_reason;
   output N_DISCONNECT_req(...,disc_reason);
   output T_DISCONNECT_ind(TCEP,disc_reason,user_reason);
END;

WHEN TSAP.T_CONNECT_resp
FROM wait_for_T_CONNECT_resp
PROVIDED (/quality of service requested <=
          quality of service proposed in T_CONNECT_ind/)
TO data_transfer
P10:BEGIN
   local_reference:=...;
   TPDU_size:=...;
   variable_part_to_send.called_address:=remote_address;
   variable_part_to_send.calling_address:=...;
   variable_part_to_send.max_TPDU_size:=TPDU_size;
   output CC(remote_reference,local_reference,class_0,normal,variable_part_to_send);
   in_buffer.clear;
   out_buffer.clear;
   out_buffer.set_max_get_size(TPDU_size);
END;

WHEN TSAP.T_CONNECT_resp
FROM wait_for_T_CONNECT_resp
PROVIDED (/quality of service requested > quality
          of service proposed in T_CONNECT_ind/)
TO idle
P11:BEGIN
   variable_part_to_send.additional_clear_reason:=...;
   output DR(remote_reference,0,connection_negotiation_failed,variable_part_to_send);
   T_DISCONNECT_ind(TCEP,inability_to_provide_the_quality,...);
END;

WHEN TSAP.T_DISCONNECT_req
FROM wait_for_T_CONNECT_resp
PROVIDED
TO idle
P12:BEGIN
   variable_part_to_send.additional_clear_reason:=...;
   output DR(remote_reference,0,TS_user_initiated_termination,variable_part_to_send);
END;

WHEN TSAP.T_DATA_req
FROM data_transfer
PROVIDED (/flow control from the user is ready/)
TO data_transfer
P13:BEGIN
   out_buffer.append(TSDU_fragment);
END;
```

```
FROM data_transfer
PROVIDED (/flow control to the Network layer is ready/)
TO data_transfer
P14:BEGIN
   mapping.DT(get_next_fragment(out_buffer));
END;

WHEN mapping.DT
FROM data_transfer
PROVIDED (/flow control from the Network layer is ready/)
TO data_transfer
P15:BEGIN
   in_buffer.append(user_data);
END;

FROM data_transfer
PROVIDED (/flow control to the user is ready/)
TO data_transfer
P16:BEGIN
   output TSAP.T_DATA_ind(get_next_fragment(in_buffer));
END;

WHEN TSAP.T_DISCONNECT_req
FROM data_transfer
PROVIDED
TO idle
P17:BEGIN
   output N_DISCONNECT_req(disconnect_reason,user_reason);
END;

WHEN mapping.N_DISCONNECT_ind
FROM data_transfer
TO idle
P18:BEGIN
   disc_reason:=...;
   output T_DISCONNECT_ind(TCEP,disc_reason,user_reason);
END;

WHEN mapping.Network_reset
FROM data_transfer
TO idle
P19:BEGIN
   disc_reason:=...;
   output T_DISCONNECT_ind(TCEP,disc_reason,...);
END;
```

Appendix B
A DFG for the Class 0 Transport Protocol




ADDRESSING
DATA TRANSFER
CR calling_T_addr
CR called_T_addr
T_CONNECT_req from_T_addr
T_CONNECT_req to_T_addr
assign_ calling_ address
assign_ called_ address
assign_ calling_T address
assign_ called_T address
get_max get_size
variable_part calling_T_addr
variable_part called_T_addr
remote address
calling_ address
called_ addr.
CR calling_T_addr
CC calling_T_addr
DR calling_T_addr
CR called_T_addr
CC called_T_addr
DR called_T_addr
T_CONNECT_ind from_T_addr
T_CONNECT_ind to_T_addr
T_DATA_req.TSDU_fragment
DT.user_data
append
clear
out_buffer
in_buffer
get_ next_ fragment
DT.user_data
T_DATA_ind.TSDU_fragment
P1
P1, P3,P4
P3,P4
P5
P6,P7,P10
P10
P11,P12
P13
P14
P15
P16



DISCONNECTION
DR.disconnect_reason
DR.additionel_clear_reason
P8,P9
P8
assign_disc_reason
assign_add_clear_reason
P17,P18,P19
P5,P11,P12
unable_to_provide_the_quality
P2,P11
TS_user_initiated_termination
disc_reason
variable_part.additional_clear_reason
user_reason
P12
P8,P9,P18,P19
P8,P9,P17
P5,P11,P12
P8
connection_negotiation_failed
P5,P11
DR.disconnect_reason
N_DISCONNECT_req.disconnect_reason
T_DISCONNECT_ind.TS-disconnect_reason
T_DISCONNECT_ind.TS-user-reason
DR.variable_part additional-clear-reason

Acknowledgment

We are grateful to an anonymous editor of TSE for constructive suggestions which has lead to many improvements in the paper.

References

[1] J. P. Ansart, "Test and certification of standard protocols," in *Protocol Testing—Towards Proof?* vol. 2, D. Rayner and R. W. S. Hale, Eds. National Physical Laboratory (NPL), 1981, pp. 119-126.

[2] E. Cerny *et al.*, "Experiments in testing communication protocol implementation," in *Proc. 14th Symp. Fault Tolerant Comput.*, 1984, pp. 204-209.

[3] L. A. Clarke and D. J. Richardson, "Symbolic evaluation methods for program analysis," in *Program Flow Analysis*, S. S. Muchnick and N. D. Jones, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[4] J. D. Day and H. Zimmermann, "The OSI reference model," *Proc. IEEE*, vol. 71, pp. 1334-1340, Dec. 1983.

[5] ISO, "ESTELLE—A FDT based on an extended state transition model," ISO TC 97/SC21 DP 9074, Sept. 1985.

[6] W. E. Howden, "Functional program testing," *IEEE Trans. Software Eng.*, vol. SE-6, Mar. 1980.

[7] —, "A survey of dynamic analysis methods," in *Software Testing Validation Techniques*, E. Miller and W. E. Howden, Eds. Washington, DC: IEEE Comput. Soc., 1981, pp. 184-206.

[8] G. v. Bochmann, "Example of transport protocol specification," Doc. de Travail, Montreal Univ., Canada, Rep. ISO TC97/SC16/WG1, Subgroup B, Nov. 1982.

[9] —, "Examples of transport protocol specifications," Doc. de Travail, Montreal Univ., Canada, Rep. ISO TC97/SC16/WGI, Subgroup B, Apr. 1982.

[10] R. E. Prather, "Theory of program testing—An overview," *Bell Syst. Tech. J.*, vol. 62, no. 10, pp. 3073-3105, Dec. 1983.

[11] D. Rayner, Ed., "Protocol implementation assessment: Philosophy and architecture," NPL Rep. DNACS 44/81, Apr. 1981.

[12] D. Rayner, "Standardizing conformance testing for OSI," in Proc. COMNET'85. Amsterdam, The Netherlands: North-Holland, 1986, pp. 47-66.

[13] B. Sarikaya and G. v. Bochmann, "Synchronization and specification issues in protocol testing," *IEEE Trans. Commun.*, vol. COM-32, pp. 389-395, Apr. 1984.

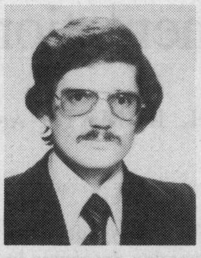
[14] B. Sarikaya, "Test design for computer network protocols," Ph.D. dissertation, McGill Univ., Montreal, Canada, Mar. 1984.

[15] B. Sarikaya, G. v. Bochmann, and J. M. Serre, "A method of validating formal specifications," Concordia Univ., Montreal, Canada, Res. Rep., Jan. 1986.

[16] S. M. Thatte and J. A. Abraham, "Test generation for general microprocessor architectures," in *Proc. 9th FTCS*, 1979, pp. 203-210.

[17] C. Vissers, R. Tenney, and G. v. Bochmann, "Formal description techniques," *Proc. IEEE*, vol. 71, pp. 1356-1364, Dec. 1983.

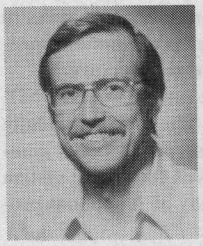
[18] M. Barbeau and B. Sarikaya, "CAD-PT: A computer-aided design tool for protocol testing," Concordia Univ., Montreal, Canada, Res. Rep., Feb. 1987.



Behçet Sarikaya (M'84) received the B.S. and M.S. degrees from the Middle East Technical University, Ankara, Turkey, in 1973 and 1976, respectively, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1984.

He is currently an Assistant Professor at Concordia University, Montreal. His research interests include protocols and software testing.

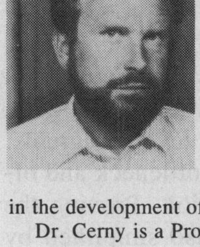
Dr. Sarikaya is a member of the Association for Computing Machinery. He has been a Session Coordinator for the 18th and 19th Hawaii International Conferences on System Sciences and Cochairman of the 6th IFIP Workshop on Protocols.



Gregor v. Bochmann (M'82–SM'85) received the Diploma in physics from the University of Munich, Munich, West Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

He has worked in the areas of programming languages, compiler design, communication protocols, and software engineering and has published over 100 papers in these areas. He is currently a Professor in the Département d'Informatique et de Recherche Operationelle, Université de Montréal, Montréal. His present work is aimed at design models for communication protocols and distributed systems. He has been actively involved in the ISO OSI standardization of formal description techniques. From 1977 to 1978 he was a Visiting Professor at the Ecole Polytechnique Federale, Lausanne, Switzerland. From 1979 to 1980 he was a Visiting Professor in the Computer Systems Laboratory, Stanford University, Stanford, CA.

Dr. Bochmann was the Program Chairman of SIGCOM'84 and Cochairman of the 6th IFIP Workshop on Protocols.



Eduard Cerny (M'73) was born in Czechoslovakia in 1946. He received the B.Sc. (Eng.) degree in electrical engineering from Loyola College, Montreal, P.Q., Canada, in 1970, and the M.Eng. and Ph.D. degrees in electrical engineering from McGill University, Montreal, in 1970 and 1975, respectively.

Currently, he is a Professor of Computer Science and Operations Research at University of Montreal. His interests are primarily in the design, verification, and test of VLSI circuits, and in the development of the associated CAD tools.

Dr. Cerny is a Professional Engineer in the Province of Quebec.