A COMPUTER-AIDED DESIGN TOOL FOR PROTOCOL TESTING

Michel Barbeau
INRS-Telecommunications
3 Place du commerce
Ile des souers, Québec
Canada H3E 1H6

Behçet Sarikaya
Concordia University
Electrical Eng. Dept.
1455 de Maisonneuve W.
Montréal, Québec
Canada H3G 1M8

ABSTRACT

Reliable communication among heterogenous computers in a distributed system will be built on standardized protocols based on the open systems interconnection reference model. Testing these protocols locally or from a distance is of prime importance for OSI. In this paper, a computer-aided design tool is presented for designing tests for protocols. CAD-PT accepts a formal specification in Estelle of the protocol and generates control and data flow graphs on a graphic workstation. The tool is explained component by component by taking a simplified ISO Class 2 transport protocol as an example.

1.INTRODUCTION

In order to facilitate interconnection of computer systems of various types, international standardization institutions such as ISO and CCITT have developed a reference model of open systems interconnection (OSI). OSI typical applications are file transfer, remote job entry, distributed data base access, teletex and electronic mail. The last two applications are also classified as teleservices in integrated services data network (ISDN) context.

What makes OSI work is the standardization of protocols and services. Standard definitions are presently given in natural language and because of this the definitions contain ambiguities and impreciseness. ISO and CCITT have been involved in defining formal languages with which protocols and services can be specified. The development of the formal description techniques has benefited from earlier efforts of protocol modelling with finite-state machines (FSM) [Dant 83] and Petri nets [Diaz 82]. A Formal Description Technique (FDT) of interest to us in this paper is Estelle [Linn 85]. Estelle is based on an extended finite-state machine model. Many of the OSI protocols have already been formally specified and formal specification of others are being developed.(an example is given in [ISO 82])

Protocols and services of OSI and ISDN are complex. They have to be throughly tested before being incorporated in the system. Another related activity is testing for conformance to the protocol specification. Conformance testing is usually undertaken by national or international institutions. There are two major aspects of protocol testing: defining the environment in which an implementation under test (IUT) will be tested and defining the tests to be applied to IUT. We are interested in test design for protocols and we will describe a tool for this purpose. The resulting tests can be used in any architecture defined by ISO Conformance Testing Methodology [ISO 86].

The computer-aided design tool CAD-PT is a partial implementation of a protocol test design methodology [SaBoCe 87]. The methodology derives graphical models of the formal specification of a protocol/ service. The control graph is used to define the interaction sequences to be applied / observed during a test while the data flow graph decomposes the protocol data flow into various simpler flows called functional blocks.

The 5th generation programming language Prolog is suggested as a modelling tool for protocol verification [Sidh 83], protocol analysis [UrPr 86] and testing [UrSh 86]. Much earlier, Prolog was used in compiler development [Warr 80]. In CAD-PT we used Prolog for syntactic and semantic analysis of Estelle specifications. The result is a quicker way of implementing compilers since Prolog programs are much shorter than those written in conventional languages.

The paper is organized as follows: Section 2 gives an overview of protocol specification and the formal description language Estelle, Section 3 introduces the tool whose components are explained in detail in sections 4 (lexical/ semantic analysis and normalization), 5 (control flow analysis) and 6 (data flow analysis). Finally, Section 7 contains authors' conclusions.

2.PROTOCOL SPECIFICATION

In this section we discuss protocol specification with Estelle formal specification language.

Estelle describes a protocol entity as a collection of modules. Each module is a finite-state machine capable of having memory, i.e., extended finite-state machine. Modules of an entity can communicate with each other as well as with the modules of adjacent layer entities over channels (FIFO queues). Service primitives (with bottom and upper layer entities), and internal interactions (with other modules) are communicated in the channels. Protocol data units (PDUs) need not explicitly be defined; they are encoded/decoded to become service primitives. Decomposition of a protocol entity into modules is usually functional: a module for timer management, a mapping module to map PDUs into (N-1) service primitives, an abstract protocol module for handling (N+1) service primitives and forming PDUs, etc.

Language of Estelle is based on Pascal with extensions to facilitate protocol specification. To save space we only describe the constructs related to transitions. **FROM/TO** clauses define initial/ final state(s) of the finite-state machine, respectively. The arrival of an input interaction is expressed using **WHEN**. Transitions with no WHEN clause are called spontaneous. They are used to describe nondeterminism (internal decisions of the entity, for example). The conditions for firing the transition are described in a **PROVIDED** clause which is a Boolean expression on interaction primitive parameters and variables of the module. Variables of the module are called context variables. Finally, the action of the transition is contained in a **BEGIN** block which can have assignments to context variables, calls to internal procedures, Pascal conditional statements and produce output with **OUTPUT** statement.

An example protocol specification in Estelle is given in Appendix A. This specification will be used as an example in the sequel. It is a simplified version of the transport protocol class 2 entity specified in [ISO 82]. Only abstract protocol module (ATP_type) is considered. This module has 5 major states : closed (idle state), wtcc (wait for CC TPDU), wtcr (wait for TCONNECTresponse from the user), open (connection established) and wfdc (wait for DC TPDU). ATP_type module has 18 transitions and communicates with modules Map (for mapping) and TS (for transport service). Only transitions of the data transfer phase along with one transition of the connection establishment phase are listed in Appendix A.

3.STRUCTURE OF THE TOOL

The first step applied to an Estelle specification is **normalization**. The process of normalization (to be explained in Section 4) gives rise to simpler specifications called normalized specification. In a normalized specification we identify two types of flows: **control flow** which models major state transitions from one transition to another; and **data flow** which models flow from input primitive parameters to context variables and from context variables to the output primitive parameters.

Control flow analysis step (to be discussed in Section 5) generates the control graph which is the FSM of the protocol. From this FSM, test squences can be generated. Data flow analysis step (to be discussed in Section 6) generates the data flow graph and partitions it. Protocol functions are obtained from this graph. These functions can be tested independently from each other with the application of the test sequences.Figure 1 shows the global structure of the tool. The rectangles are the steps and rounded squares show the product obtained from the step.

4.LEXICAL/SEMANTIC ANALYSIS AND NORMALIZATION

To ensure a correct specification for the normalization phase, the input specification is syntactically and semantically analysed.

Lexical analysis and parsing phase verify that the input Estelle specification has no syntactic errors. This phase is implemented using standard tools of Lex and Yacc available in Unix™ operating system. The output produced is a parse tree expressed as a single Prolog clause.

Lexical analysis phase accepts a subset of Estelle language [Este 87]. The constructs concerning dynamic module creation (init, attach, etc.) are excluded. In the present version, the conditional statements of for, repeat and while are not accepted.

Semantic analysis phase takes the tree produced by the previous phase and verifies the definitions, correct use of variables in the statements and prepares a symbol table. This phase is completely implemented in Prolog. The algorithm used is an adapted version of [WaGo 85] More detail on the structure of the symbol table can be found in [Barb 87].

4.1.Normalization

Symbolic execution is a technique for static program analysis [ClRi 81]. It has been used for program verification and testing. An Estelle specification can be symbolically executed for the purpose of identifying all the control paths. It is also possible to express these paths as distinct transitions using Estelle syntax [SaBoSe 86].

In our tool the normalization, i.e. symbolic execution of Estelle specifications, is done in three steps: first procedure and function calls are replaced, then conditional statements (IF, CASE) are eliminated and finally FROM and TO clauses are processed so that each transition contains maximum a single major state change. Each module of the specification is treated *independently*, resulting in a normalized specification for each module. Steps of normalization are detailed in what follows.

4.1.1.Procedure/Function Call Elimination

Each procedure / function body definition is converted to an internal representation to facilitate replacement procedure. Local variables are converted to global variables with unique identifiers and a global variable is sometimes created for each parameter which is called by value, i.e. when the variable is assigned a value. For the result returned by each function, a global variable is created.

All procedure/function calls are replaced by the corresponding begin-end block. In replacing a procedure call, here is what happens: i) The begin-end bloc is obtained from the symbol table, ii) For each value parameter which is assigned in the body of the procedure, an assignment statement is placed before call replacement occurs iii) All other parameters are symbolically replaced by the actual parameters. Function call replacement is similar and explained in [Barb 87].

Procedures/functions declared as *primitive* are not eliminated. In Estelle, this mechanism is used for defining operations on abstract data types.

4.1.2.Conditional Statement Elimination

Conditional statements in the transitions can be eliminated by creating two or more equivalent transitions and modifying the PROVIDED clause to reflect the condition from taking this path.

Modification of the PROVIDED clause is compicated in cases where the boolean expression of the conditional statement contains variables assigned in the same transition before the conditional statement. A *symbolic replacement* should be made for these variables. The present version of the tool does not accomplish this, instead the user is signaled of the case with a warning message.

Elimination of the CASE statement is a generalization of the IF statement. The loop statements FOR, REPEAT and WHILE (excluded from the language accepted by the present version) are difficult to process in symbolic execution. CAD-PT ignores these statements in the normalization phase. However, the user can eliminate them by assuming that the loop count can statically be known and by repeating the loop body accordingly. While this assumption may not be realistic in general case, it turned out to be true for the specifications we have treated so far.

4.1.3.FROM/TO Clauses

This step simplifies FROM/TO clauses in a given specification. State lists and state sets are eliminated by creating more than one transition, one for each state in the list. This elimination is facilitated by other steps of normalization since the control paths in the actions are identified.

4.2.Example

The example specification in Appendix A (the simplified transport protocol specification) will be treated for normalization. The specification contains an abstract data type definition (buffer_type), two variables of this type (send_buffer, receive_buffer) and operations on this type (store, remove, etc.) as *primitive* procedures and functions.

Normalized transport protocol specification is listed in Appendix B. The transitions are numbered starting from 1. Normalized transition 0 corresponds to initialization (initialize in Estelle). Normalization produces 24 transitions of which Appendix B contains only the normalized transitions of the data transfer phase to save space.

5.CONTROL FLOW ANALYSIS

Major state changes from one normalized transition to another is called control flow. Control flow is best shown by a state diagram. Because of availability of an algorithm to graphically display a tree, we have opted for displaying the state diagram as a tree. In [Chow 78], such a tree is called **testing tree**.

Control flow analysis module is implemented in Prolog and C. The testing tree is created by a Prolog program and then displayed by a C program using Sun workstation graphics [SUN 85]. From a tree representation of the normalized specification, Prolog program extracts the names of the inputs (WHEN clause) and outputs (output clause) and creates an intermediary file. The transition function of the FSM is easily obtained from FROM and TO clauses and also placed in an intermediary file. A C program then displays the testing tree using the information in the files. The algorithm used to display the tree can be found in [Vauc 80].

An example testing tree of the transport protocol as obtained by our tool is shown in Figure 2. Rectangles are the states and arcs are the transitions. On each arc shown are the number of the normal form transition, the input and the output. Missing input/output are represented by the word *nil*. Apart from displaying the tree, the module creates a file which describes the FSM. This file can furthermore be taken as input to test sequence generation modules based on FSMs.

6.DATA FLOW ANALYSIS

Actions of the normalized transitions can be seen as a collection of operations which process parameters of input primitives (service primitives and/or PDUs) in order to determine the values of parameters of the output primitives. This processing is done using context variables as storage and applying certain functions such as arithmetic operations or abstract data type operations on context variables. Thus we define value changes on the context variables as *data flow*. Data flows from input primitive parameters to the context variables and from context variables to output primitive parameters. This leads us to a natural graphical representation of the actions of normalized transitions, called data flow graph (DFG). Generation of DFGs is only possible when PDU (being input/ output primitives) parameters are explicitly identified. This point is discussed in Section 6.2.

A data flow graph such as shown in Figure 3 can be interpreted as displaying the overall flow of information on each context variable. Therefore, this type of flow graphs are called **global data flow graphs**. This property distinguishes them from other data flow graphs such as parallel data flow graphs described in [ArCu 86]. Parallel data flow graphs for protocols are discussed in [AhSa 87].

6.1.Description of DFGs

In the upper part of the data flow graph, data sources (input primitive parameters) are placed and similarly, data sinks (output primitive parameters) are placed in the bottom. All other nodes are placed in the middle. The arcs describe the information flow of the statements in the actions. For example, for a simple assignment statement, an arc is created from source node (the variable on the right hand side of the := operator), to the destination node (the variable on the left hand side of the := operator).

Procedure/function parameters are represented depending on whether they are passed by value or by reference: Value parameters are connected by an arc to the node representing the procedure; reference parameters are connected to the node by a two-directional arc since the parameter can be at times an input or an output.

The arcs are labelled with the number of the normalized transition in which the statement it models can be found. The same assignment statement occuring in more than one transition is represented with a single arc containing a list of transition numbers.

6.2.PDU Field Identification

Identification of individual fields of service primitives and protocol data units (PDU) is necessary for the data flow graphs. For service primitives, Estelle specifications contain an explicit list of fields but such is not the case for the PDUs. It is common practice to define a single record for PDUs and identify them using an identification field. Appendix A contains TPDU_and_control_information which is used to define the fields of all the PDUs, each identified by the field "TPDU_code_type".

The tool requires user intervention in order to identify fields of PDUs in case they are commonly defined in the specification. The user is asked to create a file in which (s)he places the name of the record defining the PDUs, the id field and list of the fields of the PDUs. In case of the transport protocol, such a file would contain:

```
pdu_type(tpdu_and_control_information).
pdu_id_field(kind).
pdu(cr,[peer_address,options_ind,credit_value]).
pdu(cc,[options_ind,credit_value]).
pdu(dr,[disconnect_reason,is_last_pdu]).
pdu(dc,[]).
pdu(dt,[user_data,send_sequence,end_of_tsdu]).
pdu(ak,[expected_send_sequence,credit_value]).
```

The Prolog clause pdu_type defines the *structure* containing the PDUs in the specification. The field of the structure which identifies the PDUs is specified by pdu_id_field clause. Following this is the pdu clauses, one for each possible value of pdu_id_field. The pdu clauses associate to each type of pdu, a list of the names of the pdu fields.

PDU identification module is part of the data flow analysis module and constitutes its first phase. When PDU identification module is called with an auxiliary file a modified normalized specification is generated. The modifications to the specification are done to reflect the changes in the field which defines the PDUs, i.e. explicit definition of the fields of each PDU of the protocol. The *structure* is replaced with two or more new structures whose names are automatically created by adding the id of the PDU to the end of the structure. Channel definitions are modified to explicitly list the PDUs exchanged. Transitions need to be modified to detect the places where PDUs are generated and used as parameters to the output statements.

The appendix B contains the normalized transport specification in which PDU fields are identified using the auxiliary file listed above. Modifications done by the PDU field identification module can be found in the Appendix by inspection.

6.3.DFG Generation

Data flow analysis module is also implemented in Prolog and C. A Prolog program generates a representation of the graph from the tree representation that is the output from the PDU identification module in an intermediary file. A program written in C displays the graph on the screen. The Sun workstation graphics facilities are used to easily show the graph. Since optimal placement of the nodes on the screen is difficult to achieve, instead the tool displays the nodes in some order, but the user can easily displace any node interactively. The final form is stored in a file for further processing.

As an example, the data flow graph as generated by the tool of the transport protocol is given in Figure 3.

6.4.Partitioning the Data Flow Graph

A data flow graph can be partitioned into blocs, a bloc representing the flow over a single context variable. The blocs in the figure 3 are separated with vertical lines. A partitioning algorithm can be found in [SaBoCe 87]. In [SaBoCe 87] it is shown that by *merging* some of the blocs, it is possible to obtain representations of most of the protocol functions. Merging can only be automated to some extent since what variables to merge in order to obtain a protocol function is sometimes left to the user. The present version of the tool does not attempt automatic merging, but it is left to the user. The functions are usually identified as data transfer for sending, data transfer for receiving and flow control for sending and receiving, etc.

6.5.Test Design

The aim of test design using the tool is to generate test sequences that guarantee coverage of each arc in the data flow graph of each function. Coverage of the complete control graph is assured by applying all the subtours obtained from the control graph. The methodology is detailed in [SaBoCe 87]. In the present version of the tool, test design is left completely to the user. As an example we will design a test for the *data transfer for sending* function shown in Figure 3.

Local user of the protocol (called upper tester in conformance testing) sends *T_Data_req* service primitive with *ts_user_data* parameter containing a single byte with 0 value and *end_of_tsdu* parameter containing true value. This information is verified by the peer user (called lower tester in conformance testing) from the *DT* PDU received. The local user continues to send *T_Data_req* and each time changes the contents of *ts_user_data*, achieving an enumeration of this parameter. It is also possible to increase the length and continue enumerating the data. To ensure coverage of all the arcs, the local tester should send some long data so that the protocol entity stores the data in *send_buffer* and then transmits them in consecutive DTs. More detail can be found in [SaBoCe 87].

7.CONCLUSIONS

A protocol test design tool is presented. The tool aids the user in designing protocol tests by graphically displaying the control and data flow graphs of the protocol. These graphs are obtained from a formal specification of the protocol in Estelle. Control graph leads to interactions to be applied in the tests and the data flow graph decomposes the protocol into various simpler functions, most of them can be tested independently.

More automation of the test design process is being planned in the future versions of the tool. This way, nonexpert users will be able to benefit from the tool. We believe that even this will not make the protocol test design a completely automatic process since the best tests require both expertise and intelligence.

Protocol test design can also be based on Lotos formal specification language. Theoretical results indicate that Lotos specifications exhibit similar structures as normal form transitions [AhSa 86]. This encourages us to work towards obtaining another version of our tool which also supports Lotos. Users interested in either formal description technique will be able to use the tool to design protocol tests.

Acknowledgements. We are grateful to Prof. G.v. Bochmann of Université de Montréal for his encouragements and fruitful discussions. The first author acknowledges his financial support.

8.REFERENCES

[AhSa 86] R. Ahooja, B. Sarikaya, "Comparing Normal Forms Obtained from Estelle and Lotos Specifications", 6th IFIP Workshop on Protocols, June 1986, North-Holland 1987.

[AhSa 87] R. Ahooja, B. Sarikaya, "A Unified Data Flow Model for Estelle and Lotos", Submitted for publication, available from Concordia University, February 1987.

[ArCu 86] Arvind, D.E. Cueller, "DataFlow Architectures", Annual Reviews in Computer Science, 1986.

[Barb 87] M. Barbeau, "Prototype d'un Système d'Aide à la Conception de Test de Protocoles", M.Sc. Thesis, Université de Montréal, February 1987.

[ClRi 81] L.A. Clarke, D.J. Richardson, "Symbolic Evaluation Methods for Program Analysis", Program Flow Analysis, S.S. Muchnick, N.D. Jones (Eds.), Prentice-Hall, 1981.

[Chow 78] T.S. Chow, "Testing Software Design Modeled by Finite-State Machines", IEEE Trans. on Software Engineering, Vol. SE-4, No.3.

[Dant 83] A.S. Danthine, "Protocol Repesentation with Finite-State Machines", Computer Networks and Protocols, P.E. Green (Editor), May 1983, pp. 579-606, Plenum Press.

[Diaz 82] M. Diaz, "Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models", 2nd IFIP Workshop on Protocols, pp.465-510, 1982.

[Este 87] ISO/TC 97/SC 21 Estelle, "A Formal

Description Technique Based on an Extended State Transition Model", DIS 9074, July 1987.
[ISO 82] ISO/TC97/SC16/WG1, "Example of a Transport Protocol Specification", Nov. 1982.
[ISO 86] ISO/TC 97/SC21/WG1 E28, "Conformance Testing Methodology and Framework", DP 11-12, Sept. 1986.
[Linn 85] R.J. Linn, "The Features and Facilities of Estelle", 5th IFIP Workshop on Protocols, pp.271-296, June 1985, North-Holland 1986.
[SaBoCe 87] B. Sarikaya, G.v. Bochmann, E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Trans. on Software Eng., May 1987.
[SaBoSe 86] B. Sarikaya, G.v. Bochmann, J-M. Serre, "A Method of Validating Formal Specifications", Submitted for publication. Available as a research report from Concordia University, 1986.
[Sidh 83] D.P. Sidhu, "Protocol Verification via Executable Logic Specifications", 3rd IFIP Workshop on Protocols, pp.237-248, 1983.
[SUN 85] SUN- MICROSYSTEMS, "Programmer's Reference Manual for Sunwindows", 1985.
[UrPr 86] H. Ural, R.L. Probert, "Step-Wise Validation of Communication Protocols and Services", Computer Networks and ISDN Systems, Vol-11, No.3, pp.183-202, 1986.
[UrSh 86] H. Ural, R. Short, "An Interactive Test Sequence Generator", Proc. of SIGCOM'86, Computer Comm. Review, Vol.16, No3., pp.241-250.
[Vauc 80] J.G. Vaucher, "Pretty-Printing of Trees", Software-Practice and Experience, Vol. 10, 553-561.
[WaGo 85] W.M. Waite, G. Goos, "Compiler Construction", Springer-Verlag, 1985.
[Warr 80] D.H.D. Warren, "Logic Programming and Compiler Writing", Software-Practice and Experience, Vol. 10, pp. 97-125.

APPENDIX A

TRANSPORT PROTOCOL SPECIFICATION IN ESTELLE

```
specification Transport;
const
max_seq_no = 127;

type
T_address_type = ...;
reference_type = ...;
option_type = ...;
seq_number_type = 0..max_seq_no;
credit_type = seq_number_type;
data_type = ...;
buffer_type = ...;
PDU_size_type = integer;
reason_type = ( TS_user_initiated, wrong_options );
TPDU_code_type = (CR,CC,DR,DC,DT,AK);
TPDU_and_control_information = record
peer_address : T_address_type;
kind : TPDU_code_type;
dest_ref, source_ref : reference_type;
credit_value : credit_type;
user_data : data_type;
options_ind : option_type;
is_last_PDU : boolean;
disconnect_reason : reason_type;
send_sequence : seq_number_type;
end_of_TSDU : boolean;
expected_send_sequence:seq_number_type;
end;

channel TCEP_primitives ( user, provider );
by user:
T_DATA_req( TS_user_data : data_type );
user_ready(length:integer);
..
by provider:
T_DATA_ind( TS_user_data : data_type;
        is_last_fragment_of_TSDU:boolean)
...

channel PDU_and_control_primitives(
        protocol,mapping );
by protocol, mapping:
frwrd(PDU:TPDU_and_control_information);
ready_for_sending;
...

module ATP_type process(
        TS:TCEP_primitives (provider);
        Map:PDU_and_control_primitives(protocol));

body ATP_body for ATP_type;

var
state:(CLOSED,WTCC,WTCR,OPEN,WTDC);
TRseq, TSseq : seq_number_type;
R_credit, S_credit : credit_type;
send_buffer,receive_buffer : buffer_type;
max_PDU_size : PDU_size_type;

( local functions )
function DT_PDU( s : seq_number_type;
        d : data_type; e : boolean ) :
TPDU_and_control_information;
begin
  DT_PDU.kind := DT;
  DT_PDU.user_data := d;
  DT_PDU.send_sequence := s;
  DT_PDU.end_of_TSDU := e;
end;

function AK_PDU(s: seq_number_type;
  c:credit_type):TPDU_and_control_information;
begin
AK_PDU.kind := AK;
AK_PDU.expected_send_sequence := s;
AK_PDU.credit_value := c;
end;
(* other functions forming CR,CC,DR and DC are
   omitted *)
procedure Strore(var Buf: buffer_type;
        data:data_type);
        primitive;
function Retrieve(Buf:buffer_type):data_type;
        primitive;
function length_available(Buf: buffer_type):integer;
        primitive;
(* indicates the length of data unit in  the buffer *)

function is_end_of_DU(Buf: buffer_type):boolean;
        primitive;
(* indicates whether the data in the buffer includes a
complete DU*)
function enough_space(Buf: buffer_type;
        length:integer): boolean;primitive;
(* indicates whether Buf has enough space for length
   octets of data*)

initialize
to CLOSED
begin end;

{ Transitions }
trans
(* transitions of call establishment/
 disconnection and call refusal are omitted *)

{ Data Transmission}
when TS.T_DATA_req
from OPEN to same
begin
Store(send_buffer, TS_user_data);
end;

when Map.ready_for_sending
from OPEN to same
provided (length_available(send_buffer)
        >= max_PDU_size)
        or is_end_of_DU (send_buffer)
begin
output Map.frwrd( DT_PDU(TSseq,
        Retrieve(send_buffer),
        is_end_of_DU(send_buffer) ) );
S_credit := S_credit - 1;
TSseq := ( TSseq + 1 ) mod ( max_seq_no + 1 );
end;

{ Receiving Data }
when Map.frwrd
provided (PDU.kind=DT) and
        enough_space(receive_buffer,
max_PDU_size)
from OPEN to same
begin
if (R_credit <> 0) and (PDU.send_sequence = TRseq)
then begin
        TRseq := TRseq + 1;
        R_credit := R_credit - 1;
        Store(receive_buffer,PDU.user_data);
end; (/ else error /)
end;

when TS.user_ready
provided length_available(receive_buffer) > length
from OPEN to same
begin
output TS.T_DATA_ind( Retrieve(receive_buffer),
        is_end_of_DU(receive_buffer));
end;

{ Acknowledgements }
from any_state to same
begin
R_credit := ...
end;

from open to same
begin
output Map.frwrd( AK_PDU( TRseq, R_credit  ) );
end;

when Map.frwrd
provided PDU.kind = AK
from OPEN to same
begin
{ Calculate new credit }
if PDU.expected_send_sequence <= TSseq
then
S_credit := PDU.expected_send_sequence
        + PDU.credit_value - TSseq
else
S_credit := ((PDU.expected_send_sequence
        + PDU.credit_value) mod 128) -TSseq;
end;
end;
end.
```

APPENDIX B
NORMALIZED TRANSPORT PROTOCOL SPECIFICATION

```
trans
(* 12 *)
when ts. t_data_req
from open
to open
begin
store(send_buffer,ts_user_data)
end;
(* 13 *)
```

```
when map. ready_for_sending
provided
(length_available(send_buffer)>= max_pdu_size) or
is_end_of_du (send_buffer)
from open to open
begin
dt_pdu_8_dt.user_data:=retrieve( send_buffer))
dt_pdu_8_dt.send_sequence:=tsseq;
dt_pdu_8_dt.end_of_tsdu:=
        is_end_of_du(send_buffer);
output map.frwrd_dt(dt_pdu_8_dt);
s_credit:= s_credit - 1;
tsseq:=(tsseq+1) mod (max_seq_no + 1)
end;
(* 14 *)
when map.frwrd_dt
provided
        enough_space
(receive_buffer,max_pdu_size))
        and ( ( r_credit <> 0)
        and ( pdu. send_sequence = trseq))
from open to open
begin
trseq:= trseq + 1; r_credit:=r_credit - 1;
store(receive_buffer, pdu. user_data)
end;
(* 15 *)
when map. frwrd_dt
provided
        enough_space(receive_buffer,max_pdu_size)
        and
        not((r_credit<>0)
        and(pdu.send_sequence=trseq))
from open to open
begin end;
(* 16 *)
when ts. user_ready
provided length_available( receive_buffer) > length
from open to open
begin
output ts. t_data_ind(
        retrieve(receive_buffer),
        is_end_of_du(receive_buffer))
end;
(* 18 *)
from open to open
begin
r_credit:= ...
end;
(* 22 *)
from open to open
begin
ak_pdu_9_ak.expected_send_sequence:=trseq;
ak_pdu_9_ak. credit_value:=r_credit;
output map. frwrd_ak(ak_pdu_9_ak)
end;
(* 23 *)
when map. frwrd_ak
provided not (pdu. expected_send_sequence<=tsseq)
from open to open
begin
s_credit:=((pdu. expected_send_sequence
        + pdu. credit_value) mod 128) - tsseq
end;
(* 24 *)
when map. frwrd_ak
provided pdu.expected_send_sequence <= tsseq
from open to open
begin
s_credit:=pdu.expected_send_sequence
        + pdu. credit_value - tsseq
end;
```

Figure 1. Global Structure of the Tool

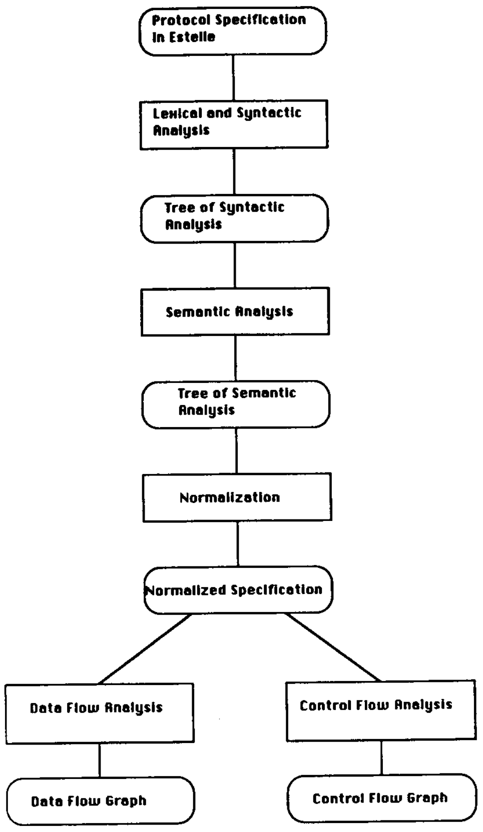
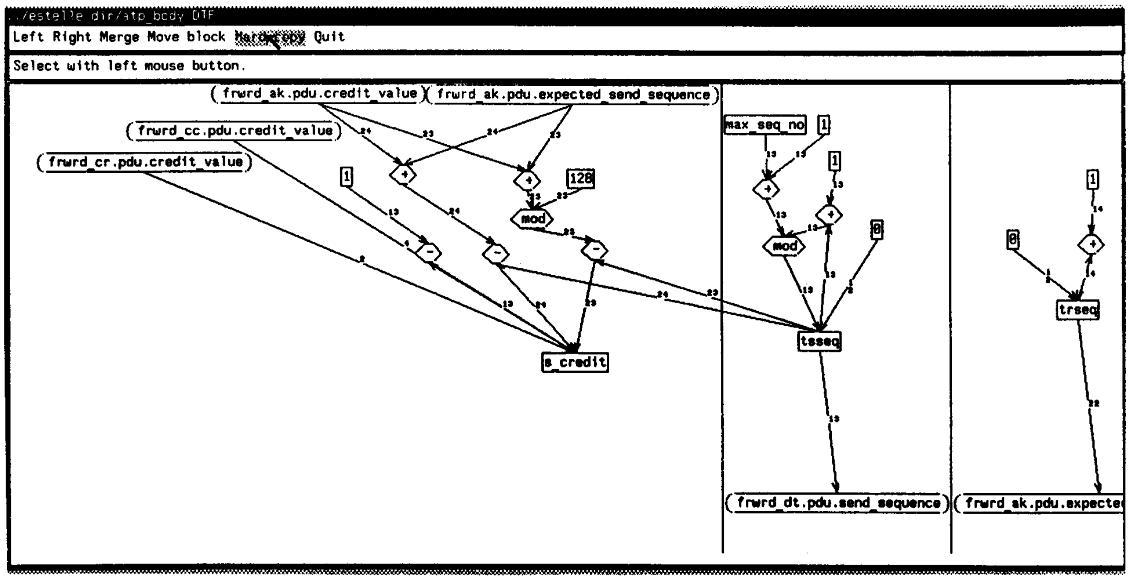


Figure 3. Data Flow Graph of the Transport Protocol

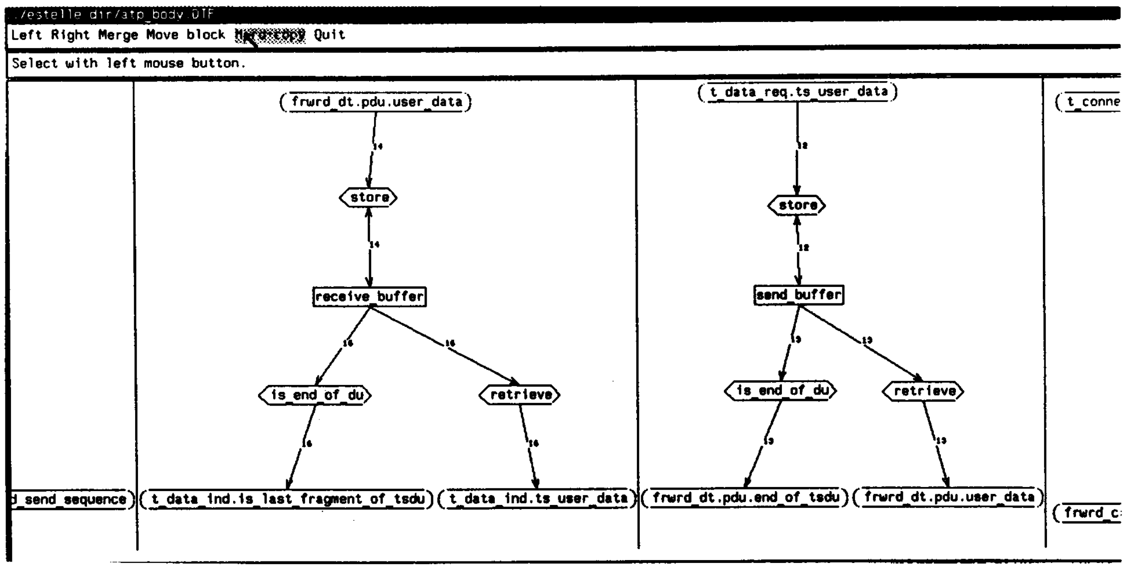


../estelle dir/atp_body DTF

Left Right Merge Move block Macro-copy Quit

Select with left mouse button.

frwrd_ak.pdu.credit_value
frwrd_ak.pdu.expected_send_sequence
frwrd_cc.pdu.credit_value
frwrd_cr.pdu.credit_value
+
+
128
mod
1
-
-
-
s_credit
max_seq_no
1
1
+
+
mod
0
tsseq
frwrd_dt.pdu.send_sequence
1
0
+
trseq
frwrd_ak.pdu.expecte

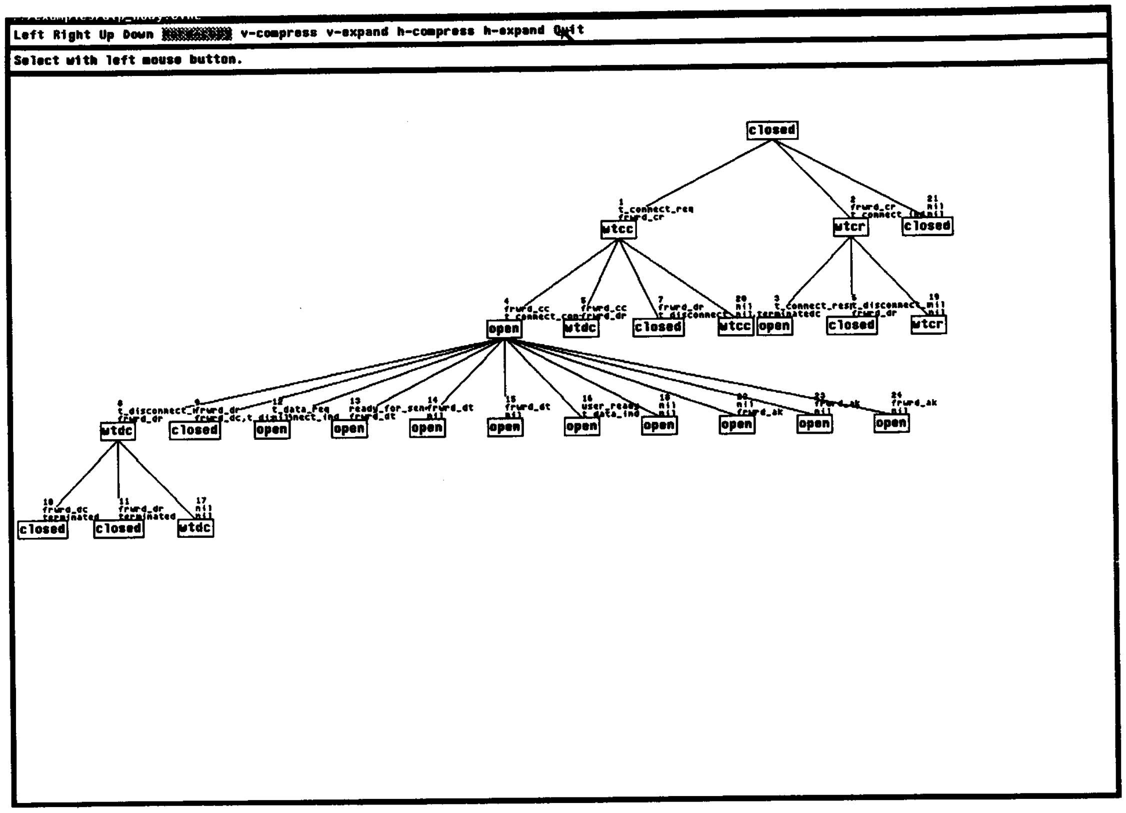


../estelle dir/atp_body DTF

Left Right Merge Move block Macro-copy Quit

Select with left mouse button.

Figure 2. A Control Graph of the Transport Protocol


Select with left mouse button.