FTAM Test Design Using an Automated Test Tool†

Michel Barbeau
Département d'I.R.O., Université de Montréal
C.P. 128, Succ. "A", Montréal, Québec, Canada, H3C 3J7

Behçet Sarikaya, Srinivas Eswara, Vassilios Koukoulidis
Concordia University, Electrical and Computer Eng. Dept
1455 de Maisonneuve W., Montréal, Québec, Canada H3G 1M8

Abstract
An experience is presented with an automated test design tool for functional analysis and test derivation of distributed systems formally specified using Estelle, a description technique based on an extended finite-state machine model. The tool accepts a formal specification of the system and generates control, data flow graphs and unparameterized test sequences. The tool has been used, on an experimental basis, for conformance test design of ISO File, Transfer, Access and Management protocol.

Keywords: Distributed system, software testing, functional analysis, communication protocol, data flow graph, file transfer protocol.

1. Introduction

Distributed system specification and analysis have recently been an area of active research and development. An area of distributed system research that is of interest to us in this paper is protocols. Protocols provide communications in distributed systems. Various tools have been developed for protocol verification[1] and specification[2]. In order to provide reliable communication among computers from different manufacturers, an architecture has been defined to structure the protocols in various layers. This architecture is called Open Systems Interconnection (OSI) reference model. What makes OSI work is the standardization of its protocols and services. Standard definitions are presently given in natural language. Natural language specifications are inherently ambiguous. Standardization institutions such as ISO and CCITT have been involved in defining formal languages with which protocols and services can be specified. The development of formal description techniques (FDT) has benefited from earlier efforts of protocol modelling by means of finite-state machines[3] (FSM), Petri nets[4], and a calculus of communicating systems[5].

There is a FDT of interest to us in this paper: Estelle[6]. Estelle is based on an extended finite-state machine (EFSM) model. Many of the OSI protocols have already been formally specified in this FDT and formal specification of others are being developed (examples are given in References 7and 8). Protocols and services of OSI are complex. They have to be throughly tested before being incorporated in the system. Another related activity is testing for conformance to the protocol specifications. 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[9]. We are interested in test design for distributed systems in general and protocols in particular. A computer-aided test design tool[10] has been introduced for the purpose of test design for conformance testing. An improved implementation of it, now nick-named **CONTEST-ESTL**, has been recently realized. Improvements are in terms of functionality, capacity and speed.

CONTEST-ESTL is an implementation of a functional formal specification based test design methodology[11]. The methodology derives two graphical models from a formal specification. The first model is called the control flow graph and is used to define interaction sequences to be applied/observed during a test. The second model is named the data flow graph and it decomposes the data flow into various simpler flows called functional blocks. Test sequences obtained from this methodology have been experimented on transport protocol implementations[11]. This paper reports unparameterized test generation for ISO's File Transfer, Access and Management[12] (FTAM) protocol. This has been realized on an experimental basis using the test design tool CONTEST-ESTL.

In order to apply the tool to FTAM, we had to make a description of this protocol in Estelle, Section 2 provides short introductions to Estelle and FTAM. An overview of our FTAM description in Estelle is given in Section 3. Section 4 introduces CONTEST-ESTL (whose components are explained in more details in References 13, 14and 15) and discusses test generation for FTAM. Finally, Section 5 contains authors' conclusions.

† This research was supported by the Department of Communications, Ottawa, Canada, with contract number DSS 10ER.36100-7-0157 and in part by the National Science and Engineering Research Council of Canada.

CH2702-9/89/0000/0251$01.00 © 1989 IEEE

2. Estelle and FTAM

2.1 Estelle

Estelle describes a system (protocol) as a collection of modules. Each module is a finite-state machine capable of having memory, i.e., a EFSM. The modules of an entity can communicate to one another as well as with the environment via FIFO channels. Service primitives (exchanged with bottom and upper layer entities), and internal interactions (with other modules) are communicated in the channels. Protocol data units (PDUs, messages exchanged between protocol entities) are introduced as parameters of service primitives. Decomposition of an entity into modules is usually functional: a module for timer management; a mapping-module to map PDUs into interactions with the environment, i.e. service primitives; an abstract-protocol-module for handling service primitives and forming PDUs, etc.

The language of Estelle is based on Pascal with extensions to facilitate protocol specification. To save space we only mention the constructs related to EFSM transitions. The **from** and **to** clauses define respectively the initial and final state of a transition. The arrival of an input interaction is expressed using a **when** clause. Transitions with no *when* clause are called spontaneous. They are used to describe nondeterminism (e.g., internal decisions of the entity). The conditions for firing the transition are described in a **provided** clause which contains a boolean expression on interaction primitive parameters and (context) variables of the module. Finally, the actions of the transition are contained inside a **begin-end** block. Actions can be assignments to context variables, calls to internal procedures, Pascal conditional statements and **output** statements.

2.2 FTAM

FTAM is a layer seven protocol for creating the means by which a client user application process can access and manage the file store of a remote open system (OS). FTAM also defines a common model, for files and their attributes, called the **Virtual File Store** (VFS). This model permits transfer, access and management of files between systems of different manufactures as well as of different level of sophistication.

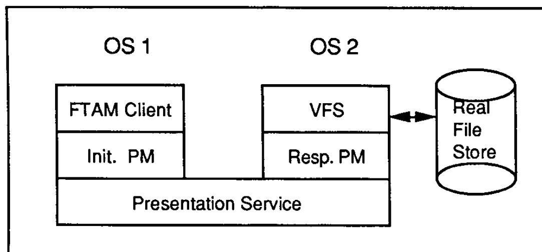

Fig. 1 Architecture of the FTAM Service

Figure 1 depicts the architecture of the FTAM service. There are two asymmetrical FTAM protocol entities called the initiator and the responder protocol machines (PMs) respectively located in OSs one and two. The FTAM client is an entity, located in OS one, that wishes to transfer files between the OSs or access and manage files of the VFS located in OS two. An implementation dependent software must be inserted between the VFS and the real file store, in order to perform the necessary mapping between the virtual and the real world.

The user operates with the FTAM service to create a series of embedded phases in which the desired file activity can take place. The main phases are:

1) **FTAM Regime Establishment** - the initiator and the responder establish each other's identity.

2) **File Selection** - identification of the needed file along with required access types.

3) **Data Access** - operations, such as read and write, are performed on the selected file.

ISO has recently defined an application layer model[16] in which fits the actual description of FTAM. This model is shown in Figure 2. A local user application process is called the **Application Process** (AP). The application protocols are contained inside the **Application Entity** (AE). The AE contains a set of **Application Service Element** (ASE). An ASE is an implementation of a given information processing service such as FTAM. The AE chooses the ASEs needed to provide the type of communication required by the AP. The **Association Control Service Element** (ACSE) is a special kind of ASE that is used by other ASEs to open and release presentation layer connections between associated AEs. A **Single Association Control Function** (SACF) along with a set of ASEs constitute a **Single Association Object** (SAO). Simultaneous associations are controled by the **Multiple Association Control Function** (MACF). A AE is identified by an **Application Title**. A AE is attached to a presentation service access point and its title is bound to the presentation address.

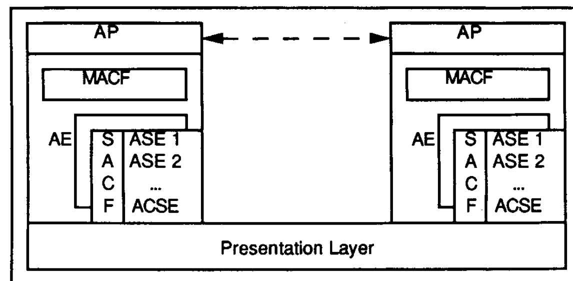

Fig. 2 Application Layer Model

3. FTAM Description in Estelle

Our description in Estelle of FTAM has been derived from the ISO document[12] which describes the protocol in natural language. We tried to design a description that would fit into the actual ISO application layer model. The description defines a FTAM ASE module that uses the facilities provided by the ACSE module to open and release presentation layer connections.

The description consists of two major parts. The first part describes the data types and the structures used by FTAM, i.e. parameters, PDUs and service primitives. The module representing the behavior of the FTAM ASE is defined in the second part. The behavior is described in terms of the interactions with the FTAM users, the presentation service and the ACSE module. The following two subsections discuss the description in Estelle of the FTAM data structures and the FTAM behavior.

3.1 Translation of FTAM Data Structures into ASN.1

Estelle borrowed from Pascal the notation for data type and structure definitions. The Pascal notation can express simple data types like boolean, char, integer and real. Ordinal types, such as enumerated types and sub-ranges, can also be described. Pascal provides as well the means for structuring data into arrays, records and sets.

ISO definitions of ACSE and FTAM make use of a standard notation named ASN.1[17]. ASN.1 has been specifically developed for application protocols data type definitions. For expressing FTAM in Estelle, it is necessary to translate ASN.1 definitions of PDUs and service primitives to Pascal. In the following we propose a mapping, typical (not all) ASN.1 data types used for defining the PDUs will be considered. The idea is to reflect as much as possible the data structures defined in the ASN.1 language. However, some aspects of the data types are dropped simply because Pascal is not as rich as ASN.1 in data description facilities. For instance *optional* declared values in ASN.1 structured types cannot be omitted in the corresponding Pascal type.

Example 1: An ASN.1 **integer** is a simple type with positive and negative whole numbers as distinguished values. Let us consider the ASN.1 definition of the *Service Level* type.

```
Service-Level ::= Integer {
 reliable(0),
 user-correctable(1) }
```

In Estelle the distinguished values are defined as Pascal constants whereas *Service-Level* is defined of integer type.

```
const
 reliable=0;
 usr_corr=1;
type
 ServLev=Integer;
```

Example 2: ASN.1 **Bitstrings** are ordered sequences of zero or more bits. The *Functional-Units* type is such a sequence.

```
Functional-Units ::= BITSTRING {
 read(0),
 write(1),
 file-access(2),
 limited-file-management(3),
 enhanced-file-management(4),
 grouping(5),
 recovery(6),
 restart-data-transfer(7) }
```

The numbers inside parentheses indicate the distinguished bits of the bitstring. ASN.1 bitstrings are translated to Pascal arrays of booleans.

```
type
 Units=(read,write,fil_acc,lim_fil_man,
       enh_fil_man,group,rec,res_dat_transf);
 FunctUn=array [Units] of boolean;
```

Example 3: An ASN.1 **sequence** is a new type consisting of an ordered list of existing types. It structures a sequence of items which order is relevant. Let us consider the *F-Select-request* type.

```
F-SELECT-request ::= SEQUENCE {
 attributes Attributes,
 requested-access Access-Request,
 access-passwords Access-Passwords,
 concurrency-control Concurrency-Control,
 commitment-control Commitment-Control,
 account Account }
```

A *sequence* type is expressed in Estelle by a Pascal record type.

```
SELrqTyp=record
 attr:Attributes;
 req_acc:AccReq;
 acc_passw:AccPassw;
 conc_contr:ConcContr;
 comm_contr:CommContr;
 acc:Account;
end;
```

Example 4: The ASN.1 type **sequence of** defines, by referencing a single existing type, a new type as an ordered list of zero, one or more values of this existing type. The number of values in a sequence of is unlimited. The type *Constraints-Sets* is a *sequence of*.

```
constraints-sets ::= SEQUENCE OF
   Constraint-Set-Name
```

A *sequence of* is mapped to an array type item and an integer type item structured inside a Pascal record. The array type item stores the sequence of values whereas the

actual number of values in the sequence is memorized in the integer type item. An implementation dependent maximum sequence length is assumed.

```
const
 maxCons=any integer;
type
 const_sets=record
  seq:array[1..maxCons] of ConsSetNam;
  num:integer;
 end;
```

Given a *Sequence of* structure named x, we add to the constant and record structure definitions the functions $x_Append, x_First, x_Last, x_Pred$ and x_Succ. Let s be an instance of a sequence type x, and v a value from the sequence elements type. The item $s.num$ is initially set to the value 0. The procedure x_Append adds v at the end of s with the following operations: $s.num := s.num + 1; s.seq[s.num] := v$. If $s.num > 0$ then $x_First(s)$ returns $s.seq[1]$ and $x_Last(s)$ returns $s.seq[s.num]$. If $1 < i \leq s.num$ $(0 < i < s.num)$ then x_Pred(i) is $s.seq[i-1]$ ($x_Succ(i)$ is $s.seq[i+1]$).

The ASN.1 **set** structured type is like *sequence* type, but the items order is unrelevant. A translation similar to the one defined for *sequence of* is adopted. Although notions of *first, last, successor* and *predecessor* do not exist for *sets*.

Example 5: A ASN.1 **set of** is a structured type defined as a unordered list of zero, one or more values of an existing type. *Insert-Values* is a *set of* type.

```
insert-values ::= SET OF Access-Control-Element
```

In Pascal, types of set elements are restricted to simple types. Therefore Pascal sets are not general enough for representing ASN.1 set types. Consequently we use record structures similar to those used for *sequence of*. Moreover to capture the unordered aspect as well as the fact that set elements must be distinct we add operations. A *set* type definition named x which elements are from domain y includes the following group of Pascal functions and procedures.

```
function x_Empty(s:x):boolean;
 primitive;
function x_Member(v:y; s:x):boolean;
 primitive;
function x_Included(s1,s2:x):boolean;
 primitive;
procedure x_Inter(s1,s2:x; var s3:x);
 primitive;
procedure x_Union(s1,s2:x; var s3:x);
 primitive;
procedure x_Add(v:y; var s:x);
 primitive;
```

These operations have the semantic suggested by their name. Their realization is left to the implementor and this is indicated by the keyword *primitive* in place of expanded bodies. Access to instances of a *set* data type must be realized via those operations.

Example 6: The **choice** data type defines a new type from a given list of distinct types. A value is chosen among them. An example of a choice data type is the *FTAM-Regime-PDU* structure.

```
FTAM-Regime-PDU ::= CHOICE {
 F-INITIALIZE-request
 F-INITIALIZE-response
 F-TERMINATE-request
 F-TERMINATE-response }
```

Choice structures are translated to Pascal variant records.

```
type
 FTAMRegPDUTyp=(INIrq,INIrp,TERrq,TERrp);
 FTAM_Reg_PDU=record
  case tag:FTAMRegPDUTyp of
   INIrq:(v0:INIrqTyp);
   INIrp:(v1:INIrpTyp);
   TERrq:(v2:TERrqTyp);
   TERrp:(v3:TERrpTyp);
end;
```

3.2 FTAM Behavior Description

The EFSM of FTAM has been obtained by first translating in Estelle the transition tables supplied by ISO. The result is a EFSM skeleton where from, to, when clauses and output statements are defined for each transition. There is a bijection between FTAM service primitives and PDUs. The PDU and the primitive associated to each other have the same name. The PDU is in general a sub-structure of the service primitive. Pascal procedures are defined for mapping service primitives into PDUs. For instance, the following procedure maps the parameters of a *f-initialize request* service primitive into a *initialize request* PDU structure. A call to this procedure will appear in the appropriate transition.

```
procedure INIrqPDU(
 { inputs }
 pi:ProtVers;pcm:BOOLEAN;le:ServLev;
 cl:ServClass;un:FunctUn;ag:AttGroups;
 ro:RollbAv;co:ContTypList;ii:UsId;
 ac:Account;fp:Password;cw:INTEGER;
 { ouput }
 var Val:FTAM_PDU);
begin
 val.tag:=INIrq;
 with Val.v0 do begin
  prot_id:=pi;pres_cont_man:=pcm;
  level:=le;class:=cl;units:=un;
  att_groups:=ag;rollback:=ro;
  contents:=co;in_id:=ii;acc:=ac;
  fs_passw:=fp;checkp_wind:=cw;
 end;
end;
```

The standard also indicates the presentation or ACSE service primitive that will contain each type of FTAM PDU. For example a *initialize request* PDU appears in a ACSE *a-*

associate request primitive whereas a *select request* PDU is contained in a presentation *p-data request* primitive. The parameter values of FTAM regime establishment primitives will be assigned to or received from ACSE primitives. The dependencies between FTAM primitive parameters and ACSE primitive parameters are provided in the standard and reflected in the Estelle specification by appropriate assignment statements. The predicate of the provided clauses are obtained from the text associated to transition tables. For the purposes of this experience we have considered the *Basic FTAM* protocol with functional units *kernel, read* and *write.* In units read and write checkpointing is omitted. The specification has three external interaction points, namely, with the user, ACSE and the presentation service. Some elements of the specification are given in Appendix A.

4. FTAM Test Design Using CONTEST-ESTL

Actions of FTAM are structured into functional units. The unit kernel contains actions related to FTAM regime establishment/ termination and file selection/ de-selection. Functional units *read* and *write* both consist of file opening/ closing, send/ receive data and end of data transfer actions. Functional unit read (write) also contains read (write) actions. Functional units are the elements of a first natural decomposition of the protocol for the purposes of testing.

The functional unit kernel is totally independent of units read and write. However, units read and write, although independent of each other, are dependent of unit kernel since before they can be activated a FTAM regime has to be established and a file has to be selected. In order to test units read and write, the test designer will have to drive the IUT into an appropriate context, i.e. in the *file selected* state.

The transition part can easily be decomposed into functional units according to the service primitives. The ISO standard relates to each functional unit the possible actions (i.e. service primitives and PDUs). To obtain the description of a given functional unit, we keep only the transitions in which the desired actions appear. We separated the FTAM description into functional units, each functional unit is considered individually.

CONTEST-ESTL takes an Estelle specification and semi-automatically produces unparameterized test suites. The first step applied to an Estelle specification is normalization. In a normalized specification, we identify two types of flow: control flow, modelling major state changes defined by the transitions; and data flow, modelling flow from input primitive parameters (service primitives and/or PDUs) to context variables and from context variables to output primitive parameters. Test sequence generation is realized for every **protocol function** that is obtained by data flow decomposition.

4.1 General Considerations

This section contains observations about testing the FTAM application protocol. Those observations originally emerged from the work of Bochmann, Deslauriers and Besette[18]. From their experience characteristics of the environment in which FTAM will be tested came up. Further details can be found in their paper. They also brought out observations on test design for FTAM. Our experience validated the four following observations.

1) The order of primitives and PDUs, in a sequence of interactions, is generally independent of parameter values. It implies that test sequences obtained from the control graph will not contain unpracticable paths. Parameters that make exception to this are those that appear in provided clause predicates. However, their value can be easily chosen to make fireable the transitions.

2) Though rules for valid sequences of interactions are simple, rules of admitted parameter values at a given point in the execution are usually very complex. It makes the non-automated part of the task more complex for the test designer, since (s)he is left with computing appropriate values of subtour interaction parameters. An example of this is the parameter of type *Attributes* of a *F-select request* PDU.

3) As already mentioned, there is a one-to-one mapping between the set of primitives and the set of PDUs. Moreover, in general values from a given primitive flow directly, without internal transformations, to the associated PDU, and vice versa. In some cases, values of output parameters are determined by internal constants. The data flow graph exhibits only simple relations between its elements. This point implies that once appropriate values are selected for input interactions, values in output interactions are easily obtained. It also implies that in a given data flow graph block, there is no incoming arcs from other blocks. Consequently there is no (data flow) dependency between data flow functions obtained by merging the blocks.

4) The problem of non-synchronizable test suites has been raised for certain protocols[19]. This problem does not arise in the case of FTAM because of the correspondence between primitives and PDUs.

4.2 Test Generation for Kernel Unit

The intent of the normalization step is to identify the control paths. They will be expressed as distinct transitions in the resulting normalized specification. The normalization is done in three steps. First procedure and function calls are symbolically substituted by the sequences of statements they represent. Next selective (if and case statements) and repetitive (for, repeat and while statements) control structures are eliminated. The first two normalized transitions of FTAM initiator kernel unit are given in appendix B. We also produced normalized specifications for functional units read

and write. In the sequel we will discuss test generation for kernel unit.

4.2.1 Control Graphs

The control flow analysis step takes as input a normalized specification and generates the control graph which corresponds to the FSM of the system (or protocol entity). This step produces a graphical representation of the FSM, Figure 3, which can help the test designer to visualize paths of execution. The nodes represent the states and the arcs indicate by their numbers the corresponding normalized transitions.

From an internal representation of the control graph, test sequences composed of transition subtours are generated. A subtour is a sub-sequence of interactions that starts and ends in the initial state. The initial state of functional unit kernel is *closed*. For kernel we obtained a total of 16 different subtours. Three of these are given in Table 1. Each transition takes one line in which its *from* state, input, output and the normalized transition number are listed in order. The *to* state of a transition appears as the initial state of the next transition.

State	Input	Output	Transition
closed	FINIrq	AASSrq_INIrq	1
initialize_pd	AASScf_INIrp	FINIcf	3
initialized	FTERrq	ARELrq_TERrq	5
terminate_pd	ARELcf_TERrp	FTERcf	6
closed	FINIrq	AASSrq_INIrq	1
initialize_pd	AASScf_INIrp	FINIcf	4
closed	FINIrq	AASSrq_INIrq	1
initialize_pd	AASScf_INIrp	FINIcf_fail	2

Table 1 Subtours

4.2.2 Data Flow Graphs

The data flow analysis step takes also as input a normalized specification and generates the data flow graph and partitions it. The graph represents the flow of values from input interaction parameters to context variables (internal variables of the protocol), and from context variables to output interaction parameters. Internal data operations (actions of the normalized transitions) of the protocol are also represented. In this graphical representation, inputs are placed in the upper part of the graph, outputs in the bottom and all others (operations, variables, etc.) are place in the middle. Directional arcs indicate dependencies between elements of the graph and they are labelled by the transition numbers in which the corresponding statements appear. The data flow graph is algorithmically partitioned into blocks. A block generally represents the data flow over a single context variable. In the case of FTAM we obtained a large number of simple blocks, generally consisting of a single input parameter interconnected by an arc to single output parameter. A sample is given in Figure 4. Straight vertical lines represent block-frontiers.

4.2.3 Block Merging

Usually a very large number of blocks is obtained and the data flow of one protocol function will be represented by several blocks. With the help of a protocol test designer, data flow representations for protocol functions are obtained by merging blocks together. Functions can be identified as data transfer for sending or receiving, flow control, etc. The test designer is guided by a number of rules for merging the blocks. The tool provides the facilities for modifying the graphical representation in order to reflect the decisions made by the test designer. Six rules for merging are provided in Reference 11. We applied those rules to the data flow graph of functional units kernel, read and write. Statistics are given in Table 2. It is important to note that the same analysis realized by a different test designer may produce different data flow functions because of different possible subjective interpretations and applications of the rules. Block merging of the FTAM data flow graph is further discussed in Reference 20.

4.2.4 Test Generation

Next is the test generation step. This step uses the control and data flow graphs to generate the transition subtours to effectively test every function. Most of these functions can be tested independently of each other.

Unit	Number of Functions
kernel	14
read	12
write	12

Table 2 Result of Data Flow Analysis

The aim of test design using the tool is to obtain test sequences that guarantee coverage of each arc in the data flow graph of each function. For this purpose a test sequence generation module produces, given as input the control flow graph and the result of functional analysis, subtours that cover the arcs of each data flow function. It is the test designer responsibility to compute appropriate values for interaction parameters in the subtours. In general values of output parameters will be determined from values of input parameters, internal variables and operations. Subtours that cover a given block will be applied several times to the implementation under test, in order to realize partial or total parameter domains enumeration and to try different combinations of values.

5. Conclusion

We presented an experience that consists in obtaining test suite skeletons for FTAM using the automated test de-

sign tool CONTEST-ESTL. We had to produce a formal description in Estelle of the protocol using the ISO standard. This step was made easier by the fact that the standard provides well described transition tables.

Improvements over the previous tool version (CAD-PT) are:

i) support of a larger Estelle language subset,

ii) speed increase (some phases of processing are now implemented in C instead of Prolog), and

iii) integration of a test sequence generation module with the data flow graph editor (test sequences are now automatically produced for each data flow function).

There is a need to investigate the integration of CONTEST-ESTL to TTCN based tools and to further explore test design in the ASN.1 context.

References

[1] G. Holzmann, *Automated Protocol Validation in Argos: Assertion Proving and Scatter Searching*, IEEE Trans. on Soft. Eng., Vol. SE-13, No. 6, June 1987.

[2] L. Logrippo, A. Obaid, J.P. Briand and M.C. Fehri, *An Interpreter for Lotos, a Specification Language for Distributed Systems*, to appear in Software Practice and Experience.

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

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

[5] R. Milner, *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag, 1980.

[6] ISO/TC 97/SC 21, *Estelle: A Formal Description Technique Based on an Extended State Transition Model*, DIS 9074, 1987.

[7] P.D.Amer, F. Ceceli and G. Juanole, *Formal Specification of ISO Virtual Terminal in Estelle*, Proceedings of IEEE INFOCOM'88, 1988.

[8] ISO/TC 97/SC 16, *Example of a Transport Protocol Specification*, November 1982.

[9] ISO/ TC 97/ SC21/ WG1, *Conformance Testing Methodology and Framework*, DP 9646, 1987.

[10] M. Barbeau and B. Sarikaya, *A Computer-aided Design Tool for Protocol Testing*, Proc. of INFOCOM'88, New Orleans, March 1988.

[11] B. Sarikaya, G.v. Bochmann and E. Cerny, *A Test Design Methodology for Protocol Testing*, IEEE Trans. on Software Eng., pp. 531-540, may 1987.

[12] ISO/TC 97/SC 21, *File Transfer, Access and Management*, DIS 8571, 1987.

[13] S. Eswara, V. Koukoulidis, M. Barbeau and B. Sarikaya, *CONTEST-ESTL User's Manual*, Concordia University, April 1988.

[14] M. Barbeau, *Prototype d'un Systme d'Aide à la Conception de Test de Protocoles*, M.Sc. Thesis, Université de Montréal, February 1987.

[15] S. Eswara, V. Koukoulidis, M. Barbeau and B. Sarikaya, *CONTEST-ESTL: A Computer Aided Design Tool for Protocol Testing*, research report from Concordia University, Montreal, 1988, (submited for publication).

[16] ISO 9545, *Application Layer Structure.*

[17] ISO/TC 97/SC 6, *Profile of Abstract Syntax Notation One*, IS 8824, 1987.

[18] G. V. Bochmann, M. Deslauriers and S. Bessette, *Application Layer Testing and ASN.1 Support Tool*, Proc. of Globecom'86.

[19] B. Sarikaya and G. V. Bochmann, *Synchronization and Specification Issues in Protocol Testing*, IEEE Trans. on Comm., COM-82, No. 4, April 1984.

[20] B. Sarikaya, V. Koukoulidis, S. Eswara and M. Barbeau, *Analysis and Testing of Application Layer Protocols with an Application to FTAM*, Submitted for publication, Available as a research report from Concordia University, 1988.

Appendix A: Transitions of Initiator Side

A.1 Data Types and Structures

```
{F T A M  P D U 's}
{ INITIALIZE-request }
INIrqTyp=record
 prot_id:ProtVers;
 pres_cont_man:BOOLEAN;
 level:ServLev;
 class:ServClass;
 units:FunctUn;
 att_groups:AttGroups;
 rollback:RollbAv;
```

```
 contents:ContTypList;
 in_id:UsId;
 acc:Account;
 fs_passw:Password;
 checkp_wind:INTEGER;
end;
{ INITIALIZE-response }
INIrpTyp=record
 stat_res:StatRes;
 act_res:ActRes;
 prot_id:ProtVers;
 pres_cont_man:BOOLEAN;
 units:FunctUn;
 attr_groups:AttGroups;
 rollback:RollbAv;
 contents:ContTypList;
 diag:Diagnostic;
 checkp_wind:INTEGER;
end;
{ TERMINATE-response }
TERrpTyp=record
 charg:Charging;
end;
{ FTAM-PDU }
FTAM_PDU_typ=(INIrq,INIrp,TERrq,TERrp);
FTAM_PDU=record
 case tag:FTAM_PDU_typ of
  INIrq:( v0:INIrqTyp);
  INIrp:( v1:INIrpTyp);
  TERrq:(); {no elements defined, shall be empty}
  TERrp:( v3:TERrpTyp);
 end;
{---- FTAM Service Primitives ----}
channel F_access_point(User,Provider);
 by User:
 { F-INITIALIZE request }
 FINIrq(
  CalledAET:AppTitl; { Called AE Title }
  CallingAET:AppTitl; { Calling AE Title }
  prot_id:ProtVers;
  pres_cont_man:BOOLEAN;
  level:ServLev;
  class:ServClass;
  units:FunctUn;
  att_groups:AttGroups;
  rollback:RollbAv;
  contents:ContTypList;
  in_id:UsId;
  acc:Account;
  fs_passw:Password;
  checkp_wind:INTEGER);
 { F-TERMINATE request }
 FTERrq;
 by Provider:
 { F-INITIALIZE confirm }
 FINIcf(
  stat_res:StatRes;
  act_res:ActRes;
  prot_id:ProtVers;
  pres_cont_man:BOOLEAN;
  units:FunctUn;
  attr_groups:AttGroups;
  rollback:RollbAv;
  contents:ContTypList;
  diag:Diagnostic;
  checkp_wind:INTEGER);
 { F-INITIALIZE confirm fail }
 FINIcf_fail;
 { F-TERMINATE confirm }
 FTERcf(
  charg:Charging);
{---- ACSE Service Primitives ----}
channel A_access_point(User,Provider);
 by User:
 { A-ASSOCIATE-request }
 AASSrq(
  protVers:integer; { Protocol Version }
  CalledAET:AppTitl; { Called AE Title }
  CallingAET:AppTitl; { CAlling AE Title }
  app_cont:AppContNam; {Application Context Name}
  us_inf:FTAM_PDU); { User Information }
 {A-RELEASE-request }
 ARELrq(
  reas:RelReqReas; { Release Request Reason }
  us_inf:FTAM_PDU); { User Information }
 by Provider:
 { A-ASSOCIATE-confirm }
 AASScf(
  protVers:integer; { Protocol Version }
  res:AssRes; { Result }
  respAET:AppTitl; { Responding AE Title }
  app_cont:AppContNam; {Application Context Name}
  us_inf:FTAM_PDU); { User Information }
 { A-RELEASE-confirm }
 ARELcf(
  reas:RelRespReas; { Release Response Reason }
  us_inf:FTAM_PDU); { User Information }
```

A.2 Transitions

```
trans
(* 1 *)
when F.FINIrq
{ F-INITIALIZE primitive acceptable }
provided (level=usr_corr)and(class=transf_class)
from CLOSED to INITIALIZE_PD
begin
 INIrqPDU({ in } prot_id, pres_cont_man, level,
    class, units, att_groups, rollback, contents,
    in_id, acc, fs_passw, checkp_wind,
    { out } PDU);
```

```
 output A.AASSrq(0 { Version1 }, CalledAET,
    CallingAET,ISO_8571_FTAM,PDU);
end;
(* 2 *)
{ F-INITIALIZE primitive not acceptable }
provided (level<>usr_corr) or
         (class<>transf_class)
from CLOSED to CLOSED
begin
 output F.FINIcf_fail;
end;
(* 3 *)
trans
when A.AASScf
{ ACSE conf. prim. indicates success of the op. }
provided ((protVers=0) and (res=ass_acc) and
   (us_inf.tag=INIrp) and
   (us_inf.v1.stat_res=succ_res) and
   (us_inf.v1.act_res=succ_act))
from INITIALIZE_PD to INITIALIZED
begin
 with us_inf.v1 do
  output F.FINIcf(stat_res, act_res, prot_id,
     pres_cont_man, units, attr_groups,
     rollback, contents, diag, checkpnt_wind);
 end;
(* 4 *)
{ ACSE conf. prim. indicates failure of the op. }
provided (us_inf.tag=INIrp) and
   ((protVers<>0) or (res<>ass_acc) or
   (us_inf.v1.stat_res<>succ_res) or
   (us_inf.v1.act_res<>succ_act))
from INITIALIZE_PD to CLOSED
begin
 with us_inf.v1 do
  output F.FINIcf(failure, act_res, prot_id,
     pres_cont_man, units, attr_groups,
     rollback, contents, diag, checkpnt_wind);
end;
(* 5 *)
{ request for termination of FTAM Regime }
trans
when F.FTERrq
from INITIALIZED to TERMINATE_PD
begin
 TERrqPDU({ out } PDU);
 output A.ARELrq(norm, PDU);
end;
(* 6 *)
trans
when A.ARELcf
provided (us_inf.tag=TERrp)
from TERMINATE_PD to CLOSED
begin
 with us_inf.v3 do
  output F.FTERcf(charg);
end;
```

Appendix B: Normalized Transitions

```
trans
(* 1 *)
when f. finirq
provided (level=usr_corr)and(class=transf_class)
from closed to initialize_pd
begin
 pdu_inirq.v0.prot_id:=prot_id;
 pdu_inirq.v0.pres_cont_man:=pres_cont_man;
 pdu_inirq.v0.level:=level;
 pdu_inirq.v0.class:=class;
 pdu_inirq.v0.units:=units;
 pdu_inirq.v0.att_groups:=att_groups;
 pdu_inirq.v0.rollback:=rollback;
 pdu_inirq.v0.contents:=contents;
 pdu_inirq.v0.in_id:=in_id;
 pdu_inirq.v0.acc:=acc;
 pdu_inirq.v0.fs_passw:=fs_passw;
 pdu_inirq.v0.checkp_wind:=checkp_wind;
 output a.aassrq_inirq(0,calledaet,callingaet,
     iso_8571_ftam,pdu_inirq)
end;

trans
(* 2 *)
when f.finirq
provided (level<> usr_corr)or(class<>transf_class)
from closed to closed
begin
 output f. finicf_fail
end;
```

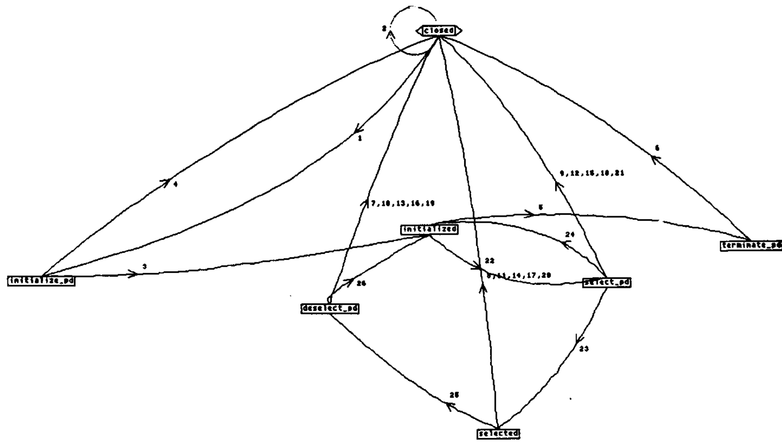

Fig. 3 Control Graph of Kernel Unit

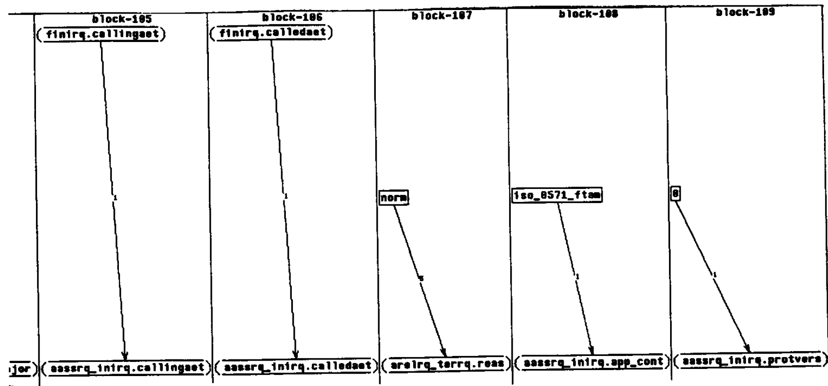

Fig. 4 Data Flow Graph