UNIVERSITÉ DE MONTRÉAL

Formal Description Techniques for Protocols

Final Report for DOC contract No. CR-CS-1982-0033

by Gregor V. Bochmann

Département d'informatique et
de recherche opérationnelle

Université de Montréal

March 1982

DÉPARTEMENT D'INFORMATIQUE

ET DE RECHERCHE OPÉRATIONNELLE

Faculté des arts et des sciences
Université de Montréal
C.P. 6128, Succursale "A"
Montréal, P.Q.
H3C 3J7

Formal Description Techniques for Protocols

Final Report for DOC contract No. CR-CS-1982-0033

by Gregor V. Bochmann

Département d'informatique et
de recherche opérationnelle

Université de Montréal

March 1982

TABLE OF CONTENT

Formal Description Techniques for Protocols

Final Report for DOC contract No. CR-CS-1982-0033

by Gregor V. Bochmann

Département d'informatique et
de recherche opérationnelle
Université de Montréal

March 1982

1. Introduction

The importance of formal description techniques (FDT) for the design and documentation of computer communication protocols and services has been acknowledged by the ISO/TC97 Subcommittee on Open System Interworking (SC16) through the establishment of a Rapporteur's Group on FDT within Working Group 1. A Rapporteurs group for studying this question has also been established within the Study Group VII of the CCITT. The work under this contract was principally aimed at contibuting to the work of these study groups, and has resulted in a number of contributions to the Canadian and international standard committees working on these questions. It is a continuation of previous work of this author in the area.

During its first meeting in Chicago (January 1980) the ISO Special Rapporteur's Group on FDT established a program of work which foresees the selection of one or more FDT's for use within SC16. The purpose of these FDT's is to provide a means for precisely specifying protocols and services of the different layers of Open Systems. These formal specifications should be unambiguous and helpful for the implementation and for the verification of the protocols. Contibutions were asked for on proposed FDT's and their application to the test cases of the Transport protocol and service.

2. Overall view of the contract activity

It is noted that the "statement of work" of the contract foresees (under point 1) the development of a formal specification for the Teletex Session and Document protocols. With the agreement of the scientific authority of the contract, this work was replaced by the activities described in section 2.1 below, which appeared to be of higher priority.

2.1 Standardization activities

Within the framework of this contract, the author was a delegate at two meetings of the CCITT Rapporteurs Group on Question VII/39 in Ottawa and Melbourne. Since the contract did not provide sufficient travel funding for the meeting in Melbourne, the travel expenses for this meeting were paid for

through a DOC contract with Dendronic Decisions Ltd. The author was also delegate at a ISO TC97/SC16/WG1 meeting on FDT in Washington and at meetings of the Subgroups A and B of the ad hoc group on FDT. The author is editor for the working papers of both subgroups (see annexes 1 and 2) and chairman of Subgroup A. The work under this and a previous contract had a strong influence on the development of the extended state transition FDT of Subgroup B of the ISO TC97/SC16/WG1 ad hoc group on FDT. Much of the effort during this contract period was aimed at bridging the gap between this FDT and the FDT developments in CCITT. The author represented the ISO ad hoc group on FDT at the CCITT meeting in Ottawa, and SC16/WG1 at the CCITT FDT meeting in Melbourne. He was also a Canadian delegate at the Melbourne meeting and presented several contributions (see annexes 7 through 11). The contributions FDT 33 and 34 (CAN COM 39 and 40, annexes 7 and 8) were discussed in detail and supported by the Canadian ad hoc group on questions VII/5, 27 and 39 and NSG VII.

We think that our contributions have advanced the development of FDT's for the specification of Open Systems protocols and services. However, further work is required for obtaining a FDT which is accepted by both ISO and CCITT.

2.2. Translator for formal specifications into implementations

A program was developed that translates formal protocol specifications given in terms of a preliminary FDT syntax (which

was submitted to the ISO TC97/SC16/WG1 ad hoc group on FDT in January 1981) into program fragments written in Pascal, which can be combined with support packages to from a complete Pascal program implementing the protocol. This translator program is documented in Annex 13. It has been tried out with some relatively simple example protocols. In addition, it was used within a course project (fall 1981) for the specification of the Teletex Document protocol, and its translation into a Pascal implementation.

3. Proposal for future work

We think that a natural continuation of the work performed under this contract would be a continuing support of the ISO and CCITT discussions on FDT's. We think that Canadian input would be much welcome in view of its past participation.

In order to increase the usefulness of the proposed FDT, the following additional reseach activities are proposed:

a) To apply the method to several protocols and services at levels higher than the transport layer in order to test its applicability in all areas of OSI.

b) To develop a protocol implementation tool which would partly automate the production of a protocol implementation from the formal specification of the protocol. (It is noted that the program described in section 2.2 and Annex 13 is only a initial attempt at approaching this problem; it should be

adapted to the final syntax of the FDT, and could be improved as far as the translation process is concerned).

c) To develop a testing tool that would be helpful to test a protocol implementation for conformance with the protocol specification. Such a tool could be useful for the certification of communication software and systems.

d) To develop a protocol simulation tool that would make simulations of communication subsystems based on the formal specifications of the protocols to be used in the system. Such a tool would be useful during the development of protocol standards for analyzing the behavior of the protocol, finding eventual malfunctions (deadlocks, etc.), and determining the efficiency of its operation.

4. More detailed account of the standardization activities

The following subsections list the different meetings that were attended, and the contributions prepared for these meetings, as well as other activities related to these standardization meetings.

4.1. Work within ISO TC97/SC16/WG1 ad hoc group on FDT

4.1.1. Meeting in Washington, DC, September 21 - 25, 1982.

Submitted contributions:

WASH-9 : "Formal specification of a Transport service" (G. Bochmann, E. Cerny, and C. Lacaille) (see Annex 3)

WASH-10: "Comments on a possible compromise on the syntax for extended state transition descriptions" (Canada) (see Annex 4)

WASH-11: "An extended state transition model as a FDT" (G. Bochmann)

These contributions were discussed during the Subgroup B meeting in Washington.

4.1.2. Work within Subgroup A

The author is chairman of Subgroup A on "Architecture". The working document WASH-1 was elaborated by correspondence during summer 1981, and discussed during the Washington meeting. The revision of the document was edited by the author.

A Subgroup A meeting was held in Ispra (Italy) in November 1981 (see Annex 1 for the minutes). The resulting working document (see Annex 1) was again revised in the recent FDT meeting in Enschede (Holland), April 1982.

4.1.3. Work within Subgroup B

The author participated in the work of Subgroup B on "Extended State Transition Model FDT" as contributor, and as the editor of the working document.

The author participated in the Subgroup B meeting held in Boston, December 1981. The Subgroup B working document resulting from this meeting is included as Annex 2.

4.2. Work within the CCITT Rapporteurs group on Question VII/39 (FDT)

4.2.1. Rapporteurs meeting in Ottawa, September 19 - 27, 1981

Submitted contributions:

FDT-2 : "Formal specification of a Transport protocol" (Canada)

FDT-21: "Formal specification of a Transport service" (G. Bochmann et al.) (see Annex 3)

FDT-27: "Time sequence diagrams as FDT" (G.Bochmann) (see Annex 5)

FDT-28: "Syntax for linear form FDT: Comparison of ISO proposal and SDL-PR" (G. Bochmann) (see Annex 6)

4.2.2. Rapporteurs meeting in Melbourne, March 9 - 16, 1982

Submitted contributions (see Annexes 7 through 10):

FDT 33 (D 205, CAN COM 39): "Proposal for a programme like FDT" (see Annex 7)

FDT 34 (D 206, CAN COM 40): "Proposal on different forms of FDT" (see Annex 8)

FDT 47 (D 207, CAN COM 42): "Translation of the ISO linear FDT into graphical SDL" (see Annex 9)

FDT 48 (D 129, CAN COM 43): "A method for specifying module interconnections" (see Annex 10)

FDT 49 (D 131, CAN COM 41): "Examples of Transport protocol specifications" (see Annex 11)

ANNEX 1

To : Members of ISO/TC97/SC16/WG1 ad hoc group on FDT

J. Day, chairman of ad hoc group
H. Zimmermann, chairman of WGI

cc : T. Steel, CCITT Rapporteur on Question VII/27
G. Dickson, CCITT Rapporteur on Question VII/39

From : G.V. Bochmann, chairman of Subgroup A

Re : Last meeting of FDT Subgroup A on "Architecture"

Please find enclosed the minutes of the last FDT Subgroup A meeting in Ispra. The result of this meeting is the revised working document "Concepts for describing the OSI architecture", which is enclosed. It is the desire of the Subgroup to give a wider circulation to this working document in order to get a broader feedback for its next revision.

Sincerely

G.V. Bochmann

P.S. In the spirit of collaboration between ISO and CCITT, copies are sent to the CCITT Rapporteurs who work on related problems.

Title: Minutes of the meeting of Subgroup A of the ISO/TC97/SC16/WGI ad hoc group on FDT, Ispra, November 20, 1981

From : Subgroup A

The following people attended the meeting:

A. Endrizzi	Italy
G.V. Bochmann (chairman)	Canada
F.H. Vogt	W. Germany
P.F. Linington	U.K.
J.P. Ansart	France
A. Faro (secretary)	Italy
G. Messina	Italy

The only item of work was the revision of the working paper "Concepts for describing the OSI architecture" which was distributed several weeks before the meeting in the version, edited by G.V. Bochmann based on the work during the Washington meeting in September.

Three contributions were presented:

- two papers from LeMoli (COMPUNET/CREI/80/17 and COMPUNET/CREI/80/16) concerning general comments on the Washington working paper and a proposal of entity structure.

- a technical report by Bochmann and Raynal concerning "structured specification of communication of systems" which was presented as a contribution to the topic of section 3.3 of the working paper, to be discussed at the next Subgroup A meeting.

The chairman proposed to revise the working paper page by page. The major points of discussion were the nature of module interections and interection mechanisms. However the section 4 on "Definition of service, protocol and interface specifications" was discussed in order to take into account the proposal of LeMoli. Some minor editorial changes were left to the discretion of the editor (G.V. Bochmann), who will distribute the new version of the working paper.

It was agreed among the members of Subgroup A that it is desirable to distribute the new version of the paper also outside the ad hoc group on FDT in order to get a wider feedback.

G.V. Bochmann

From: Subgroup A on "Architecture" of ISO TC97/SC16/WG1 adhoc group on FDT

Title: Concepts for describing the OSI architecture (Working Draft, Ispra, Nov. 1981)

1. Introduction

The scope for formal description techniques (FDT) in the development of OSI standards is described in "Statement of scope of the FDT group" (N). The present document may serve the following purposes:

(a) Provide a more precise model for the Guidelines (N 380 and N381), and

(b) define certain basic concepts that are used by the formal description techniques developed by subgroups B ("Extended finite state transition models") and C ("Sequencing expressions, temporal logic") of the FDT Rapporteur's Group.

The document is divided into several sections, discussing the concepts of system components (called "modules") and their specification, their interconnection and the description of an architecture, the definition of service, protocol and interface specifications, and possible subdivisions of modules for specification purposes.

2. Modules and their interactions

2.1 Module interactions

A module is a unit of description, and is specified by its interactions* with other modules within the specified system or its environment.

---

* In previous work the terms "message" and "command" have been used to denote interactions, but they are not used in OSI documents because the variety of previous uses has obscured their meaning.

Other terms have been used for this concept, such as "specification unit", "abstract machine" "system part" , "interlocutor" . , etc. An entity is a particular case of a module (see also sections 4 and 5). An abstract specification is considered; implementation issues are addressed in section 4.4.

A module is specified in terms of its interactions. For example, if the module is an N-entity, then the module interacts through N-service-primitives* (N-SPr, see section 4.1) and (N-1) - SPr's* with other local modules (respectively, the (N+1)-entity and the (N-1)-entity).

In general, three time instants are important for the execution of an interaction between two modules:

1) the moment that the interaction is initiated ("called") by one (i.e. the first) of the modules;

2) the moment that the interaction begins, i.e. the moment that the other module agrees to the execution of the interaction;

3) the moment when the interaction ends.

Each interaction carries explicit information (parameters) only in one direction: from the source module to the sink module. The source module is not necessarily the initiating module.

Depending on the model used for the interactions between modules, the distinction between all of those three instants may not be necessary. At least, instants (2) and (3) are considered as always relevant. It is noted, however, that other models may require the identification of instant (1). Moreover in situations where it is important to know which module is waiting (for example, performance considerations), it is proposed to distinguish between "source initiated" and "sink initiated" interactions.

---

* Service primitives are either expressed directly or in more detail by using interface data units (IDU).

The types of interaction considered for specification purposes are called "interaction primitives". They are abstract interactions in the sense that their implementation by the interface between the interacting modules is not specified. Examples of interaction primitives are:

- open connection to remote address with options;
- send data on connection
- send data to remote address;

where "connection" is a local connection identifier, "remote address" is the destination address, "options" is a list of facilities, "data" is an information which has to be transferred unchanged to "remote address".

In an implementation, the abstract interactions are realized through the real interactions of a real interface (see section 4.4).

The following points are important properties of interaction primitives:

(1) Each occurring interaction belongs to exactly one type; i.e. interaction primitive.

(2) Each interaction primitive is characterized by a number of parameters.
For example "remote address" and "options" parameters for the "connection establishment request" interaction.

(3) For each occurrence of an interaction, the value of each parameter of the interaction primitive is determined by the source module.

(4) The range of possible parameter values is specified for each interaction parameter e.g. by a data type definition.

(5) There are some models in which the execution of an interaction by a module may be considered as an atomic action (which excludes any other action by that same module at the same time). Parallel interactions by the same module (for example concerning different connections handled by the same module) are modelled by assuming an arbitrary order between these interactions. Alternatively, there are models that do not make these assumptions. In specifying any particular model the assumptions made about atomicity and synchronization must be clearly stated.

We assume that all primitive interactions involve a rendez-vous technique*, but it may be useful, as an aid to understanding, to introduce compound interactions consisting of a primitive interaction between the initiator and a queuing module, followed by a primitive interaction between the queuing module and a receiver.

Note: Further study is required to identify all the necessary compound interaction types and to demonstrate that they can be specified as indicated above.

In the following, when modules are components of the same entity, it is generally supposed that the receiving module sees exactly the same interaction as the initiating module: the case in which the "received" interaction is not the same as the "sent" one happens when two modules are connected by an unreliable communication medium: this is one of the reasons for which protocols are built, and it will be supposed that this case does not happen also in the connection among the modules used for modelling entitites performing protocols.

For certain purposes, it may be useful to specify how the interaction primitives are realized by the interface between the interacting modules. In the following, the term "real interaction" is sometimes used for the interface interactions that implement an abstract interaction primitive (see section 4.4).

2.2 Elements for the specification of a module

Within the OSI architecture the concept of module specification is used to describe layer services, protocols, management services, etc. The specification of a module contains the following parts:

---

* A rendez-vous interaction is one in which the two (or more) modules that participate in the interaction execute the interaction during a "rendez-vous", i.e. for an interaction to occur it is necessary that all participating modules execute "their part" at the same time. The interaction implies a close synchronization of the modules. One module has to wait for the other, in general.

2.2.1 Enumeration of possible interaction primitives (types of interactions and parameters)

They are specified considering the points enumerated above. For each module, all the interaction primitives for which it is the sink are enumerated: this list is the "input dictionary" of the module. Analogously, for each module the list of all the interaction primitives for which it is the source: the list is the the "output dictionary" of the module. The specification should be structured by interaction points, as explained in section 3.2.

It is assumed that the interaction parameter values are determined by the source module. It is useful for many purposes to specify for each interaction primitive which module is the source. For example in the Transport service specification, the convention of distinguishing between "requests" and "indications" for the service specifications serves this end.

2.2.2 Specification of possible execution sequences

Each module follows certain rules (constraints) on the execution of the interactions in which it is involved. Such rules could involve the parameter values of the interactions, as well as the order in which the interactions are executed. For example, a Transport entity module will execute a connection establishment indication only after it has received a connection request from a peer entity, and the remote address parameter of the indication will correspond to the value contained in the request. Such rules must be specified to determine the behavior of a module. The set of rules describes the behaviour of the module: more exactly, the behaviour of a module is known when it is known how the sequence of output interactions of the module depends on the sequence of input interaction. The set of rules which a module follows in producing its output interaction may be called the "procedure" of the module. Different specification techniques may be used for this purpose. Possible techniques are developed by the subgroups B and C of the FDT ad hoc group.

2.3 Language for module specifications

The content of this section is being studied by Subgroups B and C.

3. Interconnections of modules

The architecture of a system is defined by the modules out of which the system is built, and the structure by which they are interconnected.

The interactions of a module with other modules or with the environment of the system (as defined in section 2) occur over the interconnections between the modules. In a real system, such an interconnection is realized by a real interface. In this section we are not concerned with the specification of module interfaces, but only with the abstract properties that any real interface for a given module-to-module interconnection must satisfy. These properties may be called the "abstract interface" between two modules.

3.1 Interaction points

An "interaction point" is a useful concept for the description of the OSI architecture. It is related to the notion of "abstract interface" (see above).

The concept serves for

(a) the partitioning of the interactions of a given module into separate groups concerning different parts of the environment, (ensuring that the module has contact with the outside word only through a well defined set of "interaction points"), and

(b) the specification of the interconnections between the different modules within a system (or the sub-modules within a module). An interconnection could be specified by naming an interaction point of one module and an interaction point of another module with which the former is to be interconnected.

For example, typical interaction points of a layer entity executing the layer protocol are: (a) the service access point serviced, (b) the access point(s) of the layer below through which the underlying service is accessed, (c) an (abstract) interface to the local system management module, and possibly a local interaction point through which local services such as buffer management, time-outs, etc. can be obtained.

3.2 Abstraction and step-wise refinement

Abstraction and (inversely) step-wise refinement is supported by the concepts of interaction points and their interconnection. Figure 1 shows an example of a module consisting of three sub-modules interacting with one another. The system may be considered (at a more abstract level of description) as a module that interacts with its environment through three interaction points. If these interaction points are connected with the interaction points of other modules, the given module may be used for the construction of more complex system architectures.

More examples on possible substructures for larger entities are given in Annex 1.

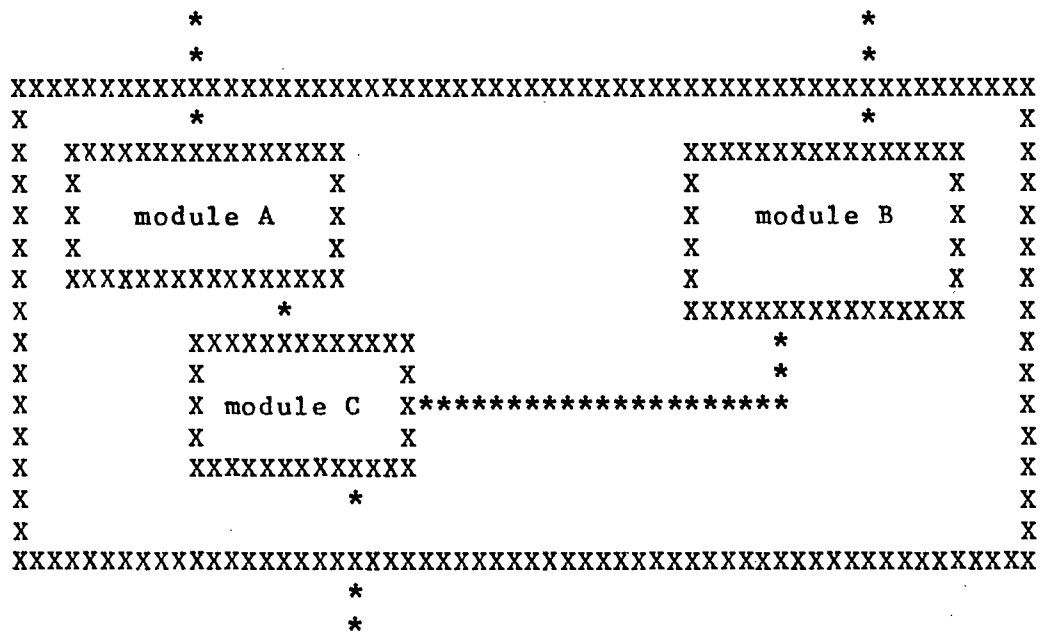


Figure 1

3.3 Description techniques for the specification of interaction points of modules and their interconnections

Further study is required to establish the necessary description means. This study might be based on the conclusions of Subgroups B and C. Simple graphical techniques, such as that used in figure 1, may provide an initial approach to the problem.

4. Definition of service, protocol, and interface specifications

Descriptions of service, protocol and interface specifications are given in the "Introduction to the Guidelines: Overall view of OSI specifications" (N 380). The purpose of this section is to make these descriptions into precise definitions, and to put them into the framework of the specification model outlined in the sections above.

4.1 Service specification for layer N

The service of a layer consists of a set of elementary services of this layer. The service specification for layer N is a specification of a module, consisting of the entities of the layer N and the layers below, given in an abstract view showing only the interactions at the (N)-service-access-points, as indicated by figure 2. The interaction primitives executed at the service access points are called "service primitives". (N)-service-data-units (SDU's) are exchanged as parameters of particular kinds of service primitives (by the T-DATA requests and indications of the Transport service, for example). These interactions would be given for any one of the elementary services and for their interrelations. We note that in this figure and the following, a double arrow represents the interactions taking place between two interaction points of two interacting modules. The name written close to it indicates the kind of interaction primitives.

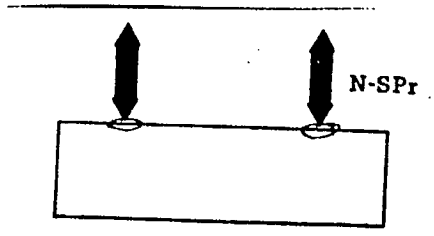


Figure 2

4.2 Protocol specification for layer N

The protocol specification for layer N is the set of the specifications of the modules which represent the entities of layer N: if all such entities have the same procedure (that is, the protocol is symmetric), then the protocol specification coincides with the specification of one module. This module(s) represents an (N)-layer entity providing service through one (or more) (N)-service-access-points, and accessing the service of the layer below through one (or more) (N-1)-service-access-points. For example, the modules A and B in figure 3 are such modules.

The protocol specification should be consistent with the service specification, i.e. the abstracted view of the system shown in figure 3 (ignoring the interactions at the (N-1)-service-access-points) should satisfy the contraints defined by the (N)-service specification.

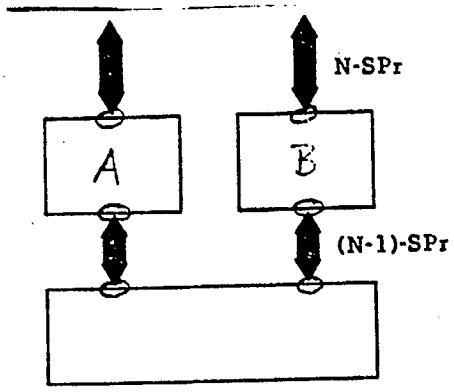


Figure 3

4.3 Abstract protocol specification

An "abstract protocol specification" is a part of a protocol specification which assumes a "mapped" (N-1)-service for the exchange of (N)-PDU's between the peer entities, and relevant control information relating to the (N-1)-service. This is a useful technique because any particular protocol may not use all aspects of the supporting service. The mapped service might, for example, provide for connection establishment and data transfer only.

The complete mapping from (N)-PDU's and control information into (N-1)-service- primitives is not specified directly, but in terms of the mapped service. The specification of the mapped (N-1)-service consists of the specification of a mapping from each of its elements to some element of the (genuine) (N-1)-service and

visa versa.

The situation is as shown by the diagram (a) of figure 4. Alternatively, the diagram (b) is sometimes used to indicate an abstract protocol specification, where the single arrow indicates the use of the mapped service.

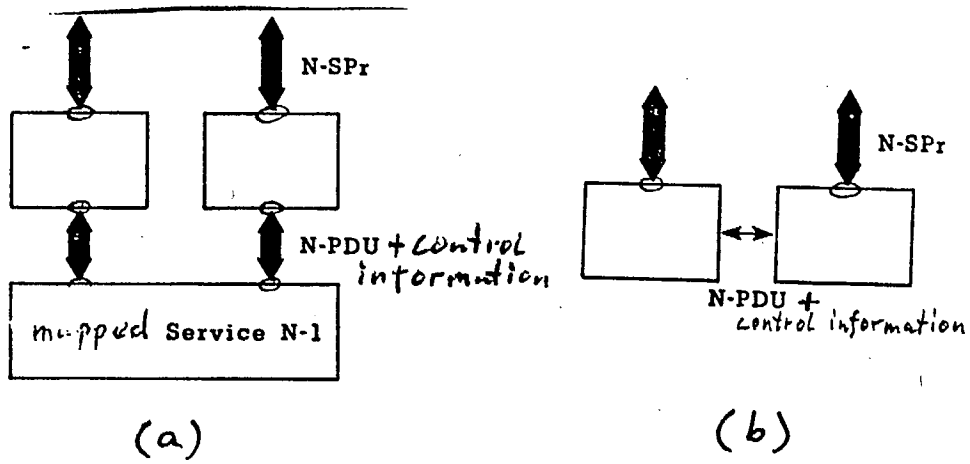


Figure 4

4.4 Implementations and real interfaces

For the module specifications considered (and in particular for protocol and service specifications) the module is assumed to interact with the other modules in a system through interaction primitives. An implementation of such a module, however, will interact by "real interactions" (of hardware or software nature) realized by a real interface. One real interface per interaction point is usually foreseen.

An implementation of the interactions over a given interaction point includes the definition of a mapping from the abstract interaction primitives into the real interaction at the interface. It defines a correspondence between the real interactions and the interaction primitives, which are not necessarily explicitely visible in the implementation. Figure 5 shows the correspondence between an abstract module specification (a) and its implementation (b).

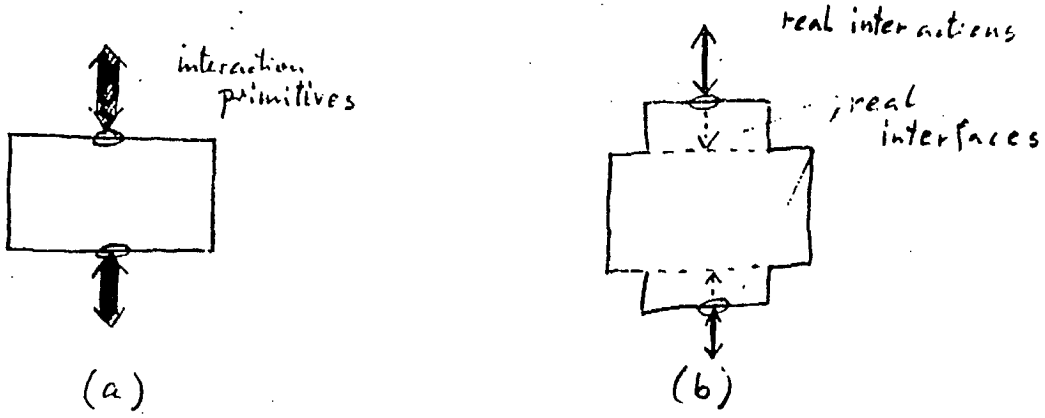


Figure 5

5. Definition of terms

...for further study...

Annex : Examples of entity substructures

For specification purposes, it seems to be useful to consider a substructure of an entity. Different kinds of substructures may be considered depending on the nature of the entity to be described. Some possible substructures are discussed in the following subsections. Further work is needed for identifying appropriate substructures for protocol specifications.

As far as the work of the FDT ad hoc group is concerned, it seems to be necessary to determine a description technique for defining a substructure. A possible approach to this end is the use of the concepts and methods described in section 3, such that the entity is considered a module which consists of several interconnected submodules.

1. Identification of a "mapping" submodule

The concept of an abstract protocol specification (see section 4.3) suggests a substructure of an entity as indicated in the figure 6.

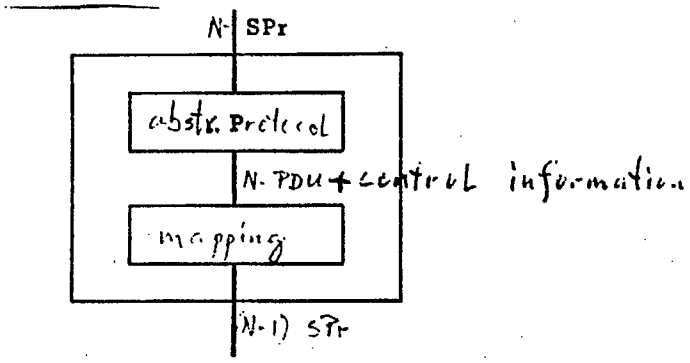


Figure 6

2. A possible entity substructure

Other entity substructures may be considered, such as the following: an entity X, or each of the submodules shown in figure 6, may be subdivided into the submodules shown in figure 7 below.

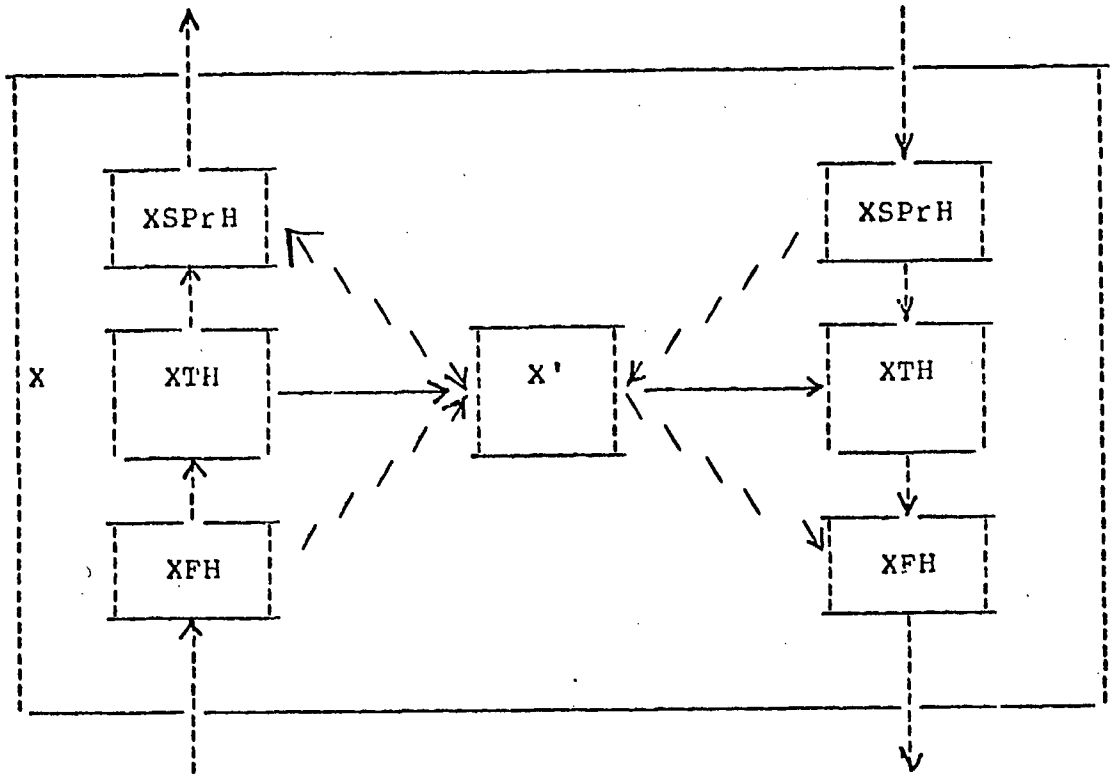


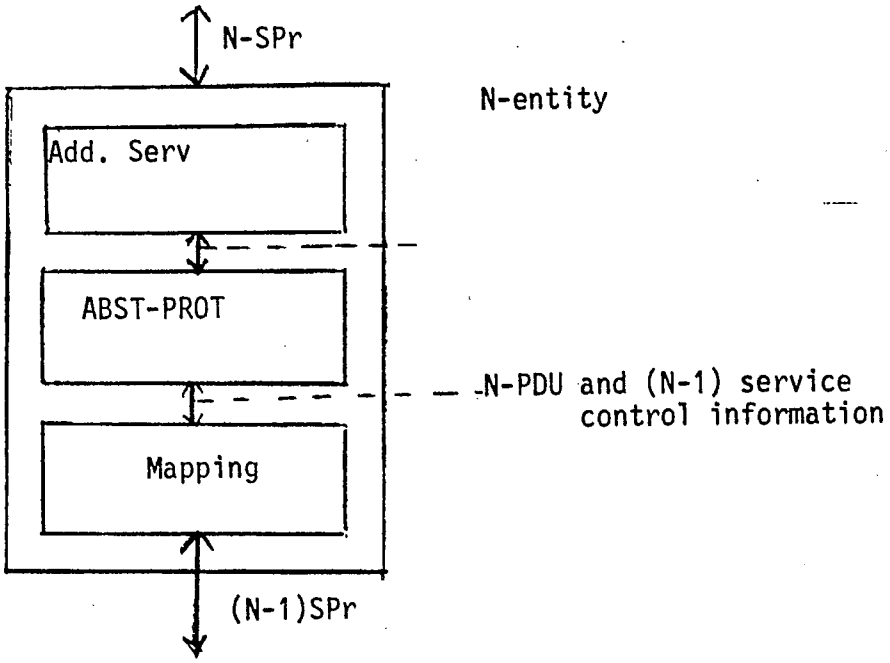
Figure 7

In this figure, the submodule X' executes the abstract protocol of the module X (and processes the control information contained in the input interactions); XFH (X Format Handler) are modules for handling Input/Output format problems for module X; XTH (X Test Handler) are modules for handling user data, e.g. for segmentation, reassembling, store for retransmission, etc., and XSPrH (X Service Primitive Handler) are modules for handling service primitives which interact with module X.

3. Possible identification of submodules

The concept of an abstract protocole specification (see section 4.3) suggests a substructure containing separate submodules for mapping and abstract protocol.

Moreover there may be cases in which the complexity of the service suggests to introduce a third box called "additionnal service" and leads to the following structure.



NOTES

1. The boxes located at the top and the bottom are optionnal. Thus, depending on the entity to be described the structure may be different.

2. Only the "protocol box" is mandatory in all cases: thus the structure can be reduced to a single protocol module.

3. The concepts described above are only suitable for description purpose and do not have to be introduced in the model for OSI as generic concepts.

4. Examples of the use of the "Additionnal Service" box can be the quarantining or blocking services at the session layer or some manipulation or transformation of the data store at the presentation layer.

5. This section 3 is more general than section 1 of this annex. Due to lack of discussion, it has been included as a separate section. It may superseed section 1 in the future.

ANNEX 2

To: Members of ISOTC97/SC16/WG1

From: Subgroup B of ad hoc group on FDT

Title: A FDT based on an extended state transition model (Working Draft, Boston, Déc. 1981)

1. Introduction

This document describes a FDT for the specification of communication protocols and services. The specification language is based on an extended finite state transition model and the Pascal programming language.

2. An Extended State Transition Model

2.1. Introduction

A system comprises interconnected modules, each of which is an extended finite state transition machine, which is described as explained below.

2.2 The model of interactions

The extended state transition model described in section 3 assumes a model of interaction where each interaction of the specified module with its environment can be considered an atomic event. The transition model distinguishes between interactions that are initiated by the environment and received by the module (inputs), and interactions initiated by the module (outputs).

The reception of an interaction from the environment produces, in general, a state transition of the specified module which may give rise to other (output) interactions.

For the interaction between two modules, the model allows for the queuing of the outputs from one module before they are considered as input by the other. Queues of infinite or finite (usually zero) length are possible. The length of the queue is determined when the modules and their interconnection are instantiated (see "Concept for describing the OSI architecture" (working document of Subgroup A), section 3). It is noted that zero buffer length means a rendez-vous type of interaction (see "Concepts...", section 2.1).

2.3 A state transition model

In order to define the possible orders in which interactions may be initiated by the entity, the state transition model introduces the concept of the "internal state" of the entity which determines, at each given instant, the possible transitions of the entity, and therefore the possible interactions with the environment.

The possible order of interactions of a module (or entity) is given in terms of

(a) the state space of the module which defines all (internal) states in which the module may possibly be at any given time, and

(b) the possible transitions. For each type of transition, the designer specifies the states from which a transition of that type may take place, and the "next" state of the module. A transition may also involve one or more interactions of the module with its environment (see below).

Since finite state diagrams or equivalant methods often lead to very complex specifications when a complete protocol specification is required (partial specifications, can be more readily comprehended) the following approach to the specification of modules in the extended state transition model is used. This approach combines the simple concept of states and transitions with the power of a programming language.

The state space of the module is specified by a set of variables. A possible state is characterized by the values of each of these variables. One of the variables is called "STATE". It represents the "major state" of the module.

The possible transitions of the module are defined by the specification of a number of transition types. Each transition type is characterized by

(a) an enabling condition: This is a combination of a boolean expression depending on some of the variables defining the module state, and (possibly) the specification of an input. A transition may occur in a given state only if the enabling condition has the value true, and the interaction in question (if it exists) is initiated by the environment.

(b) an operation: this operation is to be executed as part of the transition. It may change the values of variables, and may specify the initiation of output interactions with the environment. The operation is assumed to be atomic.

The model is non-deterministic in the sense that in a given state (at some given time) and a given input interaction, several different transitions may be possible. Only one of these transitions is executed, leading to a next state which determines which transitions may be executed next. If several transitions are possible at some given time, the transition actually executed is not determined by the specification model. An implementation of the module could choose any of these possibilities.

In many cases, the specification of a module may be deterministic, in the sense that (at most) one transition is specified in any reachable state and given input.

3. Language elements

This section gives an introduction to the different elements of the specification language based on the extended state transition model described above.

The language is largely based on the syntax and semantics of the Pascal programming language (ISO DP7185, formally TC97/SC5 N595, see also Jensen and Wirth: "Pascal: User manual and report", Springer Verlag, 1974), and uses the general approach of using type definition facilities and type checking for allowing the implementation of automatic consistency checking, which usually detects a large proportion of those errors in a specification that connot be found by syntax checks.

A complete definition of the syntax is contained in section 4.

3.1 Language elements taken from Pascal

The following language elements of the Pascal programming language are included in the specification language without any change in syntax and semantics:

(a) Type and constant definitions including
- scalar types, including enumeration types
- subranges
- record types
- array types

Predefined types:
- boolean
- integer
- character (defined by some ISO standard)

(b) Procedure and function definitions

(c) Statements

3.2 The specification of interactions

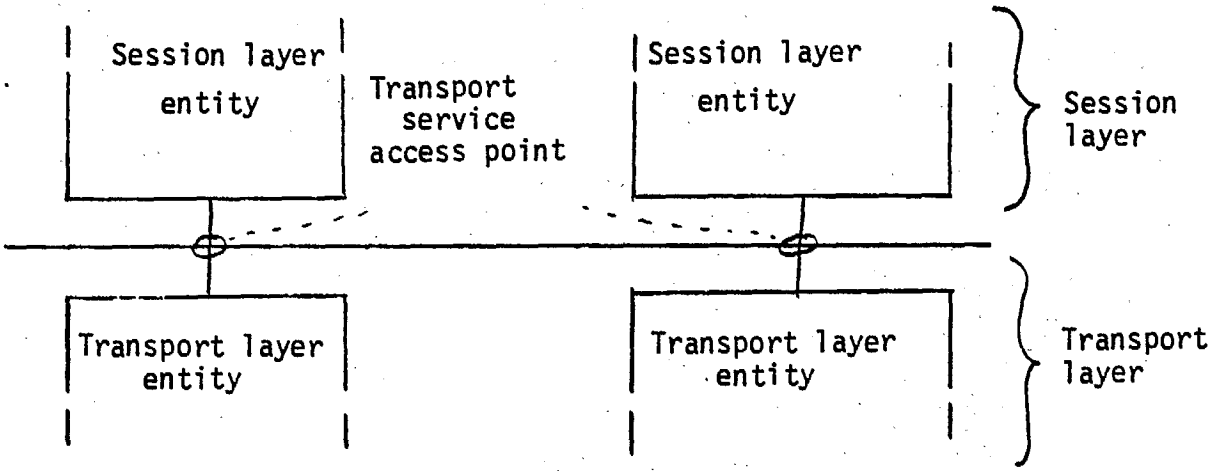
The following examples are considered. The (N)-service is provided to the entities in the layer above by the interactions through the service access points between the service providing module and its environment. The interaction model is also useful to define interactions between different entities (or "modules") of an (N)-layer subsystem. For example, it may be used for defining the timer or data buffering services used in the (N)-layer protocol.

In the following the term "channel" denotes the interactions between the given module and another module in its environment. For example, a serivce access point is a channel between the service providing module and the entity using the service through this access point. It should be noted that the abstract properties of these channels are discussed here only to the extent that they are concerned with service and protocol specifications.

The specification of a channel of a module is given by enumerating the possible interaction primitives that may occur over the channel (including possible parameter values (determined by the module initiating the interaction), and indicating whether the module, its environment, or both may initiate the interaction).

The language allows the specification of the possible interactions through a channel without explicitly defining the modules that interact through the channel. However, it is necessary to refer to the roles that these modules play in this interaction.

As an example we consider the abstract interface through which the Transport service is provided at some Transport service access point. The diagram below shows the entities involved.



Using the syntax defined in section 4, the possible service primitives may be enumerated as follows.

```
interaction

  TS_access_point(TS_user,TS_provider) is

     by TS_user:

       T_CONNECT_req(TCEP_identifier   : TCEP_identifier_type;
                     to_T_adress       : T_address_type;
                     from_T_address    : T_address_type;
                     QQTS_request        : quality_of_TS_type;
                     TS_connect_data   : TS_connect_data_type);

       T_ACCEPT_req(TCEP_identifier    : TCEP_identifier_type;
                    QOTS_request         : quality_of_TS_type;
                    options                : option_type;
                    TS_accept_data     : TS_accept_data_type);

       T_DISCONNECT_req etc.
```

by TS_provider:

T_CONNECT_ind

etc.

This specification states that a module that interacts through a Transport service access point must take the role of a "TS-user", or a "TS-provider". Depending on its role it may initiate a certain number of interactions (indicated by the BY clause), for example a user may initiate requests for connection establishment or disconnection, or the sending of a fragment of user data.

The same notation may also be used for defining the interactions between several entities within the same layer, or between an entity and some locally provided services, such as timers or buffer management. An example is the following definition of the timer services used by the Transport entity implementing the Transport protocol.

```
interaction
  timer-interface (user, server) is
    by user:
       start (period: integer);
       stop;
    by server:
       time-out;
```

We note that the possible orders of interactions are not specified. However, it is understood that the time-out interaction will only be initiated by the server "period" seconds after it has received a start interaction and no subsequent stop interaction.

3.3 Module interconnection

It is useful to separate the specification of the characteristics of channels from statements that certain modules use certain types of channels. For example, the characteristics of the (N)-service access points are relevant for the (N)-service specification, the (N + 1) - layer entities, as well as for the (N)-protocol specification. This leads to a specification method in which channel types may be defined independent of their use, and the specification of a module includes an enumeration of all the interaction points through which it interacts with its environment, with an indication of the channels type for each of these interaction points. The syntax for these specifications is given in section 4.

The language must be enhanced for specifying how the interaction points of the different modules and entities within an Open System are connected through channels. These considerations are for further study.

To demonstrate these ideas, the following lines show the general outline of Transport service and protocol specifications. It is noted that the Transport service access point definitions are used by the service as well as by the protocol specifications. The PDU and timer interface definitions are only used by the protocol specification and therefore included in that section.

Specification of the Transport layer service:

```
module TS(access_points : array[T_address_type] of TS_access_point
                                        (TS_provider));
            <global constraints of the Transport service>
```

Specification of a Transport protocol (balanced class):

```
  type
     max_TPDU_size type = ...
interaction
  TPDU (N_calling, N_called) is etc.
interaction
  local_buffer(user,buffer) is etc.
module Transport_entity  (TSAP : TS_access_point(TS_provider);
                          N    : TPDU(N_calling,N_called);
                          T    : timer_interface(user);
                          out_buffer,
                          in_buffer : local_buffer(user));
  <global constraints of the Transport entity>
```

The specification of the Transport entity states that such an entity interacts through an interaction point, called "TSAP", which uses a channel of type "TS-access-point", where it takes the role of a service provider, and also through a timer interface, where it is a user. It also interacts with a Transport mapping submodule (see "Concepts...", Annex, section 1) through the interaction point called "N" by using the interactions defined as TPDUs. The <global constraints of the Transport entity> are specified with the state transition model as described below.

3.4 Overview of the externally visible properties of a module

The external behavior of a module is determined by the following:
(a) enumeration of the interaction points through which the module interacts with its environment. The specification of each interaction point includes the following information:

(a1) enumeration of the interactions that may occur through the interaction point;

(a2) a set of rules that determine the order in which these interactions may occur.

(b) global constraints on the order in which the interactions through different interaction point of the module may occur. (In the case of service specifications, these constraints define how the interactions at the two end-points of a connection relate to one another. In the case of a protocol specification, these constraints specify the order in which different PDU's may be sent, and how the interactions at the (N)-service access point of the entity relate to the sending and receiving of PDU's through the (N-1)-layer interface).

While (a1) is explicitly defined by the interaction definitions (see section 3.2) points (a2) and (b) are implicitely determined by the state transition model (see section 3.5.).

3.5 Specification of a module in the state transition model

The state space of the module is specified by a set of variables. A possible state is characterized by the values of each of these variables. One of the variables is called "STATE". It represents the "major state" of the module.

As an example, the following lines specify the state space of an entity implementing the Transport protocol:

```
var
  state : (idle,wait_for_CC,wait_for_T_ACCEPT_req,data_transfer);
  local_reference : TP_reference_type;
  remote_reference : TP_reference_type;
  TPDU_size :max_TPDU_size_type;
  QOTS_estimate : quality_of_TS_type;
```

The possible transitions of the module are defined by the specification of a number of transition types. Each transition type is characterized by:

(a) the enabling condition: this includes
- the present major state (FROM clause)
- the input (WHEN clause)
- the "additional enabling condition" (or "predicate") (PROVIDED clause)
- the priority of the transition type (PRIORITY clause)

(b) the operation of the transition: this includes
- the definition of the next major state (TO clause)
- the "action" (BEGIN statementof the <block>) including the generation of output.

As an example, the following lines specify some transition types for a Transport entity:

```
from idle
  when TSAP.T_CONNECT_req
    provided ...(* Transport entity able to provide the quality of
                   service asked for *)
    to wait_for_CC
    begin
      local_reference := ...;
      TPDU_size := ...;
      N.CR(0,local_reference,class_0,normal,variable_part_to_send);
    end;

from data_transfer to same
  when TSAP.T_DATA_req
    provided ... (* flow control from user ready *)
      begin
        out_buffer.append(user_data);
      end;
  when out_buffer.next-fragment
    provided ... (* Network layer flow control ready *)
      begin
        N.DT(data-fragment);
      end;
```

3.6 User guidelines

4. Syntax overview

This section defines the syntax of the specification language. Large parts of the language are taken from the Pascal programming language (ISO DP 7185).

Elements of the Pascal programming language are used for the specification of constants, data types, procedures and functions, and the declaration of the state variables.

This section defines the extensions to Pascal, as well as certain restrictions.

4.1 Syntactic extension

Notation: Extended BNF where "+" means one or more occurrences, "*" means zero, one or more occurrences of an expression, and "|" separates alternatives". "**" means that the construct is the same as in Pascal.

A service or protocol specification consists of a specification of the interaction points and primitives (see section 4.1.1) and one or more module specifications (see sections 4.1.2 and 4.1.3). Only the definition of a module type is given here. Language elements for the declaration of module instances within a system and their interconnection is for further study.

4.1.1 Interaction points and primitives

The <channel definition> defines a type of interaction point.

<channel definition> ::= <constant definitions>*
<type definitions>* <channel type def>

The possible interactions at a given type of interaction point are enumerated by a definition of the following form:

```
<channel type def> ::= INTERACTION <channel type id>
        ( <role list> )  <interactions>
<role list> ::= <role id>
              | <role list> , <role id>
<interactions>::= <BY clause>
                | <interactions> <BY clause>
<BY clause> ::= BY <role list> : <interaction list>
<interaction list> ::= <interaction>
                | <interaction list> <interaction>
<interaction> ::= <interaction id> <interaction parameters> ;
```

The declaration of <interaction parameters> is in the same form as function parameter declarations in Pascal (i.e. for each parameter its name and type).

```
<interaction id>   ::=   <identifier>     (*Note1*)
<channel type id>  ::=   <identifier>
```

Note 1: Alternatively, the form of an <interaction id> could indicate whether the interaction is, for instance, a request, indication, response, or confirmation (for further study).

4.1.2 Modules and their interaction points

The definition of a module type contains the declaration of all abstract interaction points through which a module of this type interacts. This includes the service access points through which the communication service is provided as well as the system interface for timers, etc. and the access point to the layer below, through which the PDU's are exchanged.

```
<module type definition>::= MODULE  <module type id>
                            ( <interaction points> ) ;
                            <module body>
<interaction points>  ::= <interaction point declaration>
                        | <interaction points> ; <interaction point
                                                   declaration>
<interaction point declaration> ::= <interaction point id> :
                                    <channel type id>
                                    ( <role id> )
```

The <role id> indicates which role the entity plays as far as the declared interaction point is concerned. We note that the distinction of these roles permits the checking that the invocation of interactions in the conditions and actions of transitions is consistent with the possible exchanges defined in the channel definition.

4.1.3 Extended state transition model

```
<module body> ::= <label definitions>**
                  <constant definitions>**
                  <type definitions>**
                  <variable declarations>**
                  <major state declaration>
                  <state set definition>*
                  <procedure and function definition> (*Notes 2 and 3*)
                  <initialization>
                  <transition>+
                  END.

<major state declaration> ::= STATE : <enumeration type> ;
<state set definition>  ::=   <state set id> = <set definition>** ;
                                                        (*Note 4*)
<initialization>    ::=    <state initializer> <begin statement>** ;


<transition>::=
   | ANY <identifier> : <type identifier** DO <transition>+ (*Note 5a*)
   | WITH <variable>** DO <transition>+ (*Note 5b*)
   | WHEN <interaction point id> . <intraction id> <transition>+ (*Note 5c
   | FROM <major present state> <transition>+ <*Note 5d*)
   | TO <major next state> <transition>+ (*Note 5e*)
   | PROVIDED <expression>** <transition>+ (*Note 5f*)
   | PRIORITY <priority indication> <transition>+ (*Note 5g*)
   | <block>** ;


<priority indication> ::= <identifier>**  (*constant of some
                                           enumeration type*)
                        | <integer>**
<major present state> ::= <major state value>
                        | <state set id>
<major next state>    ::= <major state value>
                        | SAME
<major state value>   ::= <identifier>**   (*must be element of the
                              enumeration type of the <major
                              state declaration>*)


<output statement>::= <interaction point id> . <interaction id>
                          <effective parameter list>**  (*Note8*)
```

Note 2 : Within a transition, "..." may be written for an expression that is implementation dependent (not defined by the specification). The body of a procedure or function that is implementation dependent (not defined by the specification) is written in the form "PRIMITIVE" or "...".

Note 3 : A boolean function X(<parameters>) with no side effects may be declared in the form "predicate X(<parameters>)".

Note 4 : The elements of the set must be included in the enumeration type of the <major state declaration>.

Note 5a: These transitions may not include a ANY clause.

Note 5b: These transitions may not include a WITH clause.

Note 5c: These transitions may not include a WHEN clause.

Note 5d: These transitions may not include a FROM clause.

Note 5e: These transitions may not include a TO clause.

Note 5f: These transitions may not include a PROVIDED clause. The expression must be boolean.

Note 5g: These transitions may not include a PRIORITY clause.

Note 6 : Each <block> must be preceeded by a FROM and a TO clause.

Note 7 : To refer to the input parameters, the parameter identifiers of the interaction in the <channel type definition> are used.

Note 8 : This kind of statement (for producing an output interaction) is an extension of Pascal.

4.1.4 Other extensions

(a) A comment that starts with the word "property" describes properties that are part of the specification.

(b) A facility for describing optional parameters is introduced. To indicate that a parameter (or field of a record) is optimal, its type definition is preceeded by the keyword OPTIONAL. the value UNDEFINED means that the parameter (or field) is not present. A default value may be associated with the type definition by a succeeding "DEFAULT=<constant>" clause.

4.2 Removal of certain restrictions

4.3 Elements of Pascal not used

5. Definition of the semantics

6. Conformity rules for checking implementations

7. Verification rules for checking that an (N)-service is rendered by an (N-1)-service and an (N)-protocol.

Annex A: Terminology

Annex B: Relation to graphical description techniques

1. Introduction

Graphical description techniques are often used to give an overview of a protocol or service specification, and sometimes are enhanced to provide a complete specification. Different graphical representations of extended state transition models are in use. Some of these representations are shown in section 2. The systematic translation of linear specifications written in the FDT described in this document, into graphical representations is discussed in section 3.

2. Different graphical description techniques

The following subsections present overviews of the Transport protocol class 0 connection establishment phase (a complete specification is given in Annex D) using different graphical description techniques. This may be used for a comparison of these graphical techniques.

2.1 Common state transition diagrams

The diagram of Figure 1 gives an overview. It specifies the major states and the types of transitions, indicating for each transition only the kind of the relevant input and output. A similar description technique is used in several CCITT Recommendations, such as X.25, etc.

2.2 Enhanced state transition diagrams

The diagram of figure 2 contains the basic information of figure 1, but it also includes some additional information about conditions and actions of transitions relating to the interaction parameters and additional state variables of the extented state transition model. Such a description technique is used in several SC6 documents, such as SC6 N2231.

2.3 The System Description Language (SDL) of CCITT SGXI

The diagram of figure 3 contains the same information as figure 2, using the SDL of CCITT.

3. The translation of the linear FDT into graphical form

The translation is relatively straightforward if the linear specification contains the transitions sorted by major present states (FROM clause), input interactions (WHEN clause) and additional conditions (PROVIDED clause), as in the example below. Any specification may be put into this form by a simple rearrangements of the order of the different transitions. The following example is considered:

```
(*transitions*)
from A
    when AP.req1
       provided C1
            to B
            begin Action1; AP.ind1 end;
       provided C2
            begin Action2; AP.ind2 end;
    when AP.req2
            to C
            begin Action3; AP.ind3 end;
```

The translations of these three transitions into the different graphical representations are shown in figures 4, 5 and 6.

3.1 Translation into common state diagrams

All states shown in the diagram are declared in the <major state declaration> part of the linear specification. Each defined transition gives rise to an arrow in the diagram, as shown in figure 4 (using the information of the FROM and TO clauses). The information for the annotation of the arrows is taken from the WHEN clause and the BEGIN statement of the transition <block>. This statement must be scanned to extract the <output statements> which are used for the annotation of the arrows.

3.2 Translation into enhanced state transition diagrams

While in the overview diagrams of common state diagrams the information of the PROVIDED clauses and the BEGIN statement (except for the output) is usually lost (see figure 1), this information may be included in the enhanced transition diagrams, as shown in figure 5. The translation process is similar to the case of common state diagrams.

3.3 Translation into SDL

The process of translating a linear specification into SDL is closely related to the embedded structure of the linear specification (see example above). Each FROM clause corresponds to a "large" graphical state symbol. Each WHEN clause, within a given FROM clause, corresponds to a graphical input symbol connected to that state symbol. If for a given WHEN clause, there are embedded PROVIDED clauses, then a graphical decision symbol represents the choice between these alternative transitions, as shown in figure 6. The BEGIN statement corresponds, in general, to an action symbol and possibly some output symbols. (The relevant outputs must be extracted from the BEGIN statement, as explained in section 3.2). The TO clause corresponds to a "small" state symbol which terminates a transition.

ANNEX 3

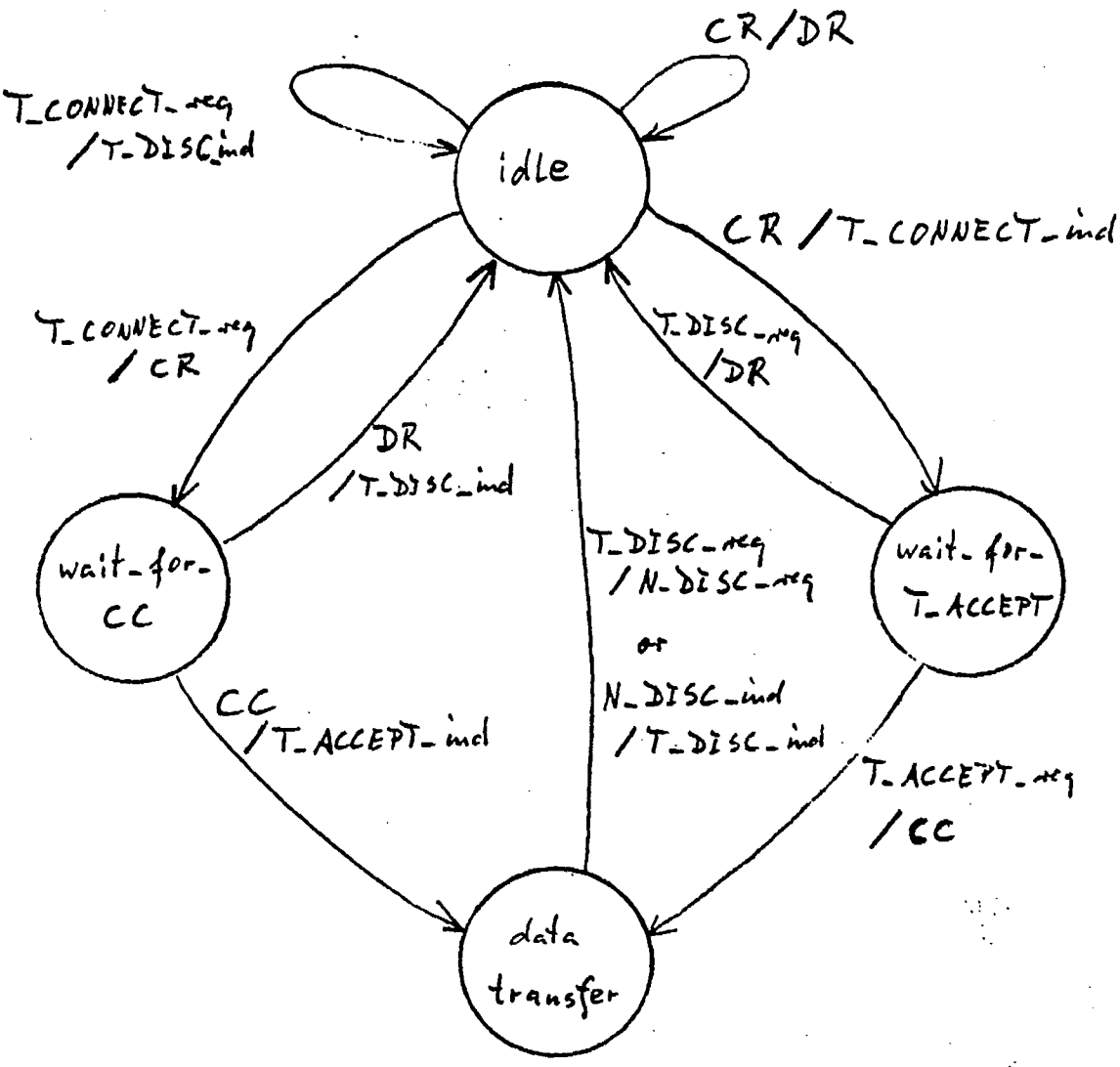

CR/DR
T_CONNECT_req
/T_DISC_ind
idle
CR /T_CONNECT_ind
T_CONNECT_req
/CR
T_DISC_req
/DR
DR
/T_DISC_ind
wait_for_
CC
T_DISC_req
/N_DISC_req
or
N_DISC_ind
/T_DISC_ind
wait_for_
T_ACCEPT
CC
/T_ACCEPT_ind
T_ACCEPT_req
/CC
data
transfer

Figure 1

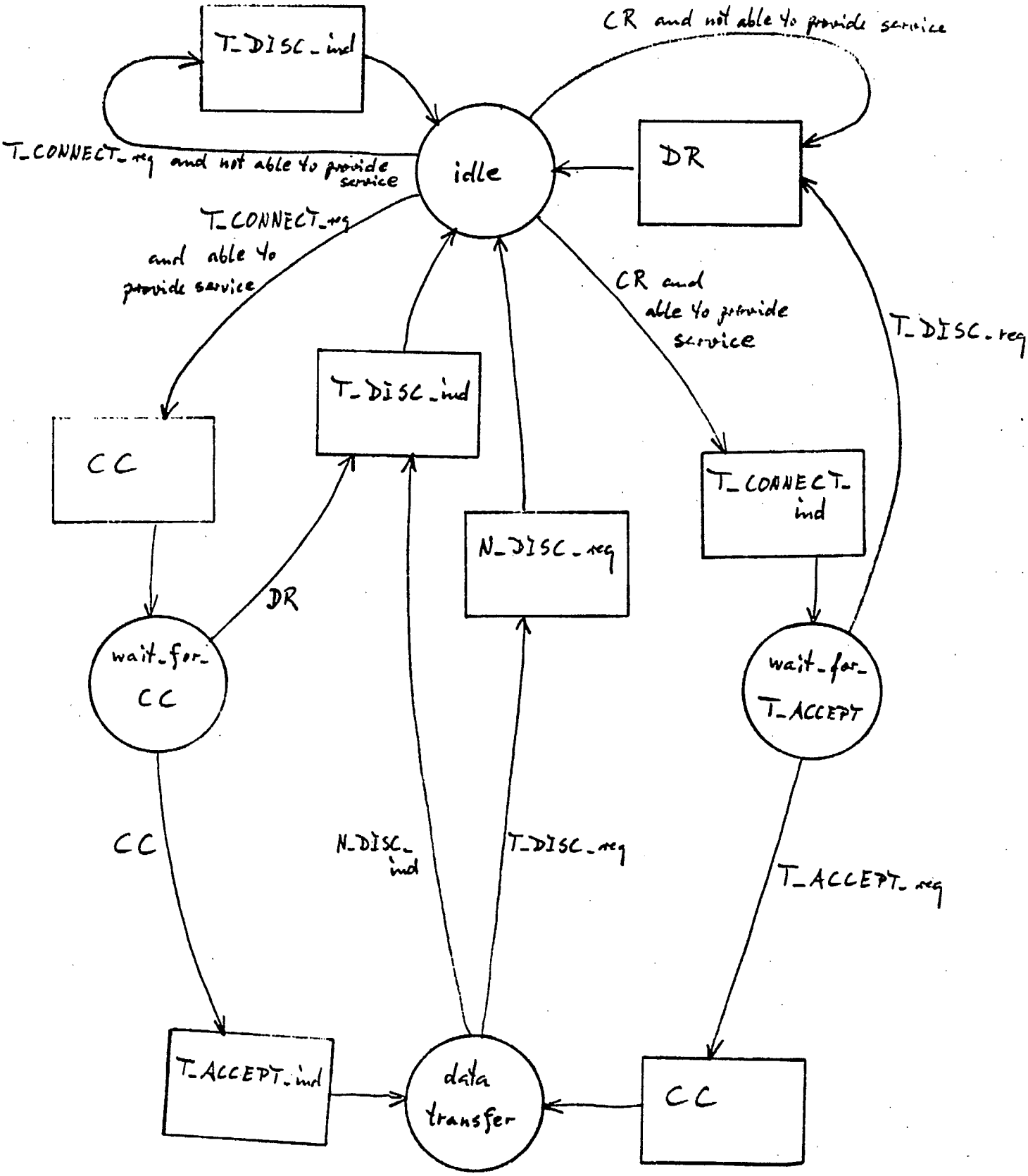

T_DISC_ind
CR and not able to provide service
T_CONNECT_req and not able to provide service
idle
DR
T_CONNECT_req and able to provide service
CR and able to provide service
T_DISC_req
T_DISC_ind
CC
T_CONNECT_ind
N_DISC_req
DR
wait_for_CC
wait_for_T_ACCEPT
CC
N_DISC_ind
T_DISC_req
T_ACCEPT_req
T_ACCEPT_ind
data transfer
CC

Figure 2

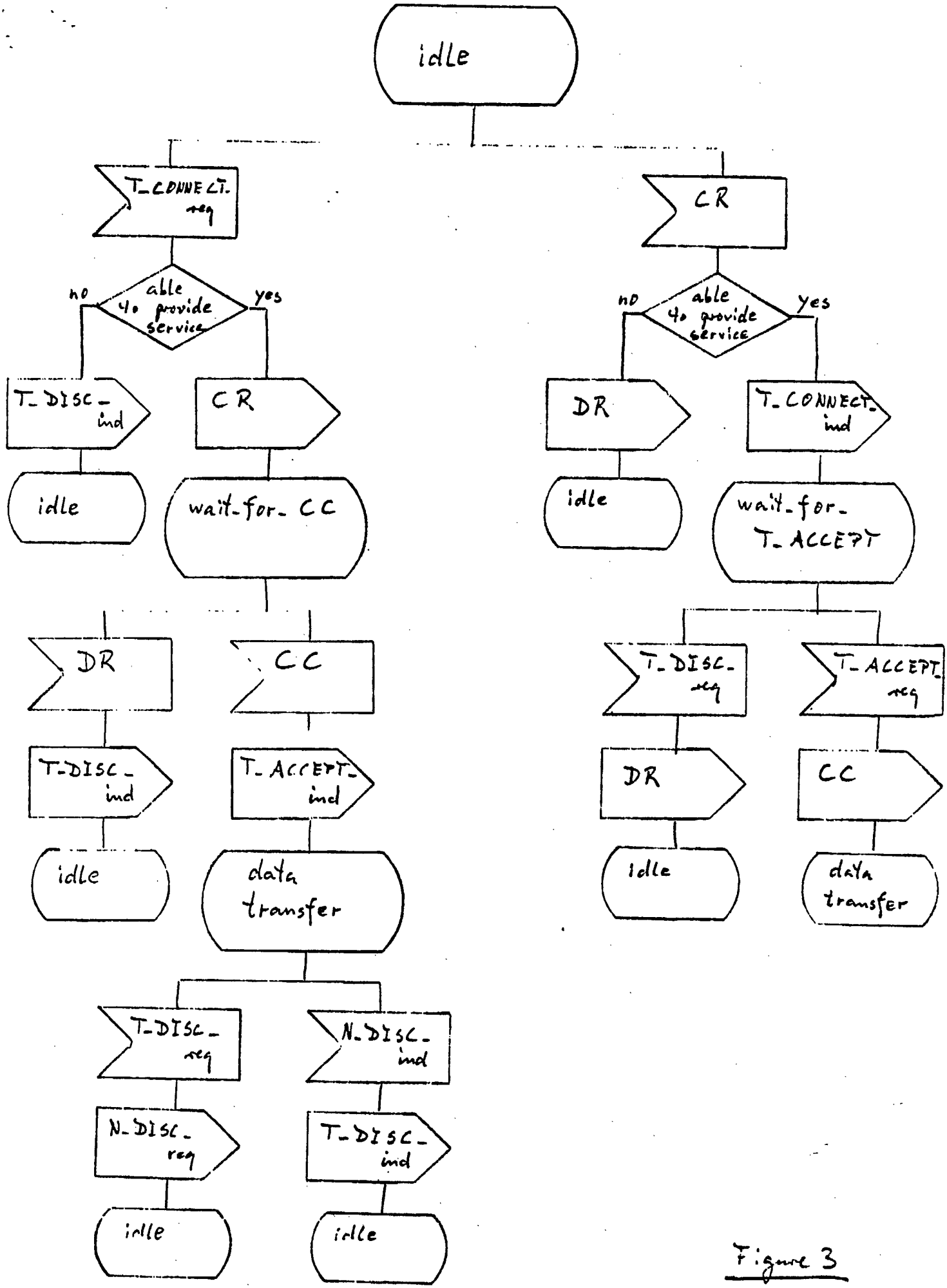

idle
T_CONNECT_req
able to provide service
no
yes
T_DISC_ind
idle
CR
wait_for_CC
DR
idle
CC
T_DISC_ind
idle
T_ACCEPT_ind
data transfer
T_DISC_req
N_DISC_req
idle
N_DISC_ind
T_DISC_ind
idle
CR
able to provide service
no
yes
DR
idle
T_CONNECT_ind
wait_for_T_ACCEPT
T_DISC_req
DR
idle
T_ACCEPT_req
CC
data transfer

Figure 3

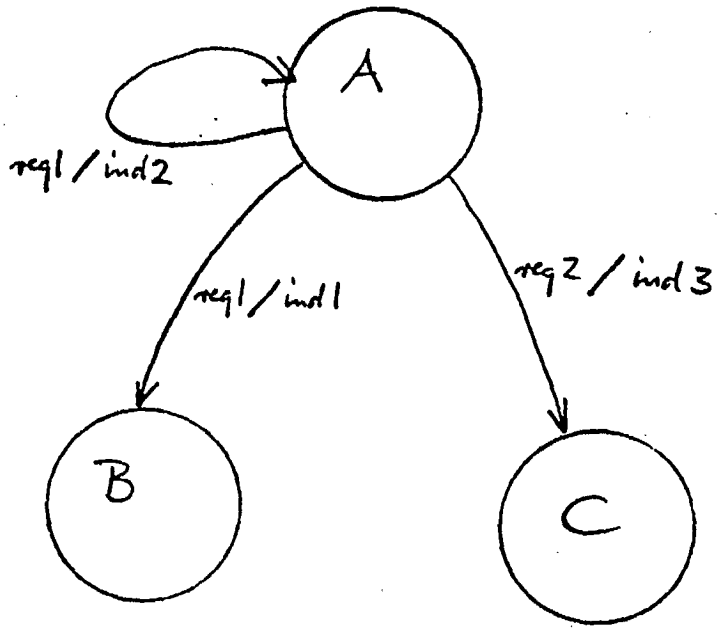


Figure 4

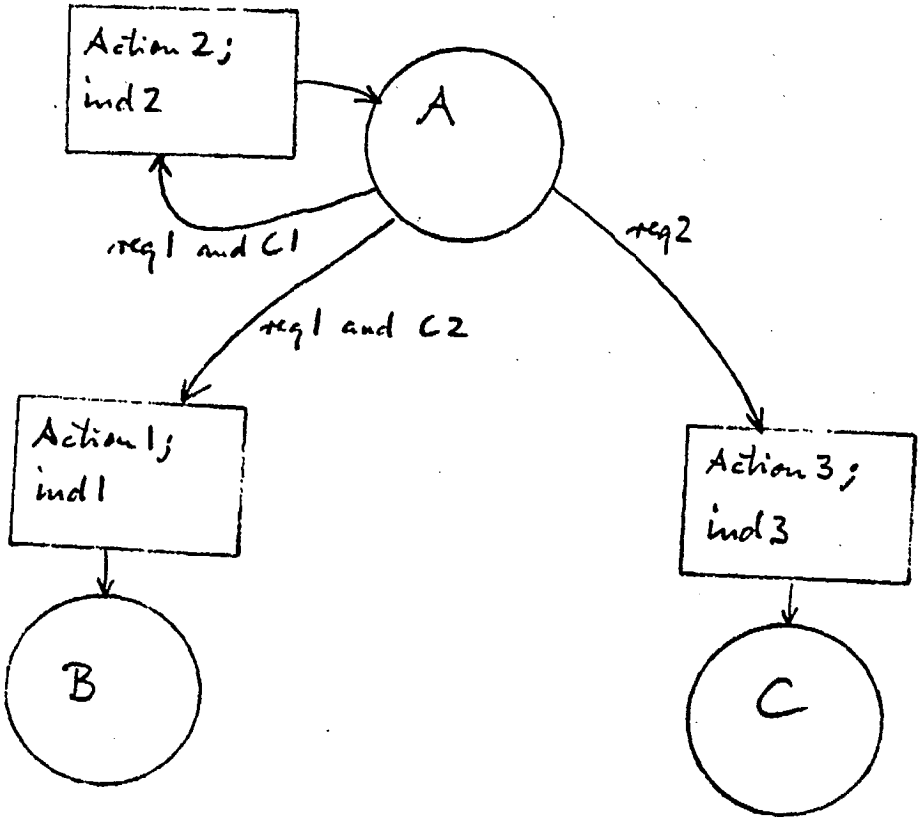


Figure 5

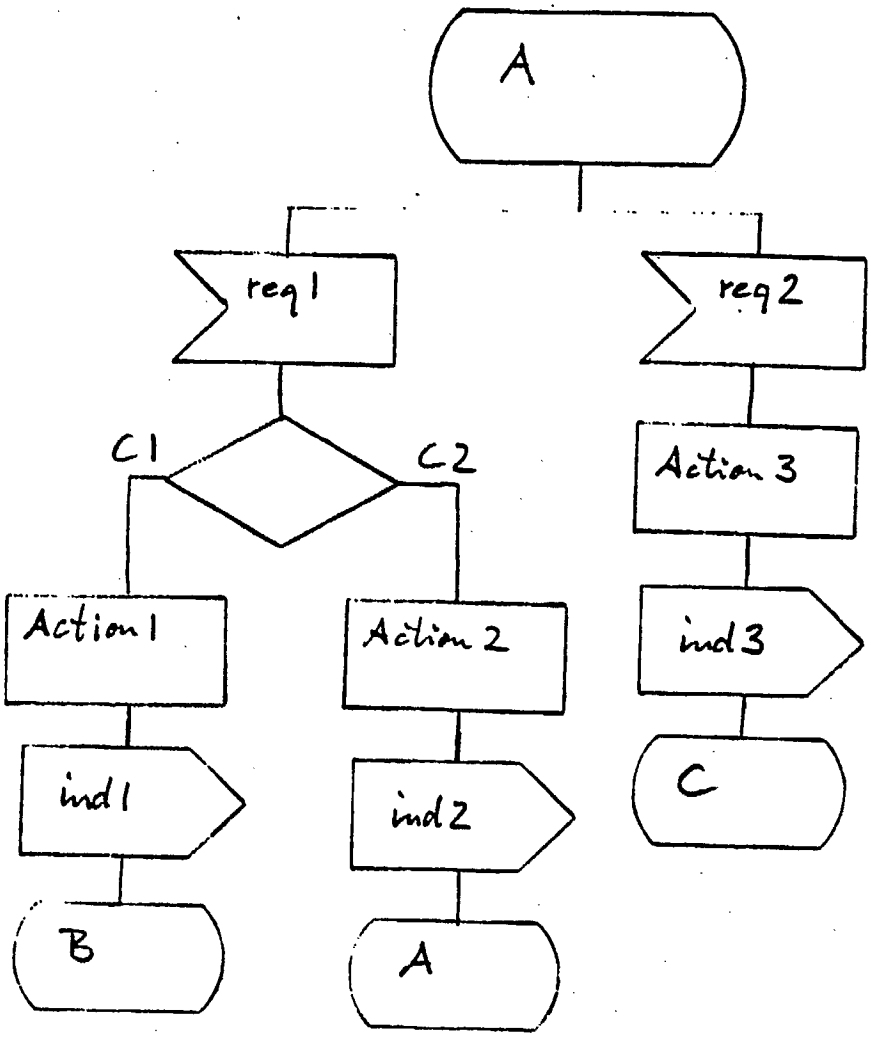


Figure 6

WASH-9

To: ISO/TC97/SC16/WG1 Rapporteur's group on FDT

Source: G.V. Bochmann, E. Cerny, C. Lacaille (Canada)

Title: Formal specification of a Transport Service

1. Introduction

The annex gives a specification of the Transport service using the extended state transition model described in "Tutorial on formal description techniques (FDT)" (SC16 N..., and TUB-11). It is intended as an example of the use of this FDT.

We note that the first part of the annex (specification of the types and service primitives) was already given in annex 1 of N706 ("Formal specification of a Transport protocol").

```
type
  T_address_type = ...;        (* note 1 *)

  TCEP_identifier_type = ...; (* note 2 *)

  quality_of_TS_type = record
    throughput_from_average    : integer;   (* bps *)
    throughput_to_average      : integer;   (* bps *)
    throughput_from_minimum    : integer;   (* bps *)
    throughput_to_minimum      : integer;   (* bps *)
    transit_delay_from_average : real;      (* seconds *)
    transit_delay_to_average   : real;      (* seconds *)
    transit_delay_from_maximum : real;      (* seconds *)
    transit_delay_to_maximum   : real;      (* seconds *)
    residual_error_rate        : real;      (* probability *)
    set_up_delay               : real;      (* seconds *)
    resilience_of_TC           : real;      (* seconds *)
    acceptable_cost            : real;      (* some monetary unit *)
    security_level_of_TC       : integer;   (* ??? *)
    connection_assurance       : real;      (* seconds *)
    priority_level             : integer;   (* ??? *)
                        end;

  option_type = (normal,fast_connect_disconnect,with_expedited); (* see note 10 *)

  TS_connect_data_type = ...;    (* string of octets of limited length *)

  TS_accept_data_type = ...;     (* string of octets of limited length *)

  TS_expedited_data_type = ...;  (* string of octets of limited length *)

  fragment_length_type = ...;    (* implementation dependent *)

  data_fragment_type = record
    last_fragment_of_TSDU : boolean;
    length : fragment_length_type;   (* length of string *)
    data : ...;                      (* string of octets *)
                        end;
```

```
TS_disconnect_reason_type = (TS_user_initiated_termination,
                             lack_of_local_resources,
                             inability_to_provide_the_quality_of_service_asked_for,
                             inability_to_maintain_quality_of_service,
                             misbehavior_of_TS_user,
                             reference_overflow,
                             mismatched_reference,
                             local_congestion,
                             remote_congestion,
                             empty,
                             ...);

TS_user_reason_type = ...;      (* string of octets of limited length *)
```

```
interactions

  TS_access_point(TS_user,TS_provider) is

    by TS_user:

      T_CONNECT_req(TCEPI              : TCEP_identifier_type;
                    to_T_address       : T_address_type;
                    from_T_address     : T_address_type;
                    QOTS_request       : quality_of_TS_type;
                    options            : option_type;
                    TS_connect_data    : TS_connect_data_type);

      T_ACCEPT_req(TCEPI               : TCEP_identifier_type;
                   QOTS_request        : quality_of_TS_type;
                   options             : option_type;
                   TS_accept_data      : TS_accept_data_type);

      T_DISCONNECT_req(TCEPI              : TCEP_identifier_type;
                       TS_user_reason     : TS_user_reason_type);

      T_DATA_req(TCEPI                 : TCEP_identifier_type;
                 TSDU_fragment         : data_fragment_type);    (* note 3 *)

      T_EXDATA_req(TCEPI               : TCEP_identifier_type;
                   TS_expedited_data   : TS_expedited_data_type);
```

```
  by TS_provider:

    T_CONNECT_ind(TCEPI              : TCEP_identifier_type;
                  to_T_address       : T_address_type;
                  from_T_address     : T_address_type;
                  QOTS_request       : quality_of_TS_type;
                  options            : option_type;
                  TS_connect_data    : TS_connect_data_type);

    T_ACCEPT_ind(TCEPI               : TCEP_identifier_type;
                 QOTS_request        : quality_of_TS_type;
                 options             : option_type;
                 TS_accept_data      : TS_accept_data_type);

    T_DISCONNECT_ind(TCEPI                  : TCEP_identifier_type;
                     TS_disconnect_reason   : TS_disconnect_reason_type;
                     TS_user_reason         : TS_user_reason_type);   (* note 4 *)

    T_DATA_ind(TCEPI              : TCEP_identifier_type;
               TSDU_fragment      : data_fragment_type);   (* note 3 *)

    T_EXDATA_ind(TCEPI              : TCEP_identifier_type;
                 TS_expedited_data  : TS_expedited_data_type);

end TS_access_point;
```

```
message_buffer(user,buffer) is   (* note 5 *)

  by user:

    clear(in_fragment_size  : integer;
          out_fragment_size : integer);

    append(data_fragment : data_fragment_type);

  by buffer:

    get_next(data_fragment : data_fragment_type);

end message_buffer;
```

```
module TS (AP1,AP2:TS_access_point(TS_provider);
           buffer12,buffer21: message_buffer (user) );

var
   state1,state2: (idle,wait_for_acc,data_transfer,disconnect);
     (* major states (note 6) *)

    TS_reason:TS_disconnect_reason_type;     (* TS provided disconnection reason *)
    user_reason:TS_user_reason_type;         (* TS user provided disconnection reason *)
    TCEP1:TCEP_identifier_type;              (* local TS user identifier (note 4) *)
    TCEP2:TCEP_identifier_type;
    caller:T_address_type;                   (* TS address of AP1 (caller) *)
    called:T_address_type;                   (* TS address of AP2 (called) *)
    TCEP1_QOTS_estimate:quality_of_TS_type; (* quality of service requested by AP1 user *)
    TCEP2_QOTS_estimate:quality_of_TS_type; (* quality of service agreed by AP2 user *)
    options:option_type;                     (* option initially requested by AP1 user *)
                                               and finally agreed by AP2 user *)
    connect_data:TS_connect_data_type;       (* connect data sent by the calling (AP1) user
                                               during the connection establishment phase *)
    accept_data:TS_accept_data_type;         (* data returned by the called  (AP2) user
                                               during the establishment phase *)

(* Global constraints *)

Initialisation
       state1:=idle;
       state2:=idle;
```

```
ransitions

when (state1=idle) and (state2=idle) do
       when AP1.T_CONNECT.req( TCEPI,to_T_address,from_T_address,
                               QOTS_request,requested_options,TS_connect_data  )
              when ...  (* no congestion *) do
                      begin
                        state1:=wait_for_acc;
                        state2:=idle
                        TCEP1:=TCEPI;
                        caller:=from_T_address;
                        called:=to_T_address;
                        options:=requested_options; (* see note 10 *)
                        connect_data:=TS_connect_data;
                        buffer21.clear;
                        buffer12.clear;
                      end
              else (* congestion *)
                      begin
                        state1:=idle;
                        state2:=idle;
                        AP1.T_DISCONNECT_ind(TCEPI,...(* congestion (note 8) *),empty)
                       end;

hen (state1=wait_for_acc) and (state2=idle) do
       when ...  (* The connection request reaches the called user *) do
              begin
                state1:=wait_for_acc;
                state2:=wait_for_acc;
                TCEP2:=...; (* some unique identifier  *)
                TCEP2_QOTS_estimate:=...; (* see note 7 *)
                AP2.T_CONNECT_ind(TCEP2,called,caller,TCEP2_QOTS_estimate,
                                  connect_data,options)
              end;
       when ...  (* internal problem (note 9) *) do
              begin
                state1:=idle;
                state2:=idle;
                AP1.T_DISCONNECT_ind(TCEP1,...  (* congestion (note 8) *),empty)
               end;
       else;
```

```
when (state1=wait_for_acc) and (state2=wait_for_acc)
      when AP2.T_ACCEPT_req(TCEPI,QOTS_request,requested_option
                            (* see note 10 *),TS_accept_data)
           and(TCEPI=TCEP2) do
              begin
                state1:=wait_for_acc;
                state2:=data_transfer
                options:=requested_options; (* see note 10 *)
                accept_data:=TS_accept_data;
              end;
      when AP2.T_DISCONNECT_req(TCEPI, TS_user_reason)
           and(TCEPI=TCEP2) do
              begin
                state1:=wait_for_acc;
                state2:=disconnect;
                TS_reason:=TS_user_initiated_termination;
                user_reason:=TS_user_reason
              end;
      when ...  (* internal problem (note 9) *) do
              begin
                state1:=wait_for_acc;
                state2:=disconnect;
                TS_reason:=...;  (* note 8 *)
                user_reason:=empty;
                AP2.T_DISCONNECT_ind(TCEP2,TS_reason,user_reason)
              end;
      when ...  (* internal problem (note 9) - alternative transition *) do
              begin
                state1:=disconnect;
                state2:=wait_for_acc;
                TS_reason:=...; (* note 8 *)
                user_reason:=empty;
                AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason)
              end;
      else;
```

```
when (state1=wait_for_acc) and (state2=data_transfer)
        when ...  (* the accept indication reaches the caller *) do
                begin
                  state1:=data_transfer;
                  state2:=data_transfer;
                  TCEP2_QOTS_estimate:=...; (* note 7 *)
                  AP1.T_ACCEPT_ind( TCEP1,TCEP1_QOTS_estimate,
                                    options,accept_data)
                end;
        when AP2.T_DATA_req(TCEPI,TSDU_fragment)
             and ...  (* flow control to Transport Entity is ready *)
             and(TCEPI=TCEP2) do
                begin
                  state1:=wait_for_acc;
                  state2:=data_transfer;
                  buffer21.append(TSDU_fragment);
                end;
        when ...  (* internal problem (note 9) *) do
                begin
                  state1:=disconnect;
                  state2:=data_transfer;
                  TS_reason:=...; (* note 8 *)
                  user_reason:=empty;
                  AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason);
                end;
        when  ...(* internal problem (note 9) *) do
                begin
                  state1:=wait_for_acc;
                  state2:=disconnect;
                  TS_reason:=...; (* note 8 *)
                  user_reason:=empty;
                  AP2.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason);
                end;
        else;
```

```
hen (state1=data_transfer) and (state2=data_transfer) do
      when AP1.T_DATA_req(TCEPI,TSDU_fragment)
           and...  (* flow control to Transport Entity is ready *)
           and(TCEPI=TCEP2) do
              begin
                state1:=data_transfer;
                state2:=data_transfer;
                buffer12.append(TSDU_fragment)
              end;
      when buffer12.get_next(data_fragment)
           and (*flow control to user is ready *) do
              begin
                state1:=data_transfer;
                state2:=data_transfer;
                AP2.T_DATA_ind(TCEP2,data_fragment);
              end;
      when AP2.T_DATA_req(TCEPI,TSDU_fragment)
           and ...(* flow control to Transport Entity is ready *)
           and(TCEPI=TCEP1) do
              begin
                state1:=data_transfer;
                state2:=data_transfer;
                buffer21.append(TSDU_fragment);
              end;
      when buffer21.get_next(data_fragment)
           and (* flow control  to user is ready *) do
              begin
                state1:=data_transfer;
                state2:=data_transfer;
                AP1.T_DATA_ind(TCEPI,data_fragment);
              end;
```

```
when AP1.T_DISCONNECT_req(TCEPI,TS_user_reason)
     and(TCEPI=TCEP1) do
        begin
          state1:=disconnect;
          state2:=data_transfer;
          TS_reason:=TS_user_initiated_termination;
          user_reason:=TS_user_reason;
        end;
when  AP2.T_DISCONNECT_req(TCEPI,TS_user_reason)
      and(TCEPI=TCEP2) do
        begin
          state1:=data_transfer;
          state2:=disconnect;
          TS_reason:=TS_user_initiated_termination;
          user_reason:=TS_user_reason;
        end;
when ...  (* internal problem (note 9) *) do
        begin
          state1:=data_transfer;
          state2:=disconnect;
          TS_reason:=...; (* note 8 *)
          user_reason:=empty;
          AP2.T_DISCONNECT_ind(TCEP2,TS_reason,user_reason)
        end;
when ...  (* internal problem (note 9)  - alternative transition -  *) do
        begin
          state1:=disconnect;
          state2:=data_transfer;
          TS_reason:=...,(* note 8 *)
          user_reason:=empty;
          AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason)
        end;
else;
```

```
when (state1=disconnect) and ((state2=data_transfer) or (state2=wait_for_acc)) do
        (* the disconnect reaches the called user *)
        begin
          state1:=idle;
          state2:=idle;
          AP2.T_DISCONNECT_ind(TCEP2,TS_reason,user_reason)
        end;
else;
when ((state1=data_transfer) or (state1=wait_for_acc)) and  (state2=disconnect) do
        (* the disconnect reaches the calling user *)
        begin
          state1:=idle;
          state2:=idle;
          AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason)
        end;
else;
end;(*transitions*)
```

note 1 : An object of this type must be able to contain the network and country code (4 bytes) and the national subscriber number (12 bytes), which together form the Network address, and also possibly some space for subaddressing in the Transport layer

note 2 : A connection endpoint identification mechanism must be provided to allow a Transport Service user to distinguish between several Transport connections at the same Transport Service access point; this identification has local significance only

note 3 : Since TSDUs are of unlimited length, they may be exchanged over the Transport Service access point in several fragments; the maximum length of fragments is implementation dependent and may be different for different interfaces of a given Open System

note 4 : The TS_user_reason parameter is significant only when TS_disconnect_reason = TS_user_initiated_termination

note 5 : "clear" is a request to empty the buffer ;"append(fragment)" adds the data fragment after the data already in the buffer (if any); "get_next(fragment)" occurs when the buffer sends a data fragment to the Transport Entity.

ote 6 : The variables state1 and state2 are associated with the two access points AP1 and AP2 respectively. The major state of a TS module is thus defined by a pair <s1,s2> where s1 stands for the state as seen at AP1 and s2 for the state as seen at AP2.

note 7 : The quality of service value indicated to the user is not precisely defined.

note 8 : The value of TS_reason is returned to both users. The choice of the value is not defined formally.

note 9 : Some internal problem causes the termination of the connection.The problem could be due to a Transport Entity malfunction or to unrecovered problems of the Network Service.

ote 10: The acceptable set of values for the item "options" depend on the maximum class of Transport protocol available to both users.Only "normal" is allowed with class 0.

ANNEX 4

WASH-10

To: SC16 WG1 Rapporteur's group on FDT

Source: Canada

Title: Comments on a possible compromise on the syntax for extended state transition descriptions

1. Introduction

A possible compromise between TUB-11 and TUB-17 on the syntax of specifications in the extended state transition model could be as follows:

(1) specifications of the interactions: as in TUB-11

(2) overall form of transitions: as in TUB-17, i.e. a transition is of the form:

< next state > <---- < present state> < incoming interaction >
{and additional conditions >} begin < actions > end ;
The additional conditions are optional.

(3) references to the parameters of interactions: as in TUB-11, but no parameters are mentioned for the incoming interactions, references to these parameters in the conditions and actions use the parameter names defined in the interaction specifications (see point (1)).

2. Comparison of the possible compromise with the proposal in TUB-11.

Advantage of compromise:

(1) The first part of each transition contains the information about the major state and the incoming interaction in a more concise form.

Disadvantages of the compromise:

(1) Embedding of several transitions with similar conditions or the same incoming interaction is not possible. A result of this seems to be the need for distinguishing between different priorities of transitions (for normal operation and error processing), which is less important when embedded transitions can be specified.

(2) The syntax of the compromise gives the major state variable a special role, while in the syntax of TUB-11 the major state is represented by a variable without any special status. The results of this special role are the following:

(a) For service specifications this syntax is not very convenient, since one needs typically two major state variables (one for each end of the connection), and not one as provided by the syntax.

(b) Additional syntax rules are needed for the following points:
-- "same state"
-- sets of states (with TUB-11, the standard Pascal syntax may be used for this purpose)
-- declaration of possible major state values (if desired)

3. Some possible improvements for the syntax of TUB-11.

(1) For an action, write "begin...end when" instead of "do begin...end"

(2) Write "end when" instead of "else".

(3) Drop the parameter list for incoming interactions. The parameter names given in the interaction specification could be automatically made available for reference in the conditions and actions of the transition.

4. Importance of formatting

It is noted that a concise format for the major state information of a transition (similar as in the case of the "compromise") may be obtained by appropriate formatting, as in the following example:

```
when state = present and SAP.incoming

   and < additional condition >
     begin                              state := next_state;
       < additional actions >
     end when;
```

With the syntax of the compromise, this could be written in the form:

```
next_state < --- present : SAP.incoming
   and < additional condition >
   begin
      < additional actions >
   end;
```

5. Conclusion

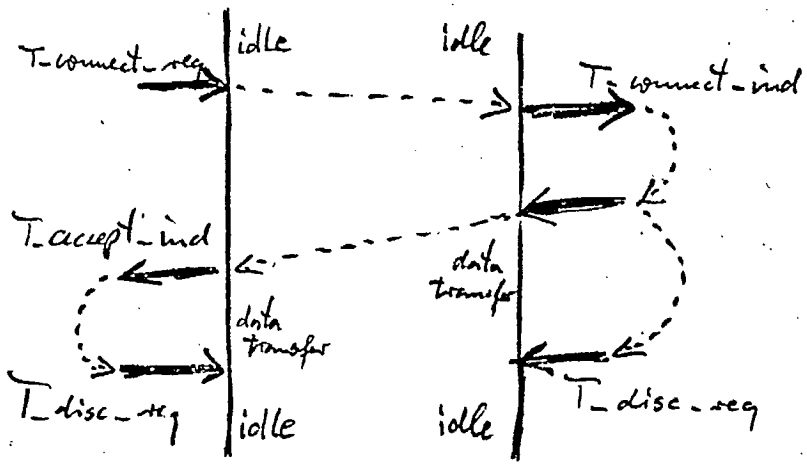
It seems that a syntax as described in section 3 above would be a better choice than the one described in section 1.

ANNEX 5

Time sequence diagrams as FDT

Time sequence diagrams seem to be a useful tool for describing service and protocols by examples of their behavior. This note proposes a notation for the use of such diagrams for service descriptions, and defines the meaning of the diagrams.

The notation and its meaning is explained with the follwoing example:



1. The two vertical lines represent two access points of the service.

2. The execution of service primitives is indicated by arrows; inward arrows represent requests, outward arrows represent indications.

3. The order of execution of service primitives is indicated by pointed arrows. A pointed arrow from primitive A to primitive B means that primitive B is executed after primitive A. The pointed arrows give a relation between the primitives in "logical time" (see Lamport, Comm. ACM). All relevant orders are indicated by pointed arrows.

4. Information about the local state of an access point may be indicated close to the vertical line representing the access point, between the arrows representing the execution of service primitives.

ANNEX 6

Syntax for linear form of FDT: comparison of ISO proposal and SDL-PR

1. Introduction

This document gives a comparison of the ISO syntax in FDT-16 for the linear form of the FDT, and the syntax of SDL-PR (see for example FDT-11, and Annex A to new Question 7/XI). This comparison could be the basis for an evaluation of the two.

An overview comparison is given in the sections 3 and 4 which contain the syntax of the two proposals with comparing annotations. The annotations are as follows:
-- The underlined terminal and non-terminal symbols have a correspondence in the other language (non-underlined symbols have no correspondense in the other language).

-- For those non-terminals that have a corresponding symbol in the other language with a quasi-identical meaning, the corresponding symbol is written on the left margin of the paper, where the symbol is defined. Where such a symbol is used, without being explicitely defined, the corresponding symbol is written close to it with an arrow indicating the correspondence. For example, <process> and <module type definition> correspond to one another, as well as "TO <process name>" and <interaction point id>.

Enumeration of differences:

1. Design goals:

The design goal for the ISO syntax was to use as much as possible the elements of standard Pascal. Pascal is a high-level modern programming language. Its design goals were, among others, simplicity, ease of use for program design and implementation, and the support of the "structured programming" approach.

The design goals for SDL-PR are (see Annex A to new Question 7/XI, section 5.1):
"This Recommendation defines a program-like form of the SDL (SDL/PR) whose primary purpose us to allow the mecanical production of graphical SDL presentations (SDL/GR). ..."

It therfore seems that SDL-PR is primarily designed to lead to comprehensive specifications in the graphical form, while the ISO syntax is designed to lead to comprehensive specifications in the linear form.

2. Comments

The ISO syntax allows comments, written "(* <text phrase> *)", anywhere between
symbols of a specification. The SDL-PR syntax allows comments only in certain places of a specification, written "COMMENT <text phrase>" (for details see section 4).

3. Different keywords (reserved identifiers)

The keywords chosen for delimiting the different elements of a specificati are different in the two languages. A comparison of corresponding keywords is given by the annotations in sections 3 and 4.

4. Identifiers -- names

To identify different elements of a specification (e.g. interactions, interaction points, processes, etc.) the ISO syntax uses "identifiers" (i.e. character strings without spaces nor special characters (except "_") while SDL-PR in addition uses "names" (arbitrary character strings). It is noted that the SDL-PR "names" are the text to be printed inside the corresponding graphical symbol in the SDL-GR form of the specification

5. Interaction points vs. process names

The source and destination of interactions (i.e. signals) are indicated by refering to the interaction points declared in the module (i.e. process) specification in the case of the ISO syntax, while they are indicated by refering to the names of the processes (i.e. module occurences) in the case of SDC-PR. A result of this is that the specification of a process in SDL-PR must be changed when the names of the other processes in the system are changed.

6. State sets

The ISO syntax supports the concept of "state sets" which means that it is possible to defined a single transition that is enabled in a certain situation (for example a given input, or a particular internal condition) for a certain set of (major) states (for example in all data transfer states, there may be several such states). This concept is not supported by SDL-PR.

7. Priority

The ISO syntax supports the distinction of several levels of priority for different transitions, for example, high priority transitions for detection of PDU coding errors, and low priority transitions for normal processing (the coding errors may then be ignored in the specification of the normal processing transitions). This concept is not supported by SDL-PR.

8. PROVIDED

The ISO syntax uses the keyword PROVIDED to define a condition (depending usually on input parameters and/or state variables) that must be satisfied when a transition is executed. In SDL-PR such a situation can usually be modeled by a DECISION. However, the PROVIDED may also be used to introduce non-determinism into the specification. In SDL-PR, for this purpose, fictitious inputs must be assumed.

9. GOTO programming

The assignment of the next (major) state in the state transition model (of both proposals) is a kind of "GOTO programming" (in the sense of the GOTO statement in many programming languages, which leads to a new program (control) state). (It is noted that "GOTO programming" is considered harmful for the design of easily understandable programs by most experts in software engineering). It seems that, in addition, the GOTO statement of Pascal is not needed in the case of the ISO syntax for the specification of communication services or protocols. In the case of SDL-PR, use of GOTO programming is encouraged (by use of the constructs JOIN and LABEL)due to the flow chart approach of structure of the DECISION construct.

10. Possible next states

In SDL-PR several different (major) states may be possible next states after a given input signal, if use of the DECISION construct is made. In the case of the ISO syntax, there is only one possible next (major) state after a given transition (the next major state is indicated by the TO construct). It is noted that there is in SDL no clear correspondence to the transition concept of the ISO syntax). This restriction (of a single next state) was introduced on purpose. It could be avoided by allowing explicit assignments to the STATE variable in the BEGIN ... END statement of the transition.

11. Additional concepts in the ISO syntax

A number of concepts are supported by the ISO syntax, which are not considered in SDL-PR, such as
-- definitions of interactions and their parameters
-- definition of types of interaction points (<channel definition>)
-- additional state variables
-- data types
-- procedures and functions
-- initialisations
As shown in FDT-11, these concepts (with the exception of the interaction points, see difference 5) can be added to SDL-PR without affecting much the other elements of the language.

Section 3

Annex to minutes of Subgroup B meeting in Washington, Sept. 1981

Working Draft

Syntax of an extended state transition specification language

Notation: Extended BNF where "+" means one or more occurrences, "*" means zero, one or more occurrences of an expression, and "|" separates alternatives . "**" means that the construct is the same as in Pascal.

<module>::=<channel definition>+<module type definition> (*Note9*)

<channel definition>::=<constant definitions>**
<type definitions>**<channel type def>

The possible interactions at a given type of interaction point are enumerated by a definition of the following form:

<channel type def>::=INTERACTION <channel type id>
(<role list>) <interactions> ;

<role list> ::= <role id>
| <role list> , <role id>

<interactions>::=<BY clause>
| <interactions> <BY clause>

<BY clause> ::= BY <role list> : <interaction list>

<interaction list> ::= <interaction>

| <interaction list> <interaction>

<interaction> ::= <interaction id> <interaction parameters> ,

The declaration of <interaction parameters> is in the same form as function parameter declarations in Pascal (i.e. for each parameter its name and type).

<interaction id> ::= <identifier> (*Note1*)

<channel type id> ::= <identifier>

The definition of a module type contains the declaration of all abstract interaction points through which a module of this type interacts. This includes the service access points through which the communication service is provided as well as the system interface for timers, etc. and the access point to the layer below, through which the PDU's are exchanged. The following syntax is proposed:

<process>

<module type definition>::=MODULE <module type id>

PROCESS (<interaction points>) ;

<module body>

<interaction points> ::= <interaction point declaration>

| <interaction points> ; <interaction point declaration>

<interaction point declaration> ::= <interaction point id> :

<channel type id>

(<role id>)

The <role id> indicates which role the entity plays as far as the declared interaction point is concerned. We note that the distinction of these roles permits the checking that the invocation of interactions in the conditions and actions of transitions is consistent with the possible exchanges defined in the channel definition.

<module body> ::= <label definitions>**

<constant definitions>**

<type definitions>**

<variable declarations>**

<major state declaration> (* this has the form of a Standard Pascal variable declaration *)

<state set definition>*

<procedure and function definition>** (*Notes 2 and 3*)

<initialization>

<transition>+

END.

↖ ENDPROCESS <process name>

<major state declaration> ::= STATE : <enumeration type>**;

<state set definition> ::= <state set id> = <set definition>**

(*Notes4*)

<initialization> ::= <state initializer> <begin statement>** ;

<state> <transition> ::= <transition part>+ <begin statement>** ; (*Note5*)

<transition string>

<transition part> ::= FROM <major present state>

<STATE

| TO <major next state>

<NEXTSTATE

| WHEN <interaction point id> . <interaction id>

<INPUT FROM<process name> (*Notes 6,7*) <signal name>

| PROVIDED <expression>** (*must be boolean*)

*part of a <decision>

| PRIORITY <priority indication>

<priority indication> ::= <identifier>** (*constant of some enumeration type*)

| <integer>**

<major present state> ::= <major state value>

| <state set id>

<major next state> ::= <major state value>

| SAME

<state name> <major state value> ::= <identifier>** (*must be element of the enumeration type of the <major state declaration*)

<output> <output statement>::=<interaction point id> . <interaction id>

TO <process name> <signal name>

<effective parameter list>** (*Note8*)

Note 1: Alternatively, the form of an <interaction id> could indicate whether the interaction is, for instance, a request, indication, response, or confirmation (for further study).

Note 2: The body of a procedure or function that is implementation dependent (not defined by the specification) is written in the form "PRIMITIVE" or "...".

Note 3: A boolean function X(<parameters>) with no side effects may be declared in the form "predicate X(<parameters>)".

Note 4: The elements of the set must be included in the enumeration type of the <major state declaration>.

Note 5: There should be at most one of each parts defined in the rule below. And there should be at least the parts FROM, TO, and WHEN. A consistent ordering throughout a specification is desirable. The possibility of nesting is for further study.

Note 5a: Within a transition, "..." may be written for an expression that is implementation dependent (not defined by the specification).

Note 6: The possibility of spontaneous transition, i.e. without an input (WHEN part) is for further study.

Note 7: To refer to the input parameters the parameter identifiers of the interaction in the <channel type definition> are used.

Note 8 This kind of statement (for producing an output interaction) is an extension of Pascal.

Note 9: Only the definition of a module type is given here. Language elements for the declaration of module instances within a system and their interconnection is for further study.

Section 4

ANNEX

Representation of the SDL/PR Syntax in the Backus Naur Form *

A.1 Preliminaries to the Backus Naur Form Representation

The context-free syntax is defined in this Annex by a context-free grammar using an extension of the Backus Naur Form (1).

Syntactic categories are indicated by one or more English Words, typed in italic characters, enclosed between the angular brackets <and>. This is called a non terminal element. Each non terminal category is defined by an expression of terminal and non-terminal elements on the right hand side of the symbol ::=.

Sometimes a non-terminal element includes an underlined part. This underlined part does not form part of the context-free description, but defines a semantic sub-category. For example: <state name> is identical to <name> in the context free sense, but semantically it defines only names of the sub-category "state".

Syntactic elements may be grouped together by using the curly brackets {and}. Repetition of a curly bracketed group is indicated by an asterisk (*) or plus (+). An asterisk denotes that the group may be repeated zero or more times, a plus that the group may be repeated one or more times. {A} * stands for any sequence of A's including zero, while {A} + stands for any sequence of at least one A. If syntactic elements are grouped using the square brackets [and], the group is optional. Groups of syntactic elements may be separated by vertical bars | ; this represents alternative groups any one of which may be chosen.

The lexical elements of SDL/PR are: the keywords, identifiers (<ident>), text strings (<text>) and note strings (<note>). Spaces may be used to delimit the lexical elements of SDL/PR. For instance BLOCKCALLHANDLING will be taken as one identifier instead of the start of a functional block identified as CALLHANDLING. Contiguous spaces have the same delimiting effect as a single space.

A note may be inserted at all places in a program where spaces are allowed as delimiters. A note has the same delimiting effect as a space.

Other delimiters are the colon, the semicolon and the comma.

A.2 Syntax of the basic SDL/PR

<functional block> :: = BLOCK <block name> [<comment>]; {<process>}
+ ENDBLOCK <block name> [<comment>];

<process> :: = PROCESS <process name> [<comment>]; {<state>} +
ENDPROCESS <process name< [<comment>];

<state> :: = STATE <state name> [<comment>]; [<state picture>]
{<save> | <input>} * ENDSTATE <state name>
[<comment>];

---

* This annex is taken from CCITT Study Group XI, Contribution No. 1, Period 1981-1984, pp. 73-75, Appendix A to Draft Recommendation Z.105.

WHEN <interaction id>

<input> :: = INPUT <signal name> {, <signal name>} * [INTERNAL]
[FROM <process name>] [<comment>]; <transition string>

<interaction point id> {<join> | <next state>}

<save> :: = SAVE <signal name> {, <signal name>} *
[FROM <process name>] [<comment>];

Pascal BEGIN statement

<transition string> :: = {[<label>] <transition element>} *

<transition element> :: = <task> | <output> | <null task> |
<decision>

Pascal statement

<task> :: = TASK <task name> [<comment>];

<null task>:: = <comment>;

<interaction id>

<output statement>

<output> :: = OUTPUT <signal name> [INTERNAL]
[TO <process name>] [<comment>];

<interaction point id>

Pascal if and case statements

<decision> :: = DECISION <decision name> [<comment>];
(<result name> {, <result name>} *)
: <transition string> [<join> | <next state>]
{(<result name> {, <result name>} *)

: <transition string> [<join> | <next state>]} +

END DECISION;

GOTO

Pascal GOTO statement

<join> :: = JOIN <label ident> [<comment>];

<nextstate> :: = NEXTSTATE <state name> [<comment>];

TO <major next state>

Pascal label definition

<label> :: = <label ident> :

<name>:: = {<ident> [<text>] | <text>}

<comment> :: = COMMENT <text>

(* *)

<text> :: = ' <text phrase> '

<text phrase> :: = {char} +

<identifier>

<ident> :: = {<letter> | <digit> | —} +

<state picture> :: = see section 5.6

<char> :: = <letter> | <digit> | <specials>

<specials> :: = +|—|*|/|(|)|>|<|=| |:|;|—|,

<letter> :: = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> :: = 0|1|2|3|4|5|6|7|8|9

ANNEX 7

International Telegraph and Telephone
Consultative Committee
(CCITT)

Period 1981-1984

Original: English

Question : 39/VII

Date: December 1981

STUDY GROUP VII - CONSTRIBUTION No.

SOURCE: CANADA

TITLE : PROPOSAL FOR A PROGRAMME-LIKE FDT.

1.0 Considering that;

1.1 Different approaches to the development of a programme-like FDT were discussed during the last Rapporteurs meeting on Q39/VII at Ottawa during Oct 1981,

1.2 The ISO TC97/SC16/WG1 ad-hoc group on FDT has develloped, after two years of intensive study, a programme-like FDT for communication protocols and services* which is based on the PASCAL programming language, and seems suitable for the purposes of SGVII's FDT requirements,

1.3 There are important advantages in adopting the same FDT in CCITT and ISO,

1.4 The adoption of a FDT based on PASCAL has many other advantages, as explained in the attached ANNEX 7 of the minutes of the last Rapporteurs meeting on Q 39/VII.

1.5 The systematic translation from a programme-like FDT given in the form proposed by ISO into graphical form, and in particular into graphical SDL, is possible as discussed in ANNEX B of the recent ISO document.

2.0 It is proposed that the CCITT and ISO adopt a common programme-like FDT based on PASCAL along the lines of the present ISO proposal*.

* ISO TC97/SC16/WG1 - ad hoc group on FDT, Subgroup B. - Title: A FDT based on an extended state transition model (Working Draft, Boston, Dec. 1981)

ANNEX 8

International Telegraph and Telephone
Consultative Committee
(CCITT)

Period 1981 - 1984

Original: English

Question : 39/VII

Date: December 1981

STUDY GROUP VII - CONTRIBUTION No.

SOURCE: CANADA

TITLE: PROPOSAL ON DIFFERENT FORMS OF FDT

---

1. Different forms of FDT are useful, such as graphical and programming-language-like forms. Some are intended for giving overviews of specifications, or may be used for both overviews and complete specifications.

2. A complete specification in the programming-language-like form should always be given and should be regarded as the authoritative specification.

3. For the programming-language-like specifications of services an approach similar to the one for protocols should be used. A possible extension for describing the local rules for the execution sequences at service access points is described in the annex.

4. The use of time-sequence diagrams, as explained in section 2 of Annex 5 of the report of the last Rapporteurs meeting on Q 39/VII should be retained for overviews of typical interaction sequences.

5. Common state diagrams, similar as used in X.25, should be retained for overviews of protocol and service specifications.

6. SDL-GR seems appropriate for a more detailed graphical specification than is possible with state diagrams.

.../2

Annex: Specifying local rules for interactions at an access point

The example below shows how the elements of the program-like FDT developed by ISO* could be used to specify the local rules that determine in which order the service primitives may be executed at one given access point. The syntax of this example assumes that the symbol <interactions> in the syntactic rule for <channel type def> in the FDT* is replaced by the two symbols <interactions> <constraints>, and the new symbol <constraint> is defined by

```
<constraint> ::= empty
               | <module body>
```

It is also assumed that the <block> of a <transition> is optional.

The example below is the specification of a Transport service access point (for a single connection, for simplicity). The first part of the specification defines the service primitives with their parameters, while the second part defines the order in which they may be executed at an access point, using the state transition model. (For instance, the first transition reads: From the "idle" state a "T_CONNECT_req" interaction will lead to the "wait_for_conf" state).

It is important to note that such a specification is not a complete service specification. A service specification should include the information given here, as well as the global end-to-end properties of the service which relates the interactions taking place at different access points.

* ISO TC97/SC16/WG1 - ad hoc group on FDT, Subgroup B. - Title: A FDT based on an extended state transition model (Working Draft, Boston, Dec. 1981).

Example

```
interaction

  TS_access_point(TS_user,TS_provider) is

    by TS_user:

      T_CONNECT_req(TCEPI            : TCEP_identifier_type;
                    to_T_address     : T_address_type;
                    from_T_address   : T_address_type;
                    QOTS_request     : quality_of_TS_type;
                    options          : option_type;
                    TS_connect_data  : TS_connect_data_type);

      T_ACCEPT_req (etc.

      T_DISCONNECT_req(etc.

      T_DATA_req(etc.

      T_EXDATA_req(etc.
```

```
    by TS_provider:
      T_CONNECT_ind(etc.
      T_ACCEPT_ind(etc.
      T_DISCONNECT_ind(etc.
      T_DATA_ind(etc.
      T_EXDATA_ind(etc.

var state : (idle, wait_for_conf, wait_for_response, data_transfer);
from idle
    when T_CONNECT_req      to wait_for_conf;
    when T_CONNECT_ind      to wait_for_response;
from wait_for_conf
    when T_DISCONNECT_ind   to idle;
    when T_ACCEPT_ind       to data_transfer;
from wait_for_response
    when T_DISCONNECT_req   to idle;
    when T_ACCEPT_req       to data_transfer;
from data_transfer
    when T_DATA_req         to same;
    when T_DATA_ind         to same;
    when T_EXDATA_req       to same;
    when T_EXDATA_ind       to same;
    when T_DISCONNECT_req   to idle;
    when T_DISCONNECT_ind   to idle;
end.
```

ANNEX 9

FDT-47

International Telegraph and Telephone
Consultive Committee
(CCITT)

CAN COM VII/42
Original: English

Period 1981-1984

Date: February 1982

Question : 39/VII

For submission to the SG VII Rapporteurs meeting on FDT, Melbourne

Title: The translation of the ISO linear FDT into graphical SDL

Source: CANADA

1. Introduction

At the last Rapporteur's meeting on FDT in Ottawa it was concluded that it would be desirable to adopt the ISO proposal for a linear FDT (working document of subgroup B of the ad hoc group on FDT of TC97/SC16/WG1) for the linear form of FDT, and to consider SDL as a possible graphical form. This paper shows how a specification given in the linear form of ISO can be translated into a graphical representation using SDL. It is noted that this translation is already considered in Annex B of the ISO working document (Dec. 1981). The same approach is used in the following, and more detailed considerations are given. It is assumed that the graphical form is to be as complete as possible. In the case that only overview information is required, the traditional state diagrams, as in X.25, seem to be a preferable graphical representation.

2. A control structure convention simplifying the translation

The following structure simplifies the translation. (It is noted, however, that this structure does not necessarily represent the best structure for obtaining readable specifications; sorting the transitions by input interactions (WHEN clauses) may be preferable):

The different transitions of the specification are sorted by
- major present state (FROM clause),
- input interaction (WHEN clause),
- additional conditions (PROVIDED clause)

and then contain the TO clause (next major state) and the <block>. It is assumed in the following that the <block> does not contain loops nor GOTO statements. It is not clear where the information about variables and procedures declared within the <block> (if such exist) should be displayed in the graphical form.

3. The translation process

The basic approach to the translation process and a simple example are descibed in the ISO working document. The following considerations are added.

3.1. The SDL decision symbol is used for the following purposes:

(a) to represent the different choices that are described by different transitions (in the linear form) having different PROVIDED clauses, but otherwise identical conditions (present state value and input interaction).

(b) To represent the different choices of a Pascal CASE or IF statement within the BEGIN .. END part of the transition <block>.

3.2. The BEGIN ... END part of the transition <block> must be parsed by the translator and the following actions must be performed, depending on the statements encountered:

(a) An output statement is to be translated into an SDL output symbol. The text within the symbol could simply be the text of the output statement.

(b) A sequence of statements which are neither output nor IF or CASE statements, are translated into a single SDL action symbol. The text within the symbol could simply be the text of the encountered statements.

(c) An IF or CASE statement is translated into an SDL decision symbol, where the different alternate statements of the IF or CASE statement are translated into different branches following the decision symbol. The branches could be annoted by "TRUE" and "FALSE", or the disciminant values of the CASE variable.

(d) At the end of the translation of the BEGIN ... END part of a transition, an SDL state symbol is added to each open branch of the resulting SDL diagram, which represent the next major state of the system. The symbol should be annotated by the state value given in the TO clause of the linear specification.

4. The inclusion of informal specification elements

A distinction is made in the ISO working document (section 4.1.4 point a) between informal specification elements that are part of the specification (although expressed in an informal, natural language format), and a comment that is merely some text which is only included for ease of understanding. This distinction seems to be an important one. In SDL such a distinction has not been made; most texts written within SDL symbols are effectivly informal specification elements, since they are written in natural language.

If informal specification elements are written with the linear syntax # <informal specification element text> # then the information of the SDL diagram below (taken from CCITT Rec. Q.704, figure 8) may be represented in the linear form as follows.

```
module signalling_route_management (
        to_STP : .....
```

```
        to_signalling_traffic_managment : .....
        tester : ..... );
var
   state : (idle);

from idle to idle
   when to_STP. # transfer prohibited #
      begin
         to_signalling_traffic_management. # signalling route unavail#;

         # start route set test procedure #
         end;

   when to_STP. # transfer allowed #
      begin
         to_signalling_traffic_management. # signalling route avail#;

         # stop route set test procedure #
         end;

   when tester. # route set test #
      begin
         if # route available #
         then tester. # transfer prohibited #
         else tester. # transfer_allowed #
         end;
```

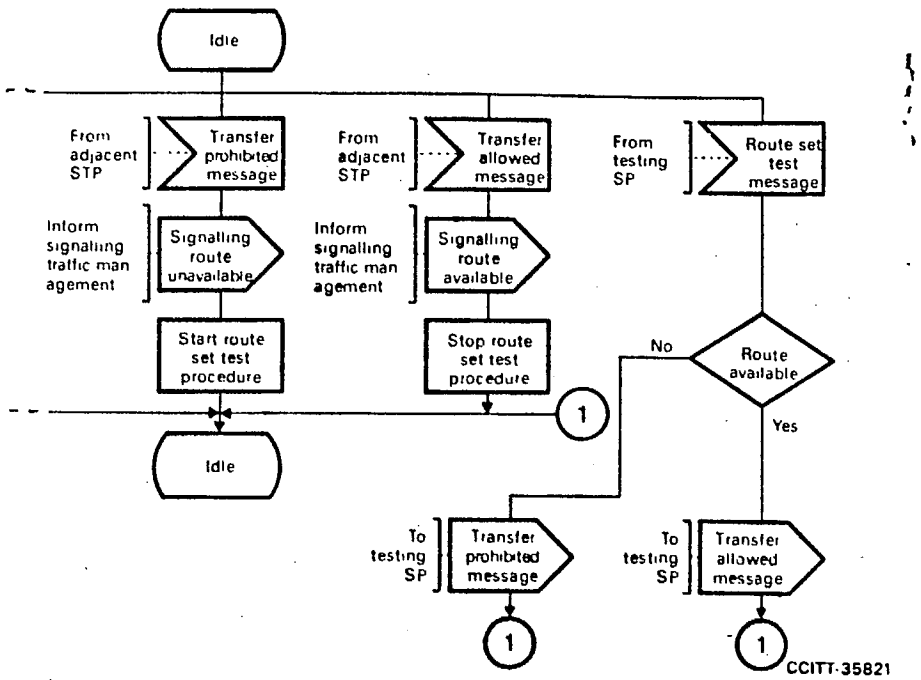


FIGURE 8/Q.704

route management overview diagram

ANNEX 10

FDT-48

International Telegraph and Telephone
Consultive Committee
(CCITT)

Period 1981-1984

CAN COM VII/43

Original: English

Date: February 1982

Question : 39/VII

For submission to the SG VII Rapporteurs meeting on FDT, Melbourne

Title: A method for specifying module interconnections

Source: Canada

1. Introduction

The working paper of Subgroup A of the ad hoc group on FDT of ISO TC97/SC16/WG1 (N....) identifies the following elements of formal specifications for communication protocols and services:
(a) enumeration of possible interaction primitives (section 2.2.1),
(b) specification of possible execution sequences (section 2.2.2), and
(c) specification of interaction points of modules and their interconnecions (section 3.3).

It is noted that the subgroups B and C of the ISO ad hoc group on FDT work on different approaches to point (b); and the approach of the CCITT SG VII on FDT is related to the one of the ISO Subgroup B. It is desirable that unique approaches to (a) and (b) could be developed by ISO and CCITT which are compatible with the different approaches for point (b).

It is noted that the syntax developed by Subgroup B contains some elements for the specification of interaction points (this is part of point (c)).

The other aspect (the interconnection of modules) is usually represented in graphical form by diagrams, such as shown in figures 1 and 2. This paper presents a possible linear form for such specifications which could be useful for certain purposes. The application of this linear specification technique to the OSI Reference Model, as shown in figure 1, is also given.

2. A possible syntax for specifying module sub-structure and module interconnections

For the specification of a module type, the syntax of section 4.1.2 of the Subgroup B working document is assumed. For the specification of a refinement (implementation) of a module in terms of a number of sub-modules and an appropriate interconnection of these sub-modules, the following syntax may be used.

```
REFINEMENT <name of refinement> FOR <name of refined module type>
      IS
   <list of sub-modules>
   INTERNAL CONNECTION  <list of internal connections
                            between sub-modules>
   EXTERNAL CONNECTION  <list of connections of sub-modules
                            to ports of refined module>
   END;
```

Each sub-module is declared as
<name of sub-module occurence> : <name of sub-module type>
Each connection is written in the form
<(sub-) module name> . <name of interaction point> =
<name of other (sub-) module> . <name of interaction point>
An example is given in the following section.

3. Linear form for the structure shown in figure 1

3.1. Introductory comments

Figure 1 shows the structure of the OSI Reference Model as far as the Transport layer is concerned. Similar diagrams (or linear forms) could be used to describe all seven layer of the Model. The linear specification given below uses the Transport and Network service specification, and the Transport protocol specification, which are assumed to be given in the form of specifications for module types named "TS", "NS", and "TP", respectively.

It is noted that the multiplexing allowed in the Transport layer, and the undefined relation between the Transport and Network addresses makes the specification below relatively complex.

3.2. Linear specification of the structure of the Transport layer

```
refinement ISO_TS_provider for TS is
   entities : array [entity_id_type] of TP;
   NS_provider : NS;
   internal connection
      for id in entity_id_type, N_addr in N_address_type
            such that ... (* the entity "id" uses the NSAP
                             identified by the Network address
                             N_addr *)
         entities [id]. NSAP [N_addr] = NS_provider. AP [N_addr]
         (* property: at most one entity connected to each
               access point (AP) of NS_provider *)
   external connection
      for id in entity_id_type, T_addr in T_addr_type
            such that ... (* the entity "id" services the TSAP
                             identified by the Transport address
                             T_addr *)
         entities [id]. TSAP [T_addr] = TS. AP [T_addr]
         (* property: at most one entity connected to each
               access point (AP) of the refined module "TS" *)
```

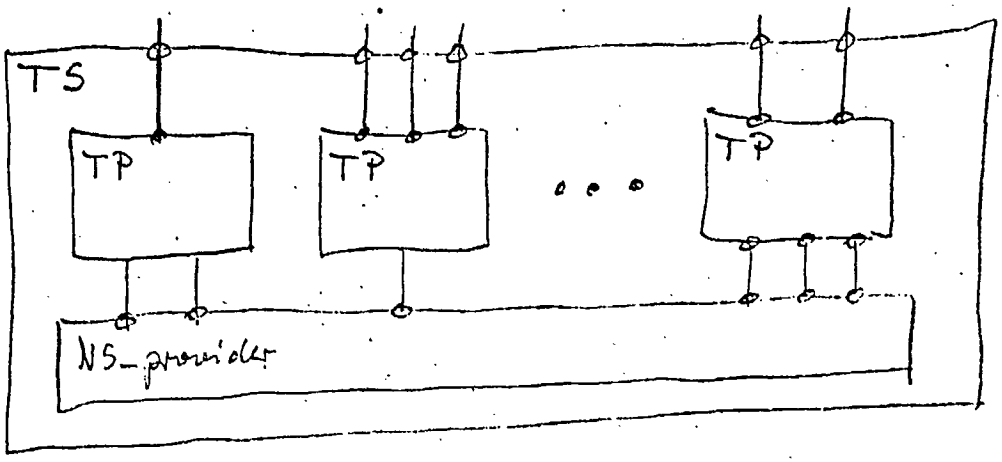


Figure 1

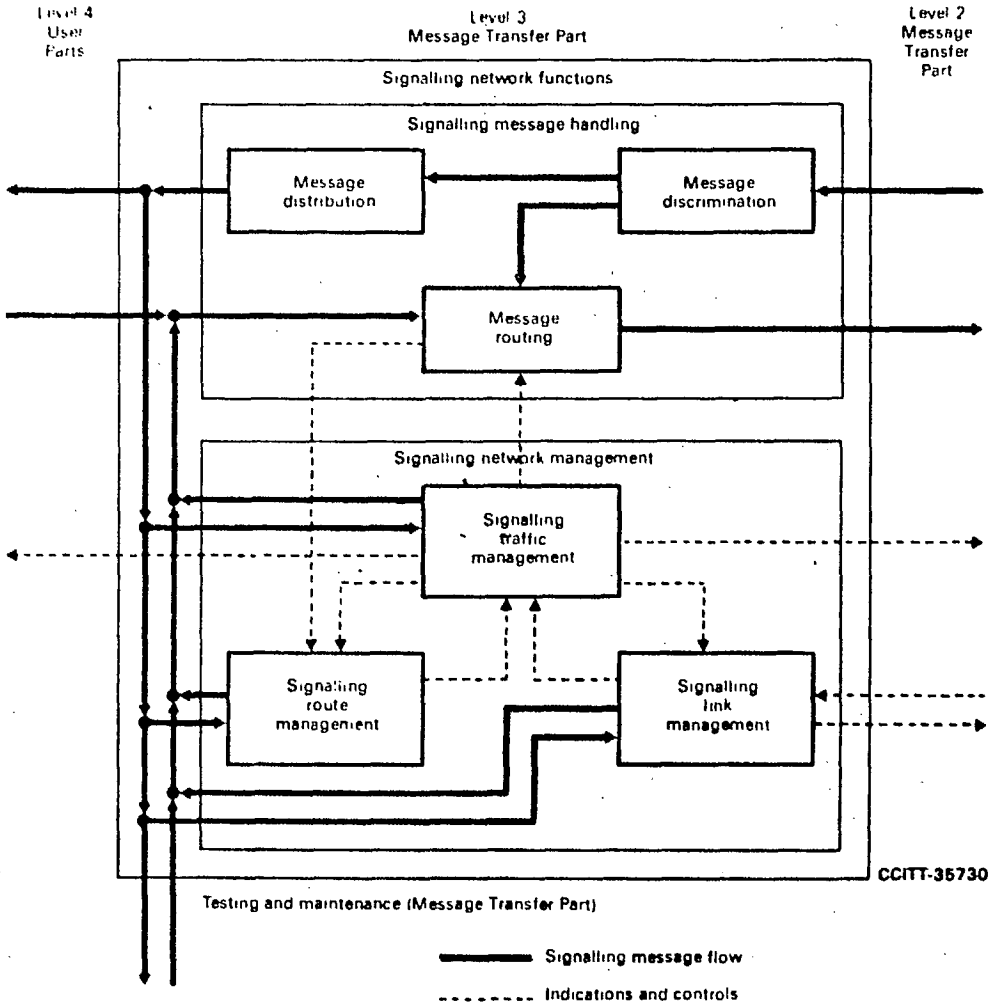


FIGURE I/Q.704

Signalling network functions

Figure 2

ANNEX 11

FDT-49

International Telegraph and Telephone
Consultive Committee
(CCITT)

CAN COM VII/41

Original: English

Period 1981-1984

Date: February 1982

Question : 39/VII

For submission to the SG VII Rapporteurs meeting on FDT, Melbourne

Title: Examples of Transport protocol specifications

Source: Canada

1. Introduction

The annexes contain specifications of the Transport protocol (classes 0 and 2) using the formal description technique (FDT) defined by ISO TC97/SC16/WG1 ad hoc group on FDT (working document December 1981, Subgroup B). The specifications are based on (informal) protocol description which is similar (but not identical) to the latest CCITT/ISO protocol description for classes 0 through 4. The purpose of this document is to show the application of the FDT to Transport protocol specifications.

Annex 1 contains a class 0 protocol specification for a single Transport connection. This specification is kept relatively simple, and certain aspects, such as the mapping of the TPDU into the Network service data units are not specified. The specification is a adaptation of the protocol specification given in the paper FDT-2 of the last Rapporteurs meeting in Ottawa, and the notes referenced in the specification can be found in that document.

Annex 2 contains a relatively complex protocol specification, including the handling of many simultaneous connections, and multiplexing of several Transport connections into a given Network connection. The following sections contain some additional remarks on certain aspects of the specification.

2. Some comments on the specifications

2.1. Structuring by functions or phases

The different transitions of the specification are grouped by functions and/or phases. The grouping has been chosen in an arbitrary fashion. More study is needed to determine which kind of grouping gives rise to most readable specifications. In particular, all PDU receiving transitions have been grouped together, in order to obtain a more compact specification. It may, however, be preferable to distribute these transitions with the other groupes of functions and phases.

2.2. Major states

In Annex 2, since the handling of many simultaneous connections is described, there is one "major" state per connection. The ISO syntax foresees only a single "major" state. Therefore the "major" states of the connections are handled as ordinary state variables, which are replicated in an array for each connection. (The same is done for the states of the used Network connections).

2.3. A possible definition of the meaning of the "FROM" and "TO" clauses of the FDT

In order to overcome the problem mentioned under point 2.2 above, the application of the "FROM" and "TO" clauses defined in the ISO working document could be generalized, and its meaning could be defined by the following equivalence rules:

(1) A "FROM <major state value>" clause is equivalent to the clause "PROVIDED state = <major state value>" or the addition of "and state = <major state value>" in an already existing PROVIDED clause.

(2) A "FROM <state set id>" clause is equivalent to the clause "PROVIDED state in <state set id>" or the addition of "and state in <state set id>" in an already existing PROVIDED clause.

(3) A "TO <major state value>" clause is equivalent to the Pascal statement "state := <major state value>;" to be included as an additional statement in the BEGIN ... END part of the <block> of the transition.

(4) A "TO SAME" clause is equivalent to a "no operation" being added to the <block> of the transition (i.e. no change).

Since these rules define the meaning of the "FROM" and "TO" clauses in terms of Pascal expressions and statements, their meaning is defined in terms of the meaning of Pascal.

2.4. The possible use of "FROM" and "TO" clauses in the specification of Annex 2

Using the equivalences defined above, the "FROM" and "TO" clauses may be used in many places of the formal specification given in Annex 2, instead of the equivalent Pascal expressions and statements used in the present version. These places are indicated by a vertical line on the right margin of the specification.

A N N E X 1

Title : Specification of class 0 Transport protocol for a single connection

Introductory comments
****

The Transport protocol specification given below uses the notation of the FDT proposed in "Tutorial on formal description techniques (FDT)" as referenced above.

Only a single Transport connection is considered. It is assumed that the interactions specified always refer to a particular Network and Transport connection which are not explicitely identified. A specification of the explicit handling of several connections, possibly over different Network and Transport access points, is given in Annex 2.

For the data transfer phase, flow control at the Network layer and Transport layer interfaces is considered. However, it is only specified informally, since the specification of the Transport service (and equally the Network service) does not include explicit service primitives for flow control. If such primitives are added to the service specifications, the flow control could be specified formally within the same formalism.

The specification below defines the "logical behavior" of a Transport entity in terms of its interactions through the exchange of protocol data units and service primitives. It does not, however, specify how the protocol data units are mapped into the service primitives of the Network layer. Some of these aspects are specified in Annex 2.

It is noted that the choice of data types for the parameters of service primitives and protocol data units is mainly oriented towards a simple logical structure of the data, and not towards the way this information may be coded as protocol data units within the service data units of the Network layer, or as interface data units depending on the implementation of the Open System.

Since the protocol specification refers to the Transport service specification, the list of Transport service primitives and the type definitions for their parameters are given below.

```
Transport Service Specification
**********************************

const
  undefined = ...;    (* note 1 *)

type
  T_address_type = ...;    (* note 2 *)

  TCEP_identifier_type = ...;    (* note 3 *)

  quality_of_TS_type = record
    throughput_from_average    : integer;    (* bps *)
       .....etc.....
                        end;

  option_type = (normal,fast_connect_disconnect,with_expedited);

  TS_connect_data_type = ...;    (* string of octets of limited length *)

  TS_accept_data_type = ...;    (* string of octets of limited length *)

  TS_expedited_data_type = ...;    (* string of octets of limited length *)

  fragment_length_type = ...;    (* implementation dependent *)

  data_fragment_type = record
    end_of_TSDU : boolean;
    length : fragment_length_type;    (* length of string *)
    data : ...;    (* string of octets *)
                        end;

  TS_disconnect_reason_type = (TS_user_initiated_termination,
                               lack_of_local_resources,
                               inability_to_provide_the_quality,
                               misbehavior_of_TS_user,
                               reference_overflow,
                               mismatched_reference,
                               local_congestion,
                               remote_congestion,
                               ...);    (* note 4 *)

  TS_user_reason_type = ...;    (* string of octets of limited length *)
```

```
interactions

  TS_access_point(TS_user,TS_provider) is

    by TS_user:

      T_CONNECT_req(TCEPI              : TCEP_identifier_type;
                    to_T_address       : T_address_type;
                    from_T_address     : T_address_type;
                    QOTS_request       : quality_of_TS_type;
                    options            : option_type;
                    TS_connect_data    : TS_connect_data_type);

      T_CONNECT_resp(TCEPI              : TCEP_identifier_type;
                     QOTS_request       : quality_of_TS_type;
                     options            : option_type;
                     TS_accept_data     : TS_accept_data_type);

      T_DISCONNECT_req(TCEPI            : TCEP_identifier_type;
                       TS_user_reason   : TS_user_reason_type);

      T_DATA_req(TCEPI             : TCEP_identifier_type;
                 TSDU_fragment     : data_fragment_type);   (* note 5 *)

      T_EX_DATA_req(TCEPI              : TCEP_identifier_type;
                    TS_expedited_data : TS_expedited_data_type);

    by TS_provider:

      T_CONNECT_ind(TCEPI              : TCEP_identifier_type;
                    to_T_address       : T_address_type;
                    from_T_address     : T_address_type;
                    QOTS_request       : quality_of_TS_type;
                    options            : option_type;
                    TS_connect_data    : TS_connect_data_type);

      T_CONNECT_conf(TCEPI              : TCEP_identifier_type;
                     QOTS_request       : quality_of_TS_type;
                     options            : option_type;
                     TS_accept_data     : TS_accept_data_type);

      T_DISCONNECT_ind(TCEPI                  : TCEP_identifier_type;
                       TS_disconnect_reason : TS_disconnect_reason_type;
                       TS_user_reason       : TS_user_reason_type);

      T_DATA_ind(TCEPI             : TCEP_identifier_type;
                 TSDU_fragment     : data_fragment_type);   (* note 5 *)

      T_EX_DATA_ind(TCEPI              : TCEP_identifier_type;
                    TS_expedited_data : TS_expedited_data_type);
```

```
Transport Protocol Specification   (* for class 0 *)
************************************************

uses Transport Service
uses Network Service

type   (* note a *)
  credit_allocation_type = 0..15;

  TP_reference_type = ...;   (* string of 2 octets *)

  max_TPDU_size_type = (128,256,512,1024,2048);

  variable_part_type = record
    calling_T_address : ...;   (* note b *)
    called_T_address : ...;   (* note b *)
    max_TPDU_size : max_TPDU_size_type;
    additionnal_clear_reason : ...;
    rejected_TPDU : ...;   (* note c *)
                    end;

  protocol_class_type = (class_0,class_1,class_2,class_3,class_4);

  TP_disconnect_reason_type = (TS_user_initiated_termination,
                               remote_congestion,
                               connection_negotiation_failed,
                               duplicate_connection_detected,
                               mismatched_references,
                               procedure_error,
                               destination_entity_not_available,
                               reference_overflow,
                               unknown_reason);

  TP_reject_cause_type = (reason_not_specified,
                          function_not_implemented,
                          invalid_TPDU_type,
                          invalid_parameter);
```

```
Interaction

  TPDU_and_control (entity, NS_provider)

    by entity, NS_provider :

      CR(credit           : credit_allocation_type;
         source_reference : TP_reference_type;
         class            : protocol_class_type;
         options          : option_type;
         variable_part    : variable_part_type;   (* note f *)
         user_data        : ...);   (* note e *)

      CC(dest_reference   : TP_reference_type;
         source_reference : TP_reference_type;
         class            : protocol_class_type;
         options          : option_type;
         variable_part    : variable_part_type);   (* note f and note i *)

      DR(dest_reference    : TP_reference_type;
         source_reference  : TP_reference_type;   (* note j *)
         disconnect_reason : TP_disconnect_reason_type;
         variable_part     : variable_part_type);   (* note g *)

      DT(user_data : data_fragment_type);

      ERR(dest_reference : TP_reference_type;
          reject_cause   : TP_reject_cause_type;
          variable_part  : variable_part_type);   (* note h *)

    <-by entity:
      N_DISCONNECT_req (NCEP_id : ......., reason : .......);

    by NS_provider :
      Network_disconnect (......);
      Network_reset (......);

interactions

  local_buffer(user,buffer) is   (* note k *)

    by user:

      clear;

      set_max_get_size(fragment_size : fragment_length_type);

      append(fragment : data_fragment_type);

    by buffer:

      get_next(fragment : data_fragment_type);
```

```
module Transport_entity(TSAP : TS_access_point(TS_provider);
                        mapping : TPDU_and_control (entity);
                        out_buffer,
                        in_buffer : local_buffer(user)) is


var
  state : (idle,wait_for_CC,wait_for_T_CONNECT_resp,data_transfer);

  local_reference : TP_reference_type;

  remote_reference : TP_reference_type;

  TPDU_size : max_TPDU_size_type;

  remote_address : ...;    (* note b *)

  QOTS_estimate : quality_of_TS_type;


  (* intermediate variables; no state information *)
  variable_part_to_send : variable_part_type;

  disc_reason : TS_disconnect_reason_type;

  user_reason : TS_user_reason_type;

  called_address : T_address_type;

  calling_address : T_address_type;



initialisations

  state := idle;
```

```
(*  transitions :   note m *)

   from idle

     when TSAP.T_CONNECT_req(TCEPI,to_T_address,from_T_address,
                             QOTS_request,options,TS_connect_data)

       provided ... (* Transport entity able to provide the quality of
                 service asked for *)
         to wait_for_CC
         begin
           local_reference := ...;    (* note ff *)
           TPDU_size := ...;    (* note n *)
           variable_part_to_send := ...;    (* note o *)
           mapping.CR(0,local_reference,class_0,normal,variable_part_to_send);
         end;

       provided ... (* Transport entity not able to provide the quality
                   of service asked for *)
         to same
         begin
           TSAP.T_DISCONNECT_ind(TCEPI,
                                 inability_to_provide_the_quality )
          end;

     when mapping.CR(credit,source_reference,class,options,
                     variable_part,user_data)

       provided ... (* Transport entity able to provide the quality of
                 service asked for *)
         to wait_for_T_CONNECT_resp
         begin
           remote_reference := source_reference;
           if variable_part.max_TPDU_size <> undefined then
             TPDU_size := variable_part.max_TPDU_size    (* note e *)
           else
             TPDU_size := 128;
           remote_address := variable_part.calling_T_address;
           ...    (* note q *)
           called_address := ...;    (* note gg *)
           calling_address := ...;
           QOTS_estimate := ...;    (* note r *)
           TSAP.T_CONNECT_ind(TCEPI,called_address,calling_address,
                              QOTS_estimate,normal,... (* no data *));
         end;

       provided ... (* Transport entity not able to provide the quality
                   of service asked for *)
         to same
         begin
          variable_part_to_send. additional_clear_reason := ... ;
          mapping. DR (source_reference, 0, connection_negociation_failed,
                       variable_part_to_send);
         end;
```

```
from  wait_for_CC

  when mapping.CC(dest_reference,source_reference,class,options,
                  variable_part)   (* note u *)
    to data_transfer
    begin
      remote_reference := source_reference;
      if variable_part.max_TPDU_size <> undefined then
        TPDU_size := variable_part.max_TPDU_size
      else
        TPDU_size := 128;
      ...   (* note v *)
      QOTS_estimate := ...;   (* note w *)
      TSAP.T_CONNECT_conf (TCEPI, QOTS_estimate, normal, ... );
      in_buffer.clear;
      out_buffer.clear;
      out_buffer.set_max_get_size(TPDU_size);
    end;

  when mapping.DR(dest_reference,source_reference,
                  disconnect_reason,variable_part)
    to idle
    begin
      disc_reason := disconnect_reason;
      if disc_reason = TS_user_initiated_termination then
        user_reason := variable_part. additional_clear_reason;
      mapping. N_DISCONNECT_req (..., disc_reasn);
      mapping.N_DISCONNECT_req(...,disc_reason);   (* note x *)
    end;
```

```
    from wait_for_T_CONNECT_resp

when TSAP.T_CONNECT_resp(TCEPI,QOTS_request,options,TS_accept_data)

  provided ... (* quality of service requested <= quality of service
           proposed in T_CONNECT_ind *)
    to data_transfer
    begin
      local_reference := ...;   (* note ff *)
      TPDU_size := ...;   (* note y *)
      with variable_part_to_send
      begin
        called_T_address := remote_address;
        calling_T_address := ...;   (* note z *)
        max_TPDU_size := ...;   (* note aa *)
      end;;
      mapping.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;

  provided ... (* quality of service requested > quality of service
              proposed in T_CONNECT_ind *)
    to idle
    begin   (* note ee *)
      variable_part_to_send. additional_clear_reason := ...;
      mapping. DR (remote_reference, 0, connection_negociation_failed,
                   variable_part_to_send);
      TSAP.T_DISCONNECT_ind(TCEPI,
                 inability_to_provide_the_quality, ...);
    end;

when TSAP.T_DISCONNECT_req(TCEPI,TS_user_reason)
  to idle
  begin
    variable_part_to_send. additional_clear_reason := ...;
    mapping. DR (remote_reference, 0, TS_user_initiated_termination,
                 variable_part_to_send);
  end;
```

```
from data_transfer
  to same

    when TSAP.T_DATA_req(TCEPI,TSDU_fragment)
       provided ... (* flow control from the user is ready *)
          begin
            out_buffer.append(TSDU_fragment);
          end;

    when out_buffer.get_next(fragment)
       provided ... (* flow control to the Network layer is ready *)
          begin
            mapping.DT(fragment);
          end;

    when mapping.DT(user_data)
       provided ... (* flow control from the Network layer is ready *)
          begin
            in_buffer.append(user_data);
          end;

    when in_buffer.get_next(fragment)
       provided ... (* flow control to the user is ready *)
          begin
            TSAP.T_DATA_ind(fragment);
          end;
  to idle

    when TSAP.T_DISCONNECT_req(TCEPI,TS_user_reason)
      begin
        mapping. N_DISCONNECT_req (..., disconnect_reason, user_reason,...);
      end;
    when mapping.Network_disconnect(...,disconnect_reason,user_data)    begin
      begin
        disc_reason := ...;    (* note cc *)
        TSAP.T_DISCONNECT_ind(TCEPI,disc_reason,...);
      end;

    when mapping.Network_reset(...,reset_reason)    (* note x *)
      begin
        disc_reason := ...;    (* note cc *)
        TSAP.T_DISCONNECT_ind(TCEPI,disc_reason,...);
      end;
```

ANNEX 2

Title: Specification of class 0 and 2 Transport protocol for multiple connections

1. Introduction

The formal description given in section 2 uses the language defined in Part II of this report, which was defined by the ISO TC97/SC16/WG 1 ad hoc group on FDT (working document December 1981). The following paragraphs are intended to explain some characteristics of the Transport protocol specification given below in order to facilitate its reading.

1.1. Local buffers

The data buffers declared in the module heading of the specification are local buffers containing Transport service data units. There are two buffers per Transport connection, one for incoming and one for outgoing data. These buffers are included for generality. A particular imlementation of the protocol may choose buffers of zero capacity.

1.2. Connection identification

Similar to the service specification, the protocol specification uses abstract data types for identifying the different Transport and Network connections. The identifier type for the Transport connections "TC_id_type" is not specified (implementation dependent). For the identification of the Network connection a particular type "NC_id_type" has been adopted for convenience, consisting of the pair of Network address and Network connection endpoint identifier.

1.3. Addressing

The relation between Network and Transport addresses is only partly defined by this standard. The formal specification remains quite general in this respect by assuming no particular relation between these two kinds of addresses. However, it is assumed that the CR and CC PDU's optionally contain "additional addressing information" such that the following mapping exists: (a) Form a Transport address one or several suitable Network addresses can be derived together with "additional information" (using the function "determine_add_address"); (b) From a Network address and some "additional information" a corresponding Transport address may be derived (using the function "determine_T_address").

1.4. Transport PDU's

The "TP" module defines the behavior of a Transport entity as a whole. It therefore interacts with the Network layer through Network service primitives. The Transport PDU's exchanged in N_DATA requests and indications are visible within the "TP" module, for example in the data type "PDU_type" and the procedure "build_PDU" which assembles the different parameters of a PDU and stores the PDU in a "PDU_buffer". For simplicity, the module contains one "PDU_buffer" for each kind of PDU (CR, CC, DR, etc.). (An implementation, of cource, will not take such an approach). A PDU sending transition (see below) collects the

PDU's and includes them in the Network service data units to be sent.

1.5. State variables

The state variables of the module are partitioned into two parts: those associated with the Network connections, and those associated with the Transport connections. The association between Transport and Network connections is given by the variable "assigned_NC" which is defined for each active Transport connection. The major state of a Transport connection is given by the value of the variable "state" associated with each Transport connection. (It is noted that a "state" is defined for each connection; therefore the concept of ONE major state (also called "state", as defined in the FDT used) is not directly applicable. In order to conform with the syntax of the FDT, a dummy "state" variable is introduced, together with dummy "from" and "to" clauses).

1.6. Sending the PDU's

As mentioned above, the PDU to be sent are stored in "PDU_buffers", one buffer per kind of PDU. The inclusion of a PDU in the next Network service data unit (NSDU) to be sent (and the possible concatenation of several PDU into one such NSDU) is defined by the first transition of the specification. The conditions of this transition determined under which circumstances such inclusion is possible. In the case of concatenation, several instances of this transition are executed. The non-deterministic nature of the specification language assures that aspects such as the priority between different kinds of PDU's, and the extent of concatentation is not specified (but left as an implementation choice). It is noted that the inclusion of a DT PDU is handled by a separate transition (second transition of the NORMAL DATA TRANSFER section).

The second transition of the specification describes the sending of a NSDU, which may occur any time the "NSDU_to_be_sent" contains at least one PDU.

1.7. Reception of PDU's

The reception of a NSDU from a peer Transport entity is described by the third transition of the specification. For each PDU included in the received NSDU, the forth transition is executed which describes the actions to be performed on the receptions of different kinds of PDU's. To simplify the description of these actions, and in particular the error processing defined by the protocol, all these actions are integrated into a single transition, which is structured by case statements according the the major state of the Transport connection to which the PDU refers, and according to the kind of PDU received.

1.8. Grouping of transitions

The remaining transitions of the specification are partitioned according to the functions they perform, i.e. into CONNECTION ESTABLISHMENT (separately for the calling and called side), NORMAL

DATA TRANSFER, EXPEDITED DATA TRANSFER, TERMINATION PHASE, and NETWORK CONNECTION ESTABLISHMENT.

1.9. Flow control

It is assumed that the user of the Transport service is always ready to receive control service primitives.

Similarly, the Transport entity is always ready to receive control service primitives from the underlying Network layer.

The flow control of expedited data is explicitly described by the T_EX_D_READY primitives at the user interface and by state variables of the "TP" module as far as the flow through the Network layer is concerned.

The flow control of normal data is described explicitly as far as the exchange of credits between the peer protocol entities is concerned, however, the flow control mecanism at the user interface is not specified in detail. It is assumed that it can be determined whether the flow control to the user is ready, and whether the flow control for N_DATA requests to the Network layer is ready. The condition for the Transport entity to be ready for the reception of N_DATA indications from the Network layer is defined by a condition in the third transition of the specification. The condition for the Transport entity to be ready for the reception of T_DATA requests from the user is determined by the flow control mecanism of the "send_buffer" (see section 1.1).

2. Formal specification

```
module TP_entity (
      NSAP : array [N_address_type] of NS_primitives (NS_user);
      TSAP : array [T_address_type] of TS_primitives (TS_provider);
      receive_buffer,
      send_buffer : array [TC_id_type] of TS_data_buffer (user) );

type  (* the type and interaction declarations of the Transport and Network
         service specifications are used *)
   class_type = (class_0, class_1, class_2, class_3, class_4);
   reference_type = 0 .. (2**16 - 1);
   seq_number_type = 0 .. 127;
   credit_type = seq_number_type;
   PDU_size_type = pos_integer;
   TC_id_type = ...;
   additional_address_information = ...;
   reject_cause_type = (reason_not_specified     (* = 0 *),
                        function_not_implemented (* = 1 *),
                        invalid_PDU              (* = 2 *),
                        invalid_parameter        (* = 3 *)  );
   reason_type = (
     (* for class 0 *)
        0    (* reason not specified *),
        1    (* terminal occupied *),
        2    (* terminal out of order *),
        3    (* address unknown *),
     (* for classes 1 through 4 *)
      128  (* normal disconnect initiated by session level *),
      129  (* remote congestion *),
      130  (* negociation failed *),
      131  (* duplicate connection detected *),
      133  (* protocol error *),
      134  (* destination entity specified not available *),
      135  (* reference overflow *),
      136  (* refuse a new TC over the same NC *),
      255  (* unknown reason *) );
```

```
TPDU_code_type = (CR, CC, DR, DC, DT, AK, EDT, EAK, ERR, undefined_code);
TPDU_type = record
   credit_value : credit_type; (* used for CR, CC, AK *)
   dest_ref : reference_type; (* used for CC, DR, DC, DT (class 2),
                                 EDT, AK, EAK, ERR *)
   source_ref : reference_type; (* used for CR, CC, DR, DC *)
   user_data : optional string_of_octets; (* see TS *)
         (* used for CR, CC, DR (not in this version of the protocol,
               DT, ED *)
   case code : TPDU_code_type of
   CR, CC : (
      proposed_class : class_type;
      proposed_options : option_type; (* see TS *)
      calling_addr,
      called_addr : optional addtional_address_information;
      proposed_TPDU_size : optional PDU_size);
   DR : (
      disconnect_reason : reason_type);
   DC :;
   DT : (
      send_sequence : seq_number_type;
      end_of TSDU : boolean );
   AK : (
      expected_send_sequence : seq_number_type);
   EDT, EAK :;
   ERR : (
      reject_cause : reject_cause_type );
   undefined_code ;
   end;

NC_id_type = record
   local_N_addr : N_address_type; (* see NS *)
   EP_id : NCEP_id_type; (* see NS *)
   end;
```

```
var
   TC : array [TC_id_type] of record
      state : (closed, wait_for_NC, open_in_progress_calling,
               open_in_progress_called, rejected, open,
               wait_before_closing, closing);
      local_T_addr,
      remote_T_addr : T_address_type; (* see TS *)
      id : TCEP_id_type; (* see TS *)
      local_ref,
      remote_ref : reference_type;
      assigned_NC : NC_id_type;
      max_PDU_size : PDU_size_type;
      options : option_type; (* see TS *)
      class : class_type;
      QTS : quality_of_TS_type; (* see TS *)
      connect_data : optional string_of_octets;
      TR,
      TS : seq_number_type;
      R_credit,
      S_credit : credit_type;
      EX_D_sent,
      EX_D_received : boolean;
      PDU_buffer : array [TPDU_code_type] of record
         full : boolean;
         PDU : TPDU_type
         end;
   NC : array [NC_id_type] of record
      NC_state : (closed, open_in_progress, open);
      remote_N_addr : N_address_type; (* see NS *)
      this_side : both_sides; (* see NS *)
      QNS : quality_of_NS_type;
      received_NSDU,
      NSDU_to_be_sent : record
         user_data_present : boolean;
         data : string_of_octets;
         end;
   state : (dummy);
```

```
function determine_add_address (
      T_addr : T_address_type;
      N_addr : N_addrtess_type) : optional additional_address_information;
   begin ... end;

function determine_T_address (
      N_addr : N_address_type;
      add_info : optional additional_address_information) : T_address_type;
   begin ... end;

function implied_PDU_size (size : optional PDU_size_type) : PDU_size_type;
   begin if size = undefined
         then implied_PDU_size := 128
         else implied_PDU_size := size  end;

function check_PDU_size_negociation_rule
      (size, new_size : PDU_size_type) : boolean;
   begin ... end;
      (* property: if new_size <> undefined then
           ( size >=  128  implies  128 <= new_size <= size
             and  size < 128  implies  (new_size <= size
                                        or  new_size = 128) ) *)

function determine_PDU_length (PDU : PDU_type) : pos_integer;
   begin ... end; (* property: determines the length of the PDU in octets *

function find_TC_id
      (T_addr : T_address_type; id : TCEP_id_type) : TC_id_type;
   begin ... end; (* property: determines the TC associated with the
                                the EP identifier *)

function find_NC_id
      (N_addr : N_address_type; NCEP_id : NCEP_id_type) : NC_id_type;
   begin with find_NC_id do begin
      local_N_addr := N_addr;
      EP_id := NCEP_id    end;
      end; (* determines the NC associated with the EP identifier *)

function determine_TS_reason
      (TP_reason : reason_type) : TS_disconnect_reason_type;
   begin case TP_reason of
      0, 130, 131, 133, 135, 255 : determine_TS_reason := TS_FAIL;
      1, 2, 128          : determine_TS_reason := TS_U_NRM;
      3, 134             : determine_TS_reason := TS_U_UNKNOWN;
      129, 135           : determine_TS_reason := TS_CONG;
      end end;
```

```
procedure build_PDU (TC_id : TC_id_type; kind : TPDU_code_type);
   begin with TC [TC_id]. PDU_buffer[kind] do begin
      full := true;
      with PDU do begin
         code := kind;
         dest_ref := remote_ref;
         if kind in [CR, CC, AK] then
            if class = class_0 then credit_value := 0
                               else credit_value := R_credit;
         if kind in [CR, CC, DR, DC] then source_ref := local_ref;
         case kind of
         CR, CC : begin
            proposed_class := class;
            proposed_options := options;
            calling_addr := determine_add_addr (local_T_addr,
                               assigned_NC.local_N_addr;
            called_addr  := determine_add_addr (remote_T_addr,
                               NC[assigned_NC].remote_N_addr;
            proposed_TPDU_size := max_PDU_size;
            user_data := connect_data;
            end;
         DC :;
         DR : ; (* disconnect_reason must be assigned *)
         DT : send_sequence := TS;
         AK : expected_send_sequence := TR;
         EDT : user_data := TS_user_data;
         EAK :;
         ERR :; (* reject_cause must be assigned *)
         end;
      end;

procedure protocol_error (TC_id : TC_id_type; cause : reject_cause_type);
   begin with TC [TC_id] do
      TSAP [local_T_addr]. T_DISCONNECT (id, TS_FAIL, ... (* dummy *) );
      build_PDU (TC_id, ERR);
      PDU_buffer [ERR]. PDU. reject_cause := cause;
      state := closing;
      end;
```

```
procedure close_all_TC (NC_id : NC_id_type;
                        TS_reason : TS_disconnect_reason_type);
   begin
      for all TC_id : TC_id_type do  with  TC[TC_id] do
         if state <> closed  and assigned_NC = NC_id
         then begin
            if state not in [wait_before_closing, closing]
            then TSAP[local_T_addr]. T_DISCONNECT_ind
                              (id, TS_reason, ... (* dummy *));
            close_and_clear_buffers (TC_id);
            end;

procedure close_TC (TC_id : TC_id_type;
                    reason : reason_type;
                    inform_TS_user : boolean);
   begin  with TC[TC_id] do begin
      if inform_TS_user
      then TSAP[local_T_addr]. T_DISCONNECT_ind
                (id, determine_TS_reason (reason), ... (* dummy *));
      build_PDU (TC_id, DR);
      PDU_buffer [DR]. PDU. disconnect_reason := reason;
      if state <> rejected then  state := closing;
      end;

procedure close_and_clear_buffers (TC_id : TC_id_type);
   begin with TC[TC_id] do begin
      state := closed;
      for kind := CR to ERR do PDU_buffer [kind]. full := false;
      end end;

procedure clear_NC_buffers (NC_id : NC_id_type);
   begin
      received_NSDU.data.length := 0;
      NSDU_to_be_sent.data.length := 0;
      end;

(* initialization: set all states to closed *)
```

```
(* TRANSITIONS *)

from dummy to dummy (* required by FDT syntax *)

(* GENERAL PURPOSE TRANSITIONS *)

(* concatenate a PDU to be sent  into the NSDU to be sent *)

any NC_id : NC_id_type, TC_id : TC_id_type, kind : TPDU_code_type do
      with NC[NC_id], TC[TC_id] do
provided not NSDU_to_be_sent.user_data_present
      and NSDU_to_be_sent.data.length +
            determine_PDU_length (PDU_buffer[kind])  <= max_PDU_size
      and assigned_NC = NC_id
      and state <> closed
      and not ((class = class_0) and (NSDU_to_be_sent.data.length <> 0))
            (* no concatenation for protocol class 0 *)
   begin
      (* encode PDU *) with NSDU_to_be_sent do begin
         data.length := data.length + determine_PDU_length (PDU_buffer[kind
         data.content := ...; (* property: code PDU and append into NSDU *)
         if user_data <> undefined then user_data_present := true;
         end;
      if kind in [DC, ERR]  or state = rejected
      then close_and_clear_buffers (TC_id);
      end;

(* send a NSDU *)

any NC_id : NC_id_type do  with NC[NC_id] do
provided  NSDU_to_be_sent.data.length <> 0
      and state = open
      and ... (* property: flow control to Network layer ready *)
   begin
      NSAP[NC_id.local_N_addr]. N_DATA_req
            (NC_id.EP_id, NSDU_to_be_sent.data, true (* complete NSDU *) );
      NSDU_to_be_sent.data.length := 0;
      end;

(* receive a NSDU with one or more PDUs *)

any N_addr : N_address_type do
when NSAP[N_addr]. N_DATA_ind
      with NC [find_NC_id (N_addr, NCEP_id)] do
   provided received_NSDU.data.length = 0 (* property: this means
                              flow control to the Transport entity is ready *
         and is_last_fragment_of_NSDU  (* it is assumed that the N-interfac
               transfers complete SDU in each N_DATA primitive *)
      begin
         received_NSDU.data := TS_user_data;
         end;
```

```
(* reception of a PDU *)
any NC_id : NC_id_type do   with NC [NC_id] do
provided received_NSDU.data.length <> 0
      and ... (* property: not ((class = class_0) and (flow control
                                to user (or to the receive buffer)
                                is not ready))  *)
   var received_PDU : TPDU_type;
       TC_id : TC_id_type;
   procedure determine_TC (NC_id : NC_id_type; ref : reference_type);
      begin ... end; (* property: determine_TC (NC_id, ref) =
         if exists TC_id such that with TC[TC_id] holds
               state <> closed  and  assigned_NC = NC_id  and local_ref = r
         then TC_id
         else TC_id' such that TC[TC_id].state = closed;
      i.e. find the TC associated with the reference "ref" over the NC;
           if "ref" = 0 then such a TC does not exist. *)
  begin
      ...; (* decode (received_NSDU, received_PDU) *)
      with received_PDU do begin
         TC_id := determine_TC (NC_id, dest_ref);
         with TC[TC_id] do case state of
         closed : (* no TC assigned *)
            if code = CR
            then  begin
               remote_ref := source_ref;
               local_ref := ...;
               if dest_ref <> 0
               then ... (* error *)
               else if ... (* property:
                     exists TC_id' <> TC_id  such that with TC[TC_id'] holds
                        state <> closed  and  assigned_NC = NC_id
                        and remote_ref = source_ref ;
                     i.e. this is a duplicated CR *)
                  then close_TC (TC_id', 131 (* duplicate connection *),
                                 true (* inform user *) )
                  else if determine_PDU_length (received_PDU) >
                            implied_PDU_length (proposed_TPDU_size)
                       or proposed_class = class_0 and this_side = calling
                     then protocol_error (TC_id, ...)
                     else if ... (* not able to provide service
                                    or destination address unknown *)
                        then begin
                           disconnect_reason := ...;
                           build_PDU (TC_id, DR);
                           state := closing;
                           end
                        else begin (* normal processing *)
                           local_T_addr := determine_T_addr
                                 (NC_id.local_N_addr, called_addr);
                           remote_T_addr := determine_T_addr
                                 (NC[NC_id].remote_N_addr, calling_addr);
                           id := ...; (* property: for all TC_id' holds
                                         not (TC[TC_id'].state <> closed
                                            and TCEP_id = id);
```

```
                                    i.e. TCEP identifier is not yet in use *)
                            remote ref := source_ref;
                            assigned_NC := NC_id;
                            options := ...;
                                  (* property: options in proposed_options *)
                            class := ...; (* property: proposed_class = class
                                             implies class = class_0 *)
                            max_PDU_size :=
                                  implied_PDU_size (proposed_TPDU_size);
                            QTS := ...;
                            TR := 0;
                            TS := 0;
                            S_credit := credit_value;
                            R_credit := 0;
                            receive_buffer [TC_id]. clear;
                            send_buffer [TC_id]. clear;
                            TSAP[local_T_addr].T_CONNECT_ind(id, local_T_add
                                  remote_T_addr, options, QTS, user_data);
                            state := open_in_progress_called;
                            end
else if code = DR
       build_PDU (TC_id, DC);
        state := closing;
        end
     else ; (* ignore othe received PDU if no TC is assigned *)

(* in the following cases a TC is already assigned *)
closing, rejected :
   if code = DC
   then state := closed
   else ; (* ignore received PDU *)
wait_before_closing :
   close_TC (TC_id, 128 (* normal disconnect reason *)
             false (* TS user not informed again *) );
wait_for_NC, open_in_progress_calling,
       open_in_progress_called, open :
```

```
case code of
CC :
   if state <> open_in_progress_calling
   then protocol_error (TC_id, invalid_PDU)
   else begin
      remote_ref := source_ref;
      if proposed_class = class_0  and  class = class_2
            and ... (* property: NC_id is multiplexed *)
      then begin
         close_all_TC (NC_id, ...);
         NSAP[NC_id.local_N_addr].N_DISCONNECT_req
                                        (NC_id.EP_id);
         end
      else if proposed_class = class_0  and class = class_2
            and this_side = called
         then protocol_error (TC_id, ...)
         else if calling_addr <> determine_add_addr
                           (local_T_addr, NC_id.local_N_addr)
               or called_addr <> determine_add_addr
                           (remote_T_addr, remote_N_addr)
               or not check_PDU_size_negociation_rule
                           (max_PDU_size, proposed_TPDU_size)
               or proposed_options  not in options
               or proposed_TPDU_size < determine_length
                           (received_PDU)
            then protocol_error (TC_id, invalid_parameter)
            else begin (* normal processing *)
               if proposed_TPDU_size <> undefined
               then max_PDU_size := proposed_TPDU_size;
               S-credit := credit_value;
               TSAP[local_T_addr]. T_CONNECT_conf (id, QTS,
                           options, user_data);
               state := open;
               end;

DR : begin
   TSAP[local_T_addr]. T_DISCONNECT_ind
         (id, determine_TS_reason(disconnect_reason), undefined);
   if state  <>  open_in_progress_calling
   then begin
      build_PDU (TC_id, DC);
      state := closing;
      end
   else close_and_clear_buffers (TC_id);
   end;
```

```
DC : ...; (* protocol error *)
ERR : begin
   TSAP[local_T_addr]. T_DISCONNECT_ind (id, TS-FAIL, undefined)
   if class = class_0  then  NSAP[assigned_NC.local_N_addr].
         N_DISCONNECT_req (assigned_NC. EP_id);
   close_and_clear_buffers (TC_id);
   end;
DT : if state <> open
      or R_credit <> 0
   then protocol_error (TC_id, invalid_PDU)
   else if send_sequence <> TR
      then protocol_error (TC_id, invalid_parameter)
      else begin
         receive_buffer[TC_id].append (user_data, end_of_TSDU);
         TR := (TR + 1) mod 128;
         R_credit := R_credit - 1;
         end;
AK : if state <> open
      or class = class_0
   then protocol_error (TC_id, invalid_PDU)
   else begin
      new_S_credit := credit_value + expected_send_sequence
                           - send_sequence;
      if new_S_credit < S-credit
      then protocol_error (TC_id, invalid_papameter)
      else S_credit := new_S_credit;
      end;

EDT : if state <> open
       or expedited_data not in options
       or EX_D_received
   then protocol_error (TC_id, invalid_PDU)
   else begin
      TSAP[local_T_addr]. T_EX_DATA_ind (id, user_data);
      EX_D_received := true;
      end;
EAK : if state <> open
       or expedited_data not in options
       or not EX_D_sent
   then protocol_error (TC_id, invalid_PDU)
   else begin
      TSAP[local_T_addr]. T_EX_D_READY_conf (id);
      EX_D_recived := false;
      end;
undefined_code : ... ;
```

```
(* CONNECTION ESTABLISHMENT : calling side *)

any T_addr : T_address_type do
when TSAP[T_addr]. T_CONNECT_req
   provided ... (* property: for all TC_id holds
                        not (TC[TC_id].state <> closed  and TCEP_id = id)
                     i.e. the TCEP identifier is not yet in use *)
         and from_T_address = T_addr
      var TC_id : TC_id_type;
      begin
         TC_id := ...; (* property: TC[TC_id].state = closed,
                             i.e. connection not in use *)
         with TC[TC_id] do begin
            local_T_addr := T_addr;
            remote_T_addr := to_T_address;
            id := TCEP_id;
            options := proposed_options;
            QTS := proposed_QTS;
            connect_data := user_data;
            TR := 0;
            TS := 0;
            receive_buffer[TC_id].clear;
            send_buffer[TC_id].clear;
            state := wait_for_NC;
            end;
         end;

any TC_id : TC_id_type do  with TC[TC_id] do
provided state = wait_for_NC
      and ... (* not able to provide service *)
   begin
      TSAP[local_T_addr]. T_DISCONNECT_ind (id, ...
            property: if mapping between Transport and Network addresses
                  is not possible then U_UNKNOWN;
               if a N_CONNECT_req was sent to establish a new network
               connection for this TC, and N_DISCONNECT was received
                  TS_disconnect_reason :=
                     if NS_disconnect_reason = NS_U_NRM
                     then TS_FAIL else TS_QUAL_FAIL;
               QTS.class_of_service = enhanced
                  implies TS_QUAL_FAIL *),
            ... (* dummy user reason *) );
         state := closed
         end;
```

```
any TC_id : TC_id_type, NC_id : NC_id_type do
      with TC[TC_id], NC[NC_id] do
   provided state = wait_for_NC
         and NC_state = open
         and QTS.class_of_service = basic
         and ... (* check throughput quality *)
         and ... (* check addressing *)
         and ... (* able to provide service *)
      begin
         assigned_NC := NC_id;
         local_ref := ...;
               (* property: <> 0  and not un use with the same NC *)
         dest_ref := 0;
         class := ...; (* select appropriate protocol class *)
           (* property: (data <> undefined) or (expedited_data in options
                     or (this_side = called)  implies  class = class_2 *)
         max_PDU_size := ...;
           (* property: class = class_0  implies
                 max_PDU_size in [256, 512, 1024, 2048]  *)
         build_PDU (TC_id, CR);
         state := open_in_progress_calling;
         end;

(* for the handling of the peer's response, see "reception of a PDU" abov
```

```
(* CONNECTION ESTABLISHMENT : called side *)

(* for the handling of the incoming CC, see "reception of a PDU" above *)

any T_addr : T_address_type do
when TSAP[T_addr]. T_CONNECT_resp
      with TC[find_TC_id (T_addr, TCEP_id)] do
   provided state = open_in_progress_called
         and proposed_options in options
      begin
         QTS := proposed_QTS;
         options := proposed_options;
         local_ref := ...;
               (* property: <> 0 and not in use with the same NC *)
         max_PDU_size := ...; (* property:
                  check_PDU_size_negociation_rule (old value, new_value) *)
         build_PDU (find_TC_id (T_addr, TCEP_id), CC);
         state := open;
         end;

(* for the case of rejection by the T user, see first transition of the
   termination phase *)
```

```
(* NORMAL DATA TRANSFER *)

any T_addr : T_address_type do
when TSAP[T_addr]. T_DATA_req
         with TC [find_TC_id (T_addr, TCEP_id)] do
   provided state = open
         and ... (* flow control to send_buffer[find_TC_id(T_addr, TCEP_i
                           is ready *)
      begin
         send_buffer [find_TC_id (T_addr, TCEP_id)].
                append (TS_user_data, is_last_fragment_of_TSDU);
         end;

any TC_id : TC_id_type do
      with TC[TC_id], NC [ TC[TC_id].assigned_NC] do
when send_buffer [TC_id]. next_fragment
   provided class = clas_0
         and NSDU_to_be_sent.user_data.length = 0
         and ((fragment.length  + 3 (* header *) = max_PDU_size)
              or is_last_fragment_of_TSDU )
      begin
         ...; (* encode_data (fragment, NSDU_to_be_sent.data) *)
         end_of_TSDU := is_last_fragment_of_TSDU;
         end;

   provided class = class_2
         and S_credit <> 0
         and fragment.length
                + 5 (* header length for DT PDU (classes 1 to 4   ) *)
                + NSDU_to_be_sent.data.length  <= max_PDU_size
         and not NSDU_to_be_sent.user_data_present
      begin
         ...; (* encode_data (fragment, NSDU_to_be_sent.data) *)
         end_of_TSDU := is_last_fragment_of_TSDU;
         TS := TS + 1;
         S_credit := S_credit - 1;
         end;

(* reception of a DT PDU, see "reception of a PDU" above *)

any TC_id : TC_id_type do  with TC[TC_id] do
when receive_buffer [TC_id]. next_fragment
   provided ... (* flow control to user ready *)
      begin
         TSAP[local_T_addr]. T_DATA_ind
                (id, fragment, is_last_fragment_of TSDU);
         end;
```

```
when receive_buffer [TC_id]. free_space
   provided state <> closed
      begin
         R_credit := R_credit + 1;
         end;

provided class = class_2
      and state = open
   begin
      build_PDU (TC_id, AK);
      end;

(* reception of an AK PDU, see "reception of a PDU" above *)

(* EXPEDITED DATA TRANSFER *)

any T_addr : T_addr_type do
when TSAP[T_addr]. T_EX_DATA_req
         with TC[ find_TC_id (T_addr, TCEP_id)] do
   provided expedited_data in options
         and state = open
         and not EX_D_sent
      begin
         build_PDU (find_TC_id (T_addr, TCEP_id), EDT);
         EX_D_sent := true;
         end;

(* reception of a EDT PDU, see "reception of a PDU" above *)

when TSAP[T_addr]. T_EX_D_READY_req
         with TC[ find_TC_id (T_addr, TCEP_id)] do
   provided expedited_data in option
         and state = open
         and EX_D_received
      begin
         build_PDU (find_TC_id (T_addr, TCEP_id), EAK);
         EX_D_received := false;
         end;

(* reception of a EAK PDU, see "reception of a PDU" above *)
```

```
(* TERMINATION PHASE *)

any T_addr : T_address_type do
when TSAP[T_addr]. T_DISCONNECT_req
      with TC [find_TC_id (T-addr, TCEP_id)] do
   provided state in [wait_for_NC, open_in_progress_calling,
                                   open_in_progress_called, open]
      var reason : reason_type;
      begin
         (* TS_user_reason is ignored *)
         if state = open_in_progress_called then state := rejected;
         case state of
            wait_for_NC : close_and_clear_buffers
                              (find_TC_id (T_addr, TCEP_id) );
            open_in_progress_calling : if class = class_0
                  then begin
                       NSAP[ assigned_NC.local_N_addr].
                             N_DISCONNECT_req (assigned_NC. EP_id);
                       close_and_clear_buffers (find_TC_id (T_addr, TCEP_i
                       end
                  else state := wait_before_closing;
            open_in_progress_called, rejected, open :
                  begin
                     if class = class_0
                        then reason := ... (* property: 1 or 2 *)
                        else reason := 128 (* normal termination *);
                     close_TC (TC_id, reason,
                             false (* TS user not informed again *) );
            end;
         end;

any N_addr : N_address_type do
when NSAP[NC_id]. N_RESET_ind
   begin
      close_all_TC (find_NC_id (N-addr, NCEP_id), TS_QUAL_FAIL);
      clear_NC_buffers (find_NC_id (N_addr, NCEP_id));
      NSAP[N_addr]. N_RESET_resp (NCEP_id);
      if ... (* property: (NC was used for class_0 TC) and
                          (this_side = calling) and (NC is not to be
                             used for a subsequent TC)  *)
      then NSAP[N_addr]. N_DISCONNECT_req (NCEP_id);
      end;

when NSAP[NC_id]. N_DISCONNECT_ind
   var TS_reason : TS_disconnect_reason_type;
   begin
      if NS_disconnect_reason = NS_U_NRM
      then if class = class_0
         then TS_reason := TS_U_NRM
         else TS_reason := TS_FAIL
      else TS_reason := TS_QUAL_FAIL;
      close_all_TC (find_NC_id (N_addr, NCEP_id), TS_reason);
      clear_NC_buffers (find_NC_id (N_addr, NCEP_id);
      end;
```

```
(* NETWORK CONNECTION ESTABLISHMENT *)

any NC_id : NC_id_type do  with NC[NC_id] do
provided NC_state = closed
   begin
      remote_N_addr := ...; (* as required by TC in "wait_for_NC" state *
      QNS := ...;           (* as required by TC in "wait_for_NC" state *
      NSAP [NC_id.local_N_addr]. N_CONNECT_req (NC_id.EP_id, remote_N_add
                                 NC_id.local_N_addr, QNS);
      this_side := calling;
      NC_state := open_in_progress;
      end;

any N_addr : N_address_type do
when NSAP[N_addr]. N_CONNECT_conf
      with NC[find_NC_id (N_addr, NCEP_id)] do
   provided NC_state = open_in_progress
      begin
         QNS := proposed_NS;
         NC_state := open;
         end;

when NSAP[N_addr]. N_CONNECT_ind
      with NC[find_NC_id (N_addr, NCEP_id)] do
   provided NC_state = closed
         and to_N_address = N_addr
      begin
         remote_N_addr := from_N_address;
         QNS := ...;
              (* value depends on, is usually equal to, the proposed_QNS
         NSAP[N_addr]. N_CONNECT_resp (NCEP_id, QNS);
         this_side := called;
         NC_state := open;
         end;
```

ANNEX 12

To : CSA Committee on OSI

From: G.V. Bochmann

Re : Report on the meeting of the ISO TC97/SC16/WG1 ad hoc group on FDT in Washington, Sept. 1981

The meeting was held during the week 21 through 25 of September. Most time was spent by discussions within the Subgroups A, B, and C, as they were formed at the end of the previous meeting in Berlin.

For a more detailed report, please refer to the minutes (a preliminary copy of the resolutions is enclosed). The main results of the meeting were the establishment of two working documents by the Subgroups A and B, a copy of which are enclosed. The working document of Subgroup B includes a proposal for a syntax of a specification language (for an extended state transition model) which was submitted by a liaison representative (G.V.Bochmann) to the CCITT Rapporteur's meeting on FDT in Ottawa (October 19 through 27).

The next meeting of Subgroup A will be held near Milano on November 20. Subgroup B is also planning another meeting beginning of December. Work on the "guidelines" is foreseen to be done during the next WG1 meeting in January. Another meeting of the ad hoc group on FDT is planned for Mai 1982.

UNIVERSITÉ DE MONTRÉAL
Département d'informatique et
de recherche opérationnelle (I.R.O.)

November 3rd 1981

From: G.V. Bochmann

To : Members of ISO TC97/SC16/WGI ad hoc group on FDT

Re : Report of the CCITT Rapporteur's meeting on FDT in Ottawa, October 1981.

Please find enclosed the meeting report of the CCITT meeting on FDT (Question 39/VII) which was adopted at the end of the meeting.

I would like to make the following comments on the work during that meeting:

a) It was considered that different descriptions at different levels of detail (abstraction) would be useful, such as time-sequence diagrams, state transition models in graphical SDL with informal text, graphical SDL descriptions with state pictures, or formally defined text (possibly based on the Pascal programming language) and a linear, programming language like description (which corresponds to the specification language developed in Subgroup B of the ISO ad hoc group on FDT).

b) The proposal from ISO to take the syntax developed by Subgroup B as a starting point for the collaboration of a linear FDT was not accepted at this time, because CCITT's SG XI has developed a linear form of SDL (called SDL-PR), which was also proposed as a candidate starting point.

c) There was much discussion of examples how to use SDL for protocol and service specifications. Relatively little time was spent on a comparison of the two proposals for the linear syntax. Some information about such a comparison is included in the report as annexes 6 and.

.../2

Case postale 6128, Succursale "A"
Montréal, P.Q., H3C 3J7

The CCITT group has expressed the desirability of adopting the same FDT in CCITT and ISO. It seems that a possible compromise could be the adoption of SDL for the graphical form of an FDT and the ISO proposal for the linear form of an FDT.

I leave these questions for your consideration.

Sincerely,

Gregor V. Bochmann

Gregor V. Bochmann

3 mai 82

Title: Delegate's Report of the CCITT SG VII Rapporteurs meeting on Question 39 (FDT) in Melbourne, March 1982.

From: G.v. Bochmann

The meeting was attended by 23 delegates, lasted six (working) days, and more than 30 contributions were discussed. Most of the work was performed in plenary meetings. It is to be noted that several representatives from SG XI participated in the meeting. A liaison report from ISO TC97/SC16/WG1 on its FDT work was presented by G.v. Bochmann.

While some time was spent with the discussion of various specification techniques (including abstract data types), and an ad-hoc group on Petri net description was formed, most time was spent with the discussion of the extended state transition FDT. The main results of the meeting are the elaboration of a "Common semantic model for CCITT and ISO" (annex 7 of the minutes), and a proposal for a linear specification sntax (annex 8 of the minutes) which is a revision of the syntax included in the working document of Subgroup B of the ISO TC97/SC16/WG1 ad hoc group on FDT. An effort has been made at the meeting to bridge differences between the ISO Subgroup B proposal and the existing SDL Recommendation by changing the ISO proposal, and to indicate how the future extensions of SDL could follow the present ISO Subgroup B language.

The proposals included in the Canadian contributions have been discussed during the meeting. The following points, as decided at the meeting, do not completely follow the Canadian proposals: (1) The syntax of the specifcation language is not based on Pascal, but several versions of specification language are foreseen, at least the following two:

(a) based on Pascal (a revision of the Subgroup B proposal, see above),

(b) based on CHILL.

(2) The use of simple state diagrams (as in X.25) are not explicitely included as a FDT. It was avoided to make any definite statement on this issue. The same applies to time sequence diagrams.

(3) No definite "priority" was given to the linear form of specifications. Both linear and graphical versions are considered at equal footing, although it is mentioned that the linear form "... should always be given and be regarded as an authoritative specification".

ANNEX 13

Un compilateur pour la traduction de spécifications de protocoles en Pascal

par

Michel Gagné

Document de travail #120

Département d'informatique et de
recherche opérationnelle
Université de Montréal

Février 1982

Tables des matières

1. Introduction

Ce document décrit un compilateur qui traduit la spécification d'un module donnée dans le langage de spécification de protocoles (LSP) en un programme en Pascal. Puisque le LSP utilise en grande partie la syntaxe et sémantique de Pascal, une grande partie de la traduction est une recopie sans modification de la spécification source. Les parties non copiées, c'est-à-dire générées par le compilateur, se conforment au règles de Pascal ISO [1].

Le LSP accepté par le compilateur est une version préliminaire, similaire au langage de spécification développé par ISO TC97/SC16/WG1 ad hoc group on FDT ("Formal description techniques") [2]. Une description du langage accepté par le compilateur est donnée dans l'annexe.

Le compilateur décrit dans ce document a été réalisé comme projet d'été 1981, et a été utilisé pour la traduction d'une spécification de protocole de Transport, et pour la traduction d'une spécification du protocole "Document" de Teletex [3] dans le cadre du cours IFT 6052 à l'automne 1981.

Le compilateur a été réalisé à l'aide d'un système d'écriture de compilateurs [4,5]. La partie de l'analyse syntaxique, incluant le traitement des erreurs syntaxiques (pas toujours satisfaisant), est faite par le système d'écriture de compi-

lateurs; une analyse sémantique de certaines parties de la spécification traduite et sa traduction en Pascal sont réalisées par des procédures écrites spécialement pour cet effet.

Dans la section 2 de ce document, on trouve une description des vérifications sémantiques faites par le compilateur, et de l'approche à la traduction en Pascal. Une notice d'utilisation est donnée dans la section 3. La section 4 contient un petit exemple qui montre la traduction effectuée par le compilateur. Les annexes contiennent une description du LSP, et la syntaxe complète (incluant les règles de syntaxe Pascal) acceptée par le compilateur.

Références:

[1] ISO DP 7185

[2] ISO TC97/SC16/WG1 ad hoc group on FDT, Subgroup B: working document, Dec. 1981.

[3] CCITT Recommendation S.62(1980).

[4] G.V. Bochmann and P. Ward, "Compiler writing system for attribute grammars", The Computer Journal 21, No.2 (1977), pp. 144-148.

[5] P. Ward, "Un système d'écriture...", Doc. de travail #55, Département d'informatique et de recherche opérationnelle, Université de Montréal.

2. Le traducteur

(Pour cette section se référer aux textes des programmes de compilateur). Le compilateur est constitué de 2 parties:

- L'analyseur lexical et syntaxique contenant les appels aux actions sémantiques.
- Les actions sémantiques (procédure SEM déclarée externe à l'analyseur lexical et syntaxique).

L'analyseur lexical et syntaxique

L'analyseur a été engendré par le générateur d'analyseur syntaxique LL(1) de l'Université de Montréal (Patrick Ward).

Les règles de grammaire dans la description intégrée ont la forme suivante:

si <A> est la 24e catégorie dans la suite de définitions des catégories et si (par exemple)

```
<A> = 'AA' <AA> <AB> | <AC> 'AB'
```

on a

```
<A> = : sem(24 000); $ 'AA' : sem(24 001);$
      <AA> : sem(24 002);$ <AB> : sem(24 003);$
          | : sem(24 004);$ <AC> : sem(24005);$ 'AB'
            : sem(24 006);$
```

(voir aussi l'exemple 3 à la section 4)

i.e. un appel aux actions sémantiques a été inséré au début, à la fin, et entre chaque lexeme et catégorie de la partie droite de la règle.

Certaines procédures produites par le générateur d'analyseur LL(1) ont été refaites, par exemple, les procédures Erreur, Lignederreur, Caractère et Déjaluc. On a ajouté au début de la procédure Lexical l'appel à la procédure Sortlex, ainsi qu'une petite modification pour permettre de retenir le dernier identificateur lu (la variable Derniermotlu).

Certaines procédures ont été ajoutées. A chaque appel de la procédure Lexical, les caractères qu'elle traite sont gardés dans la variable Reservelex. Les procédures Augmenter, Diminur Stockr sont reliées à Reservelex.

Sortlex écrit sur un (des) fichier(s) approprié(s) l'unité lexicale gardée dans Reservelex. La procédure Plisting fait le "sommaire des erreurs" et la "signification des erreurs" (s'il y a lieu). Commentaire et Carspecial traitent les commentaires LSP qui ont le même format qu'en Pascal.

Les actions sémantiques

La procédure Sem qui exécute les actions sémantiques, est composée de 3 étapes distingues:

- l'entrée des symboles dans les tables faite par la procédure ENTRER,
- les vérifications sémantiques faites par la procédure VERIFIER,
- la traduction, faite par la procédure TRADUIRE.

Les trois procédures ENTRER, VERIFIER, TRADUIRE ont une construction similaire i.e.

```
  Cas (no div 1000) de
1: ---
2: ---
------

n: Cas (no mod 1000) de
      0: ---
      1: ---
      autrement fini
 autrement fini
```

Remarquez que les actions sémantiques ont le même nombre de fichiers et le même bloc (sauf pour les procédures) que leurs correspondants dans l'analyseur lexical et syntaxique.

2.1 La structure de stockage de l'information dans LSP

Le préprocesseur LSP produit 3 classes de structures dans son exécution.

1- Une structure bloc comme en Pascal.

2- Une liste d'interactions/PDU dont le premier membre est pointé par la variable Pduinter.

3- Une description de l'entête du module pointée par la variable Module.

2.1.1 Description d'un bloc

Un bloc est une structure contenant 6 champs:

```
Tbloc = Struct
          Etiquette: Ptrlistentier;
          Constantes: Ptrlistconst;
          Types: Ptrlisttype;
          Variables: Ptrlistvar;
          Procetfonc: Ptrlistpf;
          Blocpere: Ptrbloc;
        Fin;
```

Etiquette: pointe sur une liste d'étiquettes (au sens Pascal).

Constantes:..... de constantes (au sens Pascal).

Types:..... de types .....

Variables:..... de variables .....

Procetfonc:..... de fonctions ou procédures (au sens de Pascal).

Blocpere: Dans le cas ou le bloc en question est celui associé a

une procédure ou fonction PF, blocpere pointe vers le bloc qui a PF comme un des membres de sa liste de procédures ou fonctions.

2.1.1.1 Description d'un élément de la liste d'étiquettes (Tlistentier)

```
Tlistentier = Struct
                Etiquette:Entier;
                Suivante:Ptrlistentier;
              Fin;
```

2.1.1.2 Description d'un élément de la liste de constantes (Tlistconst)

```
Tlistconst = Struct
               Nom: Lspalfa;
               Defconst: Ptrconst;
               suivante: Ptrlistconst;
             Fin;

Lspalfa = Chainident;    (* = paquet tableau [1..30] de car*)
Ptrlistconst = ^Tlistconst;
Defconst : description de constante

Ptrconst = ^Tconst;
Tconst = Struct
```

```
            signe:booleen;
            cas ctype:entier de
                0:();
                1:( creel: reel);
                2:( centier: entier);
                3:(Cdebutchaine,Clongchaine: entier);
                4:(Idconst:Lspalfa);
            Fin;
```

Signe: vrai pour +, faux pour -.

Cas 0 : pour signaler une erreur possible.

Cas 1 : la constante est réelle.

Cas 2 : la constante est entière.

Cas 3 : constante de type chaine,

Cdebutchaine: début de la chaîne lorsque stockée dans Zonechaine par la procédure Constchaine.

Clongchaine: longueur de la chaîne.

Cas 4: la constante est un identificateur de constante.

2.1.1.3 Description d'un élément de la liste des types

```
Tlisttype = Struct
            Nom:Lspalfa;
            Deftype:Ptrtype;
            Suivante:Ptrlisttype;
          Fin;
```

Suivante: le suivant dans la liste.

```
Ptrlisttype = ^Tlisttype;
```

Deftype: Descriptif de type.

```
Ptrtype = ^Ttype;
Ttype = Struct
          Pacquete:Booleen;
          Cas choixtype:Entier de
                  0:();
                  1:(Nom:Lspalfa);
                  2:(Lscalaire:Ptrlistroles);
                  3:(C1,C2:Ptrconst);
                  4:(Tpointeur:Lspalfa);
                  5:(Ltypesimple:Ptrltypsimple;
                     Ttabtype:Ptrtype);
                  6:(Typefichier:Ptrtype);
                  7:(Typeensemble:Ptrtype);
                  8:(Typeenregistrement:Ptrlistenreg);
        Fin;
```

Le descriptif Ttype correspond aux différents types possibles:

Pacquete: indique si c'est une structure pacquete ou non.

1) Nom

Dans les déclarations (de types) du genre

A = B ou B est un type défini avant

2) Lscalaire

Pointe sur une liste d'identificateurs.

Pour les déclarations de types énumérés

A = (B,C,D)

3) Dans le cas d'un type intervalle

Ex: A = 'A' .. 'Z' ;

C1: Pointe vers le descriptif de 'A'

C2: ..... 'Z'

Le descriptif étant celui d'une constante, ce que l'on a vu dans la description de la liste de constante.

4) Dans le cas d'un type pointeur

Ex: A = ^PT;

Tpointeur contient l'identificateur (PT dans l'exemple)

5) Dans le cas d'un tableau

Ex: A = array [toto , '0' .. '9'] of integer

- Ltypesimple pointe vers le début d'une liste de descriptifs de types (simples) (toto et ensuite '0' .. '9' dans l'exemple).
- Ttabtype est un pointeur vers le descriptif du type des éléments du tableau (dans l'exemple, vers le descriptif de integer).

6) Dans le cas d'un fichier

Ex: A = file of real

- typefichier pointe vers le descriptif du type des éléments du fichier (dans notre exemple, vers le descriptif de Real).

7) Dans le cas d'un ensemble

Ex: A = set of 'A'..'Z'

- typeensemble pointe vers le descriptif du type des éléments de l'ensemble (dans notre exemple vers le descriptif de 'A'..'Z').

8) Dans le cas d'un enregistrement, typeenregistrement pointe vers un descriptif d'un type enregistrement.

```
Ptrlistenreg = ^ Tlistenreg
Tlistenreg = Struct
                Partiefixe:Ptrlistvar;
                Bvariante:Booleen;
                Bidentcas:Booleen;
                Selectid:Lspalfa;
                Selecttype:Lspalfa;
                Listcas:Dtrlisteas;
             Fin;
```

Partiefixe pointe sur une liste de champs avec leurs types.
Bvariante indique si l'enregistrement contient une variante 'case'.
Bidentcas indique si on a un champ sélecteur dans le 'case'.
Selectid est le champs sélecteur (si bidentcas).
Selectype est le type dans le 'case'.
Listcas est une liste dont chaque élément contient une liste de constantes et un pointeur (de type Ptrlistenreg) sur un enregistrement.

2.1.1.3 Description d'un élément de la liste des variables

```
PTRlistvar = ^Tlistvar;
Tlistvar   =  Struct
                 Lident:PTRlistroles;
```

```
        Vtype:PTRtype;
        Suivante:PTRlistvar;
      Fin;
```

Lident est une suite d'identificateurs ayant un même type (on peut avoir une déclaration de variables de la façon suivante A,B,C: typeABC;)
Vtype est un pointeur sur un descriptif de type. Suivante pointe vers le descriptif de variable suivant.

2.1.1.4 Description d'un élément de la liste des fonctions/procédures

```
PTRlistPF = ^Tlistprocetfonc;
Tlistprocetfonc = Struct
                    nom:Lspalfa;
                    LPara:PTRlpara;
                    Bexterne,BPlusloin:Booleen;
                    PFbloc:PTRbloc;
                    Suivante:PTRlistpf;
                    Cas Fonc:Booleen de
                      vrai:(Restype:Lspalfa);
                      faux:();
                  Fin;
```

Nom est l'identificateur de la procédure ou de la fonction.
LPARA pointe sur une liste de paramètres.
Bexterne (resp. Bplusloin) indique si la procédure/fonction est

déclarée externe (resp. plusloin).

PFBloc pointe sur le "bloc" de la procédure/fonction.
Suivante pointe sur la description de procédure/fonction suivante.
Fonc indique si c'est une fonction (vrai) ou une procédure (faux).
Restype est l'identificateur du type du résultat dans le cas d'une fonction.

2.1.2 Description de la liste d'interaction/PDU

```
PDUinter:PTRPDUinter;
PTRPDUinter=^TPDUinter;
TPDUinter=Struct
            Nom:LSPalfa;
            BPDU:Booleen;
            Listeactions:PTRLroleaction;
            Listroles:PTRlistroles;
            suivante:PTRPDUinter;
          fin;
```

Nom est l'identificateur associé à l'interaction/PDU.
BPDU indique si c'est un PDU (vrai) ou une interaction (faux).
Listeactions pointe sur une liste de listes d'actions.
Listroles est la liste des roles possibles de l'interaction/PDU telle qu'indiquée au début de la déclaration.
Suivante pointe vers le descriptif de l'intersection /PDU suivant.

2.1.3 Description du module

```
Module:PTRmodule;
PTRmodule=^Tmodule;
Tmodule=Struct
        Nom:LSPalfa;
        Listinterfaces:PTRlistinterfaces;
       Fin;
```

Nom est le nom du module.

Listinterfaces pointe sur une liste de descriptifs d'interfaces.

```
PTRlistinterfaces=^Tlistinterfaces;
Tlistinterfaces=Struct
                    Lnom:PTRlistroles;
                    Listindice:PTRltypmodule;
                    Nompduinter:LSPalfa;
                    Listroles:PTRlistroles;
                    Bwith:Booleen;
                    Withpduinter:LSPalfa;
                    Listrolewith:PTRlistroles;
                    Suivante:PTRlistinterfaces;
                   Fin;
```

Lnom est une liste d'identifications ayant le même 'type' d'interface.

Listindice pointe sur liste de types correspondant aux types des indices de tableau (s'il y a lieu) dans la déclaration d'(es) interface(s).

Nompduinter est le nom de l'interaction associée à l'interface.
Listroles est la liste des roles possibles dans l'interaction.
Bwith indique si l'option 'with PDUid (...)' a été utilisée dans la déclaration. Dans ce cas withpduinter est le nom du PDU et listrolewith est la liste des roles possibles du PDU.
Suivante pointe vers le descriptif d'interface suivant.

2.2 Les vérifications sémantiques

Prenons comme exemple la déclaration d'interaction suivante: (ce qui suit s'applique aussi dans le cas d'une déclaration de PDU).

```
Interaction   Userinterface   (toto1, toto2, toto3) is
                                          (1)

    By toto1:  Connect (Inf1: Infotype1;
        (2)                Inf2: Infotype2);
    By toto2, toto3: Ceci (est: integer;  exemple: Real);
        (3)            Egalement (celui: çi; autre:typepara)
```

1) Il faut que les identificateurs dans les listes en (2) et (3) fassent partie de la liste (1). Lorsque cette règle n'est pas vérifiée une erreur no. 400 est déclarée.

2) Une liste d'identificateurs (comme en (1), (2), (3)) ne peut contenir deux fois le même identificateur (sinon erreur no. 401).

3) Dans la règle

```
<interaction-paramete> = '('<liste-ident> ':' ident
                                                    (1)
                         *[ ';' <liste-ident> ':' ident]*  ')'
                                                    (2)
                          | vide
```

les identificateurs en (1) et (2) doivent être des types déclarés auparavant (sinon erreur no. 402)

4) Une variable ne peut porter le nom d'une interface (il y aurait alors possibilité de confusion) (autrement erreur no. 403).

5) Dans la règle

```
<type-nouveau> = Ident '(' <liste-ident> ')'
                  (1)              (3)
                ['with' ident '(' <liste-ident> ')' | vide]
                    (2)              (4)
                 | 'array'  ..,
                 ...
```

a) l'identificateur en (1) doit désigner le nom d'une interaction déclarée auparavant (sinon erreur no. 405)

b) l'identificateur en (2) doit désigner le nom d'un PDU déclaré auparavant (sinon erreur no. 404)

c) les identificateurs dans les listes en (3) et (4) doivent faire partie de la liste au début de la définition de l'inte-

raction (resp.PDU) correspondante (sinon erreur no.406).

6) Dans la règle

<when-condition> = 'when' [<expression>|Ident *[<indicage>]*

(3) (4)

<designe-champs>'('<liste-ident>')']

(1) (2)

(Rem: <designe-champs> = '.' ident

(1)

<liste-ident> = ident *[',' ident]*)

d) les identificateurs en (2) doivent être les mêmes et dans le même ordre que dans l'action dont le nom est l'identificateur (1) dans l'interaction (ou le pdu) de la définition de l'interface identifiée par (3) (sinon erreur no. 409)

e) le nombre d'indices en (4) doit être conforme à la définition de l'interface identifiée par (3) (sinon erreur no. 407).

7) Dans les règles

<action-list> = ident <suite AAA>

(4)

|...

...

<suite AAA> = *[<S-indicage>]*[<désigne-champs <interA>

(1)

| <pointage> ...

...

<inter A> = '('<expressions *[',' <expression>]* ')' (2)

| <suite-affec>

| vide

(Rem: <désigne-champs> = '.' ident)

(3)

l'identificateur de champ en (3) (provenant de (1)) doit être une action possible selon la définition de l'interface (4) (sinon erreur no. 408). Par action possible on entend une action qui, dans la définition de l'interaction (PDU) correspondante, appartient à la liste d'actions (<interaction-list>) associée à une liste de rôles qui contient l'identificateur (3).

De plus le nombre de paramètres en (2) doit être égale au nombre de paramètres dans la définition de l'action correspondante en (3) (sinon erreur no. 410).

2.3 Le code Pascal engendré

(se référer à l'exemple de traduction à la section 4)

1o Les soulignés permis dans un identificateur LSP sont éliminés lors de la traduction.

2o Le nom du programme engendré est celui du module dans le programme LSP.

3o Le seul fichier de l'entête est le fichier output.

4o Dans la règle LSP

<prog> = *[<pdefconstl> |<pdeftypel> | <pdu>
|<interface-definition>]* <module>

toutes les constantes définies dans l'une ou l'autre des occurences de la règle <pdefcontl> sont regroupées dans un seul bloc de constantes de type Pascal. De même pour les types. Dans les deux cas la traduction est directe i.e. c'est une copie.

5o A partir de la déclaration du module et des interactions (resp. PDU) le compilateur construit 2 types: ZZZT et ZZZTB. ZZZT est constitué d'un enregistrement avec cas pour chaque interface déclarée dans le module (dans le même ordre d'apparition). Le champs de chaque cas est constitué de l'identificateur de l'interface correspondant. Le type du champ est

lui-même un enregistrement constitué (possiblement) d'indices I1, I2, I3,... dans le cas que l'interface est un tableau (autant d'indices qu'il y a de dimension dans le tableau) et d'une variante (avec pour sélecteur le champ CTF) pour chacune des actions possibles dans la définition de l'interaction indiquée dans la définition de l'interface. Lorsque l'option 'with Ident (<liste-ident>)' est utilisée alors la liste des actions possibles pour le PDU correspondant (à Ident) est ajoutée à la suite.

Chaque action possible a un type enregistrement composé des paramètres (et des types) de l'action dans la définition de l'interaction (resp. PDU).

ZZZTB est un enregistrement ayant comme champs les noms des interfaces. Ces champs ont comme type soit le type Booleen soit un tableau de Booleen conforme au tableau de l'interface correspondante.

6o Les variables dans <P-decl-var> de la règle <global-constraints> sont traduits directement (copie) sauf que les variables suivantes sont ajoutées

```
ZZZR, ZZZS : ZZZT;
ZZZB : ZZZTB;
```

ZZZR est une variable, conforme au format des interfaces du module, qui sert à recevoir l'information de l'extérieur. Il n'y a que la procédure wait qui peut la modifier. ZZZS, de même type, sert à envoyer de l'information vers l'extérieur. Il n'y a que le programme qui peut la modifier lors d'une

instruction 'send'.

ZZZB est une variable, modifiée par la procédure wait, qui sert à indiquer au programmeur quels sont les actions reçues depuis le dernier 'wait'.

7o Les procédures et fonctions dans <P-decl-Proc-fonc> de la règle <global-constraints> sont traduites directement (copies). Les procédures wait et send (externes) sont ajoutées.

8o Les 'transitions' sont traduites de la façon suivante:

```
Begin (*program*)
while true do
  Begin
  Wait(ZZZR,ZZZB);
        ......
     Traduction des 'when ... else ...'
        ......
  end;
end.
```

9o Considérons les règles

```
<when-clause> = <when-condition> [<when-list>|<action>]
<when-list>   = + [<when-clause>] + 'else' <action-list>
<action       = 'DO' <action-list>
<when-condition> = 'when' [<expression> |
                        ident *[<indicage>]* <designe-champs>
```

['('<liste-ident> ')' | vide]]

<when-condition> est traduit par:

if <expression>

ou

if ZZZB. Ident *[<indicage>]* and

(ZZZR. Ident.CTF=e)

Selon l'alternative de la règle.

e = l'ordre dans ZZZT de l'action indiquée par l'identificateur dans <désigne-champs>

la traduction de <when-clause> est alors:

traduction de <when-condition>'then'[traduction de <when-list>|

traduction de <action>] 'else'

la traduction de <when-list> est:

'begin'+[traduction de <when-claude>]+traduction de <action-list>'end'

la traduction de <action> est:

traduction de <action-list>

(la traduction de <action-list> est vue en 11o)

10o Si dans une expression (<expression>) un identificateur A est dans une des <liste-ident> d'un des <when-condition> qui l'imprique alors, si

when IdentA *[<indicage>]* <désigne-champs> (<liste-ident>)

est le plus rapproché (des <when-condition>) qui vérifie la

condition énoncée plus haut, A est remplacé par

ZZZR.IdentA *[<indicage>]* <désigner-champs>.A

dans l'expression.

11o Une <action-list> est soit une instruction Pascal, soit l'envoie d'une action. Une instruction Pascal est traduite sans modification sauf pour les expressions (10o).
L'envoie d'une action est traduite par un code Pascal qui affecte la variable d'envoie ZZZS par les valeurs appropriées et envoie ZZZS avec la procédure Send (voir l'exemple de traduction).

3. NOTICE D'UTILISATION

3.1 Cartes de contrôle nécessaires

```
*Job,....
*Code
 ...
LIB, L, LSP, U=1837
LSP(F1,F2,F3,N)
```

F1: Fichier contenant le programme LSP.

F2: Fichier de sortie du listing et autres informations du genre (erreurs, sommaire).

F3: Fichier contenant le programme PASCAL provenant de la traduction du programme LSP.

N: Nombre en octal spécifiant le RFL donné à la job.

Les valeurs par défaut sont:

pour F1: input,

F2: output,

F3: Lgop,

N: 60000.

3.2 Remarques

- Le préprocesseur LSP a été développé dans un environnement CDC Cyber 173. Le langage utilisé est le Pascal 6000 version 3.2.2.
- En cas de "BUG" du préprocesseur, l'utilisateur est prié de communiquer avec M. Michel Gagné ou l'usager 1837, ou M. Gregor Bochmann.
- Les utilisateurs désirant enjoliver le programme Lgop produit par le préprocesseur LSP, peuvent utiliser les formateurs du genre PASTAB, JOLI, PRETTY dont la documentation pour certains est disponible sur Bonjour.

3.3 Echantillons de résultats et comment les interpréter

Lors d'une exécution du programme LSP, en utilisant la carte LSP (F1,F2,F3,N), le fichier F2 (par défaut output) contient des informations sur le programme LSP fourni (sur F1).

Le fichier F2 est composé jusqu'à 3 parties:

(voir le "listing" à la section 4)

A) Le texte du programme, avec une numérotation par accroissement

de 1.

- Sous les lignes contenant une erreur LSP, on a une suite de 12 étoiles ainsi qu'une flèche sous le caractère ou a été détecté une erreur. La plupart du temps, le caractère pointé n'est pas en faute; c'est (en général) l'une des deux dernières unités lexicales avant le caractère pointé.
- Une erreur décelée à la toute fin du texte n'est pas indiquée sur celui-ci mais seulement dans le sommaire des erreurs.

Lorsque le programme comporte des fautes, F2 comporte aussi 2 autres sections:

B) Un sommaire des erreurs comprenant pour chaque erreur décelée sa ligne, sa colonne (position du caractère dans la ligne) et son numéro d'erreur.

C) Une brève description des numéros d'erreur chapeautée par le titre "Signification des erreurs".

4. Exemples

"Exemple 1:

Programme LSP avec erreurs"

```
LSP                                          UNIVERSITE DE MONTREAL                    12.34.28.
PREPROCESSEUR  LSP                                                                     VERSION 1

     1  CONST
     2
     3    CAA=3;
     4    AAA=2;
     5
     6  TYPE
     7
     8    ENS = 0..10;
     9    COMMON = BOOLEAN;
    10    INFO = RECORD
    11             P1:BOOLEAN;
    12             P2:INTEGER;
    13             P3:SET OF ENS;
    14           END;
    15
    16  INTERACTIONS
    17
    18     LOCALBUF(USER,BUFFER) IS
    19
    20       BY USER:   CLEAR(INFRAG:INTEGER;
    21                        OUTFRAG:COMMON));
    22
    23                  REQUESTDATA;
    24
    25       BY BUFFER : SENDDATA(DATAFRAG:INFO));
    26
    27  CONST
    28     BROUAA = 5;
    29
    30  TYPE
    31     STATUS = INTEGER;
    32
    33  INTERACTIONS
    34
    35     TSPOINT(USER,PROVIDER,PROVIDER) IS
**********                                  ^
    36
    37       BY USER :
    38                  CONNECTREQ(AA:BOOLEAN; AB:COMMON; AC:STATUS));
**********                                            ^
    39
    40                  DISCONREQ(BA:STATUS;BB:INFO));
    41
    42                  DATAREQ(CA:INFO));
    43
    44       BY PROVIBER : CONNECTIND(DA:BOOLEAN; DB:COMMON; DC:STATUS));
**********                 ^
    45                     DISCONIND(EA: STATUS; EB:INFO));
    46                     DATAIND(FA:INFO));
    47
    48  PDU
    49
    50     TPDU(USER,PROVIDER) IS
    51
    52       BY USER : SREF;
    53                 SDATE;
    54                 SCRAM(DEHORS:COMMON; DEDANS:COMMON));
    55
    56
    57
```

```
LSP                                          UNIVERSITE DE MONTREAL                    12.34.30.
PREPROCESSEUR  LSP                                                                     VERSION 1

    58       BY PROVIDER : PREF;
    59                     PDATE;
    60                     RAM(OUTSIDE:STATUS; INSIDE:INFO)); (* POURQUOI PAS *)
    61
    62    MODULE ESSAI (TRANSPORT: ARRAY[1..5] OF ARRAY[2..3] OF
    63                                STSPOINT(USER) WITH STPDU(PROVIDER));
**********                          ^                       ^
    64                  LOCBUF : LOCALBUF(USERR)) IS
**********                                    ^
    65
    66     VAR
    67
    68       VA:COMMON;
    69       VS:STATUS;
    70       VI:INFO;
    71       BVA:COMMON;
    72       BVB,BVC,BVD:BOOLEAN;
    73         LOCBUF:STATUS;
**********             ^
    74
    75
    76  TRANSITIONS
    77
    78
    79     WHEN TRANSPORT[VS].PREF
**********                     ^
    80       WHEN VA
    81         WHEN BVA DO LOCBUF.CLEAR(10,TRUE)
    82         WHEN BVB DO VI.P2:=3
    83       ELSE VI.P3 := [1..4]
    84
    85       WHEN BVA DO TRANSPORT[VS,VI.P2].CONNECTREQ(TRUE,FALSE,33)
    86
    87     ELSE VS:=0
    88
    89     WHEN TRANSPORT[VS,VS].RAM (INSIDE,OUTSIDE)
    90       WHEN VA DO VS:=OUTSIDE
**********                    ^
    91     ELSE VS:=3;
    92
    93
```

SOMMAIRE DES ERREURS

```
LIGNE:    37      COLONNE:   38      ERREUR NO:401
LIGNE:    40      COLONNE:   54      ERREUR NO:402
LIGNE:    46      COLONNE:   22      ERREUR NO:400
LIGNE:    63      COLONNE:   39      ERREUR NO:405
LIGNE:    63      COLONNE:   56      ERREUR NO:404
LIGNE:    64      COLONNE:   43      ERREUR NO:406
LIGNE:    73      COLONNE:   19      ERREUR NO:403
LIGNE:    79      COLONNE:   26      ERREUR NO:407
LIGNE:    90      COLONNE:    1      ERREUR NO:409
```

SIGNIFICATION DES ERREURS

```
400: UN DES ROLES DE LA DERNIERE LISTE NE FAIT PAS PARTIE DE LA LISTE DE LA DEFINITION INITIALE.
401: REPETITION D'IDENTIFICATEURS DANS UNE LISTE INTERDITE.
402: TYPE NON DECLARE.
403: UN IDENTIFICATEUR DE VARIABLE NE PEUT ETRE LE MEME QUE CELUI D'UNE INTERFACE DECLAREE DANS LE MODULE
404: PDU NON DECLARE.
405: INTERACTION NON DECLARE.
406: UN DES ROLES DE LA LISTE EST ABSENT DANS LA DEFINITION DE L'INTERACTION (OU DU PDU).
407: NOMBRE D'INDICE DE TABLEAU EN DESACCORD AVEC LA DECLARATION.
409: IDENTIFICATEUR(S) DE PARAMETRE(S) EN DESACCORD AVEC LA DEFINITION DE L'ACTION.
```

```
CONST
  CAA=3;
  AAA=2;

TYPE
  ENS = 0..10;
  COMMON = BOOLEAN;
  INFO = RECORD
           P1: BOOLEAN;
           P2: INTEGER;
           P3: SET OF ENS;
         END;

INTERACTIONS

   LOCALBUF(USER, BUFFER) IS

     BY USER:   CLEAR(INFRAQ: INTEGER;
                      OUTFRAQ: COMMON);

                REQUESTDATA;

     BY BUFFER : SENDDATA(DATAFRAQ: INFO);

CONST
   BROUAA = 5;

TYPE
   STATUS = INTEGER;

INTERACTIONS
   TSPOINT(USER, PROVIDER) IS

     BY USER :
                CONNECTREQ(AA: BOOLEAN; AB: COMMON; AC: STATUS);

                DISCONREQ(BA: STATUS; BB: INFO);

                DATAREQ(CA: INFO);

     BY PROVIDER : CONNECTIND(DA: BOOLEAN; DB: COMMON; DC: STATUS);
                   DISCONINO(EA: STATUS; EB: INFO);
                   DATAIND(FA: INFO);

PDU
   TPDU(USER, PROVIDER) IS

      BY USER : SREF;
                SDATE;
                SCRAM(DEHORS: COMMON; DEDANS: COMMON);

      BY PROVIDER : PREF;
                    PDATE;
                    RAM(OUTSIDE: STATUS; INSIDE: INFO); (* POURQUOI PAS *)
```

"Exemple 2:

Programme LSP sans erreurs;

la traduction est à la page suivante"


```
MODULE ESSAI (TRANSPORT: ARRAY[1..5] OF ARRAY[2..3] OF
                         TSPOINT(USER) WITH TPDU(PROVIDER);
              LOCBUF : LOCALBUF(USER)) IS

   VAR

      VA: COMMON;
      VS: STATUS;
      VI: INFO;
      BVA: COMMON;
      BVB, BVC, BVD: BOOLEAN;

TRANSITIONS

   WHEN TRANSPORT[VS, VS].PREF
     WHEN VA
       WHEN BVA DO LOCBUF.CLEAR(3, TRUE)
       WHEN BVB DO VI.P2:=3
     ELSE VI.P3 := [1..4]

     WHEN BVA DO TRANSPORT[VS, VI.P2].CONNECTREQ(TRUE, FALSE, 33)

   ELSE VS:=8

   WHEN TRANSPORT[VS, VS].RAM (OUTSIDE, INSIDE)
     WHEN VA DO VS:=OUTSIDE
   ELSE VS:=3;
```

```
    1  PROGRAM ESSAI(OUTPUT);

    3  CONST

    5       CAA=3;
    6       AAA=2;

    8         BROUAA = 5;

   11  TYPE


   14       ENS = 0..10;
   15       COMMON = BOOLEAN;
   16       INFO = RECORD
   17                 P1:BOOLEAN;
   18                 P2: INTEGER;
   19                 P3:SET OF ENS;
   20               END;


   23         STATUS = INTEGER;

   25  ZZZT =
   26  RECORD
   27   CASE TF: INTEGER OF
   28  1 : ( TRANSPORT :  RECORD
   29  I1:1..5;
   30  I2:2..3;
   31   CASE CTF:INTEGER OF
   32  1 :  ( CONNECTREQ :  RECORD
   33  AA : BOOLEAN;
   34  AB : COMMON;
   35  AC : STATUS;
   36   END; );
   37  2 :  ( DISCONREQ :  RECORD
   38  BA : STATUS;
   39  BB : INFO;
   40   END; );
   41  3 :  ( DATAREQ :  RECORD
   42  CA : INFO;
   43   END; );
   44  4 :  ( CONNECTIND :  RECORD
   45  DA : BOOLEAN;
   46  DB : COMMON;
   47  DC : STATUS;
   48   END; );
   49  5 :  ( DISCONIND :  RECORD
   50  EA : STATUS;
   51  EB : INFO;
   52   END; );
   53  6 :  ( DATAIND :  RECORD
   54  FA : INFO;
   55   END; );
   56  7 :  ( SREF :  RECORD
   57   END; );
   58  8 :  ( SDATE :  RECORD
   59   END; );
   60  9 :  ( SCRAM :  RECORD
```



```
   61  DEHORS : COMMON;
   62  DEDANS : COMMON;
   63   END; );
   64  10 :  ( PREF :  RECORD
   65   END; );
   66  11 :  ( PDATE :  RECORD
   67   END; );
   68  12 :  ( RAM :  RECORD
   69  OUTSIDE : STATUS;
   70  INSIDE : INFO;
   71   END; );
   72   END;
   73  );
   74  2 : ( LOCBUF :  RECORD
   75   CASE CTF:INTEGER OF
   76  1 :  ( CLEAR :  RECORD
   77  INFRAG : INTEGER;
   78  OUTFRAG : COMMON;
   79   END; );
   80  2 :  ( REQUESTDATA :  RECORD
   81   END; );
   82  3 :  ( SENDDATA :  RECORD
   83  DATAFRAG : INFO;
   84   END; );
   85   END;
   86  );
   87  END;
   88
   89  ZZZTB =
   90  RECORD
   91  TRANSPORT:  ARRAY[1..5] OF ARRAY[2..3] OF
   92   BOOLEAN
   93  ;
   94                       LOCBUF :  BOOLEAN
   95  ;
   96  END;
   97
   98
   99  VAR
  100
  101  ZZZR, ZZZS: ZZZT;
  102  ZZZB: ZZZTB;
  103
  104            VA: COMMON;
  105  (*   69*)          VS: STATUS;
  106  (*   70*)          VI: INFO;
  107  (*   71*)          BVA: COMMON;
  108  (*   72*)          BVB, BVC, BVD: BOOLEAN;
  109  (*   73*)
  110    PROCEDURE WAIT(VAR T:ZZZT; VAR B:ZZZTB); EXTERN;
  111    PROCEDURE SEND(T:ZZZT); EXTERN;
  112
  113  BEGIN  (* PROGRAM *)
  114
  115   WHILE TRUE DO
  116  BEGIN
  117
  118  WAIT(ZZZR,ZZZB);
```



```
  119
  120  IF ZZZB.TRANSPORT[VS
  121  ,VS
  122  ] AND (ZZZR.TRANSPORT.CTF = 10 )(*PREF*) THEN
  123
  124  BEGIN
  125
  126  IF VA
  127   THEN
  128
  129  BEGIN
  130
  131  IF BVA
  132   THEN
  133
  134  BEGIN  (* SEND OPERATION *)
  135  ZZZS.TF :=          2;
  136   WITH ZZZS.LOCBUF DO
  137
  138   BEGIN
  139
  140  CTF :=          1;
  141
  142  END;
  143   WITH ZZZS.LOCBUF.CLEAR DO
  144
  145  BEGIN
  146  INFRAG := 3
  147  ;
  148  OUTFRAG := TRUE
  149  ;
  150
  151  END;
  152  SEND(ZZZS);
  153  END (* SEND OPERATION *)
  154   ELSE
  155  IF BVB
  156   THEN
  157
  158   VI.P2:=3
  159   ELSE
  160
  161   VI.P3 :=[1..4]
  162   END
  163   ELSE
  164  IF BVA
  165   THEN
  166
  167  BEGIN  (* SEND OPERATION *)
  168  ZZZS.TF :=          1;
  169   WITH ZZZS.TRANSPORT DO
  170
  171   BEGIN
  172
  173  I1 := VS
  174  ;
  175  I2 := VI.P2
  176  ;
```



```
  177
  178  CTF :=          1;
  179
  180  END;
  181   WITH ZZZS.TRANSPORT.CONNECTREQ DO
  182
  183  BEGIN
  184  AA := TRUE
  185  ;
  186  AB := FALSE
  187  ;
  188  AC := 33
  189  ;
  190
  191  END;
  192  SEND(ZZZS);
  193  END (* SEND OPERATION *)
  194   ELSE
  195
  196   VS:=8
  197   END
  198   ELSE
  199  IF ZZZB.TRANSPORT[VS
  200  ,VS
  201  ] AND (ZZZR.TRANSPORT.CTF = 12 )(*RAM*) THEN
  202
  203  BEGIN
  204
  205  IF VA
  206   THEN
  207
  208   VS:=ZZZR.TRANSPORT.RAM
  209  .OUTSIDE
  210   ELSE
  211
  212   VS:=3
  213   END
  214   ELSE
  215
  216  END;
  217
  218  END.
```

COMPILER ESTIMATED 'W' OPTION = 2505B.

"Exemple 3:

Format de la description intégrée"


```
                    $EM(47000)!
                    <LISTE=IDENT>
                    DEBUT
                    LOCALCHAMP:=LCHAMPCOURANT;
                    LOCALTYPE:=DESCTYPECOURANT;
                    FIN;
                    $EM(47001)!
                    ';'
                    $EM(47002)!
                    <DENOTE=TYPE>
                    DEBUT
                    LOCALCHAMP^.VTYPE:=DESCTYPECOURANT;
                    DESCTYPECOURANT:=LOCALTYPE;
                    LCHAMPCOURANT:=LOCALCHAMP;
                    FIN;
                    $EM(47003)!
<PART=VARIANTE>     $
                    #
                    $EM(48000)!
                    'CASE'
                    $EM(48001)!
                    <SELECT=VARIANT>
                    $EM(48002)!
                    'OF'
                    $EM(48003)!
                    <VARIANTE>
                    $EM(48004)!
                    '|'
                    $EM(48005)!
                    ','
                    $EM(48006)!
```



```
                    $
                    <VARIANTE>
                    $EM(48007)!
                    '*'
                    $EM(48008)!
<SELECT=VARIANT>    $
                    #
                    $EM(49000)!
                    IDENT
                    $EM(49001)!
                    $EM(49002)!
                    ':'
                    $EM(49003)!
                    IDENT
                    $EM(49004)!
                    $EM(49005)!
                    VIDE
                    $EM(49006)!
                    $EM(49007)!
<VARIANTE>          $
                    #
                    $EM(50000)!
                    <LIST=CONST=CAS>
                    $EM(50001)!
                    ':'
                    LOCALCASCOURANT:=LISTCASCOURANT;
                    LOCALTENREG:=TENREGCOURANT;
```

Annexe 1

Introduction au langage LSP

ISO

INTERNATIONAL ORGANISATION FOR STANDARDISATION

ORGANISATION INTERNATIONALE DE NORMALISATION

ISO/TC97/SC16

OPEN SYSTEMS INTERCONNECTION

Source : Canada

Title: Tutorial on formal description techniques (FDT)

1. Introduction

The purpose of this paper is to describe some key techniques which can be used as part of the formal description techniques in specifying services and protocols for Open System Interconnection (OSI). The so-called formal description techniques (FDT) for OSI may be envisaged, at this stage, as a set of techniques used to accurately specify the complex nature of services and protocols. This paper particularly discusses two techniques, i.e. an "interaction model" for describing layer services, and a "state and transition model" for describing protocols.

The first part (section 2) describes an "interaction model" which is based on the principles outlined in the "Introduction to the Guidelines: Overall View of OSI Specifications" (section 1 of ISO/TC97/SC16 N 380). It provides a framework for specifying the interactions through which a layer provides its service. A possible syntax for this is defined in Annex 1.

The second part of the paper (section 3) describes a state transition model which was presented in a previous contribution (Amsterdam 13, "Comments on Formal Description Techniques"). This model may be applied to protocol specification by defining the behavior of a layer entity. Such a specification uses the concepts of (a) the state of the entity and (b) transitions between such states initiated by interactions and internal events. A possible syntax for this specification method is given in Annex 2. This specification technique may be complemented with additional specification techniques, such as state transition diagrams or transition tables.

Although the state and transition model has been found very useful for protocol descriptions, it is, however, not clear, at this stage, whether it is also useful for describing services. Probably other techniques may be more suitable for this purpose.

2. The interaction model

Section 1 of ISO/TC97/SC16/N 380 ("Introduction to the Guidelines: Overall View of OSI Specifications") gives an introduction to the main characteristics and the role of service, and protocol specifications for OSI. Many of the concepts discussed in the present paper are further explained in this "Introduction to the Guidelines".

A certain similarity exists between the requirements for service and protocol specifications. It is therefore possible to use certain techniques for both services and protocols. The following discussion uses the term "module" mainly in two different connotations: In the case of a (N)-service specification, the module considered consists of the layers of all Open Systems below the (N)-layer interface, i.e. the layers up and including the (N)-layer, as observable by the (entities within the) (N+1)-layer (see "Introduction", section 3.1.1). (The module is the functional unit that provides the service). In the case of a (N)-protocol specification, the module considered consists of the part of an Open System corresponding to the (N)-layer of the model of an Open System (also called (N)-layer subsystem) or of an entity contained in such a part, as observable by other entities within the same layer (see "Introduction", section 3.2 and 3.2.1). (The module is the functional unit that conforms to the protocol).

The following subsections discuss concepts for specifying the interactions of the module.

2.1. Interactions

The following examples are considered. The (N)-service is provided to the entities in the layer above by the interactions through the service access points between the service providing module and its environment. The interaction model is also useful to define interactions between different entities (or "modules") of an (N)-layer subsystem. For example, it may be used for defining the timer or data buffering services used in the (N)-layer protocol.

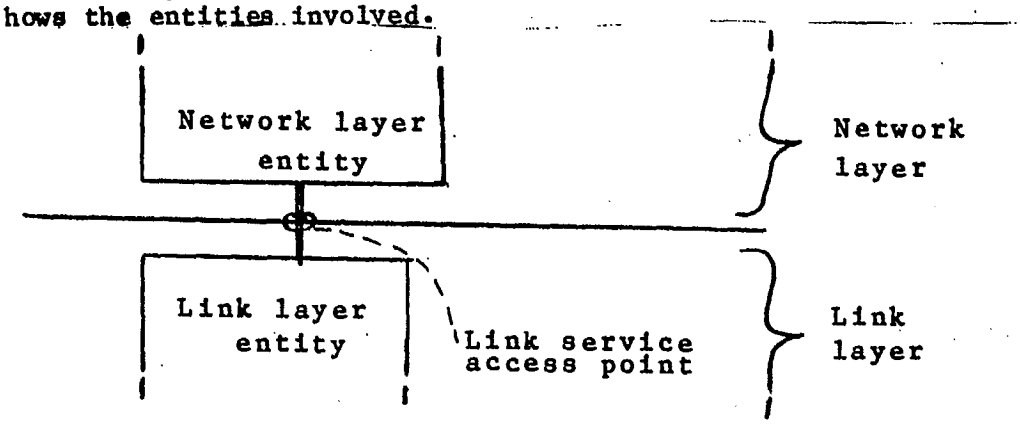
In the following the term "abstract interface" denotes the interactions between the given module and another module in its environment. For example, a service access point is an abstract interface between the service providing module and the entity using the service through this access point. It should be noted that the abstract properties of these interfaces are discussed here only to the extent that they are concerned with service and protocol specifications.

The specification of an abstract interface of a module is given by enumerating the possible interaction primitives that may occur over the

interface (including possible parameter values (determined by the module initiating the interaction), and indicating whether the module, its environment, or both may initiate the interaction), and defining the possible orders of interaction. We note that the latter is often only given informally, or not at all (it is assumed to be understood).

Annex 1 defines a possible notation which allows to specify the poossible interactions through an abstract interface without explicitly defining the modules that interact though the interface. However, it is necessary to refer to the roles that these modules play in this interaction.

As an example we consider the abstract interface through which the Link service is provided at some Link service access point. The diagram below shows the entities involved.



Using the syntax defined in Annex 1, the possible service primitives may be enumerated as follows.

```
interactions
   Link-service-access-point (link-entity, user) is
      by user:
         init-request;
         term-request;
         send (d: user-data);
         ...
      by link-entity:
         init-indication;
         term-indication;
         receive (d: user-data);
         ...
   end Link-service-access-point;
```

The interacting modules have the roles "user" and "Link-entity", respectively.

The user may initiate the following interactions:

request for the initiation of the link

indication that an initiation of the link is requested by the peer user.

This specification states that a module that interacts through a Link interface must take the role of a user, or a Link entity. Depending on its role it may initiate a certain number of interactions (indicated by the BY clause), for example a user may initiate requests for link initialization, or termination, or the sending of a block of user data.

The same notation may also be used for defining the interactions between several entities within the same layer, or between an entity and some

locally provided services, such as timers or buffer management. An example is the following definition of the timer services used by the Link entity implementing the Link protocol.

```
interactions
   timer-interface (user, server)  is
      by user:
         start (period: integer);
         stop;
      by server:
         time-out;
   end timer-interface;
```

The interactions of the timer module are "start", "stop", and "time-out".

Here again, the possible orders of interactions are not specified. However, it is understood that the time-out interaction will only be initiated by the server "period" seconds after it has received a start interaction and no subsequent stop interaction.

2.2. Protocol data units (PDU's)

A (N)-layer PDU is the unit of interaction that is exchanged between peer (N)-entities through the (N-1)-service. As suggested in the "Guidelines for the specification of Protocols for OSI" (N 381), the specification of a protocol is clarified by separating the specification of the mapping of the PDU's into (N-1)-layer service primitives (clause 6.2), and the "interaction behavior" (clause 6.1) of the protocol entity.

It is therefore suggested to use the same formalism for the enumeration of PDU's as for the specification of the other interaction primitives considered in the subsection above. Such an enumeration does not include the specification of the mapping of the PDU's into the (N-1) service primitives, which must be given separately.

The following example defines some Link PDU's using the notation of Annex 1.

```
type
   sequence-count: 0 .. 7;
PDU
   Link-PDUs (primary, secondary, balanced) is
      by balanced:
         SABM;
         ...
      by primary:
         SNRM;  SARM;
         ...
      by balanced, primary, secondary:
         I (N, R: sequence-count;
            PF-bit: boolean;
            data: user-data);
```

Three types of interacting modules are distinguished. (But only two are involved for a given link, this is not shown by this specification).

If there is an entity implementing the balanced class of procedures, it may send the following PDU's:

"Set asynchronous response mode"

Information frame with "NS", "NR", and user data fields, as well as P/F bit.

```
        DM;
         ...
end Link-PDUs;
```

2.3. The externally visible properties of a module

The behavior of a module, as seen by its environment, is characterized by the following points:
(a) enumeration of the abstract interfaces through which the module interacts with its environment. The specification of each interface includes the following information:
(a1) Enumeration of the interactions that may occur through the interface (for a possible notation see annex 1);
(a2) Specification of the permissible order of execution.
(b) global constraints on the order in which the interactions through different interfaces of the module may occur. (In the case of service specifications, these constraints define how the interactions at the two end-points of a connection relate to one another. In the case of a protocol specification, these constraints specify the order in which different PDU's may be sent, and how the interactions at the (N)-service access point of the entity relate to the sending and receiving of PDU's through the (N-1)-layer interface).

Different approaches may be useful for the specification of the global constraints. The state transition model described in section 3 seems to be a useful specification method in the case of protocol specification. Another method may be preferable in the case of service specifications.

It is useful to separate the specification of the characteristics of abstract interfaces from statements that certain modules use certain types of interfaces. For example, the characteristics of the (N)-service access points are relevant for the (N)-service specification as well as for the (N)-protocol specification. This leads to a specification method in which interface types may be defined independent of their use, and the specification of a module includes an enumeration of all the interfaces through which it interacts with its environment, with an indication of the interface type for each of these interfaces. A possible syntax for these specifications is defined in Annex 1.

A separate "connection language" may be used for specifying how the different modules and entities within the Open Systems are connected through these interfaces. However, such consideration go beyond the scope of this paper.

To demonstrate these ideas, the following lines show the general outline of Link service and protocol specifications, using the syntax defined in Annex 1. It is noted that the Link service access point definitions are used by the service as well as by the protocol specifications. The PDU and timer interface definitions are only used by the protocol specification and therefore included in that section.

Specification of the Link layer service:

```
type
   user-data = ...;
interactions
   Link-service-access-point  ... (see above)
   <local constraints>
module
   link-service (access-point-1, access-point-2 :
                    Link-service-access-point (Link-entity) )  is
      <global constraints for the Link service>
```

The module "link-service" provides the service through two access points.

The link-service module plays the "link-entity" role over both abstract interfaces.

Specification of a Link protocol (balanced class):

```
<type and abstract interface specifications of the Link layer service>
<type and abstract interface specifications of the Physical layer
service>
interactions
   timer-interface  ... (see above)
PDU
   Link-PDUs  ... (see above)
module
   Link-entity (access-point: Link-service-access-point (service);
                peer: Physical-interface (user)
                   with Link-PDUs (balanced);
                timer: timer-interface (user) ) is
      <global constraints for the Link entity>
```

Link-entity

May be specified using the state transition model discussed below.

The specification of the Link entity states that such an entity interacts through a Link interface, where it takes the role of a service provider, and also through a timer interface, where it is a user. It also interacts through a Physical interface with a peer entity by using the interactions defined as Link-PDUs. The <global constraints of the Link entity> may be specified with the state transition model described below.

3. A state transition model

The state transition model discussed in this section is a descriptive model that seems to be useful for the specification of protocols. Given the specification of the abstract interfaces of an (N)-layer entity implementing the (N)-layer protocol, as discussed in section 2, the specification of the possible orders of interactions (point (b) of section 2.3) may be given with a state transition model as described below.

In order to define the possible orders in which interactions may be initiated by the entity, the state transition model introduces the concept of the "internal state" of the entity which determines, at each given instant, the possible transitions of the entity, and therefore the possible interactions with the environment. In contrast to this model, other models (for example algebraic, and abstract data type approaches) try to provide an equivalent specification without introducing any notion of an internal structure of the specified entity.

3.1. States and transitions

The specification of the possible order of interactions of a module (or entity) is given in terms of
(a) the state space of the module which defines all (internal) states in which the module may possibly be at any given time, and
(b) the possible transitions. For each type of transition, the designer specifies the states from which a transition of that type may take place, and the "next" state of the module. A transition may also involve one or more interactions of the module with its environment (see below).

The model is non-deterministic in the sense that in a given state (at some given time), several different transitions may be possible. Only one of these transition is executed, leading to a next state which determines which transitions may be executed next. If several transitions are possible at some given time, the transition actually executed will be determined either by the module's environment (which may initiate a paricular interaction) or by the implementation of the module (which will usually determine in which order different independent abstract interfaces, connections, etc. are serviced) or by the local system manager (which may determine in which situations certain services are supported). These choices among several possible transitions are not specified by the state transition model of the protocol specification; it specifies all possible transitions.

An example is given in the figure below which shows the major states of a Link protocol entity. The state space is defined graphically. Each place of the diagram corresponds to a possible state. The transitions are also shown graphically. Each arrow corresponds to a transition which is possible when the entity is in the state which is left by the arrow, and the next state of the entity is pointed by the arrow. The names of the places have no formal meaning, but are useful for the understanding of the specification.

3.2. Interactions

As mentioned above, some transitions (for many specifications, all) are associated with interactions. Some transitions are initiated by interactions from the environment of the module, others initiate interactions with the environment, and some do both.

Assuming that the environment of the module consists of modules that are specified with the same state transition model, and assuming that the transitions are not executed infinitely fast, the problem may arise that the environment may not be ready for executing the interaction initiated by a transition of the module (for example, the environment may be in the process of executing another transition involving other interactions). There are the following two submodels which differ in the way they handle this problem:
(a) the "simple" model: the module initiating the interaction must wait

with the transition until the environment is ready to execute the interaction.
(b) the model with queues: There is a queue associated with each abstract interface through which a module receives interactions initiated by the environment. If the module is not ready for executing an interaction initiated by the environment, this interaction (i.e. all information concerning the interaction, including possible parameter values) is stored in the queue of the interface through which it is initiated, and the transition of the module corresponding to this interaction and its next state will be executed as soon as possible.

It is not clear whether the more complex model with queues is needed for the specification of OSI protocols.

3.3. An approach to specification

Since finite state diagrams, as shown in the figure below, or equivalant methods often lead to very complex specifications when a complete protocol specification is required (partial specifications, such as the one in the figure below are usually quite nice), the following approach to the specification of modules in the state transition model is proposed. This approach combines the simple concept of states and transitions as shown in the figure below with the power of a programming language.

The state space of the module is specified by a set of program variables. A possible state is characterized by the values of each of these variables. One of the variables may be called "STATE". It represents the "major state" of the module and its values may be graphically represented as shown in the figure below.

As an example, the following lines specify the state space of an entity implementing the Link protocol, using the syntax of Annex 2.

```
var
   VS, VR, VB, Unack: sequence-count;
   Count: 0 .. N2;
   State: (disconnected, information-transfer,
           FRMR-condition, DISC-requested);
```

These major states are graphically represented in the diagram at the end of the paper.

The possible transitions of the module are defined by the specification of a number of transition types. Each transition type is characterized by
(a) an enabling condition: This is a combination of a boolean expression depending on some of the variables defining the module state, and (possibly) the specification of an interaction initiated by the environment. A transition may occur in a given state if and only if the enabling condition has the value true, and the interaction in question (if it exists) is initiated by the environment.
(b) an action: This is a programming language statement which defines an action to be executed as part of the transition which may change the values of (some of) the variables, and may specify the initiation of interactions with the environment.

As an example, the following lines specify some transition types for a Link entity, using the syntax of Annex 2.

```
when peer.DISC                                  when a disconnect frame arrives and
   when State = information-transfer            the entity is in the
      do begin                                  "information-transfer" state ...
         access-point.term-indication;
         peer.UA;  send a UA frame to the peer entity
         timer.stop;  stop the timer
         State := DISC-requested end
   when State = DISC-requested                  when a disconnect frame arrives and
      do begin                                  the entity is in the
         peer.DM;                               "DISC-requested" state ...
         State := disconnected  end
   else;
 ...

when (VS not = VB) and (VS < Unack + modulus - 2)   Anytime during the
     and (State = information-transfer)             "information-transfer" state, the
   do begin                                         entity may send an information
      peer.I (VS, VR, "VS-th buffer");              frame (provided VS points to user
      VS := (VS + 1 ) mod modulus  end;             data not yet sent, and the send
 ...                                                window is open).

when peer.I (NS, NR, data)                      when the next expected information
   when State = information-transfer) and (VR = NS)  frame is received ...
      do begin
         access-point.receive (data);
         VR := (VR + 1) mod modulus;
         Unack := NR  end
   else;
 ...
```

The first transition type reads as follows: When a DISC protocol-data-unit arrives from the peer entity and the given entity is in the information transfer state, a disconnect indication is passed on to the user and a UA PDU is sent to the peer entity in response to the DISC. The timer is stopped. The next major state is "DISC-requested".

Annex 1: Syntax for specifying the interactions of a module

1. Introduction

This annex describes a possible syntax for specifying types of abstract interfaces and modules.

This syntax is largely based on the syntax and semantics of the Pascal programming language (see for example Jensen and Wirth: "Pascal: User manual and report", Springer Verlag, 1974), and uses the general approach of using type definition facilities and type checking for allowing the implementation of automatic consistency checking, which usually detects a large proportion of those errors in a specification that cannot be found by syntax checks.

2. Language elements taken from Pascal

The following language elements of the Pascal programming language are included in the specification language without any change in syntax and semantics:

Type and constant definitions including
- scalar types
- subranges
- record types
- array types

Predefined types:
- boolean
- integer
- character (defined by some ISO standard)

3. Additional language elements

The following additional language elements are defined to support the specification of service primitives, PDU's and other interaction primitives, and the definition of modules and their abstract interfaces.

3.1. Abstract interface definitions

The possible interactions at a given type of abstract interface are enumerated by a definition of the following form:

```
<interface definition> ::= INTERACTIONS <interface type id>
            ( <role list> ) IS <interactions> ;
<role list>  ::=  <role id>
             |   <role list> , <role id>
<interactions> ::=  <BY clause>
                 |  <interactions>  <BY clause>
<BY clause> ::= BY <role list> : <interaction list>
<interaction list> ::= <interaction>
                 | <interaction list>  <interaction>
```

```
<interaction> ::= <interaction id> <interaction parameters> ;
```

The declaration of <interaction parameters> is in the same form as function parameter declarations in Pascal (i.e. for each parameter its name and type).

3.2. Module definitions

The definition of a module contains the declaration of all abstract interfaces through which the module interacts. This includes the service access points through which the communication service is provided as well as the system interface for timers, etc. and the access point to the layer below, through which the PDU's are exchanged. The following syntax is proposed:

```
<module definition> ::= MODULE  <module type id>
                            (  <interfaces>  )  IS
                            <global constraints>
<interfaces>  ::=  <interface declaration>
                |  <interfaces>  ;  <interface declaration>
<interface declaration> ::= <interface id> :
                                <interface type id> ( <role id> )
```

The <role id> indicates which role the entity plays as far as the declared interface is concerned. We note that the distinction of these roles permits the checking that the invocation of interactions in the conditions and actions of transitions is consistent with the possible exchanges defined in the interface definition.

3.3. PDU definitions

The definition of PDU's is given in the same form as the definition of interactions over interfaces. The syntax is as follows.

```
<PDU definition> ::= PDU <id for PDU's> ( <role list> )
                        IS <interactions>  ;
```

The use of PDU's over a given interface, for instance over the access point to the service provided by the layer below, is declared together with the interface declaration in the module header in question. The syntax for such a combined interface and PDU declaration is the following.

```
<interface declaration>  ::=  <interface id>  :  <interface type id>
     ( <role id> )  WITH <id for PDU's> ( <role id> )
```

Annex 2: Syntax of state transition model

1. Introduction

This annex describes a syntax for the state transition model described in section 3 of the paper. It uses the same approach as Annex 1 as far as it uses many language elements from Pascal, extended with some elements which are particular to the transition model.

It is assumed that the overall structure of a module is specified in the notation defined in Annex 1. This annex is only concerned with the <global constraints>, i.e. a specification of an internal structure of the module which determines the possible order of interactions with the environment.

2. Specification of the state space

The specification of the variables defining the state space of the module follows the Pascal syntax for variable declarations. The (major) state variable (which has the identifier "STATE") is handled like any other variable of the entity.

3. Specification of the transition types

In the simplest case, each transition type is specified by a clause of the form WHEN "enabling condition" DO "action". In order to simplify the specification of different transitions with similar enabling conditions, constructions with embedded conditions, such as the following, are allowed:

```
WHEN "condition 1"
   WHEN "condition 2" DO "action 12"
   WHEN "condition 3" DO "action 13"
   ELSE;
```

which specifies the following two transition types

```
WHEN "condition 1 and condition 2" DO "action 12";
WHEN "condition 1 and condition 3" DO "action 13";
```

The ELSE keyword makes the construction non-ambiguous.

The specification of a state transition module, defining the possible orders of interactions of a module, has the general form

```
<global constraints> ::= <state space definition (see above)>
                         <definitions of functions and procedures>
                         TRANSITION  <transitions>
```

The syntax of the transition clause with embedded conditions has the following syntax.

```
<transitions>  ::=  <embedded transitions>
                 | <transitions> <embedded transitions>
<embedded transitions>  ::= <when clause>  ;
                          |  <when clause> <embedded transitions>
```

```
<when clause>  ::=  <when condition>  <when list>
                 |  <when condition>  <action>
<when list> ::= <when clause> ELSE
              |  <when clause>  <when list>
<when condition> ::=  WHEN  <boolean expression>
              |   WHEN <incoming interaction>
<action> ::= DO <action list>
```

where <action list> is a Pascal statement making reference to interactions initiated by the transition.

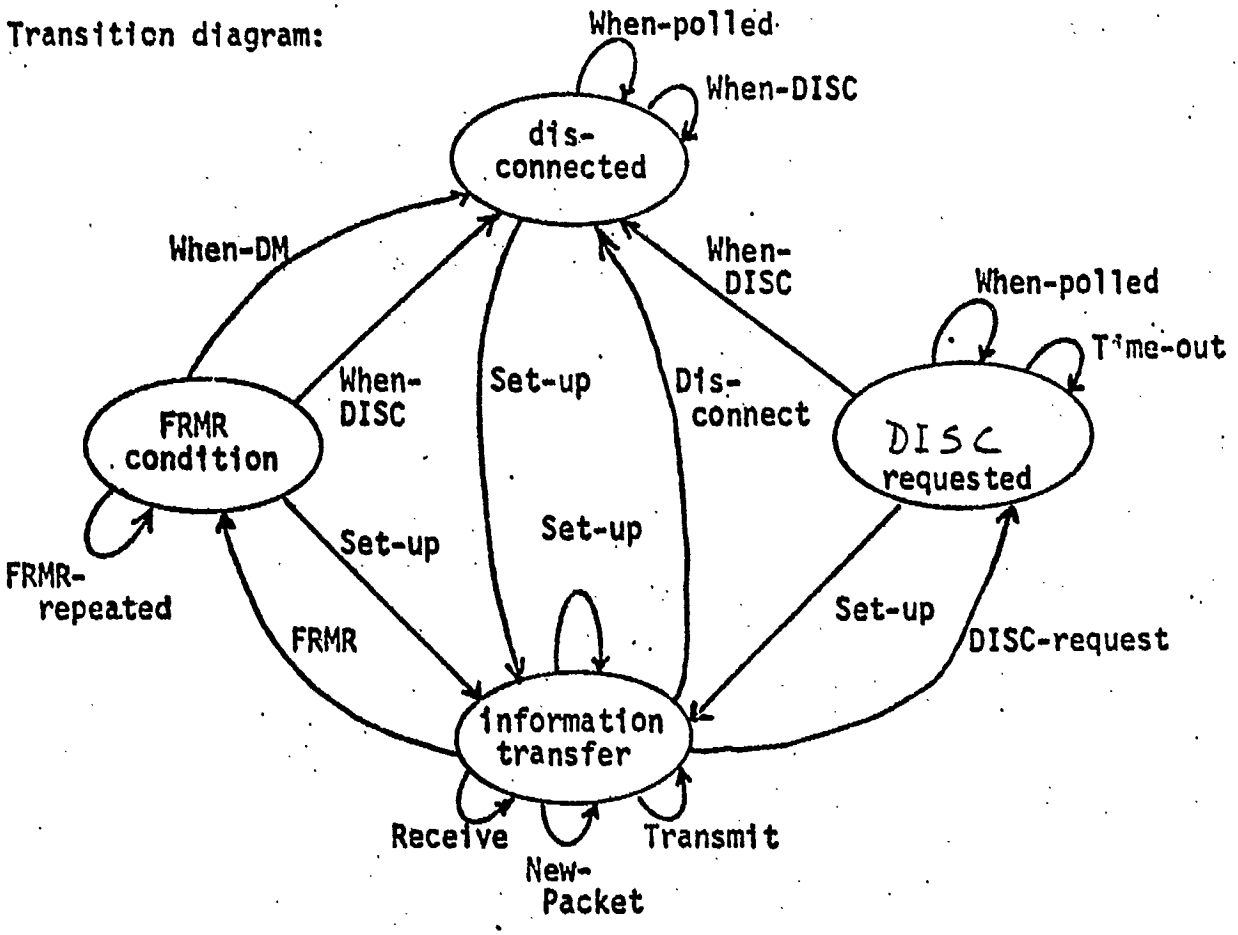
References to interactions (to incoming interactions in the enabling conditions, and to initiated interactions in the actions) are written in the "dot notation" (which is also used by other languages for the interaction with internal and/or external program modules). The notation is demonstrated by the example in section 3.3 of the paper.

The parameter identifiers used with incoming interactions are to be considered formal parameters within the scope of the transition, in the same way as the parameter identifiers of a function definition are considered formal parameters within the body of the function. This implies that all assignments to variables must be made explicitely (see for example the last assignment in the last transition of the example in section 3.3 of the paper.

Figure

Formal specification of LAP B link establishment and clearing procedure executed by the DCE (as of 1978)

Transition diagram:



Annexe 2

Syntaxe du langage LSP

DESCRIPTION DU LANGUAGE LSP

1. LES UNITES LEXICALES

A) LES IDENTIFICATEURS

IDENT = <LETTRE> #[<LETTRE> ! <CHIFFRE> ! '_']#

<LETTRE> = 'A' ! 'B' ! .....! 'Z'

<CHIFFRE> = '0' ! '1' ! .... ! '9'

B) LES ENTIERS

ENTIER = +[<CHIFFRE>]+

<SIGNE> = '+' ! '-' ! VIDE

C) LES REELS

REEL = ENTIER ['.' ENTIER ['E' <SIGNE> ENTIER ! VIDE] !
'E' <SIGNE> ENTIER]

D) LES CHAINES DE CARACTERES

CHAINE = ''' +[<CARS]+ '''

<CARS> = TOUS LES CARACTERES SAUF ''' , PLUS '''' (LA SUITE DE 2 APPOSTROPHES)

E) FIN DE FICHIER

FDF

F) LES OPERATEURS

```
':='
'..'
'+'
'-'
'*'
'/'
'('
')'
'='
','
';'
'['
']'
'<>'
'<='
'<'
'>='
'>'
'^'
'.'
```

G) LES MOTS RESERVES

```
'AND'
'ARRAY'
'BEGIN'
'BY'
'CASE'
'CONST'
'DIV'
'DOWNTO'
'DO'
'ELSE'
'END'
'EXTERN'
'FILE'
'FORWARD'
'FOR'
'FUNCTION'
'GOTO'
'IF'
'INTERACTIONS'
'IN'
'IS'
'LABEL'
'MODULE'
'MOD'
'NIL'
'NOT'
'OF'
'OR'
'PACKED'
'PDU'
'PROCEDURE'
'RECORD'
'REPEAT'
'SET'
'THEN'
'TO'
'TRANSITIONS'
'TYPE'
'UNTIL'
'VAR'
'WHEN'
'WHILE'
'WITH'
```

H) LES IDENTIFICATEURS PREDEFINIS

'ABS'
'BOOLEAN'
'CHAR'
'CHR'
'COS'
'ARCTAN'
'DISPOSE'
'EOF'
'EOLN'
'EXP'
'FALSE'
'GET'
'INPUT'
'INTEGER'
'LN'
'MAXINT'
'NEW'
'ODD'
'ORD'
'OUTPUT'
'PACK'
'PAGE'
'PRED'
'PUT'
'READ'
'READLN'
'REAL'
'RESET'
'REWRITE'
'ROUND'
'SIN'
'SQR'
'SQRT'
'SUCC'
'TEXT'
'TRUE'
'TRUNC'
'UNPACK'
'WRITE'
'WRITELN'

I) LES LIMITES RELIEES A L'ANALYSE LEXICALE

- LA LONGUEUR MAXIMALE (EN CARACTERES) D'UN IDENTIFICATEUR EST 100,000
- LE NOMBRE DE CARACTERES SIGNIFICATIFS D'UN IDENTIFICATEUR EST 30.
- UNE CHAINE DE CARACTERES COMPORTE AU PLUS 140 CARACTERES ET NE PEUT ETRE ETALEE SUR 2 LIGNES.
- LE MAXIMUM DE CARACTERES RECONNUS DANS UNE LIGNE EST DE 120.
- SUR LE "LISTING" PRODUIT PAR LE PROGRAMME LSP , LE NOMBRE DE LIGNES DANS UNE PAGE EST DE 60.

2. LES REGLES SYNTAXIQUES DU LANGUAGE LSP

```
<AXIOME>                = <PROG> FDF
*
<PROG>                  = *[ <PDEFCONST1> ! <PDEFTYPE1> ! <PDU> !
                           <INTERFACE-DEFINITION> ]* <MODULE>
*
<PDEFCONST1>            =  'CONST'  +[ <DEF-CONST> ';' ]+
*
<PDEFTYPE1>             =  'TYPE'  +[ <DEF-TYPE> ';']+
*
<INTERFACE-DEFINITION> = 'INTERACTIONS' IDENT
                           '(' <LISTE-IDENT> ')' 'IS' <INTERACTIONS>
*
<LISTE-IDENT>           = IDENT *[',' IDENT ]*
*
<INTERACTIONS>          = +[ <BY-CLAUSE> ]+
*
<BY-CLAUSE>             = 'BY' <LISTE-IDENT> ':' <INTERACTION-LIST>
*
<INTERACTION-LIST>      = +[ <INTERACTION>]+
*
<INTERACTION>           = IDENT <INTERACTION-PARAMETE> ';' ! ';'
*
<INTERACTION-PARAMETE> = '(' <LISTE-IDENT> ':' IDENT
                             *[ ';' <LISTE-IDENT> ':' IDENT ]* ')'
                         ! VIDE
*
<PDU>                   = 'PDU' IDENT '(' <LISTE-IDENT> ')' 'IS'
                          <INTERACTIONS>
*
<MODULE>                = 'MODULE' IDENT '(' <INTERFACES> ')'
                          'IS' <GLOBAL-CONSTRAINTS>
*
<INTERFACES>            = <INTERFACEDECLARATION>
                          *[';' <INTERFACEDECLARATION> ]*
*
<INTERFACEDECLARATION> = <LISTE-IDENT> ':' <TYPE-NOUVEAU>
*
<TYPE-NOUVEAU>       =  IDENT '(' <LISTE-IDENT> ')'
                        [ 'WITH' IDENT '(' <LISTE-IDENT> ')' ! VIDE ]
                      ! 'ARRAY' '[' <TYPE-SIMPLE> *[ ',' <TYPE-SIMPLE> ]* ']' .
                              'OF' <TYPE-NOUVEAU>

*
<GLOBAL-CONSTRAINTS>   = <P-DECL-VAR> <P-DECL-PROC-FONC> 'TRANSITIONS'
                          <TRANSITIONS>
*
<TRANSITIONS>          = +[ <EMBEDDED-TRANSITIONS> ]+
*
<EMBEDDED-TRANSITIONS> = +[ <WHEN-CLAUSE> ]+ ';'
*
<WHEN-CLAUSE>          = <WHEN-CONDITION> [ <WHEN-LIST> ! <ACTION> ]
*
<WHEN-LIST>            = +[ <WHEN-CLAUSE> ]+ 'ELSE' <ACTION-LIST>
*
<ACTION>               = 'DO' <ACTION-LIST>
*
<WHEN-CONDITION>   = 'WHEN' [  <EXPRESSION>  !
                               IDENT *[ <INDICAGE>  ]*  <DESIGNE-CHAMPS>
                     [ '(' <LISTE-IDENT> ')'.! VIDE ] ]
*
<ACTION-LIST>          =
                         IDENT <SUITEAAA>

                       ! 'BEGIN' <ACTION-LIST> *[ ';' <ACTION-LIST> ]* 'END'
    ! 'CASE' <EXPRESSION> 'OF' [ <W-LIST-CONST-CAS> ':' <ACTION-LIST> ! VIDE ]
                            *[ ';' <W-LIST-CONST-CAS> ':' <ACTION-LIST> ]* 'END'
                       ! 'REPEAT' <ACTION-LIST> *[ ';' <ACTION-LIST> ]* 'UNTIL'
                            <EXPRESSION>
                       ! 'WHILE' <EXPRESSION> 'DO' <ACTION-LIST>
                       ! 'FOR' IDENT ':=' <EXPRESSION>
                      [ 'TO' ! 'DOWNTO'] <EXPRESSION> 'DO' <ACTION-LIST>
                       ! 'WITH' <BL-ACCES-VAR> 'DO' <ACTION-LIST>
                       ! <ENONCE-SI-NOUVEAU>
                       ! VIDE
                       *
```

```
<SUITEAAA> =
                *[ <INDICAGE> ]* [ <DESIGNE-CHAMPS> <INTERA>
                                 ! <POINTAGE> <SUITE-AFFEC>
                                 ! ':=' <EXPRESSION>
                                 ! VIDE ]

              ! '(' <EXPRESSION> *[',' <EXPRESSION> ]* ')'
                ! <L-PARA-LIRE>
                ! <L-PARA-LIRELN>
                ! <L-PARA-ECRIRE>
                ! <L-PARA-ECRIRELN>

<INTERA>      = '(' <EXPRESSION> *[ ',' <EXPRESSION> ]* ')'
              ! <SUITE-AFFEC>
              ! VIDE

<ENONCE-SI-NOUVEAU> = 'IF' <EXPRESSION> 'THEN' <ACTION-LIST>

                      [ 'ELSE' <ACTION-LIST> ! VIDE ]

<BLOC>        = <P-DECL-ETIQU> <P-DEF-CONST> <P-DEF-TYPE> <P-DECL-VAR>
                <P-DECL-PROC-FONC> <P-ENONCE>

<P-DECL-ETIQU> =  [ 'LABEL' ENTIER *[ ',' ENTIER ]* ';' ]
                ! VIDE

<P-DEF-CONST> =  [ 'CONST' <DEF-CONST> ';' *[ <DEF-CONST> ';' ]* ]
                ! VIDE

<P-DEF-TYPE>  =  [ 'TYPE' <DEF-TYPE> ';' *[ <DEF-TYPE> ';' ]* ]
                ! VIDE

<P-DECL-VAR>  =  [ 'VAR' <DECL-VAR> ';' *[ <DECL-VAR> ';' ]* ]
                ! VIDE

<P-DECL-PROC-FONC>
              =  *[ [ <DECL-PROC> ! <DECL-FONC> ] ';' ]*

<P-ENONCE>    =  <ENONCE-COMPOSE>

<DEF-CONST>   =  IDENT '=' <CONSTANTE>

<CONSTANTE>   =  [ <SIGNE> [ ENTIER ! REEL ! IDENT ] ]
                ! CHAINE

<SIGNE>       =  '-' ! '+' ! VIDE

<DEF-TYPE>    =  IDENT '=' <DENOTE-TYPE>

<DENOTE-TYPE> =  <TYPE-SIMPLE> ! <TYPE-STRUCTURE> ! <TYPE-POINTEUR>

<TYPE-SIMPLE> =  <TYPE-ENUMERE>
              ! [['+' ! '-'] [ ENTIER ! REEL ! IDENT] ! CHAINE ]
                  '..' <CONSTANTE>
              ! [ ENTIER ! REEL ] '..' <CONSTANTE>
              ! IDENT [ '..' <CONSTANTE> ! VIDE ]

<TYPE-ENUMERE> =  '(' <LISTE-IDENT> ')'

<TYPE-STRUCTURE>
               =  [ [ 'PACKED' ! VIDE ] <TYPE-STRUCT-DET> ]

<TYPE-STRUCT-DET>
               =  <TYPE-TABLEAU> ! <TYPE-STRUCT> ! <TYPE-ENSEMBLE>
                  <TYPE-FICHIER>
```

```
<TYPE-TABLEAU> =   'ARRAY' '[' <TYPE-SIMPLE> #[ ',' <TYPE-SIMPLE> ]# ']'
                   'OF' <DENOTE-TYPE>
                $

<TYPE-STRUCT>  =     'RECORD'
                     <LISTE-CHAMPS>  'END'
                $

<LISTE-CHAMPS> = [ <SECTION-STRUCT> [ ';' <LISTE-CHAMPS> ! VIDE ]
                 ! <PART-VARIANTE> ! VIDE ]

<SECTION-STRUCT>
               =  <LISTE-IDENT> ':' <DENOTE-TYPE>
                $

<PART-VARIANTE>=     'CASE' <SELECT-VARIANT> 'OF'
                       <VARIANTE>
                     #[ ';' <VARIANTE> ]#
                $

<SELECT-VARIANT>
               = IDENT [':' IDENT ! VIDE ]
                $

<VARIANTE>     = <LIST-CONST-CAS> ':' '(' <LISTE-CHAMPS> ')' ! VIDE

                $

<LIST-CONST-CAS> = <CONSTANTE> #[ ',' <CONSTANTE> ]#
                $

<TYPE-ENSEMBLE>=   'SET' 'OF' <TYPE-SIMPLE>
                $

<TYPE-FICHIER> =   'FILE' 'OF' <DENOTE-TYPE>
                $
<TYPE-POINTEUR>=    '^' IDENT
                $

<DECL-VAR>     =   <LISTE-IDENT> ':' <DENOTE-TYPE>
                $
<ACCES-VAR>    =  IDENT #[ <INDICAGE> ! <DESIGNE-CHAMPS> ! <POINTAGE> ]#
                $

<INDICAGE>     =     '['  <EXPRESSION>  #[  ','  <EXPRESSION>  ]#  ']'
                $

<DESIGNE-CHAMPS>
               =     '.' IDENT
                $

<POINTAGE>     =      '^'
                $

<DECL-PROC>    =      'PROCEDURE'  IDENT  [ <L-PARA-FORMEL> ! VIDE ] ';'
                      [ <BLOC> ! 'EXTERN' ! 'FORWARD' ]
                $

<DECL-FONC>    =     'FUNCTION' IDENT
                       [ [ <L-PARA-FORMEL> ! VIDE ] ':' IDENT ]
                     ';' [ 'EXTERN' ! 'FORWARD' ! <BLOC> ]
                $

<L-PARA-FORMEL>=   '(' <S-PARA-FORMEL> #[ ';' <S-PARA-FORMEL> ]# ')'
                $
<S-PARA-FORMEL>=   <SPEC-PARA-VAL> !  <SPEC-VAR-PARA> !  <SPEC-PARA-PROC> !
                   <SPEC-PARA-FONC>
                $
<SPEC-PARA-VAL>=   <LISTE-IDENT> ':' IDENT
                $
<SPEC-VAR-PARA>=    'VAR' <LISTE-IDENT> ':'  IDENT
                $

<SPEC-PARA-PROC>
               =     'PROCEDURE' IDENT [ <L-PARA-FORMEL> ! VIDE ]
                $

<SPEC-PARA-FONC>
               =     'FUNCTION' IDENT
                      [ <L-PARA-FORMEL> ! VIDE  ] ':' IDENT
                $
```

```
<FACTEUR>       = REEL ! CHAINE ! 'NIL' ! ENTIER
                  ! <CONSTR-ENS> ! '(' <EXPRESSION> ')' ! 'NOT' <FACTEUR>
                ! IDENT [ #[ <INDICAGE> ! <DESIGNE-CHAMPS> ! <POINTAGE> ]#
                        ! '(' <EXPRESSION> #[',' <EXPRESSION> ]# ')' ]
                  #
<CONSTR-ENS>    =   '[' [ [ <DESIGNE-MEMBRE> #[ ',' <DESIGNE-MEMBRE> ]# ] !
                    VIDE ] ']'
                  #
<DESIGNE-MEMBRE>=     <EXPRESSION> [ [ '..' <EXPRESSION> ] ! VIDE ]
                  #
<TERME>         =   <FACTEUR> #[ <OPER-MULT> <FACTEUR> ]#
                  #
<EXPR-SIMPLE>   =     <SIGNE> <TERME> #[ <OPER-ADD> <TERME> ]#
                  #
<EXPRESSION>    =   <EXPR-SIMPLE> [ [ <OPER-REL> <EXPR-SIMPLE> ] !  VIDE ]
                  #
<OPER-MULT>     =    '*' !  '/' !  'DIV' !  'MOD' !  'AND'
                  #
<OPER-ADD>      =     '+'  !  '-'  !  'OR'
                  #
<OPER-REL>      =   '=' !  '<>' !  '<' !  '>' !  '<=' !  '>=' !  'IN'


<ENONCE>        =   [ [ ENTIER ':' ] !  VIDE ]
                    [ <ENONCE-SIMPLE> !  <ENONCE-STRUCT> ]
                  #
<ENONCE-SIMPLE>=    VIDE !  <AFFEC-APPEL> !  <ENONCE-ALLERA>
                  #
<AFFEC-APPEL>   = IDENT [ <SUITE-AFFEC>
                        ! '(' <EXPRESSION> #[ ',' <EXPRESSION> ]# ')'
                        ! <L-PARA-LIRE>
                        ! <L-PARA-LIRELN>
                        ! <L-PARA-ECRIRE>
                        ! <L-PARA-ECRIRELN>
                        ! VIDE ]
                  #
<SUITE-AFFEC>   = #[ <SS-INDICAGE> ! <DESIGNE-CHAMPS> ! <POINTAGE> ]#
                    ':=' <EXPRESSION>
                  #
<ENONCE-ALLERA>=    'GOTO' ENTIER
                  #
<ENONCE-STRUCT>=    <ENONCE-COMPOSE> ! <ENONCE-COND> ! <ENONCE-BOUCLE>
                    <ENONCE-AVEC>
                  #
<ENONCE-COMPOSE>=     'BEGIN' <SEQ-ENONCES> 'END'
                  #
<SEQ-ENONCES>   =   <ENONCE> #[ ';' <ENONCE> ]#
                  #
<ENONCE-COND>   =   <ENONCE-SI> !  <ENONCE-CAS>
                  #
<ENONCE-SI>     =     'IF' <EXPRESSION> 'THEN' <ENONCE>
                        [ <PART-SINON> ! VIDE ]
                  #
<PART-SINON>    =   'ELSE' <ENONCE>
                  #
<ENONCE-CAS>    =     'CASE' <EXPRESSION> 'OF' <L-ELEMENT-CAS>
                      #[ ';' <L-ELEMENT-CAS> ]# 'END'
                  #
```

```
<L-ELEMENT-CAS>=    [ <W-LIST-CONST-CAS> ':' <ENONCE> ]
                  !  VIDE
                  *
<ENONCE-BOUCLE>=   <ENONCE-REPETER> !  <ENONCE-TANTQUE> !  <ENONCE-POUR>
                  *
<ENONCE-REPETER>
                  =    'REPEAT' <SEQ-ENONCES> 'UNTIL' <EXPRESSION>
                  *
<ENONCE-TANTQUE>
                  =    'WHILE' <EXPRESSION> 'DO' <ENONCE>
                  *
<ENONCE-POUR>  =   'FOR' IDENT ':=' <EXPRESSION>
                  [ 'TO' !  'DOWNTO' ] <EXPRESSION> 'DO' <ENONCE>
                  *


<ENONCE-AVEC>  =   'WITH' <L-ACCES-VAR> 'DO' <ENONCE>
                  *
<L-ACCES-VAR>  =    <ACCES-VAR> #[ ',' <ACCES-VAR> ]#
                  *
<L-PARA-LIRE>  =    '(' <ACCES-VAR> #[ ',' <ACCES-VAR> ]# ')'
                  *
<L-PARA-LIRELN>=    <L-PARA-LIRE> ! VIDE
                  *
<L-PARA-ECRIRE>=    '(' <PARA-ECRIRE> #[ ',' <PARA-ECRIRE> ]# ')'
                  *     .
<PARA-ECRIRE>  =    <EXPRESSION>
                    [   [ ':' <EXPRESSION> [ [ ':' <EXPRESSION> ] ! VIDE ] ]
                    !   VIDE   ]
                  *
<L-PARA-ECRIRELN>
                  =    <L-PARA-ECRIRE> ! VIDE

                  *

<S-INDICAGE>   = '[' <EXPRESSION> #[ ',' <EXPRESSION> ]# ']'
                  *
<SL-ACCES-VAR>  =  <S-ACCES-VAR> #[ ',' <ACCES-VAR> ]#
                  *
<S-ACCES-VAR>   = IDENT #[ <SS-INDICAGE> ! <DESIGNE-CHAMPS>
                                         ! <POINTAGE> ]#
                  *
<SS-INDICAGE>   = '[' <EXPRESSION> #[ ',' <EXPRESSION> ]# ']'
                  *
<SW-INDICAGE>   = '[' <EXPRESSION> #[ ',' <EXPRESSION> ]# ']'
                  *
<W-LIST-CONST>  = <W-CONSTANTE> #[ ',' <W-CONSTANTE> ]#
                  *
<W-CONSTANTE>   =  [ <SIGNE> [ ENTIER ! REEL ! IDENT ] ] ! CHAINE
                  *
```

LISTE DES MESSAGES D'ERREURS

1: IDENTIFICATEUR ATTENDU.
2: ENTIER ATTENDU.
3: REEL ATTENDU.
4: CHAINE DE CARACTERES ATTENDU.
5: FIN DE FICHIER ATTENDU.
6: 'CONST' ATTENDU.
7: ';' ATTENDU.
8: 'TYPE' ATTENDU.
9: 'INTERACTIONS' ATTENDU.
10: '(' ATTENDU.
11: ')' ATTENDU.
12: 'IS' ATTENDU.
13: ',' ATTENDU.
14: 'BY' ATTENDU.
15: ':' ATTENDU.
16: 'PDU' ATTENDU.
17: 'MODULE' ATTENDU.
18: 'WITH' ATTENDU.
19: 'ARRAY' ATTENDU.
20: '[' ATTENDU.
21: ']' ATTENDU.
22: 'OF' ATTENDU.
23: 'TRANSITIONS' ATTENDU.
24: 'ELSE' ATTENDU.
25: 'DO' ATTENDU.
26: 'WHEN' ATTENDU.
27: 'BEGIN' ATTENDU.
28: 'END' ATTENDU.
29: 'CASE' ATTENDU.
30: 'REPEAT' ATTENDU.
31: 'UNTIL' ATTENDU.
32: 'WHILE' ATTENDU.
33: 'FOR' ATTENDU.
34: ':=' ATTENDU.
35: 'TO' ATTENDU.
36: 'DOWNTO' ATTENDU.
37: 'IF' ATTENDU.
38: 'THEN' ATTENDU.
39: 'LABEL' ATTENDU.
40: 'VAR' ATTENDU.
41: '=' ATTENDU.
42: '-' ATTENDU.
43: '+' ATTENDU.
44: '.' ATTENDU.
45: 'PACKED' ATTENDU.
46: 'RECORD' ATTENDU.
47: 'SET' ATTENDU.
48: 'FILE' ATTENDU.
49: '^' ATTENDU.
50: '.' ATTENDU.
51: 'PROCEDURE' ATTENDU.
52: 'EXTERN' ATTENDU.
53: 'FORWARD' ATTENDU.
54: 'FUNCTION' ATTENDU.
55: 'NIL' ATTENDU.
56: 'NOT' ATTENDU.
57: '*' ATTENDU.
58: '/' ATTENDU.
59: 'DIV' ATTENDU.
60: 'MOD' ATTENDU.
61: 'AND' ATTENDU.
62: 'OR' ATTENDU.
63: '<>' ATTENDU.
64: '<' ATTENDU.
65: '>' ATTENDU.
66: '<=' ATTENDU.
67: '>=' ATTENDU.
68: 'IN' ATTENDU.
69: 'GOTO' ATTENDU.
303: CARACTERE ILLEGAL.
304: ENTIER TROP GROS.
305: PARTIE FRACTIONNAIRE TROP GRANDE.
307: CHAINE DEBORDANT LA LIGNE INTERDITE.
308: TROP DE CHAINES DE CARACTERES DANS LE PROGRAMME.
309: CHAINE TROP LONGUE, UNE CHAINE PEUT CONTENIR AU PLUS 140 CARACTERES.
320: PARASITE.
321: SUBSTITUTION.
322: DEBUT DU MECANISME DE RECUPERATION D'ERREUR.
323: FIN DU MECANISME DE RECUPERATION D'ERREUR.
400: UN DES ROLES DE LA DERNIERE LISTE NE FAIT PAS PARTIE DE LA LISTE DE LA DEFINITION INITIALE.
401: REPETITION D'IDENTIFICATEURS DANS UNE LISTE INTERDITE.
402: TYPE NON DECLARE.
403: UN IDENTIFICATEUR DE VARIABLE NE PEUT ETRE LE MEME QUE CELUI D'UNE INTERFACE DECLAREE DANS LE MODULE.
404: PDU NON DECLARE.
405: INTERACTION NON DECLARE.
406: UN DES ROLES DE LA LISTE EST ABSENT DANS LA DEFINITION DE L'INTERACTION (OU DU PDU).
407: NOMBRE D'INDICE DE TABLEAU EN DESACCORD AVEC LA DECLARATION.
408: ROLE INTERDIT D'APRES LA DEFINITION DU MODULE.
409: IDENTIFICATEUR(S) DE PARAMETRE(S) EN DESACCORD AVEC LA DEFINITION DE L'ACTION.
410: NOMBRE DE PARAMETRES EN DESACCORD AVEC LA DEFINITION.