2-4218

UNIVERSITÉ DE MONTRÉAL

Formal Description Techniques for Protocols

Final Report for DOC contract No. OSU82-00218

by Gregor v. Bochmann

Département d'informatique et

de recherche opérationnelle

Université de Montréal

March 1983

da

24218

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

Checked 10

Formal Description Techniques for Protocols

Final Report for DOC contract No. OSU82-00218

by Gregor v. Bochmann

Département d'informatique et

de recherche opérationnelle

Université de Montréal

March 1983

24218

This report was prepared for the Department of Communications Canada under contract No. OSU82-00218. The report presents the views of the author. Publication of this report does not constitute DOC approval of the report's findings or conclusions. This report is available outside the Department by special arrangement.

TABLE OF CONTENT

Annex 7: Proposals for contents for sections 3, 4 and 7 of the Draft Recommendation, Canadian contributions to CCITT Q39/VII (Nov. 1982).

Annex 8: "Comparison of FDT proposals from ISO (Subgroup B) and CCITT" (July 1982)

Annex 9: "Towards a common FDT for ISO and CCITT" (Canadian contribution to WGI meeting in Paris (Febr. 1983)

Annex 10: "Proposal to produce an FDT standard" (Canadian contribution to WGI meeting in Paris (Febr. 1983)

Annex 11: Meeting reports

Annex 12: "A parser for an FDT language" by G. Gerber and G.v. Bochmann (March 1983)

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 contributing 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. Contributions 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

2.1. Standardization activities

Within the framework of this contract, the author was a Canadian delegate at a meeting of the CCITT Rapporteurs Group on Question VII/39 in Geneva. The author was also delegate at a ISO TC97/SC16/WG1 meeting in Paris, as well as in a meeting of its ad hoc group on FDT in Catania and a meeting of Subgroups B in Montreal. The author is editor for the working papers of both subgroups A and B (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 Geneva.

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 single FDT which is accepted by both ISO and CCITT.

2.2. Translator for formal specifications

One of the activities undertaken under the present contract was an improvement of a translator program which was developed in 1981-82 at the Univerity of Montreal. The purpose of such a program is to automatically translate a formal description of a protocol given in the extended state transition model into a set of Pascal declarations and procedures which could be incorporated into some run-time support system, thus leading to a semi-automatic implementation approach for protocol entities from the formal protocol specification. The improvements performed under the present contract concern the following issues:

(a) Adjusting the accepted syntax to follow more closely the present language syntax developed by Subgroup B of the ISO ad hoc group on FDT.

(b) Improving the handling of syntax errors found in formal specifications. The above mentioned translator program has only very rudimentary error recovery facilities.

(c) Developing a method by which several modules, each specified by an extended state transition machine, could be translated into Pascal procedures such that they can be combined into a single Pascal program executing all these modules in a semi-parallel fashion. This would allow the integration of several protocol layers into a single program implementing the protocol entities of all those layers.

Work on points (a) and (b) have led to the parser for formal descriptions described in Annex 12. This parser has good error recovery properties, and accepts a syntax which is close to the present proposal of subgroup B of the ISO ad hoc group on FDT. (The syntax is still undergoing slight changes within the standardization groups; it is not possible to follow these changes immediately).

Work on point (c) has led to a model for translating formal specifications which has been tried out with the example of the alternating bit protocol. It seems quite general in nature, and we plan to implement a translation scheme based on this new approach.

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 research 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 improve the protocol implementation tools which could partly automate the production of a protocol implementation from the formal specification of the protocol.

c) To improve the tools that could be used to test that a protocol implementation conforms with the protocol specification. Such tools could be useful for the certification of communication software and systems.

d) To develop a protocol simulation tools 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

4.1. Contributions discussed by topic

The following paragraphs describe the different contributions which were submitted to the international standardization meetings, mentioned in section 2.1. Most of these contributions were first submitted to the responsible Canadian standardization committee (CSA comittee on OSI, or NSG VII for CCITT), and some of the contributions were submitted as "Canadian" papers. Others were submitted to the international meetings as "expert papers".

The different contributions are discussed in the following by topic. A complete list of contributions is given in section 4.2. below. Meeting reports concerning international meetings attended for the work under this contract are included in Annex 11.

4.1.1. Transport protocol specifications

A Transport protocol specification for the classes 0 and 2 [TP 2], which was prepared for a separate DOC research contract, was presented at the Catania and Geneva meetings. A revised version [TP 3] was presented at the Paris meeting. In contrast to previously presented specifications [TP 1], this protocol specification uses a later version of the FDT syntax, and defines the protocol in terms of several modules, one module per connection and a common "mapping" module.

4.1.2. Transport service specifications

A Transport service specification describing explicitely multiple simultaneous connections between an arbitrary number of service access points was presented at the Enschede meeting (see Annex 3). In order to demonstrate the separation of "local" and "global" service properties, as proposed in [Boch 83], a new service specification was elaborated (Annex 4) and presented at the Subgroup B meeting in Montreal. The separation of "global" and "local" properties is similar to the approach presented in

[Logrippo 82], however, our specification remains within the "extended state transition model".

4.1.3. Refinement of the extended state transition model

A number of contributions have been presented with the aim of better defining the Subgroup B working document in the "extended state transition model" FDT. For more detail, we refer the reader to the Annexes 5 and 6, and the contributions (3) of section 4.2.1, (3) of section 4.2.2, (4) of section 4.2.3, and (3) of section 4.2.5.

4.1.4. Harmonizing the FDT developments in ISO TC97/SC16 and CCITT SG VII and XI

A number of contributions were prepared in order to harmonize the development of FDT's for OSI applications in ISO TC97/SC16 and the CCITT Study groups VII and XI. (SG XI has been involved for some time in the development of the SDL language). We refer the interested reader to the Annexes 7 through 9.

4.1.5. Editing the Subgroup A and B working documents

The author has been the editor for the working documents of the ISO Subgroups A and B during the last year. During this time, most of the working document of Subgroup A has been completely rewritten, and large parts of the working document of Subgroup B have been added and revised. The preparation of the subsequent versions of these documents was a non-negligeable task during the last year. The present versions of these documents are included as Annexes 1 and 2.

4.2. List of contributions presented at international meetings

4.2.1. ISO meeting in Enschede (ad hoc group on FDT, April 1982)

(1) "Formal description of the transport service" (TWENTE-2, see Annex 3)

(2) "Examples of Transport protocol specifications" (TWENTE-3, see [TP 1])

(3) "A simple state transition foundation for the "Common semantic model for CCITT and ISO" (TWENTE-6; parts of this contribution had an impact on Section 5 of the present working document of Subgroup B, see Annex 2).

4.2.2. Subgroup B meeting in Montreal (July 1982)

(1) "Comparison of FDT proposals from ISO (Subgroup B) and CCITT" (UM-1, see Annex 8)

(2) "Example description of the Transport service" (UM-2, see Annex 4)

(3) "Preliminary draft of Section 5 (Formal Semantics) of Subgroup B working document" (UM-7)

4.2.3. ISO meeting in Catania (ad hoc group on FDT, November 1982)

(1) "Comparison of FDT proposals ISO-CCITT" (CAT-11 (Canada), similar to Annex 8)

(2) "Some enhancements to the syntax of Subgroup B FDT" (CAT-12 (Canada), see Annex 5)

(3) "Example of a Transport protocol specification" (CAT-13), see [TP 2])

(4) "Section 2.y for working document of Subgroup A" (CAT-17)

4.2.4. CCITT Rapporteurs meeting of FDT (Geneva, December 1982)

(1) "Specification of Transport service using finite-state transducers and abstract data types" (FDT 77, prepared by L. Logrippo)

(2) "Example of a Transport protocol specification" (FDT 78, see [TP 2])

(3) "Constructive and executable specifications of protocols and services" (FDT 79, prepared by L. Logrippo)

(4) "Examples for the use of non-deterministic extended finite state machines" (FDT 86, see Annex of Annex 6)

(5) "Proposal for contents for Section 3 (semantic model) of the Draft Recommendation" (FDT 87, see Annex 7a)

(6) "Proposal for contents for Section 4 (Language for describing system structure) of the Draft Recommendation" (FDT 88, see Annex 7b)

(7) "Proposal for the contents for Section 7 (language for describing synamic behavior based on Pascal) of the Draft Recommendation" (FDT 89, see Annex 7c)

4.2.5. <u>ISO meeting in Paris (WG1 and had hoc group on FDT, February 1983)</u>

(1) "Example of a Transport protocol specification (revised)", see [TP 3]

(2) "Towards a common FDT for ISO and CCITT" (Source: Canada; see Annex 9)

(3) "Comments on avoiding collisions and the zero-queue option" (13 pages)

(4) "Semantics of spontaneous transitions" (see Annex 6)

(5) "Proposal to produce and FDT standard" (Source: Canada; see Annex 10)

REFERENCES

[TP 1] G.v. Bochmann, "Examples of Transport protocol specifications", contribution to ISO TC97/SC16/WG1 ad hoc group on FDT, Twente-3, 1982. Originally prepared under contract for COST 11 bis (CEE).

[TP 2] G.v. Bochmann, "Example of a Transport protocol specification", prepared for CERBO Informatique Inc. under contract for Department of Communications Canada, Oct. 1982.

[TP 3] G.v. Bochmann, "Example of a Transport protocol specification (revised)", Annex 1, Final Report, DOC research contract OST82-0092, March 1983.

[Boch 83] G.v. Bochmann and M. Raynal, "Structured specification of communicating systems", IEEE Trans. Computers, Febr. 1983.

[Logr 82] L. Logrippo, "Specification of Transport service using finite-state transducers and abstract data types", CCITT Q39/VII, FDT-77, Geneva, Dec. 1982.

<u>ANNEX 1</u>

ISO/TC 97/SC 16 N 1346
Date: November 1982
Project:

ISO
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

ISO/TC 97/SC 16
OPEN SYSTEMS INTERCONNECTION
SECRETARIAT: USA(ANSI)

Source: ISO TC97/SC16/WG1, Subgroup A of adhoc group on FDT

Title: Concepts for describing the OSI architecture (Working document, Catania, November 1982)

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) Define certain architectural concepts which are used by the FDT's (see sections 2 and 3),

(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 (see section 4), and

(c) provide a more precise model for the Guidelines (N 380 and N381) (see section 5)

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.

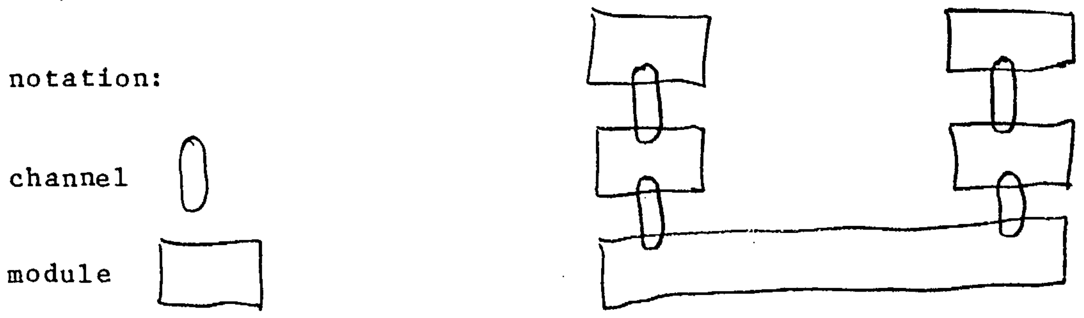
2. Modules, channels, and interaction points

2.1. The concepts

The architecture of a system is defined by a set of interacting 'modules' and the structure by which they are interconnected.

Modules share channels with each other and with modules in the system's environment. The channels embody the interactions between the modules, and between the modules in the system and those in the system's environment. The modules embody the actions exclusively allocated to modules.

The configuration of channels and modules represent the system's structure. An example is shown in the figure below.

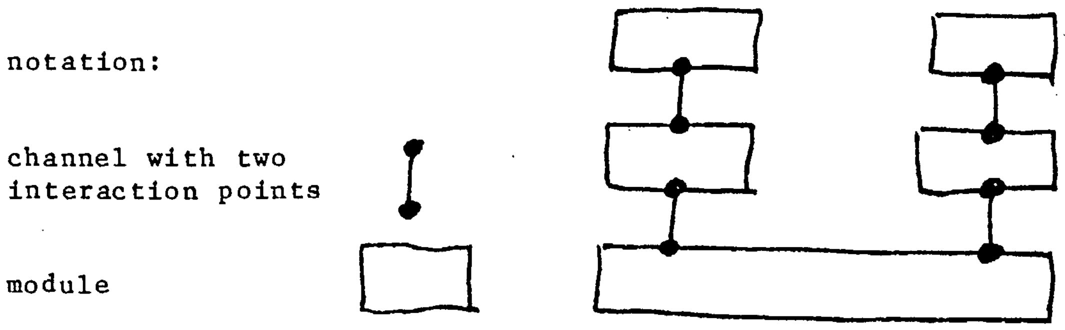


Modules bear different responsibilities in the performance of interactions. For example, if in an interaction a value is passed, then one module is responsible for providing that value, and the other module is responsible for accepting the value.

To allow for modelling of these different responsibilities, we introduce the concept of 'interaction point'.

An 'interaction point' is a view of a channel as seen from one of the modules that is connected to the channel.

Using an alternative graphical notation, the above example can be represented as follows:



The concepts of 'channel' and 'interaction point' are useful for the description of the OSI architecture. They are related to the notion of 'abstract interface' in the following sense: the interactions of a module with other modules or with the environment of the system occur through channels between the modules. In a real system, such a channel is realized by an '(real) interface'. In this section we are not concerned with the specification of real module interfaces, but only with the abstract properties that any such interface for a given module-to-module interconnection must satisfy. These properties are called the 'abstract interface' between the two modules.

The concepts serve for:

a) the partitioning of the interactions of a given module into separate groups concerning different modules forming the module's environment. A module has contact with its environment only through a well-defined set of 'channels'.

b) the specification of the interconnections between the different modules withing a system (or the sub-modules within a module).
A channel connecting two modules could be specified by naming an interaction point of one module and an interaction point of the other module with which the former is to be connected.

For example, typical channels of a layer entity executing the layer protocol are:

a) the access point(s) to the layer above through which the service is provided,

b) the access point(s) to the layer below through which the underlying service is accessed,

c) an (abstract) interface to the local system management module, and possibly a local channel through which local services such as buffer management, time-outs, etc. can be obtained.

2.2. The specification of a channel

The purpose of a channel type definition is to be used in the specification of a module (see section 2.3.), where each interaction point of a module is characterized by the type of channel which it represents.

The specification of a channel type includes:

a) an enumeration of the possible interaction primitives that may be invoked through a channel of that type.

b) the names of two 'roles' which distinguish the two sides of the channel, and hence the two connected modules (e.g. 'service provider' and 'service user').

c) the properties of the interaction primitives, which may include such properties as:
--parameters including their data type,
--an indication by which 'role' parameters are established, including the responsability for provision and accepting parameter values,
--time dependencies in relation to these different roles, e.g.atomic interactions, or interactions extended in time and interruptible,
--etc.

d) possibly certain rules about the order in which the interaction primitives may be executed over a given channel of that type

2.3 The specification of a module

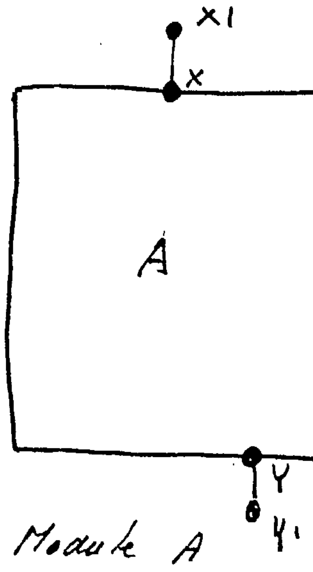
The purpose of a module specification is to define the behavior of the module as <u>observable at the interaction points</u> to which it is connected. Therefore a module specification cannot be given without a definition of the interaction points through which the module interacts with its environment.

The specification of a module may be given in each of the following forms:

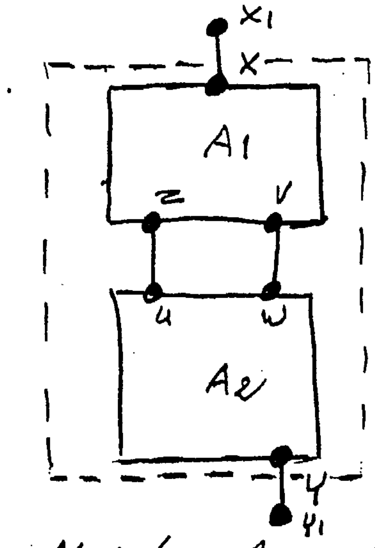
(a) by a fixed substructure definition (see section 3), where each submodule in the substructure can be defined either according to (a) or to (b).
(b) by defining the behavior of the module using one of the specification languages developed by Subgroups B or C.

3. <u>Substructure definitions</u>

A specification of a module may be given in the form of a substructure definition, as shown in the figure below. If the behavior of each of the submodules is defined, such a substructure defines the behavior of the module.



Module A

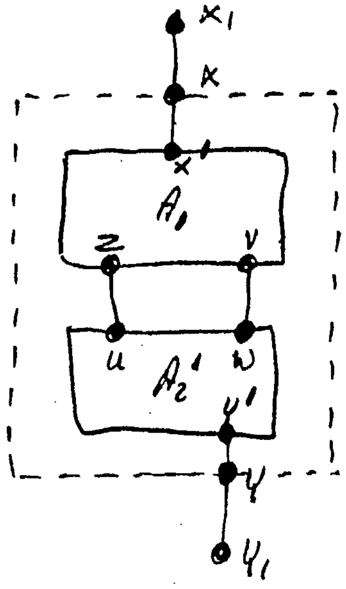


Modules A1 and A2 representing the substructure of A.

In the example above, the module A interacts with other modules in the system through the channels X-X1 and Y-Y1. The substructure of module A consists of two submodules A1 and A2. The connections Z-U and V-W are called <u>internal channels</u> and connect interaction points by which the modules A1 and A2 interact. The notation of the example also means that the interactions of A at X and Y are realized by the interactions of A1 at X, and A2 at Y, respectively.

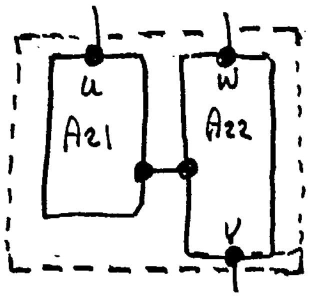
The above structuring has assumed that the interaction points X and Y of A and X and Y of A1 and A2 remained unaltered, i.e. only the functionality of A was represented by two submodules A1 and A2 connected by internal channels.

One could also consider a substructuring for the interaction points X and Y, and represent this by an alternative way of picturing

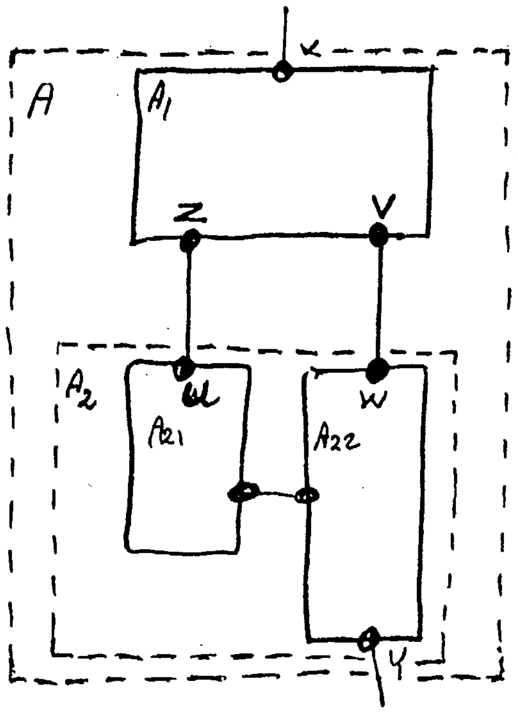


We leave this possibility for further study.

It is possible to further subdivide the structure of a module. For example à possible substructure of module A2 would be as follows:



Sometimes several steps of refinement are shown in a single diagram. For example, the figure below shows the two steps of refinement for module A given above:



A syntax for describing substructure definitions is for further study.

4. The nature of interactions

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-SP, see section 4.1) and (N-1) - SPr's* with other local modules (respectively, the (N+1)-entity and the (N-1)-entity).

Two time instants** are important for the execution of an interaction between two modules:

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

2) the moment when the interaction ends.

Each interaction carries explicit information (parameters, sometimes refered to as associated information).

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;

---

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

** It is noted that certain models distinguish an additional time instant: the moment that the interaction is initiated ("called") by one (i.e. the first) of the modules. This may be useful, for example, in situations where it is important to know which module is waiting (for example, performance considerations).

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 5.4).

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 one of the interacting modules, or both.

(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). In these models 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.

An interaction is always seen the same by the two interacting modules.

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 5.4).

5. 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.

5.1 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.

---

* 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.

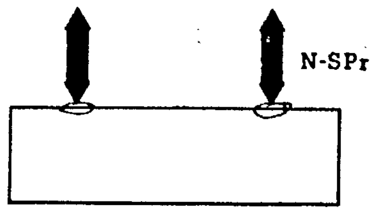


Figure 2

5.2 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.

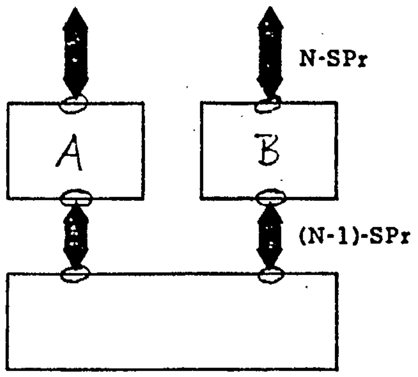


Figure 3

5.3 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.

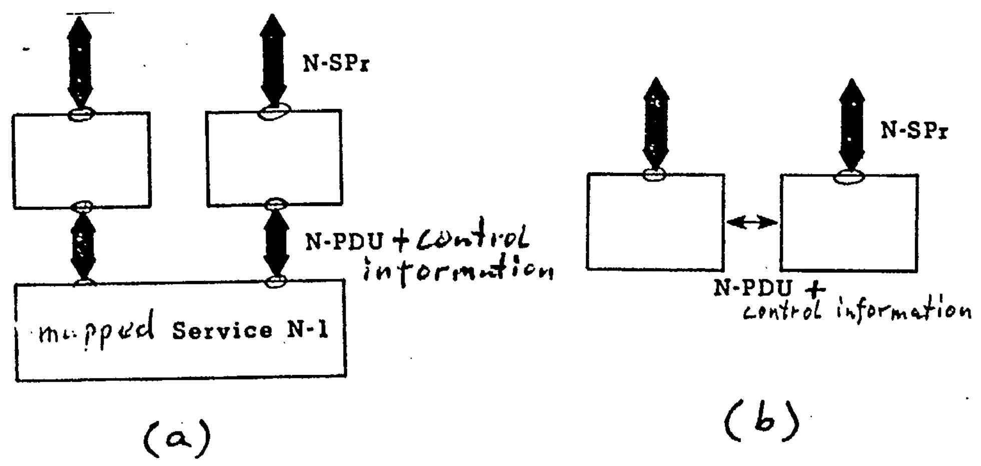


Figure 4

5.4 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).

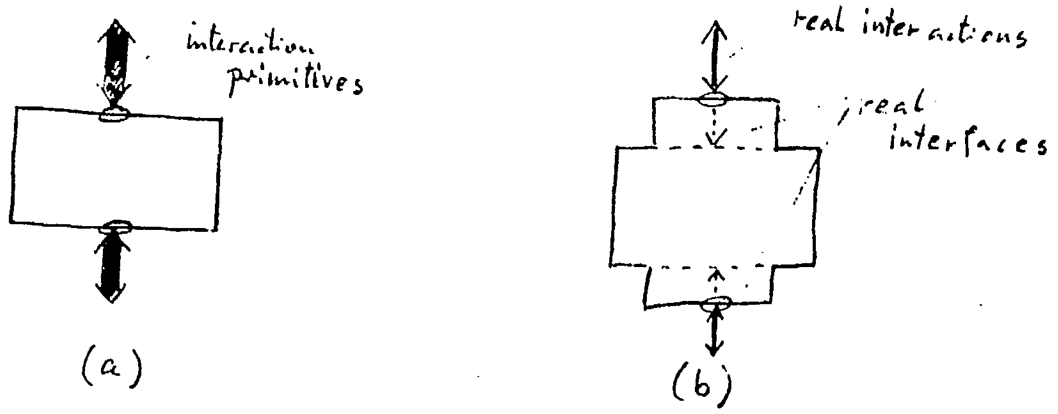


Figure 5

6. 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.

A module can be decomposed into submodules according to several criteria, e.g.:

- to encapsulate well defined functional blocks in submodules which are activated sequentially to accomplish the more complex service of the module. The objective could be to introduce more abstract service primitives describing the service provided by the next lower layer, etc.

- to separate data flow and control flow.

- to consider inherent concurrency of the module. This can be accomplished by assigning one submodule to each connection since the connections are the sources of different unsynchronized sequences of events or transitions. The service of the module is in this case implemented in a distributed manner.

1. Possible identification of submodules

The concept of an abstract protocole specification (see section 5.3) suggests a substructure containing separate submodules for mapping and abstract protocol.

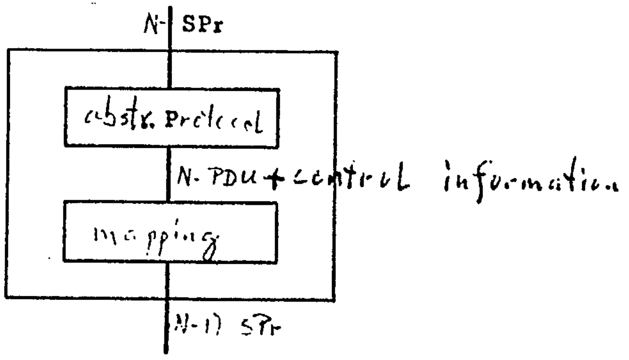
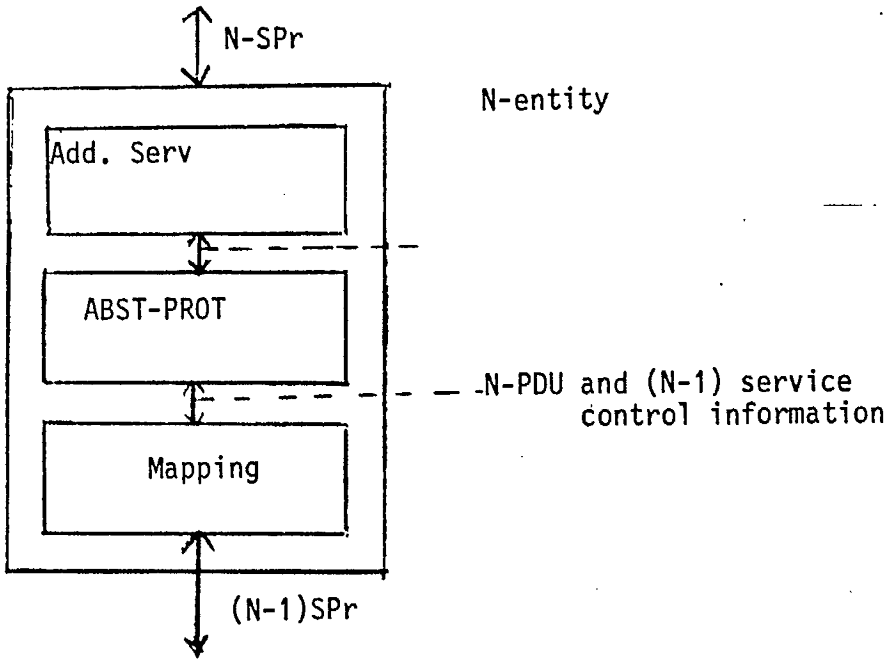


Figure 6

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.

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.

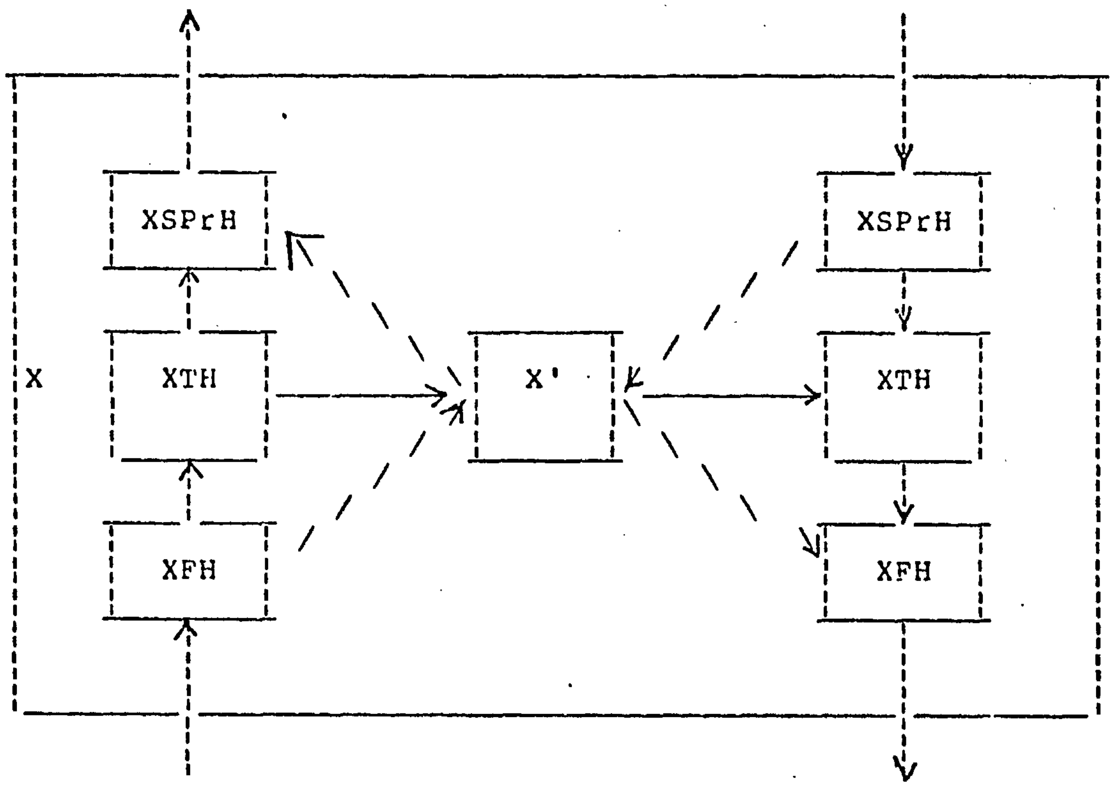


Figure 7

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. Decomposition of an entity according to its inherent concurrency

One possible criterion for decomposition is to what degree parallelism in the entity is to be modelled. Ultimately each event or state transition could be represented as an independent module. This will, however, not contribute to a clear and comprehensible structure. The decomposition should rather reflect the structure of independent event-sequences relevant to the intended level of description. Such independent event-sequence are initiated by input events at the connection end points of the module to be decomposed.

In fig. 9, three such input handling modules are introduced: the service request handler (SRH), the service indication handler (SIH) and the time-out handler (TOH).

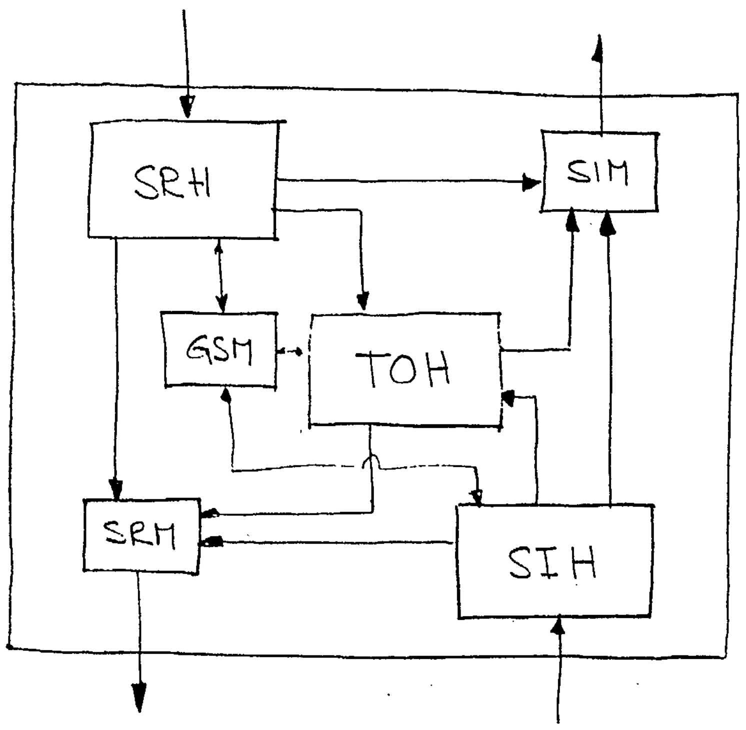


Figure 9

The submodules must cooperate to perform the function of the entity i.e. the function is distributed. In fig. 9 the communication between the input handlers is accomplished by state variables encapsulated in a monitor module providing mutual exclusion (GSM). Since more than one submodule can produce output events on the same channels, these are also encapsulated in monitor modules (SRM, SIM).

The distribution of a protocol function can be illustrated by time-out handling. A submodule (SRH or SIH), having submitted for transmission a message on which a response is expected, sets the time-out interval. The submodule receiving the response resets the time. The time-out handler performs the protocol actions prescribed when a time-out occurs.

A N N E X 2

ISO/TC 97/SC 16 N 1347
Date: November 1982
Project:

ISO
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

ISO/TC 97/SC 16
OPEN SYSTEMS INTERCONNECTION
SECRETARIAT: USA(ANSI)

Source: ISO TC 97/SC16/WG1, Subgroup B of ad hoc group on FDT

Title: A FDT based on an extended state transition model (Working Document, November 1982)

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. Model

2.1. Modules, channels, and interaction points

2.1.1. The concepts
(see Subgroup A document, section 2.1)

2.1.2. The specification of a channel
See Subgroup A document, section 2.2)

2.1.3. The specification of a module
(see Subgroup A document, section 2.3)

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 (including 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", 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 <u>true</u>, and the interaction in question (if it exists) is initiated by the environment. A transition without input is called a spontaneous transition.

(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.

2.4. The specification of module substructures

(see Subgroup A document, section 3)

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 ... (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 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

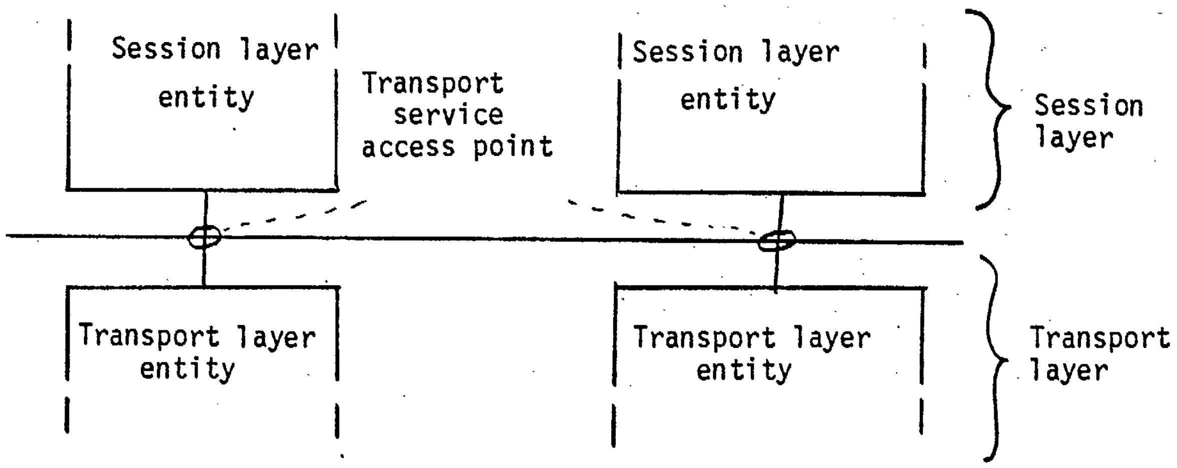
3.2 The specification of interactions

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.

The specification of a channel 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 an indication about which module may initiate the interaction).

In order to distinguish between the two modules that use the channel for their interactions the concept of a "role" is introduced. For each type of channel two roles are defined. These two roles are 'played' by the respective module instances that are connected to an instance of a channel. 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.

```
channel

  TS_access_point(TS_user,TS_provider);

     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_CONNECT_resp(TCEP_identifier    : TCEP_identifier_type;
                    QOTS_request        : quality_of_TS_type;
```

```
                                    options              : option_type;
                                    TS_accept_data    : TS_accept_data_type);

          T_DISCONNECT_req

        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 "user", or a "Transport 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 connection establishment or disconnection, or the sending of a fragment of user data.

The same concept of a channel 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.

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

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

The FDT provides for a separation of 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. A channel type may be defined independent of its 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. The same enhancement could be used to define the substructure of a module in term of submodule and their interconnection. These considerations are for further study.

3.4. Specification of a module as an extended state transition machine

3.4.1. State variables

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_CONNECT_resp,data_trans-
fer);
  local_reference : TP_reference_tupe;
  remote_reference : TP_reference_type;
  TPDU_size :max_TPDU_size_type;
  QOTS_estimate : quality_of_TS_type;
```

3.4.2. State transitions

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 statement of the <block>) including the generation of output.

An input interaction to the module is either considered immediatly by the state machine or first put into the (conceptualy infinite) input queue of the module (depending on the queuing option used for the interaction point over which the interaction reaches the module); if it is put into the input queue it is considered by the machine when the input comes to the head of the queue. When an input interaction is considered by the state machine, one of the transitions enabled for the given input parameters and the present module state is executed. A lower priority transition can only be executed when no higher priority transition is enabled. If no transition is enabled (depending on the option used) either the input is ignored, or an undefined situation occurs which may be considered a "user error" or an indication of a design error in the specifications of the interacting modules.

A transition which has no input (no WHEN clause) is called spontaneous. It can be executed, independently of input, whenever the enabling condition is satisfied.

A spontaneous transition may include a delay clause with two parameters, d_1 and d_2. The transition may not occur until the enabeling condition has remained true continuously for d_1 time. It must be considered immediately if the enabeling condition remains true continuously for d_2 time. If the delay clause is absent, a delay of $d_1 = 0$, d_2 = infinity is assumed. (This is written "delay (0,*)".) It means that the transition may occur at any time the enabeling condition is true, possibly never.
A delay(0,0) has the semantic meaning of the immediate spontaneous transition of the basic semantic module (see section 5).

<u>Notes</u>

1. An input transition which has been received through a channel having the "not queued" option can not contain any output statement in its operation. Outputs are allowed only for spontaneous transitions or input transitions received through the queue. Relaxing this rule require further study.

2. Executing an output statement may imply some short delay*. However the module is not observable from outside during this time, and no other event can be presented to the module during this time. This short delay is that delay necessary for the output to be presented to the receiving module (directly or to its queue).

*Note: However, this delay is not present in the first output of a spontaneous transition.

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

```
trans
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);
      out
    end;
from data_transfer to same
  when TSAP.T_DATA_req
    provided ... (* flow control from user ready *)
      begin
        out_buffer.append(user_data);
        out
      end;
  when out_buffer.fragment_ready(TPDU_size)
  provided ... (* Network layer flow control ready *)
    begin
      N.DT(out_buffer.get_fragment(TPDU_size));
      out
    end;

trans
provided no_tc_uses_ne and ne_locally_open
  begin
    out N.DISCONNECT_req    (*close any unused network connection *)
  end;

trans
from data_transfer to same
  provided credit_to_be_sent  delay (0,evaluate_delay_max_agreed)
   begin
       N.ACK(credit,tpdu_nr)    (* send credit if any *)
       out
   end;
```

3.4.3. Enbedding of transitions

The syntax for transitions permits the different clauses (FROM, WHEN or DELAY, PROVIDED, PRIORITY, and TO) to be written in arbitrary order, followed by the <block> which includes at least BEGIN END. The order has no influence on the meaning of the construct.

The syntax also permits the embedding of the different clauses. This embedding structure is simply a shorthand notation with the following rules: The "scope" of a clause is defined to be the specification text corresponding to "<transition>+" in the syntactic rule of the clause (see section 4.1.3). The meaning of the clause extends over its entire scope. Each BEGIN END statement of a block within the specification text identifies a transition. All clauses in the scope of which a given transition falls apply to this transition. For example

```
trans
  when AP.I
    from A             provided  E  to  B
      begin X end;
      provided F to C
      begin Y end;
    from B to C
      begin Z end;
trans
  from C to D begin U end;
```

is a short hand notation for

```
trans
  when AD.I  from A provided E to B begin X end;
trans
  when AD.I  from A provided F to C begin Y end;
trans
  when AD.I  from B to C begin Z end;
trans
  from C to D begin U end;
```

It is noted that the following scope rules must be followed:

(a) The parameters of the input interaction (declared in the corresponding channel type definition) become accessible within the scope of the WHEN clause.

(b) As in Pascal, the WITH clause makes the fields of a record variable directly accessible within the scope of the clause.

(c) The ANY clause introduces a "variable" identifier with an arbitrary value within the range defined by the type identifier. The meaning is that the embedded transitions are defined for each of the possible values of this variable.

3.4.4. Continuous output functions

While interactions represent "events" and are generated during state transitions, continuous output functions provide steady output from one module through a channel to another module. The "receiving" module may use the value of such a function (provided by its neighbour module) within a PROVIDED clause, that is, it may influence which transitions are enabled.

The name and type of output functions are declared in a channel definition. The value provided by the function is determined by the function body which is defined within the module body which plays the role of the outputting module.

3.5 Predefined language elements

Some predefined language elements are provided. These include types, procedures, functions and modules. The predefined identifiers may be redefined by the user of the FDT. In this case, the user's element is the one used.

3.5.1. Predefined types

The following have been identified as candidates for predefined types:
user_data_type and connection_endpoint_id_type.
Their precise definitions require further study.

3.5.2. Predefined procedures and functions

The procedure 'error', taking no arguments and having an implementation dependent action is a predefined procedure. Routines for manipulating user data, coding and decoding PDUs, and manipulating addresses are currently being considered.

3.5.3 Predefined modules

The module 'timer' is currently being considered.

3.6 An Example

The following is an example of the Subgroup B method of protocol description in use. It is a specification of an alternating bit protocol. Although the example shows many of the basic constructs of the language, simplicity dictates that some of the features of the language cannot be shown here.

The first section of the example contains declarations of constants and types, in a style familiar to a reader of Pascal. One obvious addition is the notation "..." which is used to indicate that the specifier is leaving the interpretation to the implementor. Often this is accompanied by a comment to guide the implementor in his choice. A notation was needed to indicate the properties of the connections between modules. These are called "channels". Each channel may have players, the role of which are indicated in parentheses after the channel name. The various interface events of a channel are indicated after the role list. For each role, the events that the player may initiate are listed along with their parameters. These parameters are available within a transition that is initiated by the event.

The module header line includes names for the channels it uses, as well as an indication of the role the module plays on that channel. Thus, the Alternating_Bit module is the Provider of the U channel, which is a U_access_point channel. The inputs from this channel and from the N channel are placed in a common queue. The U_access_point channel supports three kinds of interface events. Two of these may be initiated by the User (and are thus inputs for the Alternating_Bit module), and one of these is initiated by the Provider (and is thus an output of the module).

Following the module header, variables local to the module are declared. Although not used in the example, if there were any labels or types local to the module, they would preceed the variables, as they do in Pascal. Then the major states and major state sets are declared. State sets are a convenient way to specify that a transition may take place from any of several major states.

Next is an initialization section. In this, the major state and the variables are given initial values. This determines the initial state of the module.

Then functions and procedures are declared. In addition to the standard Pascal definitions, either the keyword "primitive" or the notation "..." is used to indicate that the details are left to the implementor. Often, the choice of a data structure and the details of the primitives must be coordinated choices. In the example, the choice of the structure of "buffer_type" will determine the details of the procedures "store", "remove", and "retrieve". Furthermore, the actual details of these structures and the routines that manipulate them are not particularly relevant to the action of the protocol.

Output from the module over a channel is specified by the keyword OUT. The actual channel and event are indicated by naming the channel, followed by a ".", followed by the output interaction with its parameters.

Finally, the transitions are listed. The clauses corresponding to the keywords "from", "to", "when", etc. are all optional, and may appear in any order, and may be nested (though they are not in this example). They describe the major state before the transition, after the transition, and the required input, respectively. The "provided" clause describes an enabeling predicate that must be satisfied for the transition to take place. An optional "priority" may be assigned to any transition.

One the input is listed, the parameters associated with the input may be accessed in much the same manner as the fields of a record within the scope of a "with" statement. This enhances the readability ot the resultant specification.

Notice the transition from state ESTAB back to itself when a S.TIMER_response input occurs. This corresponds to the case in which the retransmit timer expires for data that have been acknowledged. In this case, clearly noting need be done. Another approach to dealing with this situation would be to "cancel" the retransmit timer when the acknowledgment is received by generating an S.TIMER_request with a Time value of zero.

```
const
    retran_time = 10;
    empty       = 0;
    null        = 0;

type
    data_type   = ...;
    seq_type     = ...; (* for alternating bit, use 0..1 *)
    id_type      = (DATA,ACK);
    timer_type   = (retransmit);
    ndata_type   =
        record
            id: id_type;
            data: data_type;
            seq: seq_type;
        end;
    msg_type      =
        record
            msgdata: data type;
        end;
    buffer_type  = ...;
    int type       = ...; (* would usually be "integer' *)


(* channel definitions *)

channel U_access_point(User, Provider);
    by User:
        SEND_request(UData: data_type);
        RECEIVE_request;
    by Provider:
        RECEIVE_response(UData: data_type);

channel S_access_point(User, Provider);
    by User:
        Timer_request(Name: timer_type; Time:int_type);
    by Provider:
        Timer_response(Name: timer_type);

channel N_access_point(User, Provider);
    by User:
        Data_request(NData: ndata_type);
    by Provider:
        Data_response(NData: ndata_type);
module Alternating_Bit(U: U_access_point(Provider) queued;
                        N: N_access_point(User)    queued;
                        S: S_access_point(User)    not queued);

var
    send_seq:    seq_type;
    recv_seq:    seq_type;
    send_buffer: buffer_type;
    recv_buffer: buffer_type;
    p,q:           msg_type;

state:
    (ACK_WAIT, ESTAB);
    EITHER = [ACK_WAIT, ESTAB];

initialize
    begin
```

```
            state to ESTAB;
            send_seq := 0;
            recv_seq := 0;
            send_buffer := empty;
            recv_buffer := empty;
        end;

    procedure send_data(msg: msg_type);
    var s: ndata_type;
    begin
        s.id   := DATA;
        s.data := msg.msgdata;
        s.seq  := msg.msgseq;
        N.DATA_request(s)
        out
    end;

    procedure send_ack(msg: msg_type);
    var a:ndata_type;
    begin
        a.id   := ACK;
        a.data := msg.msgdata;
        a.seq  := null;
        N.DATA_request(s)
        out
    end;

    procedure deliver_data(msg: msg_type);
    begin
        U.RECEIVE_response(msg.msgdata)
        out
    end;

    procedure store(var buf: buffer_type; msg: msg_type);
    primitive;

    procedure remove(var buf: buffer_type; msg: msg_type);
    primitive;

    function retrieve(buf: buffer_type): msg_type;
    primitive;

    procedure inc_send_seq;
    begin
        send_seq := (send_seq + 1) mod2
    end;

    procedure inc_recv_seq;
    begin
        recv_seq := (recv_seq + 1) mod 2
    end;

    (* transitions *)
```

```
trans
from ESTAB to ACK_WAIT when U.SEND_request
begin
    p.msgdata := UData;
    p.msgsdq  := send_seq;
    store(send_buffer,p);
    send_data(p);
    S.TIMER_request(retransmit, retran_time)
    out
end;

form ACK_WAIT to ACK_WAIT when S.TIMER_response
                              provided Name = retransmit
begin
    p := retrieve(send_buffer);
    send_data(p);
    S.TIMER_request(retransmit,retran_time)
    out
end;

from ACK_WAIT to ESTAB when N.DATA_response
                         provided Ack_OK
begin
    remove(send_buffer, NData.nsg);
    incr_send_seq;
end;

from ESTAB to ESTAB when S.TIMER_response
                    provided Name = retransmit
begin
    (* do nothing *)
end;

from EITHER to SAME when N.DATA_response
                    profided NData.id = DATA
begin
    q.msgdata := NData.data;
    q.msgseq  := NData.seq;
    send_ack(q);
    if NData.seq = recv_seq then
        begin
            store(recv_buffer, q);
            incr_recv_seq
        end
end;

from EITHER to SAME when U.RECEIVE_request,
                    provided not buffer_empty(recv_buffer)
begin
    q := retrieve(recv_buffer);
    deliver_data(q);
    remove(recv_buffer, q.msgseq)
end;
```

```
module Timer(S:S_access_point(Provider)
                              not queued)

var
  timervalue:array[timer_type] of integer;
    index:timer_type;
initialize
  begin
    for index := retransmit to retransmit do
            (* index must run through all possible timer type *)
        timervalue [index] := 0
  end;

trans
    when S.Timer_request
       begin
           timervalue[Name] := 0; (* This cancels the previous timer
                     of this name;  see next transition and semantic
                     of delay. *)
           timervalue[Name] := Time (* This sets the new timer. *)
       end;

trans
    any timer_index:timer_type do
       provided timervalue [timer_index] > 0
            delay (timervalue [timer_index],timervalue[timer_index])

            begin
                timervalue[timer_index] := 0;
                out S.Timer_response[timer_index]
            end;
```

Note: It has not yet been decided whether to use the keyword OUT or OUTPUT. In the examples, OUT is currently used.

Note: An improved version of this example is under study.

3.7. User guidelines

to be provided

4. Syntax overview

This section defines the syntax of the specification language. Large parts of the language are taken from the Pascal programming language.

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.

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 channels 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 Syntactic extension

4.1.0 Overall structure of a specification

The overall structure of a protocol or service specification (in the following simply called "system") is as follows:

```
<system>    ::=   SYSTEM <system id>;
               <channel type definition>*
               <module type definition>*
               <system structure>
<system id> ::= <identifier>
```

The <system structure> is for further study.

4.1.1 Channels and interaction primitives

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

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

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

```
<channel>    ::=  CHANNEL <channel type id>
        (  <role list>  )   <exchanges> ;
<role list>  =  <role id>
             |   <role list> , <role id>
<exchanges> ::= <BY clause>
                | <exchanges> <BY clause>
<BY clause> ::= BY <role list> : <exchange list>
<exchange list> ::= <exchange>
                | <exchange list> <exchange>
<exchange> = <interaction id> <interaction parameters> ;
                 | <function heading>**
```

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: Identifiers may include both upper and lower case letters as well as the underscore character ("_"), which is considered to be a letter, and numerals.

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 heading>  <internal definition>
<module heading>::=MODULE <module type id>
                                   (  <interaction points>  ) ;
<interaction points>  ::=  <interaction point declaration>
                     | <interaction points> ; <interaction point
                                                 declaration>
<interaction point declaration> ::= <interaction point id> :
                     <interaction point type>
                     ( <role id> ) <queue discipline>
<queue discipline> = NOT QUEUED|QUEUED
<interaction point type> = <channel type id>
                         | ARRAY [ <index type> ]
```

```
                                              OF <channel type id>
                                              (* Note 9 *)
<internal definition> ::= <module body>
                        | <substructure definition>
```

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 module

```
<module body> ::= <label definitions>**
                  <constant definitions>**
                  <type definitions>**
                  <variable declarations>**
                  <major state declaration>
                  <state set definition>*
                  <proc func or init etc.>*
                  <embedded transitions>+


<embedded transitions> = TRANS <transition>+
<major state declaration> ::= STATE : <enumeration type> ;
<state set definition>  ::=   <state set id> = <set definition>** ;
                                                        (*Note 4*)
<proc func or init etc.> = <procedure definition>** (* Note 2 *)
                         | <function definition>    (* Note 2 and 3 *)
                         | <continuous output definition>
                         | <initialization> (* it is suggested that
                                               the initialization be
                                               placed at the beginning *)
<continuous output definition> = FUNCTION <interaction point ref>.
                                 <function name> ; <block>
                                 (* the parameters of the function
                                  are already declared in the channel
                                  definition *)
<interaction point ref> ::= <interaction point id>
                          | <interaction point id> [<index variable>]
<index variable> : :=<identifier>
<function name> ::= <identifier>
<initialization>    ::=   INITIALISE BEGIN
                           STATE TO <major state value>
                           <additional init>;
<additional init> ::= END

                                  |; <statement sequence>** END
```

```
<transition> =
   |  ANY <identifier> : <type identifier** DO <transition>+ (*Note 5a*)
   |  WITH <variable>** DO <transition>+ (*Note 5b*)
   |  WHEN <interaction point ref> .  <intraction  id>  <transition>+  (*Note
5c*)
   |  DELAY(<delay value>,<delay value>)<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>** ;


<delay value> ::= <expression> | *        (* Note 10 *)
<priority indication> ::= <identifier>**  (*constant of some
                                           enumeration type*)
                        | <integer>**
<major present state> ::= <major state value list>
                        | <state set id>
<major state value list> = <major state value>
                        | <major state value list><major state value>
<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 ref> . <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 "...". Other possible uses of ... are for further study.
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 nor DELAY 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.

Note 9 : The usual multi-dimensional array notation, e.g. ARRAY [index1,index2], is also allowed.

Note 10: The delay value must be either an integer valued expression or '*', which represents infinity.

4.1.4. Other extensions

(a) Informal specification elements, which define system properties that are part of the specification (not merely comments), are written as text enclosed in "(/" and "/)" and may be placed wherever comments or ... may be placed.

(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

Functions are permitted to return arbitrary values.

4.3. Elements of Pascal not used

To date, we have not found the following features of Pascal to be necessary: pointers, and files (and go to and labels).

5. Formal semantics

5.1. General approach

The semantics of the specification language is defined by a translation of the language into a basic semantic model described in section 5.2. The translation is explained in section

The semantics of the specification language is defined by a translation of the language into a basic semantic model described in section 5.2. The translation is explained in section 5.4. The semantics of the basic semantic model (BSM) is defined in section 5.3 in terms of an abstraction which considers the system as a single module, the semantics of which is directly defined by some interpretation rules.

More precisely, the semantic meaning of a specification written in the Subgroup B FDT (in the following simply called FDT) may be obtained by successive translations which lead from the FDT representation through the BSM to a definition of the behavior of the specified system. In general, these translations may be the following:

```
  FDT ---> abstract repres. ---> BSM ---> BSM ---> semantics
      (1)                    (2)      (3)      (4)
```

The translation from the FDT into the BSM may be considered in two steps (1) and (2). However, it is defined in section 5.4 by a single translation process. The definition of the semantics of the BSM (translation (4)) is given for the case of a module without interactions (see section 5.3.1). On the other hand, the BSM obtained through the translations (1) and (2) contains at least as many modules as the original specification in the FDT. Therefore the translation step (3) is introduced which constructs a BSM specification with a single module equivalent to the multi-module specification obtained from the translation steps (1) and (2).

The following sections contain text preceeded by "Note:" or "Explanation:". This text should be considered as comments which is added for ease of understanding. However, it is not part of the definitions.

5.2. The basic semantic model

5.2.1. Architectural definitions

1) A specified system consists of a number of modules $M_i (i=1,2,...)$ and a number of interaction points $IP_j (j=1,2,...)$.

   Note: Only module and interaction point <u>instances</u> are considered here. However the FDT specification language includes facilities for defining module and interaction point <u>types</u>.

2) For each interaction point IP_j, a set of two roles is defined (Note: for instance, "upper" and "lower"). We write r for one role and $\bar{r}$ for the other role.

3) a) For each interaction point IP_j, there are two sets of possible interactions:
$I_j^{(r)}$ and $I_j^{(\bar{r})}$

Note: The interactions which are output by the connected module with role r and $\bar{r}$, respectively.

b) For each interaction point IP_j, there are two sets $CE_j^{(r)}$ and $CE_j^{(\bar{r})}$.

Note: These sets are the range of functions defined by the connected modules; the function of the module with role r determines a value in $CE_j^{(r)}$ depending on the state of the module. These values may be considered continuous signals from the defining module to its neighbours.

4) The specification of a module M_i refers to a number $(k=1,2,...K_i)$ of interactions points. These references are written $j(i,k)$. It also indicates, for each reference, the role $r(i,k)$ the module assumes at the referred interaction point.

5) An interconnection structure is given for the specified system which defines, for each interaction point reference $j(i,k)$, the referred interaction point $IP_{j(i,k)}$. The following interconnection properties are assumed:

a) If modules M_i and $M_{i'}$ refer to the same interaction point, i.e.

$IP_{j(i,k)} = IP_{j(i',k')}$, then they assume opposite roles, i.e. $r(i,k) \neq r(i',k')$, and they are different modules, i.e. $i \neq i'$. Note: This implies that at most two modules are connected to a given interaction point. To achieve loop-back, one must introduce a specific loop-back module.

b) Each interaction point is referred to by at least one module. If an interaction point $IP_{j(i,k)}$ is only referred to by M_i then it is called an external interaction point. Note: An external interaction point describes some interaction of module M_i with the environment of the specified system.

5.2.2. The behavior of a module

Note: The behavior of a module is defined by a state-transition model.

1) At any given instant of time, a module M_i is either in a state or doing a transition.

2) a) The set of possible states of module M_i is written S_i; a particular state is written s_i.

b) For each referred interaction point $IP_{j(i,k)}$, there is a function $C_{i,k} : S_i \dashrightarrow CE_{j(i,k)}^{(r(i,k))}$.

Explanation: The notation "f : D --> R" means that f is a (possibly partial) functions which defines a value in R for values of the argument in D.

Note: These functions define a continuous output of the module which may be read by the neighbouring modules.

3) The set of (names for) possible transitions of module M_i consists of transitions of the following forms:

a) Internal transitions, written $t_i^{(int)}$, with

$$P : S_i \times (\underset{k}{+} CE_{j(i,k)}^{(\bar{r}(i,k))}) \dashrightarrow \text{Bool}$$

$$F : S_i \times (\underset{k}{+} CE_{j(i,k)}^{(\bar{r}(i,k))}) \dashrightarrow S_i$$

Explanation: P is the so-called enabling predicate. For the transition to be executed, it is necessary that the predicate is true. F is the so-called transition function. It defines the next module state for the case that the transition is executed. Both P and F, depend on the present module state (S_i) and possibly on the values of the continuous output functions of the neighbouring modules.

b) Output transitions initiating an output interaction at the interaction point $IP_{j(i,k)}$, written $t_{ik}^{(out)}$, with

$$P : S_i \times CE_{j(i,k)}^{(\bar{r}(i,k))} \dashrightarrow \text{Bool}$$

$$F : S_i \times CE_{j(i,k)}^{(\bar{r}(i,k))} \dashrightarrow S_i$$

$$O : S_i \times CE_{j(i,k)}^{(\bar{r}(i,k))} \dashrightarrow I_{j(i,k)}^{(r(i,k))}$$

Note: O is the output function which determines the output produced. The output produced depends on the present module state (S_i) and possibly on the values of the continuous output functions of the neighbouring modules.

c) Input transitions initiated by an input interaction at the interaction point $IP_{j(i,k)}$, written $t_{ik}^{(in)}$, with

$$P : S_i \times I_{j(i,k)}^{(\bar{r}(i,k))} \dashrightarrow \text{Bool}$$

$$F : S_i \times I_{j(i,k)}^{(\bar{r}(i,k))} \dashrightarrow S_i$$

d) Explanation: The enabling predicate P and the next state function F depend on the present module state (S_i) and the value of the input received.

5.3. The semantics of the basic semantic model

The semantics of a system specified in the BSM is defined in two steps. The first step, described in section 5.3.2, consists of a translation of the system specification into an equivalent simple module. As second step, the semantics of a simple module is directly defined by some interpretation rules, as explained in section 5.3.1.

5.3.1. Semantics of a simple module

In the case that the system consists of a single module M_o which has no interaction point references, all its transitions are internal. In this case, the semantics of the module is defined by the following interpretation rules:

1) If the system is in state s, the next transition t to be executed is any transition for which $P_t(s)$ is <u>true</u>. Which one of these will be executed is not determined by the model. The model assumes that, if such transition exist, one such transition will be executed <u>eventually</u> (in the sense of temporal logic, liveness).

2) If the transition t is executed starting in state s then the next state will be $F_t(s)$. The execution of a transition takes finite (possibly arbitrarily short) time.

5.3.2. Abstraction from module boundaries

The following rules define a translation of a system specification consisting of modules M_i and interaction points IP_j, as explained in section 5.2, into the specification of a single module M_o with interaction point references $j(o,k')$. The semantics of the system specification is, <u>by definition</u>, equal to the semantics of the module M_o.

1) The state space S_o of the module M_o is the Cartesian product of the state spaces of the individual modules $M_i (i=1,2,...)$:
$$S_o = \times S_i$$
A particular state s_o is written as $s_o = <s_1, s_2, ..., s_i, ...>$.

2) The interaction point references $j(o,k')$ of M_o refer to the external interaction points $IP_{j(i,k)}$ of the system specification.

3) Each transition t of M_o is formed by one of the following rules:

a) Rendezvous interaction between modules M_i and $M_{i'}$ at the interaction point $IP_{j(i,k)} = IP_{j(i'k')}$: this leads to an internal transition of module M_o of the following form:

$$t=<t_{ik}^{(out)}, t_{i'k'}^{(in)}>$$

$$P_t(<s_1,...,s_i,...,s_{i'},...>) =$$

$$P_{t_{ik}}^{(out)}(s_i, C_{i'k'}^{(r(i',k'))}(s_{i'}))$$

$$\text{and } P_{t_{i'k'}}^{(in)}(s_{i'}, C_{i,k}^{(r(i,k))}(s_i),$$

$$O_{t_{ik}}^{(out)}(s_i, C_{i'k'}^{(r(i',k'))}(s')))$$

$$F_t(<s_1,...,s_i,...,s_{i'},...>)=<s_1,...,\hat{s}_i,...,\hat{s}_{i'},...>$$

$$\text{where } \hat{s}_i = F_{t_{ik}}^{(out)}(s_i, C_{i'k'}^{(r(i',k'))}(s_{i'}))$$

$$\text{and } \hat{s}_{i'} = F_{t_{i'k'}}^{(in)}(s_{i'}, C_{i,k}^{(r(i,k))}(s_i),$$

$$O_{t_{ik}}^{(out)}(s_i, C_{i'k'}^{(r(i',k'))}(s_{i'})))$$

Explanation: P_t defines when a rendezvous interaction is possible. It is possible when the enabling predicates of both involved transitions (output from M_i and input to $M_{i'}$) are enabled. The enabling predicate of the output transition depends, in general, on the (internal) state of (internal) state of $M_{i'}$ and the value of the interaction ouput produced by M_i. The next state of the module M_o is identical to its state before the rendezvous, except that the state components corresponding to the modules M_i and $M_{i'}$ are changed as defined by the respective transition functions. The change for $M_{i'}$ also depends on the output received.

b) The internal transitions of a module M_i and the input and output transitions of a module M_i at an external interaction point $IP_{j(i,k)}$ are included as transitions of M_o. Their P and F definitions remain inchanged, reading and updating only the state components of the module M_i and its input and output. Note: this is also the input and output of the single module M_o, because of translation rule (2).

Note: The translation described here leads to a module M_o which is an abstraction of the original system of interacting modules M_i. Such abstractions may be applied recursively, corresponding to multi-level module substructures, as described in Subgroup A working document, section 3.

Note: Different transitions t and t′ may proceed in parallel if they do not influence one another <u>directly</u>. A sufficient condition is the disjointness of the set of state components on which P and F depend and which are affected by F, for the two transitions respectively.

5.3.3. Semantics of a system and its environment

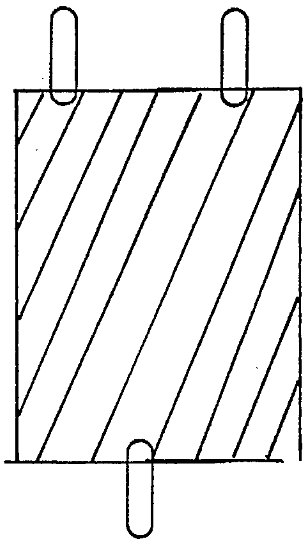
In the case that the original system specification includes external interaction points (and therefore the module M_o would include references to these interaction points), the semantics of the system specification may be defined for the case of interaction with an arbitrary system environment. Such an arbitrary system environment may be modeled by additional environment modules: One environment module $EM_{<i,k>}$ for each interaction point reference j(i,k) refering to an external interaction point. The module $EM_{<i,k>}$ has a single interaction point reference j(<i,k>,1) refering to $IP_{j(i,k)}$ and assuming the role $r(<i,k>,1) = \bar{r}(i,k)$. The behavior of the environment module $EM_{<i,k>}$ is assumed arbitrary, e.g. one could define the following transitions:

a) Input transition
    P(s,input) = <u>true</u>
    F(s,input) = s(*no change*)

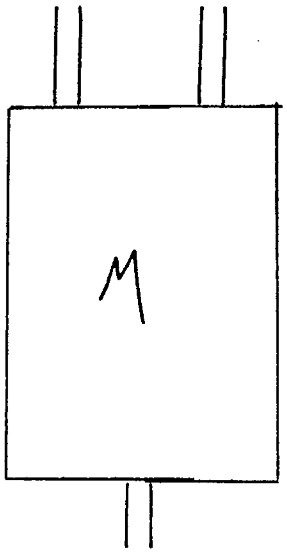
b) Output transitions, one transiton for each value "out" in $I_{j(i,k)}^{(r(i,k))}$
    P(s) = <u>true</u>
    F(s) = s(*no change*)
    O(s) = "out"

If such environment modules $EM_{<i,k>}$ are added to the system specification, as explained above, the translation (described in section 5.2.1) of the so extended system specification leads to a simple

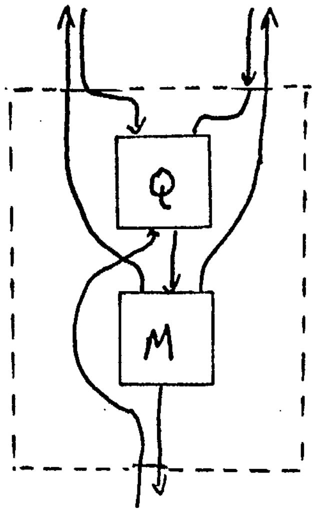
Figures for section 5:



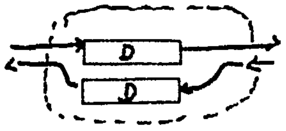
(a)



(b)



(c)



Notation:

▨	FDT module	▭	BSM module
⬭	FDT channel	═	BSM channel
		→	BSM one-directional channel

b) Module with input queue for interactions: A module in the FDT is translated into two modules of the BSM, one input queue module Q and module M similar to the module M under point (a), as shown in figure (c). Each channel in the FDT is translated into two (one-directional) interaction points in the BSM: the input from each channel leads to corresponding (input) interaction point references of Q, and the output is directly produced through corresponding (output) interaction point references of M. In addition, there is one interaction point connecting Q and M through which M receives the next input to be processed.

c) Channels with delay and modules with input queues: A channel with delay in the FDT gives rise to two additional delay modules D, one for each direction, as indicated in figure (d).

5.4.1. Module with zero-queue option

1) A module specification with zero-queue option written in the FDT is translated into a single module M_i of the basic semantic model, together with the interaction points it is connected to.

2) A single transition of the specification will, in general, be translated into several transitions of the basic semantic model. The latter transitions will be called in the following "minor transitions" to distinguish them from the former, which are simply called "transitions" and have the form defined in the specification.
In addition to an initiating minor transition, which is an input or an internal one, one may obtain one minor transition per output statement in the transition. Other translation schemes could also be envisaged. The initiating minor transition leads into a module state which is called an "intermediate" state. The last minor transition to be executed for a given transition leads back to a "major" state.

3) a) The components of the state space S_i of the module M_i correspond to the variables of the specified module, including the STATE variable as first component. The state space contains also a component, called "last-input", of type $\underset{k}{+} I_{j(i,k)}^{(r(i,k))}$ into which the last input interaction of the module is stored.
Note: this is necessary since the sequence of minor transitions resulting from the translation from a single FDT transition may all use the parameter values of the input that triggered the transition.
b) The possible values of the first component of a state s_i, called "STATE$_i$", are partitioned into "major-states$_i$" which correspond to the major states defined by the FDT specification, and into "intermediate-states$_i$", as introduced under point(2).

Note: Usually, input transitions are only possible in major states, i.e. STATE <u>in</u> intermediate-states$_i$ <u>implies</u> P(<STATE,...>, input) = <u>false</u> for any input. Note: Due to potentiel deadlocks identified, precise rules for the translation of the zero-queue case require further investigation.

Note: Due to lack of time, the formal semantics of DELAY has not yet been included.

4) After having obtained all minor transitions of a module M_i as explained under points (2) and (4) above, the following minor transitions are added. They define the action of the module in the case that a received input is not matched by any transition:
For each interaction point reference j(i,k) the following minor input transition t of the form $t_{ik}^{(in)}$ is added

$P\hat{t}(s_i,\text{input})$ = (STATE$_i^{(in)}$ <u>in</u> major-states$_i$)
<u>and</u> for all t_{ik} as obtained under point (2) and (4)
$P(s_i,\text{input})$ = <u>false</u>

$F\hat{t}(s_i,\text{input})$ = s_i (option of "NULL transition")
or
arbitrary value
(not defined by the specification)
(option of "undefined error handling")

Note: The first option implies that the input is ignored. With the second option, the resultant next state is defined by the implementation.

<u>5.4.2 Module with infinite queue option</u>

A module specification with infinite queue option written in the FDT is translated into two modules Q_i and M_i of the BSM. Q_i is the input queue described below, and M_i is as described in section 5.4.1 with the following exceptions:

(Q1) There is an additional interaction point IP_{j_q} with the sets of possible interactions

$$I_{j_q}^{(queue)} = \underset{k}{+}\, I_{j(i,k)}^{(\bar{r}(i,k))}$$

$$I_{j_q}^{(queue)} = \text{the empty set (no interaction)}$$

(Note: this is a one-directional interaction point) and the range of functions

$$CE_{j_q}^{(queue)} = \text{the empty set}$$

$$CE_{j_q}^{(queue)} = (\text{ready, wait})$$

Explanation: The module M_i indicates whether it is ready for the next input.

(Q2) The module M_i has an additional reference k_q to the interaction point IP_{j_q} with role "queue". The function C_{i,k_q} has the form

C_{i,k_q} (<STATE,...>) = if (STATE <u>in</u> major-states$_i$)
then ready else wait

Explanation: The module is ready for the next input when the module is in a major state.

(Q3) The other interaction point references (corresponding to the channels of the module in the FDT) are only used for output.

(Q4) The input queue module Q_i has interaction point references $j(i,k)$ ($k = 1,2,\ldots,K_i$; corresponding to the channels of the module in the FDT), which are only used for input. An additional reference $j(i,k_q)$ refers to the interaction point IP_{j_q}

with role "queue".

(Q5) The state space of Q_i has a single component: a queue of elements of type $\underset{k}{+} I_{j(i,k)}^{(\bar{r}(i,k))}$.

(Q6) The following transitions are defined for Q_i:
(a) For each interaction point reference $j(i,k)$, there is an input transition $t_{ik}^{(in)}$ with

  P = true

  F(q, input) = append (q, input)

  Note: the input is appended to the queue.

(b) For the interaction point reference $j(i,k_q)$, there is an output transition $t_{ik_q}^{(out)}$ with

  P(q,f) = (q is not empty) and (t = ready)

  F(q,f) = tail (q)

  O(q,f) = first(q)

Explanation: Output to the module M_i may be initiated when the queue contains on input and M_i is ready to receive one (i.e. in a major state, according to Q2). The transition has the first element of the queue on output and retains the other elements in the queue.

Note: The minor transitions corresponding to two transitions of two different modules in the FDT do not influence one another directly (except possibly enabling the other). Therefore they may be executed in parallel.

5.4.3 Module with infinite queue and SAVE options

A module specification in the FDT with infinite queue and SAVE options is translated into two modules Q_i and M_i of the basic semantic model as described in section 5.4.2 with the following exceptions:

(S1) $CE_{j_q}^{(queue)}$ = set of $(+I_{k_{j(i,k)}}^{(\bar{r}(i,k))})$, however, the values of the interaction parameters may be ignored.

Explanation: the module indicates which kinds of interactions are presently in the "save set".

(S2) C_{i,k_q}(<STATE,...>) = if STATE in major-states$_i$
then "save set" of STATE
else "all kinds of interactions".

Note: No change to (Q3) through (Q5).

(S6) The transition $t_{k_q}^{(out)}$ has the form

P(q,f) = q contains an element not part of the "save-set" f.
O(q,f) = the first such element of q.
F(q,f) = the queue q with the output element removed.

5.4.5. Examples of translations:

Example A:

```
from A to B
when IP-reference$_k$.primitive-in
provided <exp> (*may depend on parameters of primitive-in*)
begin <statements 1>;
    out   IP-reference$_{k'}$.primitive-out(<effective   parameter
list>);
      <statements 2> end;
```

may be translated into the following two minor transitions of M_i:

```
$t_{ik}^{(in)}$ : P = (STATE=A) and input is a case of primitive-in
                                          and <exp>
          F : last-input := input;
              STATE:=intermediate-1;
              other components changed according to
                 <statements 1>
```

$t_{ik'}^{(out)}$: P = (STATE = intermediate-1)
O : defined by output statement (depending on the module state at the beginning of the minor transition $t_{ik'}^{(out)}$)
F : STATE:=B;
other components changed according to <statements 2>;

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

To be provided.

7. Conformity rules for checking implementations

to be provided

8. Terminology

to be provided

Annex 1: User guidelines

Annex 2: Applications to draft standards

Annex 3: Language support tools

Annex 4: Check against evaluation criteria

Annex 5: 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 N2281.

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.

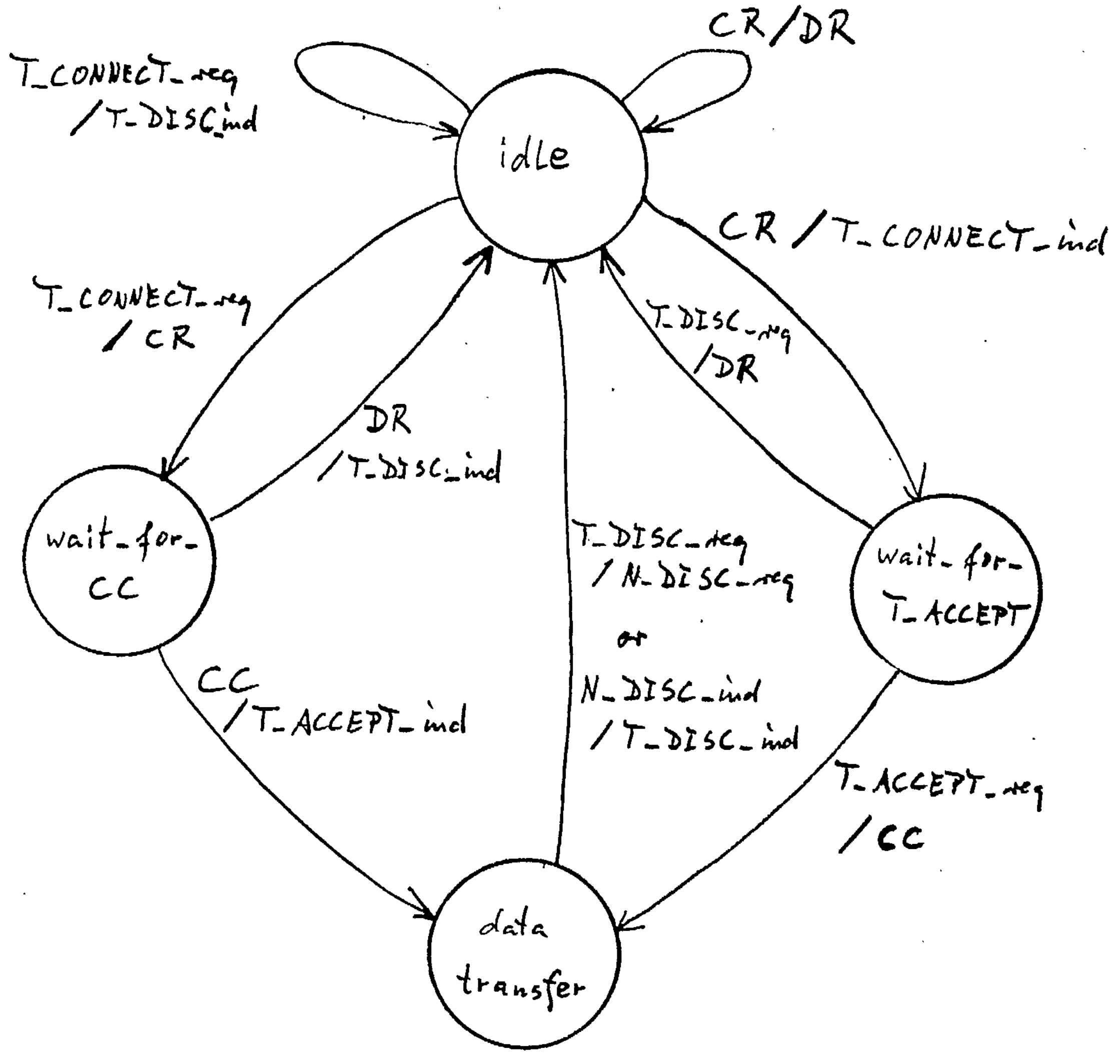

CR/DR
T_CONNECT_req /T_DISC_ind
idle
CR /T_CONNECT_ind
T_CONNECT_req / CR
T_DISC_req /DR
DR /T_DISC_ind
wait_for_CC
T_DISC_req / N_DISC_req or N_DISC_ind / T_DISC_ind
wait_for_T_ACCEPT
CC /T_ACCEPT_ind
T_ACCEPT_req /CC
data transfer

Figure 1

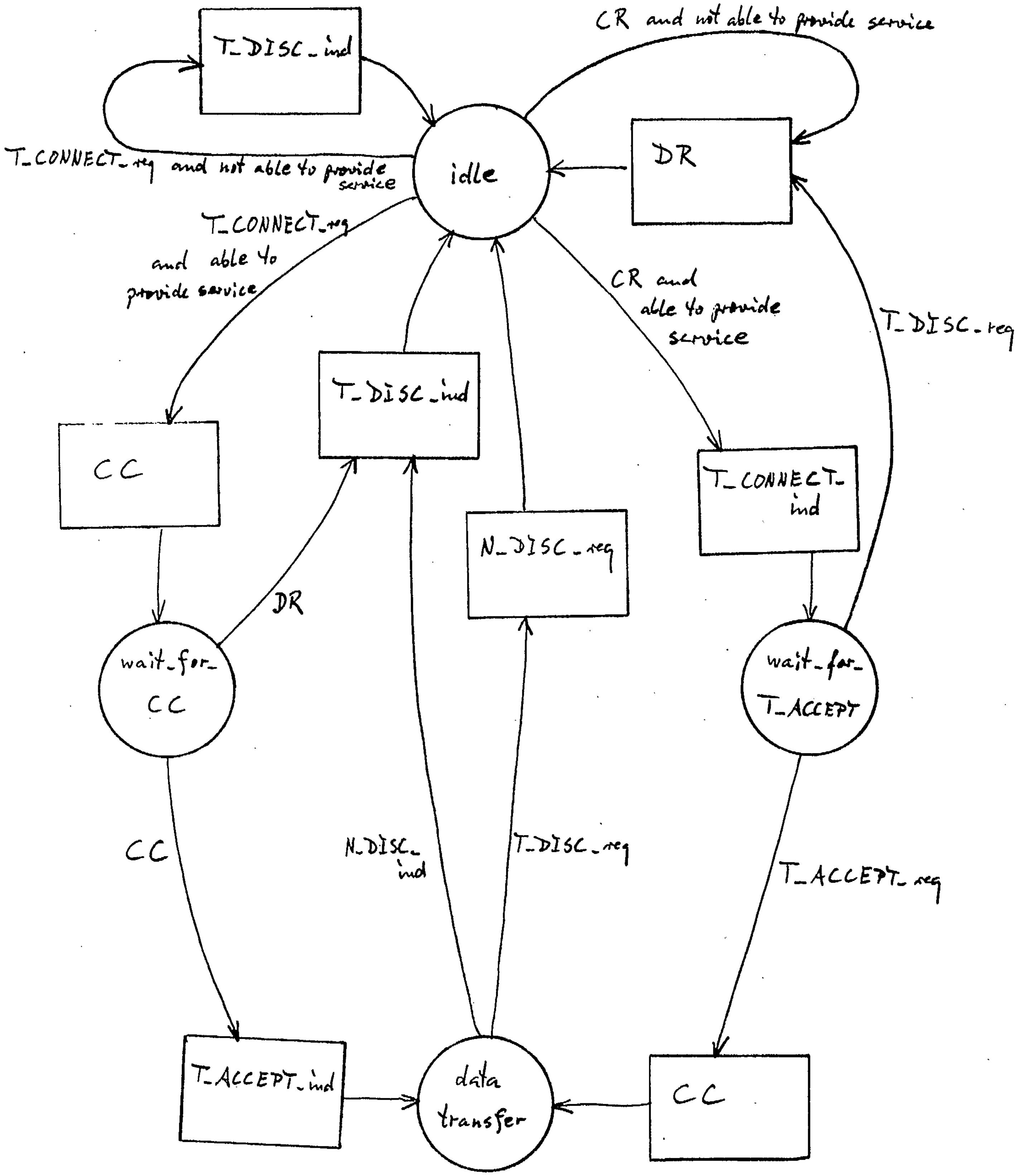

T_DISC_ind
CR and not able to provide service
T_CONNECT_req and not able to provide service
idle
DR
T_CONNECT_req and able to provide service
CR and able to provide service
T_DISC_req
T_DISC_ind
CC
T_CONNECT_ind
N_DISC_req
DR
wait_for_CC
wait_for_T_ACCEPT
CC
N_DISC_ind
T_DISC_req
T_ACCEPT_req
T_ACCEPT_ind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

Figure 2

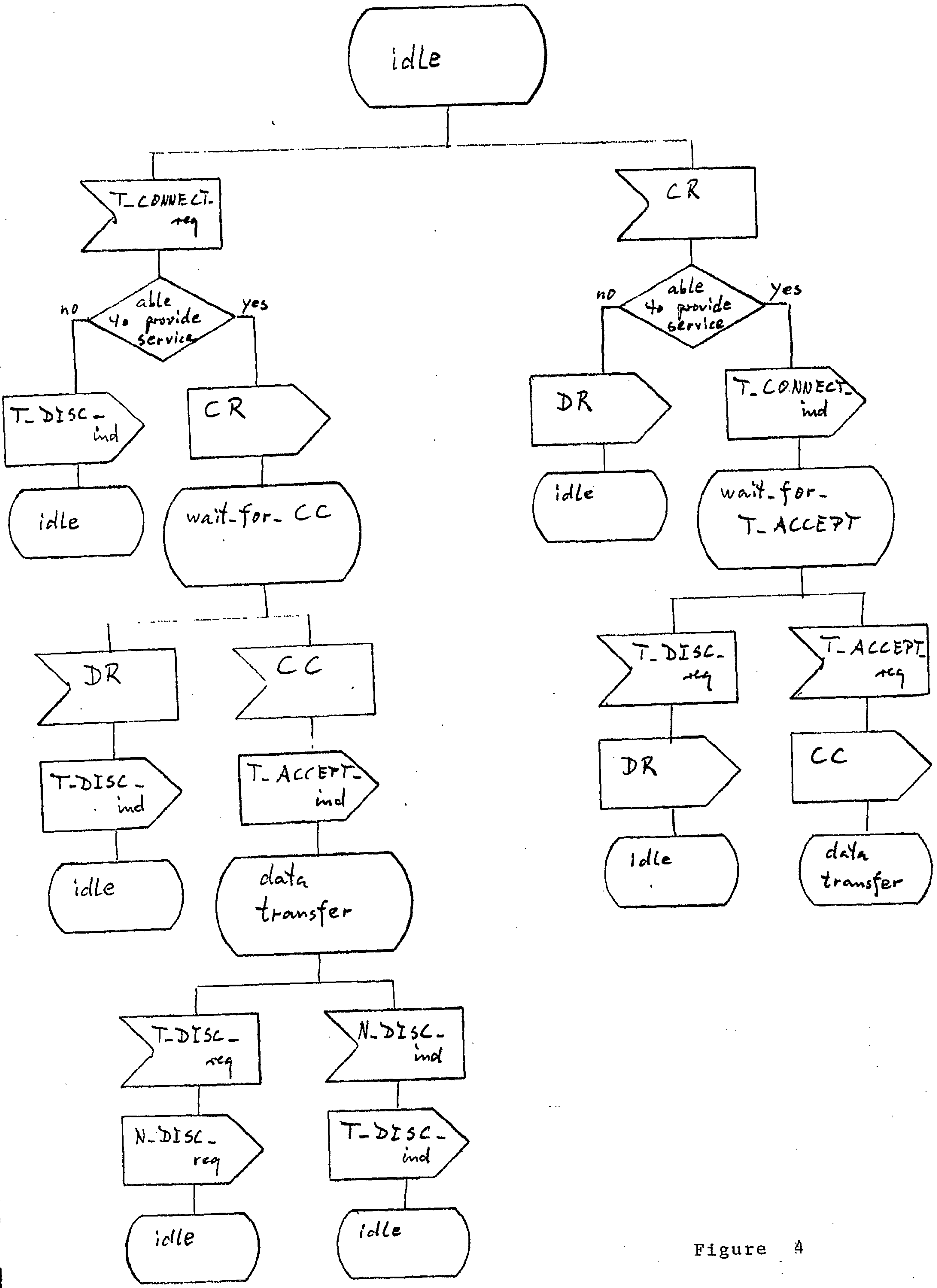

idle
T_CONNECT_req
able to provide service
no
yes
T_DISC_ind
idle
CR
wait_for_CC
DR
T-DISC_ind
idle
CC
T_ACCEPT_ind
data transfer
T_DISC_req
N_DISC_req
idle
N_DISC_ind
T_DISC_ind
idle
CR
able to provide service
no
yes
DR
idle
T_CONNECT_ind
wait_for_T_ACCEPT
T_DISC_req
DR
Idle
T_ACCEPT_req
CC
data transfer

Figure 4

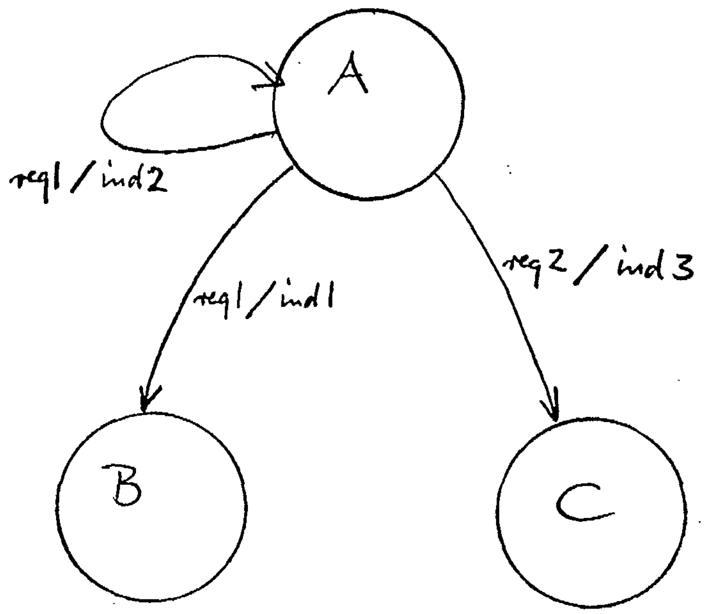


Figure 4

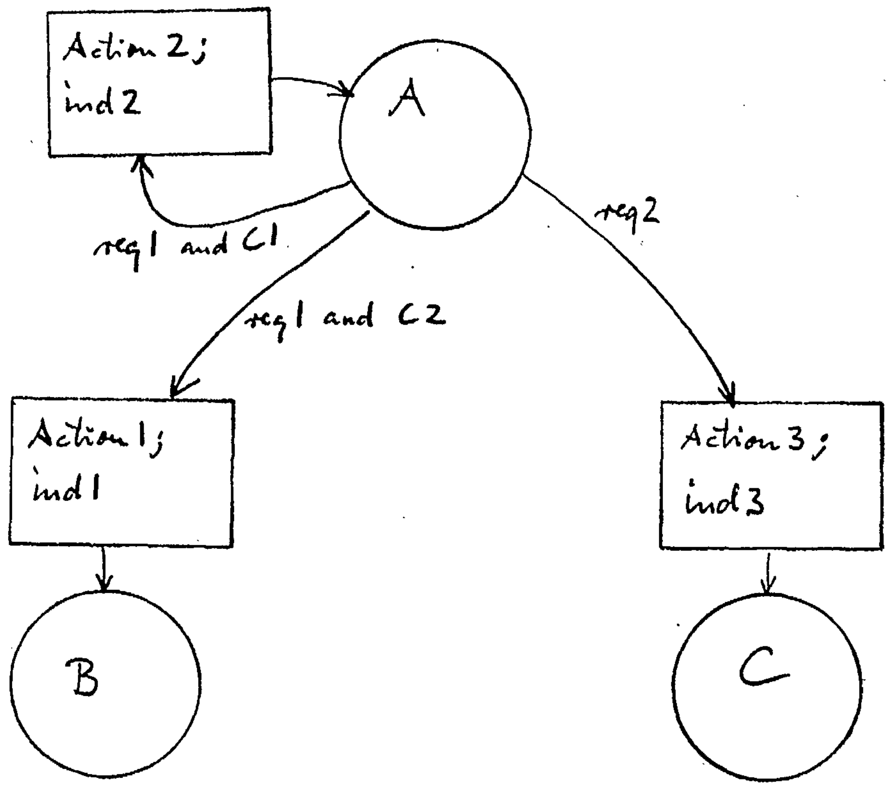


Figure 5

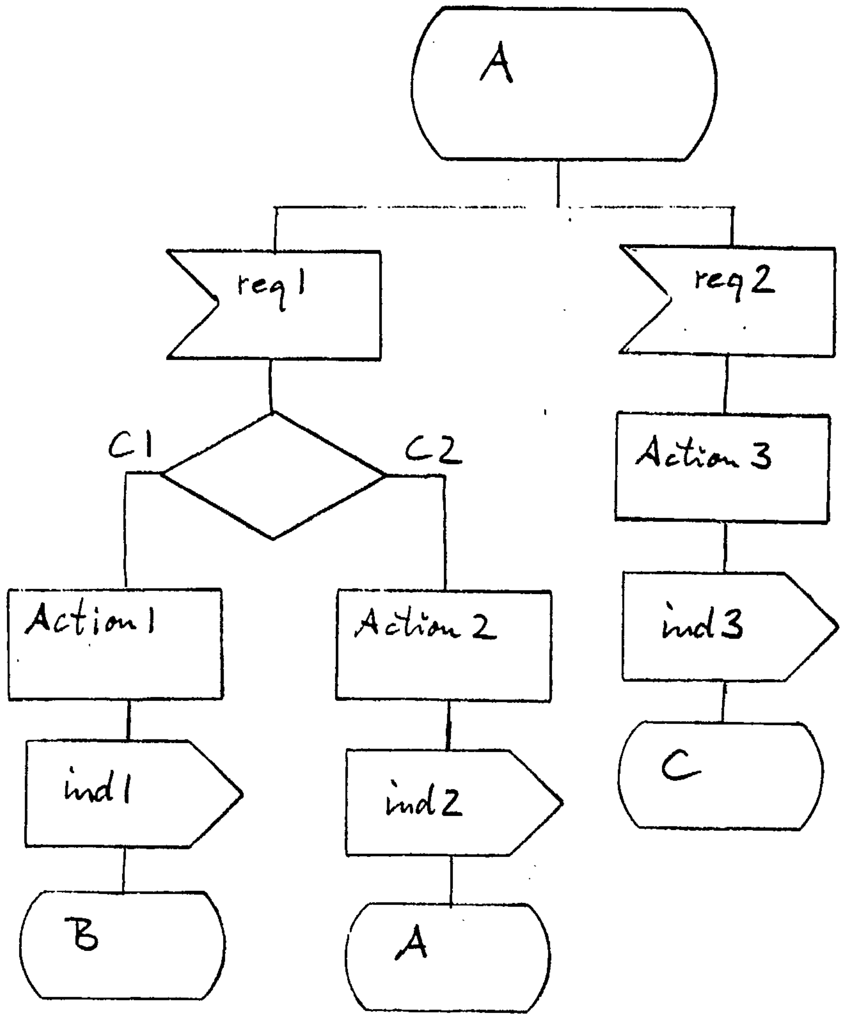


Figure 6

Annex 6: Language elements for further extension

This annex gives a list of elements for further extension in the sense that solving the issues which are raised here is not considered as necessary before considering the language as stable.

a - user_data_type operation
b - coding/decoding
e - address manipulation
d - SAP management
f - module management
e - include
g - channels with states and other intelligence

Note: however point b is considered of primary importance (further study)

The following contain a preliminary proposal made by some experts which has been presented during the Catania meeting but neither deeply discussed nor agreed.

Some preliminary proposal for user data type, coding functions, SAP and address manager functions.

(this requires further consideration and is viewed as a candidate for extension of the language)

1. The following functions and procedures (predefined) are associated with the predefined user data type.

```
dassemble (ul, u2:  userdatatype);
    ul  and  u2  are assembled and the result is assigned to
    ul

dfragments (ul:  userdatatype, l:integer): userdatatype
    ul is fragmented into two parts.  The first part  of  ul
    (from octet 1 to octet l) is assigned to  u2;
    ul  contains the remaining part
dcopy (u : userdatatype):  userdatatype;
    a  "copy"  of   u   is  retained  by the user and can be
    affected to an other variable.

dlength (u :  userdatatype): integer;
    gives the length of variable  u.
```

2. Particularity

Since the predefined type is not specified (i.e. abstract) the so-called 'octet' could be anything (e.g. digit, byte of 7 bits...)

3. Function allowed with this type

The only operations allowed on user_data_type are the predefined function or procedures. Assigment or tests of variables of the type are prohibited. The only way to assign a variable of the type is to use a for or procedure

exemple

```
u:=v   is illegal
u:=dcopy(v) is legal.
```

Note

as soon as a variable of this type is put in the parameters of an interaction passed through a channel then the variable has a length of o. This modelizes the fact that the data received from the user are passed to the layer above.

4. <u>Additional operation</u>
- dassign (u:udata_type):udatatype

  this allows an entity to move a userdata from a variable to an other one - for instance because a received user data cannot be transmitted immediately
- durger (u: udatatype)

  a received user data is dropped

5. <u>Pdu coding/decoding</u>

A new keyword in the language is created called 'PDU' which allows to declare under this new section the PDU sent or received in the same way interaction are declared (Pascal like record).

This section allows the following types and functions for each 'name' declared in the PDU Section.
- an 'encode-name' function is predeclared having as parameters the parameters of the PDU called 'name' and as a result result a variable of the u_data_type [this function allows to formally describe that before sending a PDU it is necessary to transfer the 'logical' format (i.e. the local variable) into the 'physical' format (i.e. the real string of bits in the real encoding method used by the protocol)]

- a 'decode_name' function is predeclared making the opposite translation

- a 'identify-pdu' function

<u>exemple</u>
```
PDU
  CR (credit:integer;destref,localref:integer;
      clan:clan_type;alt_class:class_type;
      option:option_type;udata:user_data_type);

  CC(... idem)

  DT (etc: boolean;u_data:userdata_type);

when N.DTindication

provided identify_pdu (netudata)=CR
         decode_CR;
         decode_CR;

provided identify_pdu(netudata)=CC
         decode_CC;

provided identify_pdu(netudata)=DR
         decode_CC;

when N.DTindication priority 0    (*protocol error*)
  begin
     out N_DISCONNECT 0
  end

when CR
    .. similarly

when CC
    .. similarly

when DT
    .. similarly
```

- notice that the 'decode-name' function performs an internal 'call' the interaction labelled by the PDU_name.
  This feature allows to deal in the same automaton with (N-1) service events and (N) PDU events (which are in fact internally generated when receiving an other interaction).

- notice that this allows to have PDU received and decoded in user data of different (N-1) interactions.

- notice that this proposes coherent solution for

    - coding/decoding
    - (N-1)SDU/(N)PDU
    - abstract user data manipulation.

[complete example can to found in CATANIA 19]

<u>Address SAP manipulation function</u>

the following predefined procedure or function can be defined

```
mappaddress ((N)-address, (N-1) address, N_suffix);
buildaddress((N)-address, (N-1) address, N_suffix);
testaddress ((N)-address, (N-1) address):boolean
testhigSAP((N)address):boolean
testlowSAP((N)-1) address):boolean
createhighCEP((N)-1)address):CEP_id_type
createlowCEP ((N)-1)address):CEP_id_type
```

They use the predefined types

```
N_address_type
N_suffix
Nminsl_address_type
CEP_id_type.
```

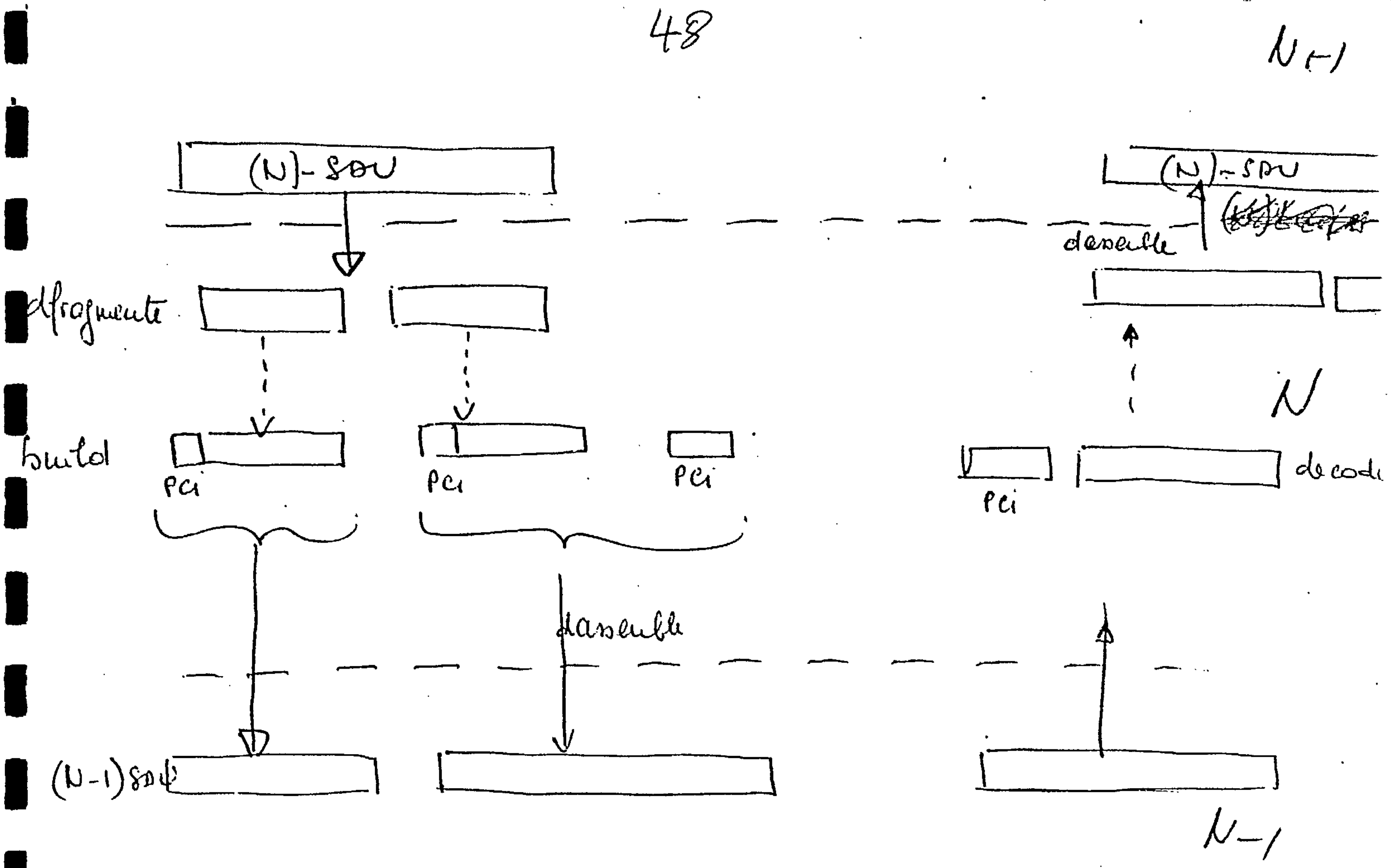


figure-1 : OSI model and ~~user-data-t~~ operations on user-data-type

ANNEX 3

This paper is submitted to the meeting of ISO TC97/SC16/WG1 ad hoc group on FDT, Enschede, April 1982.
Title : Formal description of the Transport service

Source: G.V. Bochmann (Canada)

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. Connection identification

The specification of the communication service foresees an arbitrary number of service access points identified by the address, and distibuted over the different systems of the Open Systems environment. Each access point can handle an arbitrary number of simultaneous connections to different other access points. The connections at a given accesss point are distinguished by the connection end point identifiers (CEP_id).

Since the above connection identification is specific to each end point of a connection, the service specification introduces an independent connection identification bases on connection identifiers ("conn_id" of type "TC_id_type"). The mapping between this internal identification and the identification at the endpoints is given by the functions "find_TC_id" which finds the internal identification from the identification at one endpoint (address and TCEP_id), and "this_side" which determines for a given connection end point identification (address and TCEP_id) whether this endpoint is the calling or called side of the connection.

1.2. Buffers for data transfer

For each connection, two data buffers are used to hold the user data in transit in the two directions of data transfer. The interactions of these data buffers are described, but their properties are not formally defined. We note, however, that service data units (TSDU's) are exchanged between the user and the Transport service in fragments (fragmentation may be different at the two endpoints of a connection), and the end-of-TSDU mark is handled as well. Flow control for normal data at an access point is related to the flow control into the corresponding data buffer.

1.3. The states of the Transport service

The internal states of the Transport service is an array which contains for each connection the following state information:
(a) for each TCEP (i.e. calling and called side)
- a major "state" indicating whether the connection is open etc.
- the endpoint identification of the conection,
- some additional information related to the endpoint;

(b) information about options used, quality of service parameters, ect.

1.4. Grouping of transitions

The transitions of the specification are grouped according to the funtion they perform, i.e. into CONNECTION ESTABLISHMENT, DATA TRANSFER, EXPEDITED DATA TRANSFER, and TERMINIATION.

2. Formal specification

```
type
   T_address_type = ...;
   known_T_addresses = ...; (* sub-type of T_address_type *)
   TCEP_id_type = ...; (* implementation dependent *)
   quality_of_TS_type = record
      class_of_service : (basic, enhanced);
      throughput_average : integer; (* in seconds *)
      (* and possibly other quality parameters *)
      end;
   option_type = set of (expedited_data (* and possibly other options *) )
   string_of_octets = record
      length : pos_integer;
      contents : array [1 .. ...] of 0 .. 255;
      end;
   TS_connect_data_type (* property: maximum length = 80 *),
   TS_accept_data_type  (* property: maximum length = 80 *),
   TS_expedited_data_type (* property: maximum length = 16 *),
   TS_user_reason_type  (* property: maximum length = 80 *)
        = string_of_octets;
   TS_disconnect_reason_type = (TS_U_NRM, TS_CONG, TS_FAIL, TS_QUAL_FAIL,
                                 U_UNKNOWN);
   both_sides = (calling, called);
```

```
interaction
   TS_primitives (TS_user, TS_provider)
   by TS_user :
      T_CONNECT_req (TCEP_id : TCEP_id_type;
                     to_T_address,
                     from_T_address : T_address_type;
                     proposed_options : option_type;
                     proposed_QTS : quality_of_TS_type;
                     data : TS_connect_data_type):

      T_CONNECT_resp (TCEP_id : TCEP_id_type;
                      proposed_QTS : quality_of_TS_type;
                      proposed_options : option_type;
                      data : TS_accept_data_type);

      T_DISCONNECT_req (TCEP_id : TCEP_id_type;
                        TS_user_reason : TS_user_reason_type);

      T_DATA_req (TCEP_id : TCEP_id_type;
                  TS_user_data : string_of_octets;
                  is_last_fragment_of_TSDU : boolean );

      T_EX_DATA_req (TCEP_id : TCEP_id_type;
                     TS_user_data : TS_expedited_data_type);

      T_EX_D_READY_resp (TCEP_id : TCEP_id_type)

   by TS_provider :
      T_CONNECT_ind (TCEP_id : TCEP_id_type;
                     to_T_address,
                     from_T_address : T_address_type;
                     proposed_options : option_type;
                     proposed_QTS : quality_of_TS_type;
                     data : TS_connect_data_type);

      T_CONNECT_conf (TCEP_id : TCEP_id_type;
                      proposed_QTS : quality_of_TS_type;
                      proposed_options : option_type;
                      data : TS_accept_data_type);

      T_DISCONNECT_ind (TCEP_id : TCEP_id_type;
                        TS_disconnect_reason : TS_disconnect_reason_type;
                        TS_user_reason : TS_user_reason_type);
      T_DATA_ind (TCEP_id : TCEP_id_type;
                  TS_user_data : string_of_octets;
                  is_last_fragment_of_TSDU : boolean );

      T_EX_DATA_ind (TCEP_id : TCEP_id_type;
                     TS_user_data : TS_expedited_data_type);

      T_EX_D_READY_conf (TCEP_id : TCEP_id_type);
```

```
type
   TC_id_type = ...;

interaction
   TS_data_buffer (user, buffer)
   by user :
      clear;
      append (fragment : string_of_octets;
              is_last_fragment_of_TSDU : boolean);
          (* another data fragment is inserted into the buffer *)
   by buffer :
      next_fragment (fragment : string_of_octets;
                     is_last_fragment_of_TSDU : boolean);
          (* the buffer delivers another data fragment to the user *)
      free_space; (* used for Transport protocol specification *)

module TS (AP : array [T_address_type] of TS_primitives (TS_provider);
           buffer : array [TC_id_type, both_sides] of TS_data_buffer (user
var
   TC : array [TC_id_type] of record
      EP : array [both_sides] of record
         state : (closed, open_in_progress, open, close_in_progress);
         address : T_address_type;
         id : TCEP_id_type;
         QTS_estimate : quality_of_TS_type;
         EX_D_state : (idle, EX_D_in_transit, wait_for_resp,
                       EX_D_conf_in_transit);
         (* this state concerns the transfer of expedited data
            from this EP to the opposite one *)
         last_EX_data : TS_expedited_data_type;
         end;
      options : option_type;
      connect_data : TS_connect_data_type;
      accept_data: TX_accept_data_type;
      TS_reason : TS_disconnect_reason_type;
      user_reason : TS_user_reason_type;
      end;

function opposite (side : both_sides) : both_sides;
   begin
      if side = calling
      then opposite := called  else opposite := calling
   end;

function find_conn_id (T_addr : T_address_type;
                       TCEP_id : TCEP_id_type) : TC_id_type;
   begin ... (* property: see property below *) end;

function this_side (T_addr : T_address_type;
                    TCEP_id : TCEP_id_type ) : both_sides;
   begin ... end;

   (* property:
         with TC [find_TC_id (T_addr, TCEP_id)].
                 EP [this_side (T_addr, TCEP_id)] holds
            address = T_addr
            and  id = TCEP_id
            and  state <> closed
         or TC [find_TC_id (T_addr, TCEP_id)]. EP [side]. state = closed
               for both sides;
      i.e. find_TC_id returns the TC identified by TCEP_id at the AP
              at the address T_addr;
           this_side indicates whether this end of the connection is
              calling or called. *)
```

```
(* CONNECTION ESTABLISHMENT PHASE *)

any T_addr : T_address_type do
when AP[T_addr]. T_CONNECT_req
   provided ... (* property: for all conn_id : TC_id_type,
                                all side : both_sides  holds
                             with TC[conn_id]. EP[side] do
                                state <> closed implies
                                    address <> T_addr  or
                                    id <> TCEP_id          *)
                (* i.e. the TCEP identifier is not yet in use at the
                          same AP *)
            and from_T_address = T_addr
            and ... (* able to provide the service asked for *)
       var conn_id : TC_id_type;
       begin
          conn_id := ...; (* property: for all side : both_sides holds
                                        TC[conn_id].EP[side].state = closed
                          (* i.e. connection not yet in use *)
          initialize (conn_id);
          with TC [conn_id] do begin
             with EP [calling] do begin
                state := open_in_progress;
                id := TCEP_id;
                address := T_addr;
                QTS_estimate := proposed_QTS;
                EX_D_state := idle;
                end;
             with EP [called] do begin
                address := to_T_address;
                EX_D_state := idle;
                end;
             options := proposed_options;
             connect_data := data;
             end;
          buffer[conn_id, calling].clear;
          buffer[conn_id, called].clear;
          end;

   provided ... (* not able to provide the service asked for *)
      begin
         AP[T_addr]. T_DISCONNECT_ind (TCEP_id, ... (* property:
              if not to_T_address in known_T_addresses  then U_UNKNOWN *),
              ... (* dummy *) );
      end;
```

```
any conn_id : TC_id_type do   with TC[conn_id] do
provided   (EP[calling]. state = open_in_progress)
   and  (EP[called]. state = closed)
   (* when the connection request reaches the called side *)
   begin  with EP[called] do begin
      state := open_in_progress;
      id := ...; (* property: for all conn_id' <> conn_id,
                                     all side : both_sides  holds
                                 with TC[conn_id'].EP[side] holds
                                    state <> closed implies
                                       address <> TC[conn_id].EP[called].addr
                                       or  id <>  TC[conn_id].EP[called].id
                  (* i.e. the identifier is not yet in use *)
      QTS_estimate := ...; (* not defined by this standard *)
      AP[address]. T_CONNECT_ind (id, address, EP[calling].address,
                     options, QTS_estimate, connect_data);
   end end;

any T_addr : T_address_type do
when AP[T_addr]. T_CONNECT_resp   with TC [find_conn_id (T_addr, conn_id)]
   provided (this_side (T_addr, conn_id) = called)
        and (EP[called].state = open_in_progress)
        and ((data = undefined)  or  (connect_data <> undefined))
        and (proposed_options in options)
     . begin
        EP[called]. state := open;
        options := proposed_options;
        accept_data := data;
      end;

(* There is also the possiblity of disconnection, see termination phase *)

any conn_id : TC_id_type do    with TC[conn_id] do
provided  EP[calling]. state = open_in_progress
      and EP[called]. state = open
      (* when the connect response reaches the calling side *)
   begin  with EP[calling] do begin
      state := open;
      QTS_estimate := ...; (* not defined by standard *)
      AP[address]. T_CONNECT_conf (id, options, QTS_estimate, accept_data)
      end end;
```

```
(* DATA TRANSFER *)

any T_addr : T_address_type do
when AP[T_addr]. T_DATA_req
   with TC[find_conn_id (T_addr, TCEP_id)].
           EP [this_side (T_addr, TCEP_id)] do
   provided  state = open
         and ... (* property:
                    flow control to the Transport entity is ready *)
      begin
         buffer[find_conn_id (T_addr, TCEP_id),
                this_side (T_addr, TCEP_id)].
                    append (TS_user_data, is_last_fragment_of_TSDU);
         end;

any conn_id : TC_id_type do
any side : both_sides do
      with TC[conn_id].EP [opposite(side)] do
when buffer[conn_id, side]. next_fragment
   provided  state = open
         and ... (* property: flow control to user is ready *)
         and ... (* there is no expedited data in transit
                    (in the same direction of transfer) which was sent pri
                    to the next normal data fragment *)
      begin
         AP[address]. T_DATA_ind
              (id, TSDU_fragment, is_last_fragment_of_TSDU);
         end;
```

```
(* EXPEDITED DATA TRANSFER *)

any T_addr : T_address_type do
when AP[T_addr]. T_EX_DATA_req
      with TC [find_conn_id (T_addr, TCEP_id)] do
      with EP [this_side (T_addr, TCEP_id)] do
   provided  expedited_data  in  options
         and state = open
         and EX_D_state = idle
      begin
         last_EX_data := TS_user_data;
         EX_D_state := EX_D_in_transit;
         end;

any conn_id : TC_id_type do        with TC[conn_id] do
any side : both_sides do
provided  EP[side]. state = open
      and EP[side]. EX_D_state = EX_D_in_transit
      (* when the expedited data reaches the destination *)
   begin
      AP [EP [opposite(side)].address]. T_EX_DATA_ind (
                 EP[opposite(side)]. id,
                 EP[side]. last_EX_data );
      EP[side]. EX_D_state := wait_for_resp;
      end;

when AP[T_addr]. T_EX_D_READY_resp
      with TC [find_conn_id (T_addr, TCEP_id)] do
   provided  expedited_data  in options
         and EP [this_side (T_addr, TCEP_id)]. state = open
         and EP [opposite (this_side (T_addr, TCEP_id))].
                   EX_D_state = wait_for_resp
      begin
         EP [opposite (this_side (T_addr, TCEP_id))].
                   EX_D_state := EX_D_conf_in_transit;
         end;

any conn_id : TC_id_type  do
any side : both_sides do
       with TC[conn_id]. EP[side] do
provided  state = open
      and EX_D_state = EX_D_conf_in_transit
      (* when the confirmation reaches the sender of the expedited data *)
   begin
      AP [address]. T_EX_D_READY_conf (id);
      EX_D_state := idle;
      end;
```

```
(* TERMINATION PHASE *)

any T_addr : T_address_type do
when AP[T_addr]. T_DISCONNECT_req
     with TC[find_conn_id (T_addr, TCEP_id)] do
     with EP [find_side (T_addr, TCEP_id)] do
  provided state in [open_in_progress, open]
     begin
        state := close_in_progress;
        TS_reason := TS_user_initiated_termination;
        user_reason := ... (* property: either equal to "TS_user_reason"
            or undefined, i.e. the transmission of this additional
            information is not guaranteed by the TS_provider *);
        end;

any conn_id : TC_id_type do    with TC[conn_id] do
any side : both_sides do       with EP[side] do

  provided state in [open_in_progress, open]
        and ... (* when internal problem of TS_provider *)
     begin
        state := close_in_progress;
        TS_reason := ...; (* property: not U_UNKNOWN *)
        AP[address]. T_DISCONNECT_ind (id, TS_reason, ... (* dummy *) );
        end;

  provided  EC [opposite (side)]. state = close_in_progress
            (* when the disconnect reaches the other end of the connectio
     begin
        if state in [open_in_progress, open]
           then AP[address]. T_DISCONNECT_ind (id, TS_reason, user_reason
        state := closed;
        EP [opposite(side)].state := closed;
        end;
```

ANNEX 4

Contribution to the meeting (July 1982) of the Subgroup B of the ISO TC97/SC16/WG1 ad hoc group on FDT

Title: Example description of the Transport service

Author: G. v. Bochmann and K. S. Raghunathan

June 1982

1. Introduction

1.1. Connection identification

The specification of the communication service foresees an arbitrary number of service access points identified by the address, and distibuted over the different systems of the Open Systems environment. Each access point can handle an arbitrary number of simultaneous connections to different other access points. The connections at a given accesss point are distinguished by the connection end point identifiers (CEP_id).

Since the above connection identification is specific to each end point of a connection, the service specification introduces an independent connection identification based on connection identifiers ("conn_id" of type "TC_id_type"). The mapping between this internal identification and the identification at the endpoints is given by the functions "find_TC_id" which finds the internal identification from the identification at one endpoint (address and TCEP_id), and "this_side" which determines for a given connection end point identification (address and TCEP_id) whether this endpoint is the calling or called side of the connection.

1.2. Buffers for data transfer

For each connection, two data buffers are used to hold the user data in transit in the two directions of data transfer. The interactions of these data buffers are described, but their properties are not formally defined. We note, however, that service data units (TSDU's) are exchanged between the user and the Transport service in fragments (fragmentation may be different at the two endpoints of a connection), and the end-of-TSDU mark is handled as well. Flow control for normal data at an access point is related to the flow control into the corresponding data buffer.

1.3. The states of the Transport service

The internal states of the Transport service is an array which contains for each connection the following state information:

(a) for each TCEP (i.e. calling and called side)
- a major "state" indicating whether the connection is open etc.
- the endpoint identification of the conection,
- some additional information related to the endpoint;

(b) information about options used, quality of service parameters, etc.

1.4. Specifying local rules for interactions at an access point

In the description of the Transport service, the local rules that determine the order in which the service primitives may be executed at a given connection end point (TCEP), has been specified as <constraint> of the TCEP. The syntax assumes that the symbol <interactions> in the syntactic rule for <channel type def> in the FDT is replaced by the two symbols <interactions> <constraint>, and the constraint is defined by
    <constraint> ::= empty | <module body>

The first part of the description defines the service primitives with their parameters (using the Subgroup B syntax), while the second part, viz <constraint>, defines the local order in which these primitives may be executed (using the extended state transition model, without outputs). For instance, the first transition of the <constraint> is interpreted as: From 'closed' state a 'T_CONNECT_req' interaction will lead to the 'open_in_progress' state with the variable 'side' assigned the value 'calling'. The <constraint> defines essentially the state transitions of the extended state transition model of the TCEP.

The third part of the description, viz <module>, defines the global end-to-end properties of the service which relate the interactions taking place at diffent access points. In this part, the state variables of the TCEP's are sometimes refered to (using the dot notation: <endpoint name> . <state variable name>).

1. 5. Grouping of transitions

The transitions of the specification are grouped according to the funtion they perform, i. e. into CONNECTION ESTABLISHMENT, DATA TRANSFER, EXPEDITED DATA TRANSFER, and TERMINIATION.

2. Formal specification

```
 type
    T_address_type = ...;
    known_T_addresses = ...; (* sub-type of T_address_type *)
    TCEP_id_type = ...; (* implementation dependent *)
    quality_of_TS_type = record
       class_of_service : (basic, enhanced);
       throughput_average : integer; (* in seconds *)
         (* and possibly other quality parameters *)
       end;
    option_type = set of (expedited_data (* and possibly
                                             other options*));
    string_of_octets = record
       length : pos_integer;
       contents : array [1 .. ...] of 0 .. 255;
       end;
    TS_connect_data_type (* property: maximum length = 80 *),
    TS_accept_data_type  (* property: maximum length = 80 *),
    TS_expedited_data_type (* property: maximum length = 16 *),
    TS_user_reason_type  (* property: maximum length = 80 *)
         = string_of_octets;
    TS_disconnect_reason_type = (TS_U_NRM, TS_CONG, TS_FAIL,
                                 TS_QUAL_FAIL, U_UNKNOWN);
    both_sides = (calling, called);
```

```
interaction
   TCEP_primitives (TS_user, TS_provider)
   by TS_user :
      T_CONNECT_req (to_T_address,
                     from_T_address : T_address_type;
                     proposed_options : option_type;
                     proposed_QTS : quality_of_TS_type;
                     data : TS_connect_data_type);

      T_CONNECT_resp (proposed_QTS : quality_of_TS_type;
                      proposed_options : option_type;
                      data : TS_accept_data_type);

      T_DISCONNECT_req (TS_user_reason : TS_user_reason_type);

      T_DATA_req (TS_user_data : string_of_octets;
                  is_last_fragment_of_TSDU : boolean );

      T_EX_DATA_req (TS_user_data : TS_expedited_data_type);

      T_EX_D_READY_resp;

   by TS_provider :
      T_CONNECT_ind (to_T_address,
                     from_T_address : T_address_type;
                     proposed_options : option_type;
                     proposed_QTS : quality_of_TS_type;
                     data : TS_connect_data_type);

      T_CONNECT_conf (proposed_QTS : quality_of_TS_type;
                      proposed_options : option_type;
                      data : TS_accept_data_type);

      T_DISCONNECT_ind (TS_disconnect_reason :
                                TS_disconnect_reason_type;
                        TS_user_reason : TS_user_reason_type);
      T_DATA_ind (TS_user_data : string_of_octets;
                  is_last_fragment_of_TSDU : boolean );

      T_EX_DATA_ind (TS_user_data : TS_expedited_data_type);

      T_EX_D_READY_conf;

interaction
   TSAP = array [TCEP_id_type] of TCEP_primitives;
```

```
constraint

   var
      state : (closed, open_in_progress, open);
      side : both_sides;
      EX_D_send_state,
      EX_D_receive_state : (EX_idle, EX_transfer);

   from closed
      when T_CONNECT_req                        to open_in_progress
         begin
            side := calling;
            end;
      when T_CONNECT_ind                        to open_in_progress
         begin
            side := called;
            end;

   from open_in_progress
      when T_DISCONNECT_req                     to closed;
      when T_DISCONNECT_ind                     to closed;
      when T_CONNECT_resp
         provided side = called                 to open;
      when T_CONNECT_conf
         provided side = calling                to open;

   from open
      when T_DISCONNECT_req                     to closed;
      when T_DISCONNECT_ind                     to closed;
      when T_DATA_req                           to open;
      when T_DATA_ind                           to open;
      when T_EX_DATA_req
         provided EX_D_send_state = EX_idle                 to open
            begin
               EX_D_send_state := EX_transfer;
               end;
      when T_EX_DATA_ind
         provided EX_D_receive_state = EX_idle              to open
            begin
               EX_D_receive_state := EX_transfer;
               end;
      when T_EX_D_READY_resp
         provided EX_D_receive_state = EX_transfer    to open
            begin
               EX_D_receive_state := EX_idle;
               end;
      when T_EX_D_READY_conf
         provided EX_D_send_state = EX_transfer       to open
            begin
               EX_D_send_state := EX_idle;
               end;
```

```
init begin
   state := closed;
   EX_D_send_state := EX_idle;
   EX_D_receive_state := EX_idle;
   end;
```

```
 type
    TC_id_type = ...;

 interaction
    TS_data_buffer (user, buffer)
    by user :
       clear;
       append (fragment : string_of_octets;
               is_last_fragment_of_TSDU : boolean);
       (* another data fragment is inserted into the buffer*)
    by buffer :
       next_fragment (fragment : string_of_octets;
                      is_last_fragment_of_TSDU : boolean);
      (* the buffer delivers another data fragment to the user *)
       free_space; (* used for Transport protocol
                               specification *)

 module TS (AP : array [T_address_type] of TSAP (TS_provider);
            buffer : array [TC_id_type, both_sides] of
                     TS_data_buffer (user));
```

```
var
   TC : array [TC_id_type] of record
      EP : array [both_sides] of record
         close_in_progress : boolean; (* this side has initiated
                                         a disconnect *)
         address : T_address_type;
         id : TCEP_id_type;
         (* property:
          AP[address][id].side  =  index (both_sides) in EP *)
         QTS_estimate : quality_of_TS_type;
         EX_in_transit : boolean; (* from this side to
                                     opposite side *)
         EX_data : TS_expedited_data_type; (* ..., if any *)
         end;
      options : option_type;
      connect_data : TS_connect_data_type;
      accept_data: TS_accept_data_type;
      TS_reason : TS_disconnect_reason_type;
      user_reason : TS_user_reason_type;
      end;

function opposite (side : both_sides) : both_sides;
   begin
      if side = calling
      then opposite := called  else opposite := calling
   end;

function find_conn_id (T_addr : T_address_type;
                       TCEP_id : TCEP_id_type) : TC_id_type;
   begin ... (* property: see property below *) end;

function this_side (T_addr : T_address_type;
                    TCEP_id : TCEP_id_type ) : both_sides;
   begin ... end;

   (* property:
    with TC [find_TC_id (T_addr, TCEP_id)].
    EP [this_side (T_addr, TCEP_id)] holds
    address = T_addr
    and  id = TCEP_id
    and  state <> closed
    or TC [find_TC_id (T_addr, TCEP_id)]. EP [side]. state =
    closed for both sides;
    i.e. find_TC_id returns the TC identified by TCEP_id
    at the AP at the address T_addr;
    this_side indicates whether this end of the connection is
    calling or called. *)
```

```
init begin
   close_in_progress := false;
   EX_in_transit := false;
   end;

 (* CONNECTION ESTABLISHMENT PHASE *)

 when AP[T_addr][TCEP_id]. T_CONNECT_req
    provided ... (* property: for all conn_id : TC_id_type,
                    all side : both_sides  holds
                    with TC[conn_id]. EP[side] do
                    state <> closed implies
                    address <> T_addr  or
                    id <> TCEP_id          *)
                  (* i.e. the TCEP identifier is not yet in use
                          at the same AP *)
              and from_T_address = T_addr
              and ... (* able to provide the service asked for *)
          var conn_id : TC_id_type;
          begin
             conn_id := ...; (* property:
                                for all side : both_sides holds ·
                                TC[conn_id].EP[side].state = closed *)
                               (* i.e. connection not yet in use *)
             initialize (conn_id);
             with TC [conn_id] do begin
                with EP [calling] do begin
                   id := TCEP_id;
                   address := T_addr;
                   QTS_estimate := proposed_QTS;
                   end;
                with EP [called] do begin
                   address := to_T_address;
                   end;
                options := proposed_options;
                connect_data := data;
                end;
             buffer[conn_id, calling].clear;
             buffer[conn_id, called].clear;
             end;

     provided ... (* not able to provide the service asked for *)
        begin
           AP[T_addr][TCEP_id]. T_DISCONNECT_ind
          (...(* property:
                 if not to_T_address in known_T_addresses
                 then U_UNKNOWN *),...(* dummy *) );
        end;
```

```
any conn_id : TC_id_type do    with TC[conn_id] ᵈ
provided  ( AP [EP[calling].address]
                [EP[calling].id].   state = open_in_progress)
   and    ( AP [EP[called].address]
                [EP[called].id].  state = closed)
   (* when the connection request reaches the called side *)
   begin  with EP[called] do begin
       id:=...;
       (* property: for all conn_id' <> conn_id,
                    all side : both_sides  holds
                    with TC[conn_id'].EP[side] holds
                    state <> closed implies
                    address <> TC[conn_id].EP[called].address
                    or  id <>  TC[conn_id].EP[called].id  *)
                   (* i.e. the identifier is not yet in use *)
      QTS_estimate := ...; (* not defined by this standard *)
      AP[address][id]. T_CONNECT_ind (address,
                                     EP[calling].address,
                                     options, QTS_estimate,
                                     connect_data);
   end end;


when AP[T_addr][TCEP_id]. T_CONNECT_resp
       with TC [find_conn_id(T_addr,TCEP_id)] do
   provided  ((data = undefined)  or
               (connect_data <> undefined))
         and (proposed_options in options)
      begin
         options := proposed_options;
         accept_data := data;
      end;
```

(* There is also the possiblity of disconnection, see termination phase *)

```
any conn_id : TC_id_type do    with TC[conn_id] do
provided  ( AP [EP[calling].address]
               [EP[calling].id].  state = open_in_progress )
      and ( AP [EP[called].address]
               [EP[called].id].  state = open )
      (* when the connect response reaches the calling side *)
   begin  with EP[calling] do begin
      QTS_estimate := ...; (* not defined by standard *)
      AP[address][id]. T_CONNECT_conf (options,
                                       QTS_estimate,
                                       accept_data);
      end end;
```

```
(* DATA TRANSFER *)

when AP[T_addr][TCEP_id]. T_DATA_req
   with TC[find_conn_id (T_addr, TCEP_id)].
           EP [this_side (T_addr, TCEP_id)] do
   provided  ... (* property:
                    flow control to the Transport entity
                    is ready *)
      begin
         buffer[find_conn_id (T_addr, TCEP_id),
                this_side (T_addr, TCEP_id)].
                    append (TS_user_data,
                            is_last_fragment_of_TSDU);
         end;

any conn_id : TC_id_type do
any side : both_sides do
      with TC[conn_id].EP [opposite(side)] do
when buffer[conn_id, side]. next_fragment
   provided  AP[address][id].state = open
         and ... (* property: flow control to user is ready *)
         and ... (* there is no expedited data in transit
                    (in the same direction of transfer)
                    which was sent prior
                    to the next normal data fragment *)
      begin
         AP[address][id]. T_DATA_ind
              (TSDU_fragment, is_last_fragment_of_TSDU);
         end;
```

```
(* EXPEDITED DATA TRANSFER *)

when AP[T_addr][TCEP_id]. T_EX_DATA_req
      with TC [find_conn_id (T_addr, TCEP_id)] do
      with EP [this_side (T_addr, TCEP_id)] do
   provided  expedited_data  in  options
      begin
         EX_data := TS_user_data;
         EX_in_transit := true;
         end;

any conn_id : TC_id_type do          with TC[conn_id] do
any side : both_sides do
provided  AP [EP[opposite(side)].address]
                [EP[opposite(side)].id]. state = open
      and EP[side]. EX_in_transit
      (* when the expedited data reaches the destination *)
   begin
      AP [EP [opposite(side)].address] [EP[opposite(side)].id].
            T_EX_DATA_ind (EP[side]. EX_data );
      end;

when AP[T_addr][TCEP_id]. T_EX_D_READY_resp
      with TC [find_conn_id (T_addr, TCEP_id)] do
   provided  expedited_data  in options
      begin
         EP [opposite (this_side (T_addr, TCEP_id))].
                    EX_in_transit := false;
         end;

any conn_id : TC_id_type  do
any side : both_sides do
       with TC[conn_id]. EP[side] do
provided  AP [address, id].state = open
      and EX_in_transit = false
      and AP [EP[opposite(side)].address]
                [EP[opposite(side)].id]. EX_D_send_state = EX_idle
      and AP [address][id]. EX_D_send_state = EX_transfer
      (* when the confirmation reaches the sender
            of the expedited data *)
   begin
      AP [address][id]. T_EX_D_READY_conf;
      end;
```

```
(* TERMINATION PHASE *)

when AP[T_addr][TCEP_id]. T_DISCONNECT_req
      with TC[find_conn_id (T_addr, TCEP_id)] do
      with EP [find_side (T_addr, TCEP_id)] do
      begin
         close_in_progress := true;
         TS_reason := TS_user_initiated_termination;
         user_reason := ... (* property:
                              either equal to "TS_user_reason"
                                or undefined, i.e. the transmission
                                of this additional  information  is
                                not  guaranteed  by the TS_provider
                                *);
         end;

any conn_id : TC_id_type do      with TC[conn_id] do
any side : both_sides do         with EP[side] do

  provided AP [address][id]. state in [open_in_progress, open]
        and ... (* when internal problem of TS_provider *)
     begin
        close_in_progress := true;
        TS_reason := ...; (* property: not U_UNKNOWN *)
        AP[address][id].T_DISCONNECT_ind (TS_reason,...(*  dummy
*));
        end;

   provided  EP [opposite (side)]. close_in_progress
  (* when the disconnect reaches the other end of the connection
*)
      begin
         if AP [address][id]. state in [open_in_progress, open]
          then
AP[address][id].T_DISCONNECT_ind(TS_reason,user_reason);
         EP [opposite(side)].close_in_progress := false;
         end;
```

ANNEX 5

ISO

International Organization for Standardization

Organisation Internationale de Normalisation

ISO/TC97/SC16/WG1

Contribution to the meeting of the WG1 ad hoc group on FDT,
Nov. 1982

Title: Some enhancements to the syntax of the Subgoup B FDT

Source: Canada

Contribution to the meeting of the ad hoc group on FDT, Nov. 1982

1. Introduction

Canada would recommend that the following elements be proposed to be included in the Subgroup B FDT. These are minor elements which are either important to make the language complete, or add some feature that makes the use of the language more convenient.

2. Queuing option

The option of zero and infinite input queue should be specified for each specification of a module type. The following syntax could be used:

```
  <input queue option> : : = INPUT <yes or no>  QUEUED ;
  <yes or no>  : : =  empty
                    |  NOT
```

This should be the first part in a <module body>.

3. More flexible embedding structure for transitions

The present syntax allows embedded transitions only in a certain way, where throughout a module specification all transitions must start with the same clause (e.g. the WHEN clause). In order to indicate the end of a certain embedding structure (e.g. the starting with WHEN clauses), and to allow in the subsequent transitions a different embedding structure, the construct explained below is proposed.

```
Example: when <input>
  provided <condition>
    from <present state>
      to <next state>
        begin ... end;
  provided <condition>
    ........ end when; provided <condition>
  from ......
```

```
    (* spontaneous transition, not requiring input *)
       ..........
```

The construct " END WHEN ; " is used to terminate the WHEN embedding structure. Similarly, END FROM, END PROVIDED, etc. constructs could be included in the language definition.

A possible syntax, equivalant to the present syntax of the working document, except for the introduction of the END WHEN etc. constructs, is as follows:

```
  <transition> : : =   <some  clause>
  <some  clause>  : : =  <ANY clause>
                       | <WITH clause>
                       | <WHEN clause>
                       | <FROM clause>
                       | <To clause>
                       | <PROVIDED clause>
                       | <PRIORITY clause>
                       |  <block>   **   ;

  <FROM clause>  : : =  FROM  <major present state>  <some clause>
                            <FROM suite>

  <FROM suite>  :  :  =
                     <FROM clause>
                   | END FROM
                   | empty
```

etc. for the other type of clauses

4. Arrays of interaction points

In the case that an entity services several service access points, for instance, it is important to be able to specify a number of access points refering to the same type of channel. For this purpose, the use of "interaction point arrays" in the <interaction points> declaration of a module type definition is proposed. Examples have appeared in several previous contributions on Transport service and protocol specifications.

The precise syntax could be as follows: <interaction point declaration> : : = <interaction point id>

```
   :   <interaction  point  type>  <interaction  point type> : : =
<channel type id> ( <role id> )
   |  ARRAY ( <index type> )  OF  <channel type id> ( <role id> )
```

5. Separation of module type definitions and internal definitions

5.1. It is foreseen that the <internal definition> of a module may either be given in the form of an extended state transition machine (i.e. <module body>) or in the form of a <substructure definition> which declares sub-module instances and their interconnections. In order to make such alternate specifications more easily replaced by one another, within an overall system specification, it is proposed to separate somehow the <module type definition> from the internal definition.

The following syntax could be used: <system> : : = SYSTEM <system id> ; <specification part>* <specification part> : : = <channel type definition>

```
                              | <module type definition>
                              | < extended state transition machine>
                              |   <substructure definition> <module
type definition> : : = MODULE <module type id>
                                  ( <interaction points> ) ;
<extended state transtion machine> : : =
    ESTM <module type id> ; <module body>
```

5.2. Instead of the reserved words MODULE and ESTM, the words BLOCK and PROCESS could possibly be used.

6. End indications

For the overall structure of specifications, it seems to be useful to indicate clearly the end of a specification part. For this purpose the construct

END <module type id> ; or

END <channel type id> ; could be used, to be placed at the end of a module or channel type definition.

7. Including informal elements in a specification

In many cases, certain properties of a specified module are not defined in a formal way, but as an informal specification element in (semi-) natural language. The notation " (/ <informal text> /) " is included for this purpose in the Subgroup B language.

It is proposed to allow this construct to be used instead of identifiers anywhere in a specification. To simplify the syntax analysis, it should not be placed where comments may be placed (as now defined in the working document).

This approach allows a more flexible use of this construct, as shown in the example below.

Example: (a) graphical specification with informal text
(b) equivalent linear specification

<u>A N N E X   6</u>

Contribution to the ISO TC97/SC16/WG1 ad hoc group on FDT meeting in Paris, Febr. 1983.

Title: Semantics of spontaneous transitions

Source: G.v. Bochmann

The semantics of spontaneous transitions has been refined during the last ad hoc group meeting. This contribution gives some examples of applications for spontaneous transitions in the Annex, and proposes the following revision to the semantics of these transitions.

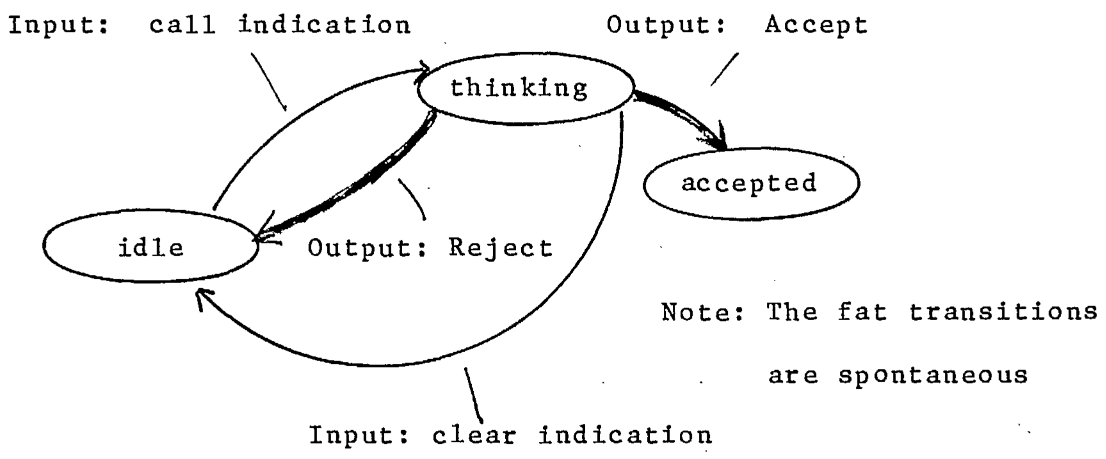
Proposal:
(a) To distinguish two kinds of spontaneous transitions: "facultative" and "required" ones, as explained in Annex 2 of the companion contribution "Comments on ...".
(b) To leave for further study the association of performance attributes to "required" spontaneous transitions. This study should address the question of time-out transitions, and probabilistic performance considerations for the execution of these transitions related to protocol and service performance characteristics.

Rational for proposal:
(1) The semantics of "immediate" execution (as defined now, DELAY(0,0)) is not well defined (consider for example the Example 3 in the Annex).
(2) The proposed semantics of "required" transitions is similar to "immediate execution".
(3) The semantics of "facultative" transitions is the same as now defined for DELAY(0,*).
(4) The use of DELAY(d1, d2) transitions for the definition of time-out transitions is not as straightforward as may seem from the example in section 3.6 of the working document. In fact, that example is in contradition with the semantics of the ESTM considering a transition as an undivisible operation (see also comments in the companion contributeion).

Annex: Examples for the use of non-deterministic extended finite state machines.

The following examples include situations where more than one transitions are possible from a given state of process, and the specification does not define which one will be executed. However, it is assumed that any implementation of the specified process must make a choice in some way or another.

<u>Example 1:</u> A user process may accept or reject an incoming call, or while "thinking" the call may be cleared by the other party:



<u>Example 2:</u> Considering the "mapping" process of a Transport entity which handles the mapping of Transport connections onto Network connections, any Network connection not in use, at any given time, may or may not be disconnected. This is most naturally modeled by a spontaneous transition which is enabled when a Network connection is idle. This transition may, or may not be executed (this decision is up to an implementation). This transition applies to all major states of the mapping process.

<u>Example 3:</u> Considering again the mapping process and assuming that it looks after concatenating the PDU's to be sent for a given Transport connection into the Network service data units (NSDU), as modeled in FDT 78. If there is a PDU buffer for each type of PDU to be sent and one NSDU buffer then the following spontaneous transitions can be identified:

(a) Sending the NSDU, enabled when the NSDU buffer contains at least one PDU.

(b) For each type of PDU:
Appending the PDU in the corresponding PDU-buffer into the NSDU buffer, enabled when the PDU buffer is not empty and the

NSDU buffer has enough free space.

Depending on the overall state of the mapping process, all these transitions may be enabled at the same time. The specification does not indicate which one to choose, since this is an implementation issue.

<u>A N N E X 7 a</u>

International Telegraph and Telephone
Consultative Committee
(CCITT)

Original: English

Period 1981-1984

Question : 39/VII

Date: November 1982

STUDY GROUP VII - CONTRIBUTION No.

This contribution is for input to the Special Rapporteurs meeting on Q 39/VII held in Geneva, November 1982.

Title: Proposal for contents for section 3 (Semantic Model) of the Draft Recommendation

Source: Canada

1. Introduction

The semantic model is defined in three parts:

(a) the part relating to the system structure defined in terms of functional blocks, subblocks and channels (see sections 2 and 3 below);

(b) the part defining the model of an extended state transition machine (see sections 4 and 5 below), and

(c) the part defining the handling of data structures as used by the extended state machine.

2. Blocks, channels, and interaction points

2.1. The concepts

A system is defined by a set of interacting 'blocks' and the structure by which they are interconnected.

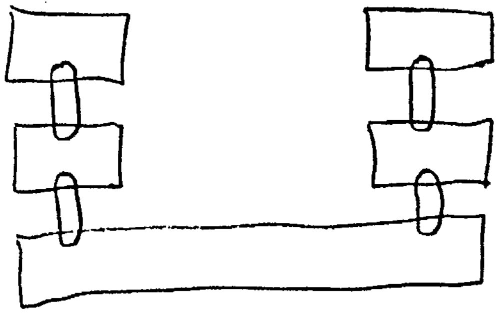
Blocks share channels with each other and with blocks in the system's environment. The channels embody the interactions between the blocks, and between the blocks in the system and those in the system's environment. The blocks embody the actions exclusively allocated to blocks.

The configuration of channels and blocks represent the system's structure. An example is shown in the figure belows.

notation:

channel

module



Blocks bear different responsibilities in the performance of interactions. For example, if in an interaction a value is passed, then one module is responsible for providing that value, and the other block is responsible for accepting the value.

The allow for modelling of these different responsibilities, we introduce the concept of 'interaction point'.

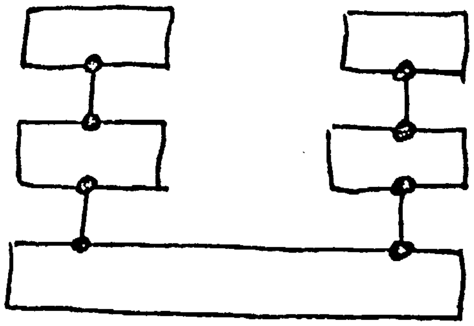
An 'interaction point' is a view of a channel as seen from one of the blocks that is connected to the channel.

Using an alternative graphical notation, the above example can be represented as follows:

notation:

channel with two interaction points

module



The concepts of 'channel' and 'interaction point' are useful for the description of the OSI architecture. They are related to the notion of 'abstract interface' in the following sense: the interactions of a blocks with other blocks or with the environment of the system occur through channels between the blocks. In a real system, such a channel is realized by an '(real) interface'. For the specification of communication protocols and services we are not concerned with the specification of real interfaces, but only with the abstract properties that any such interface for a given block-to-block interconnection must satisfy. These properties are called the 'abstract interface' between the two blocks.

The concepts serve for:

a) the partitioning of the interactions of a given block into separate groups concerning different blocks forming the module's environment. A block has contact with its environment only through a well-defined set of 'channels'.

b) the specification of the interconnections between the different blocks within a system (or the sub-blocks within a block). A channel connecting two blocks could be specified by naming an interaction point of one block and an interaction point of the other block with which the former is to be connected.

For example, typical channels of a layer entity executing the layer protocol are:

a) the access point(s) to the layer above through which the service is provided,

b) the access point(s) to the layer below through which the underlying service is accessed,

c) an (abstract) interface to the local system management block, and possibly a local channel through which local services such as buffer management, time-outs, etc. can be obtained.

2.2 The specification of a channel

The purpose of a channel type definition is to be used in the specification of a block (see section 2.3.), where each interaction point of a block is characterized by the type of channel which it represents.

In order to distinguish between the two blocks that use the channel for their interactions the concept of a "role" is introduced. For each type of channel two roles are defined. These two roles are 'played' by the respective blocks instances that are connected to an instance of a channel. It is then possible to define the possible interactions through a channel without explicitly defining the blocks that interact through the channel. However, it is necessary to refer to the roles that the blocks play in this interaction.

The specification of channel type includes:

a) an enumeration of the possible types of interaction primitives that may be invoked through a channel of that type.

b) the names of two 'roles' which distinguish the two sides of the channel, and hence the two connected blocks (e.g. 'service provider' and 'service user').

c) the properties of the interaction primitives. The invokation of an interaction primitive consists of the exchange of an interaction, called 'signal', form the 'outputting' block to the 'inputting' block. A signal may include parameters of various data types. The values of these parameters are determined by the outputting block.

d) possibly certain rules about the order in which the interaction primitives may be executed over a given channel of that type.

2.3 The specification of a block

The purpose of a block specification is to define the behavior of the block as <u>observable at the interaction points</u> to which it is connected. Therefore a block specification cannot be given without a definition of the interaction points through which the module interacts with its environment. The set of these interaction points is called the "boundary" of the block.

The specification of a block may be given in each of the following forms:

(a) by a fixed substructure definition (see section 3), where each subblock in the substructure can be defined either according to (a) or to (b).

(b) by defining the behavior of the block modeled as one or several instances of 'processes'. The behavior of a process is defined in terms of an extended state machine, as explained in section 4.

(c) Other techniques for defining the behavior of a block are for further study.

Process instances may exist from the beginning of the system or they may be created during the life of the system. Process instances may cease to exist during the life of the system.

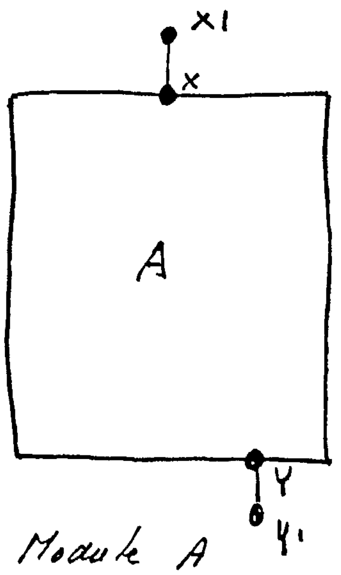
The environment of a process instance is all the other process instances within the block and all the channels connected to the block. Process instances interact with their environment solely by means of interaction primitives.

A process instance is created by requesting that an initialization action be performed. This initialization action can be requested initially when the system is created or dynamically during the life of the system.

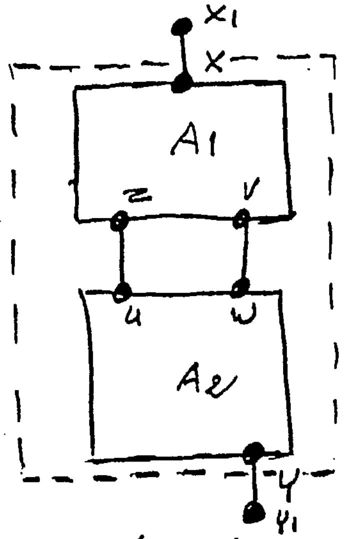
The initialization action initializes the process instance variables, sets an initial major state and initializes the input queue of that process instance.

3. Substructure definitions

A specification of a block may be given in the form of a substructure definition, as shown in the figure below. If the behavior of each of the subblocks is defined, such a substructure defines the behavior of the block.



Module A

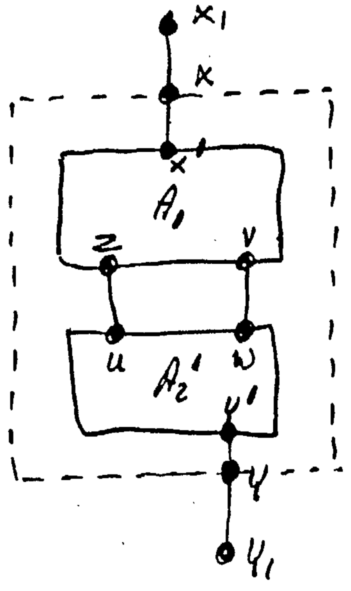


Modules A1 and A2 representing the substructure of A.

In the example above, the block A interacts with other blocks in the system through the channels X-X1 and Y-Y1. The substructure of block A consists of two sub-blocks A1 and A2. The connections Z-U and V-W are called <u>internal channels</u> and connect interaction points by which the block A1 and A2 interact. The notation of the example also means that the interactions of A at X and Y are realized by the interactions of A1 at X, and A2 at Y, respectively.

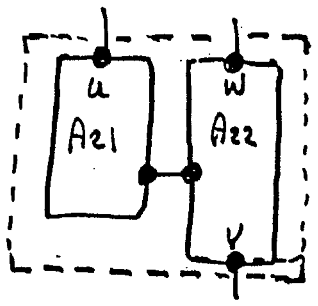
The above structuring has assumed that the interaction points X and Y of A and X and Y of A1 and A2 remained unaltered, i.e. only the functionality of A was represented by two sub-blocks A1 and A2 connected by internal channels.

One could also consider a substructuring for the interaction points X and Y, and represent this by an alternative way of picturing

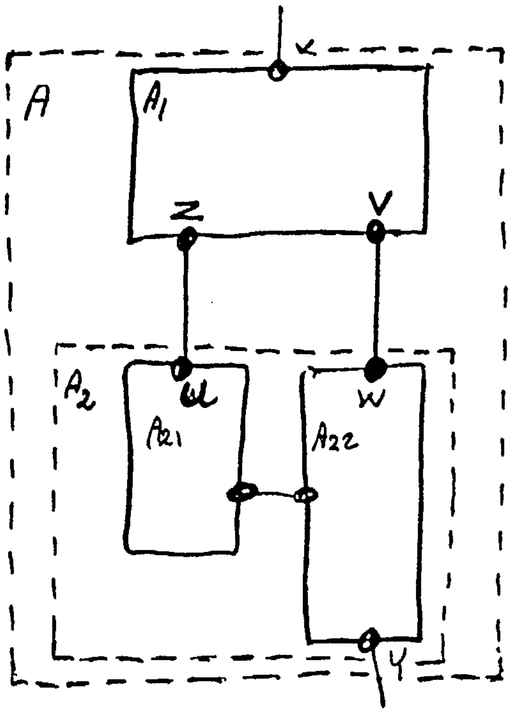


We leave this possibility for further study.

It is possible to further subdivide the structure of a block. For example a possible substructure of block A2 would be as follows:



Sometimes several steps of refinement are shown in a single diagram. For example, the figure below shows the two steps of refinement for block A given above:



4. The extended state transition model

The extended state transition model assumes a model of interaction where each interaction of the specified process 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 process (inputs), and interactions initiated by the process (outputs). The reception of an interaction from the environment leads to a transition of the specified process which may give rise to other (output) interactions.

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

The possible order of interactions of a process is given in terms of

(a) the state space of the process which defines all (internal) states in which the process 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 process. A transition may also involve one or more interactions of the process 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 languages.

The state space of the process 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 process.

The possible transitions of the process 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, called ′enabling predicate′, depending on some of the variables defining the process state, and (possibly) the specification of an input. A transition may occur in a given state only if the enabling predicate has the value <u>true</u>, and the interaction in question (if it exists) is initiated by the environment. A transition without input is called a spontaneous transition. It can be executed, independenly of input, whenever the enabling predicate is satisfied.

(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 process could choose any of these possibilities. In many cases, the specificaton of a process may be deterministic, in the sense that (at most) one transition is specified in any reachable state and given input.

An input interaction to the process is either considered immediatly by the state machine or first put into the (conceptualy infinite) input queue of the module (depending on the queuing option used); if it is put into the input queue it is considered by the machine when it becomes the first in the queue.

The model allows inputs to be "saved". For each major state of the machine a set of input interaction types may be declared to be saved. This means that inputs of these types should remain in the input queue and not be considered by the state machine when the machine is in the given major state. The first input in the queue not corresponding to any of the saved interaction types should be considered as input to the state machine.

In addition to input-output interactions, the model provides for continuous output functions. While interactions represent "events" and are generated during state transitions, continuous output functions provide steady output from one process through a channel to another process. The "receiving" process may use the value of such a function (provided by its neighbour process) within an enabling predicate, that is, it may influence which transitions are enable.

The name and type of output functions are declared in a channel definition. The value provided by the function is determined by the function body which is defined within the process body which plays the role of the outputting process.

5. Formal semantics of the extended state transition model

(to be provided, based on the "Common Semantic Model..." (Melbourne meeting)).

ANNEX 7b

International Telegraph and Telephone
Consultative Committee
(CCITT)

Original: English

Period 1981-1984

Question : 39/VII

Date: November 1982

STUDY GROUP VII - CONTRIBUTION No.

This contribution is for input to the Special Rapporteurs meeting on Q 39/VII held in Geneva, November 1982.

Title: Proposal for contents for section 4 (Language for describing system structure) of the Draft Recommendation

Source: Canada

1. Introduction

(to be provided)

2. Graphical language

The concepts of 'block', 'channel' and 'process' as defined in section 3 of the Recommendation correspond to the concepts of 'functional block', 'channel' and 'process' as defined in SDL (Draft Recommendations Z101 and Z102). Therefore the graphical representations, as defined in Z101 and Z102, can be used for these concepts. Substructure diagrams can also be drawn as shown by the examples of section 3 of the Recommendation.

It is noted that Recommendations Z101 and Z102 do not define block and channel types, only instances of blocks and channels are considered.

3. Program-like language

Note: The options of multiple processes per block and dynamic process creation are not supported by the language defined below (they are left for further study). A single process per block may be defined using the program-like language for extended state machines (see section 7 of Recommendation).

3.1. Syntax overview

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

3.1.1. Overall structure of a specification

(to be provided; includes channel types, block type, substructure, and extended state machine definitions, as well as possibly block instance declarations).

3.1.2. Channels and interaction primitives

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

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

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

```
<channel> ::= CHANNEL <channel type id>
        ( <role list> ) <exchanges> ;
<role list> ::= <role id>
            |  <role list> , <role id>
<exchange> = <BY clause>
            |  <exchanges> <BY clause>
<BY clause> ::= BY <role list> : <exchange list>
<exchange list> ::= <exchange>
            |  <exchange list> <exchange>
<exchange> = <interaction id> <interaction parameters> ;
            |  <function heading>**
```

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: Identifiers may include both upper and lower case letters as well as the u nderscore character ("_"), which is considered to be a letter, and numerals.

3.1.3. Blocks and their interaction points

The definition of a block type contains the declaration of all abstract interaction points through which a block 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.

```
<block type definition> ::= BLOCK
                           <block type id>
                           ( <interaction points> ) ;
<interaction points> : := <interaction point declaration>
                     |  <interaction points> ; <interaction point
                                                declaration>
<interaction point declaration> ::= <interaction point id> :
                    <interaction point type>
                                 ( <role id> )
<interaction point type> = <channel type id>|ARRAY [<index type>]
                                     OF <interaction point type>
                                     (* Note 9 *)
```

The <role id> indicate 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.

3.1.4. Substructure definitions

```
<substructure definition> ::= REFINEMENT FOR <block type id>;
                          <list of sub-blocks>
                           INTERNAL CONNECTION
                          <list of connections between sub-blocks>
                           EXTERNAL CONNECTION <list of connections
                            of sub-blocks
                           to interaction point(s) of refined block>
                           END REFINEMENT;

<list of sub-blocks> : := <sub-block declaration>
                       (, <sub-block declaration>)*

<sub-block declaration> ::= <sub-block id> : <block type id>

<list of connections between sub-blocks> : :=
     (<sub-block id> . <interaction point id> =
      <sub-block id> . <interaction point id>;) +   (*Note 10*)
```

Note 10 : The two sides identify the connected interaction points (which should be of the same type).

```
<list of connections of sub-blocks to interaction point(s) of refined
         block> : :=

       (<block type id> . <interaction point id> =

             <sub-block id> . <interaction point id>) + (*Note 10*)
<process definition> : := PROCESS <process type id>
             <input interaction mechanism definition>
             <process body>
```

ANNEX 7c

International Telegraph and Telephone
Consultative Committee
(CCITT)

Period 1981-1984

Original: English

Question : 39/VII

Date: November 1982

STUDY GROUP VII - CONTRIBUTION No.

This contribution is for input to the Special Rapporteurs meeting on Q 39/VII held in Geneva, November 1982.

Title: Proposal for contents for section 7
(Language for describing dynamic behavior based on Pascal)
of the Draft Recommendation

Source: Canada

1. Introduction

(to be provided)

2. Language elements

2.1. State variables

The state space of the process 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 process.

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_CONNECT_resp,data_transfer);
  local_reference : TP_reference_type;
  remote_reference : TP_reference_type;
  TPDU_size :max_TPDU_size_type;
  QOTS_estimate : quality_of_TS_type;
```

2.2. State transitions

The possible transitions of the process 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 (INPUT 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 possible next major states (TO clause) -
the operation (BEGIN statement of the <operation>) including the generation of output.

A spontaneous transition may include a delay clause with two parameters, d_1 and d_2. The transition may not occur until the enabling condition has remained true continuously for d_1 time. It must be considered immediately if the enabling condition remains true continuously for d_2 time. If the delay clause is absent, a delay of $d_1 = 0$, d_2 = infinity is assumed. (This is written "delay (0,*)".) It means that the transition may occur at any time the enabling condition is true, possibly never.
A delay(0,0) has the semantic meaning of the immediate spontaneous transition of the basic semantic module.

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

```
trans
from idle
  input 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 := ...;
      output N.CR(0,local_reference,class_0,normal,variable_part_to_send);
    end;
from data_transfer to same
  input TSAP.T_DATA_req
    provided ... (* flow control from user ready *)
```

```
        begin
          output out_buffer.append(user_data);
        end;
    input out_buffer.fragment_ready(TPDU_size)
    provided ... (* Network layer flow control ready *)
      begin
        output N.DT(out_buffer.get_fragment(TPDU_size));
      end;

trans
provided no_tc_uses_ne and ne_locally_open
  begin
    output N.DISCONNECT_req    (*close any unused network connection *)
  end;

trans
from data_transfer to same
  provided credit_to_be_sent  delay (0,evaluate_delay_max_agreed)
   begin
       output N.ACK(credit,tpdu_nr)   (* send credit if any *)
   end;
```

2.3. Embedding of transitions

The syntax for transitions permits the different clauses (FROM, INPUT or DELAY, PROVIDED, PRIORITY, and TO) to be written in arbitrary order, followed by the <block> which includes at least BEGIN END. The order has no influence on the meaning of the construct.

The syntax also permits the embedding of the different clauses. This embedding structure is simply a shorthand notation with the following rules: The "scope" of a clause is defined to be the specification text corresponding to "<transition>+" in the syntactic rule of the clause (see section 3). The meaning of the clause extends over its entire scope. Each BEGIN END statement of a block within the specification text identifies a transition. All clauses in the scope of which a given transition falls apply to this transition. For example

```
trans
  input AP.I
    from A           provided  E  to  B
      begin X end;
      provided F to C
      begin Y end;
    from B to C
      begin Z end;
trans
  from C to D begin U end;
```

is a short hand notation for

```
trans
  input AD.I  from A provided E to B begin X end;
trans
  input AD.I  from A provided F to C begin Y end;
trans
  input AD.I  from B to C begin Z end;
trans
  from C to D begin U end;
```

It is noted that the following scope rules must be followed:

(a) The parameters of the input interaction (declared in the corresponding channel type definition) become accessible within the scope of the INPUT clause.

(b) As in Pascal, the WITH clause makes the fields of a record variable directly accessible within the scope of the clause.

(c) The ANY clause introduces a "variable" identifier with an arbitrary value within the range defined by the type identifier. The meaning is that the embedded transitions are defined for each of the possible values of this variable.

2.4. Predefined language elements

Some predefined language elements are provided. These include types, procedures, functions and blocks. The predefined identifiers may be redefined by the user of the FDT. In this case, the user's element is the one used.

Details are for further study.

3. Syntax overview for extended state transition machine model

This section defines the syntax for extended state transitions specifications, excluding the part that deals with data structure definitions and manipulation. The latter part is specified in Pascal, as explained in section 4 below.

The same notation is used as in section 4 of the Recommendation.

```
<process definition> ::= PROCESS FOR <block type id>;
                         <input interaction mechanism definition>
                         <process body>
```

```
<input interaction mechanism definition> ::= RECEPTION <reception
                                                         mode>
<reception mode>
 ASYNCHRONOUS|SYNCHRONOUS

<process body> ::= <label definitions>**
                  <constant definitions>**
                  <type definitions>**
                  <variable declarations>**
                  <major state declaration>
                  <state set definition>*
                  <proc func or init etc.>*
                  <embedded transitions>+


<embedded transitions> ::= TRANS <transition>+
<major state declaration> ::= STATE : <enumeration type> ;
<state set definition>  ::=   <state set id> = <set definition>** ;
                                                      (*Note 4*)
<proc func or init etc.> ::= <procedure definition>** (* Note 2 *)
                           | <function definition>   (* Note 2 and 3 *)
                           | <continuous output definition>
                           | <initialization> (* it is suggested that
                                                 the initialization be
                                                 placed at the beginning *)
<continuous output definition> ::= FUNCTION <interaction point ref>.
                                   <function name> ; <block>
                                   (* the parameters of the function
                                    are already declared in the channel
                                    definition *)
<interaction point ref> ::= <interaction point id>
                          | <interaction point id> [<index variable>]
<index variable> : :=<identifier>
<function name> ::= <identifier>
<initialization>    ::=   INITIALISE BEGIN
                           STATE TO <major state value>
                           <additional init>;
<additional init>   : :=  END
                               |; <statement sequence>** END


<transition> ::=
    | ANY <identifier> : <type identifier** DO <transition>+ (*Note 5a*)
    | WITH <variable>** DO <transition>+ (*Note 5b*)
    | INPUT <interaction point ref> . <interaction id> <transition>+
                                                     (*Note 5c*)
    | DELAY(<delay value>,<delay value>)<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>** ;
    | SAVE <interaction point ref> . <interaction id> <transition>+
<delay value> ::= <expression> | *          (* Note 10 *)
<priority indication> ::= <identifier>**  (*constant of some
                                            enumeration type*)
                          | <integer>**
<major present state> ::= <major state value list>
                          | <state set id>
<major state value list> ::= <major state value >
                          | <major state value list>,<major state value>
<major next state>    ::= <major state value list>
                          | SAME
<major state value>   ::= <identifier>**   (*must be element of the
                                 enumeration type of the <major
                                 state declaration>*)


<output statement> : := OUTPUT <interaction point ref> . <interaction id>
                         <effective parameter list>**  (*Note8*)
<nextstate statement> ::= NEXTSTATE <major next state>
                          | NEXTSTATE SAME
```

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 "...". Other possible uses of ... are for further study.
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 nor DELAY 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.
Note 9 : The usual multi-dimensional array notation, e.g. ARRAY [index1,index2], is also allowed.
Note 10: The delay value must be either an integer valued expression or '*', which represents infinity.

Other Syntax elements

(a) Informal specification elements, which define system properties that are part of the specification (not merely comments), are written as text enclosed in "(/" and "/)" and may be placed wherever comments or ... may be placed. It may also replace a procedure call statement or a <interaction id>.

(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. Elements of Pascal used

4.1. General

The elements of the Pascal programming language are used for the following parts of the specifications:

(a) For channel definitions:
- defining the parameters of interactions and their data type;

(b) For process definitions:
- defining the variables of a process and their data types;
- defining the enabling predicates and operations of the transitions using Pascal expressions and statements. This includes the use of Pascal functions and procedures.
- defining continuous output using Pascal function definitions.

Note that two additional kinds of statements are added to those provided by Pascal, namely the <output statement> and the <nextstate statement>.

4.2. Removal of certain restrictions

Functions are permitted to return arbitrary values.

4.3. Elements of Pascal not used

To date, we have not found the following features of Pascal to be necessary: pointers, and files (and go to and labels).

ANNEX 8

ISO
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

ISO/TC 97/SC16/WG1
OPEN SYSTEM INTERCONNECTION

Contribution to the meeting of the WG1 ad hoc group on FDT, Nov. 1982

Title: Comparison of FDT proposals from ISO (subgroup B) and CCITT

Source: Canada

1. Introduction

During the last Subgroup B meeting in Enschede (April 1982), there was not enough time to discuss the FDT proposal from CCITT Rapporteurs group on Q39/VII (CCITT liaison report ..., TWENTE-4) fully. The present paper points out the main differences in the syntax and semantics of the CCITT proposal with the Subgroup B working document. The purpose of the paper is to simplify the discussion at the next meeting towards a resolution of the differences.

It is to be noted that the formal semantics of the two FDT proposals from Subgroup B and CCITT are not yet finalized. Nevertheless, the following differences in the syntax and semantics may be identified.

2. List of differences

2.1. Choice of reserved identifies

2.1.1. The CCITT syntax uses PROCESS instead of MODULE.

2.1.2. The CCITT syntax uses SIGNAL instead of INTERACTION.

2.1.3. The CCITT syntax uses INPUT instead of WHEN.

2.1.4. The CCITT syntax uses the additional reserved identifier OUTPUT, which is omitted in the Subgroup B syntax.

2.1.5. The CCITT syntax uses ENDPROCESS <process identifier> instead of END at the end of a module specification.

2.2. In the CCITT syntax, the <major present state> in a FROM clause may be a list of states, thus allowing the introduction of state sets without declaring them in a <state set definition>.

2.3. The CCITT syntax provides the <reception mode> clause for specifying whether a module has a zero or infinite input queue. Such a specification is missing in the Subgroup B syntax.

2.4. In the CCITT syntax, several possible next states are foreseen for a single group of transitions (such a group is simply called a "transition" in the CCITT document), while the Subgroup B syntax only foresees a single next major state per transition. This difference was discussed at the last meeting in Enschede (see minutes for April 15, last paragraph).

2.5. The CCITT proposal includes the SAVE construct. This construct is not foreseen in the Subgroup B proposal. Subgroup B considers that there is no need for a SAVE construct for OSI specifications (see minutes of last meeting).

2.6. Appendix 1 of the CCITT proposal outlines a method by which the interconnection structure between several submodules may be expressed. Such considerations have been outside the scope of Subgroup B, however, the separation of a module specification into a <block heading> and a <process definition> (see Appendix 1) is directly related to the Subgroup B syntax.

3. Conclusions

It is Canada's opinion that the following points be seriously considered by the FDT ad hoc group to resolve the differences between the ISO and CCITT FDT proposals.

(a) Concerning point 2.1.1: The separation of heading and body definitions (see point 2.6) seems useful, and the proposal in a companion paper (see section 5 of "Some enhancements to the syntax of the Subgroup B FDT") should be adopted, using the reserved identifiers BLOCK and PROCESS.

(b) Concerning point 2.4: The CCITT proposal is based on upward compatibility with SDL, which is considered to be important in CCITT. Multiple next states should be included.

(c) Concerning point 2.5: The SAVE construct seems not to be required for the specification of OSI protocols and services. It may, however, be considered as an option of the FDT when the FDT is used for applications outside the scope of OSI.

ANNEX 9

ISO/TC97/SC16

ISO/TC97/SC16/WG1 N 58

FEBRUARY 1983

ISO
International Organization for Standardization
TC 97 - Computers and Information Processing
SC16 - Open Systems Interconnection

Source: Canada

Title : Towards a common FDT for ISO and CCITT

Discussions for harmonizing the development of FDT's for OSI specifications between ISO and CCITT have been going on for a while. The following approach is proposed as a possible position for the ISO working group on this topic:

Given that remaining differences between the ISO Subgroup B FDT and the FDT developed under Q39/VII in CCITT are summerized in the CCITT liason report (Comparison of SDL and Subgroup B FDT language) prepared at the Catania meeting, and further understanding on these differences has resulted from the discussions at the last Q39/VII Rapporteurs meeting (Geneva Dec. 1982), the following approach is proposed:

(a) The semantic model of the Subgroup B FDT will be included in the semantic model of the Q39/VII FDT in a form as outlined at the Geneva meeting.

(b) A linear syntax based on the Pascal programming language will be included as a possible syntax of the Q39/VII FDT. This linear form should be compatible with the syntax of the Subgroup B FDT.

(c) The Subgroup B FDT will be changed in order to accomodate the following:
(1) An option for multiple next states for a single transition.
(2) A SAVE option (probably not required for OSI specifications).
(3) Multiple processes per module and dynamic process creation are considered for further study.
(4) The reserved identifiers INPUT and OUTPUT will be used in the linear syntax, instead of WHEN and OUT respectively.
(5) The term "module" will be replaced by "block" without implying any change in the semantics.
(6) The reserved identifier PROCESS will be used in the linear syntax to introduce a module body.

ANNEX 10

ISO/TC97/SC16/N_____

ISO/TC97/SC17/WG1 N 59

FEBRUARY 1983

ISO
International Organization for Standardization
TC 97 - Computers and Information Processing
SC16 - Open Systems Interconnection

Source: Canada

Title: Proposal to Produce an FDT Standard

INTRODUCTION

Canada believes that the work on FDT should be aimed at producing an International Standard on formal description techniques for OSI protocols and services. The reasons are the following:

1) The FDT document would be referenced by future protocol and service standards which will include formal specifications.

2) The standard status would give the FDT a stronger stability. This is desirable in view of the future investments in the form of consistency checkers, compilers, simulators and similar software, and also in view of the formal specifications written in the future.

3) CCITT's Rapperteurs group on Question 39/VII is working on a Draft Recommendation on an FDT, which hopefully will be as close as possible to the FDT developed by Subgroup B of the adhoc group on FDT.

4) Various formal description techniques are being used by different groups to add more precision to their protocol and service specifications in natural language. Therefore it is useful to have a standard FDT for ensuring a common understanding.

Canada therefore proposes that urgent consideration be given to producing such a standard.

ANNEX 11

3 mai 82

Title: Delegate's Report of the ISO TC97/SC16/WG1 meeting
on Formal Description Techniques (FDT) in Enschede, April 1982.

From: G.v. Bochmann

The meeting was held at the Twente University, April 13 through 16, with fourteen (14) participants. C. Vissers chaired the meeting with short notice from the chairman of the ad hoc group on FDT, who could not come. Most of the time was spent by discussions within the subgroups B and C in parallel. Some time was also spent with discussions in the plenary and Subgroup A.

Two delegates from CCITT made the liaison with the CCITT Rapporteurs group on Question VII/39 (FDT) and SG XI/WP 3-1 (SDL).

In Subgroup B, most time was spent on the discussion of the formal semantics (section 5 of the working document, which was not yet written). The discussion was stimulated by the "Common semantic model for CCITT and ISO" submitted by the CCITT Rapporteurs group on Question VII/39 and a personal contribution from G.v. Bochmann on a possibly simpler exposition of such a common model. The subgroup agreed that G.v. Bochmann should write an initial draft of section 5 of the working document based on the contributions and the discussion during the meeting.

There was not enough time to discuss the questions related to the CCITT proposal for revisions of the Pascal oriented syntax of the Subgroup B language. Among some minor points, one important issue was discussed: Should several different major final states be allowed for a single transition, as proposed by CCITT in order to make the syntax compatible with the existing SDL Recommendation ? No agreement could be obtained. Consultation from the member bodies is sought on this question.

The task of Subgroup A was extended according to its present activities. The subgroup met shortly to discuss some revisions of its working document.

Subgroup C met in parallel with Subgroup B. For a review of its progress we refer the reader to the minutes.

The next FDT meeting is foreseen for next October, in time to prepare a reply to the CCITT Rapporteurs group on Q VII/39 which meets in December. The different subgroups may meet in between.

We believe that Canada should continue its participation in the work on FDT's.

Title: Delegate's Report of the ISO TC97/SC16/WG1 meeting on Formal Description Techniques (FDT) in Catania, November 1982.

From: G.v. Bochmann

1. Introduction

The meeting was held in Catania, Sicily, 8 - 12 November, with 23 participants. Most time was spent with parallel discussions in Subgroups B and C. Some time was also spent with discussions in the plenary and Subgroup A.

2. Liaison with CCITT

A delagate from CCITT SWP XI/3-1 (SDL) attended the meeting for some days. He reported on recent meetings on SDL and mentioned that the liaison document from Subgroup B (prepared during the previous Subgroup B meeting in July) was not considered during the Rapporteurs meeting in Brazil in October (since the person carrying it arrived only for the second week of the Rapporteurs meeting. The document would be considered during the December meeting in Geneva.

Liaison documents to CCITT were written by the three subgroups and presented by G.v. Bochmann at the meetings of Q39/VII and SWP XI/3-1 in Geneva in December. A list of remaining differences between the FDT of Subgroup B and SDL was elaborated by Subgroup B and included in the liaison document.

3. Work in Subgroup A

Subgroup A revised its working document based on the proposals for a new section 2 submiited by G.v. Bochmann and C. Vissers. Also the annex was extended based on a contribution from experts from Sweden. The new version of the document is submitted as a N-document for the next WG1 meeting in February.

The new item of work on the possible interworking of specifications made in the respective FDT's of the Subgroups B and C was initiated. A possible approach to such interworking was identified. However, further work is required on this topic.

4. Work in Subgroup B

The work on Subgroup B centered around refinements of the extended state transition model, possible extensions for future study, and a detailed comparison with SDL and the "Common semantic model ..." developed in CCITT Q39/VII. An editing party revised the working document, including many additions in the informal explanations of the model and editorial improvements. The new version was reviewed in the Subgroup, and is submitted as a N_document for consideration at the next WG1 meeting in February.

5. Work of Subgroup C

The author was unable to attend the Subgroup C meetings. The reader is refered to the minutes of the meetings.

6. Next meeting and future work

The next meeting of the ad hoc group on FDT is held jointly with the next WG1 meeting in Paris, February 1983. The FDT issues will be discussed during the WG1 meeting. The following issues are of particular interest: FDT work item, the application of an FDT (possibility of setting up an editing group to develop a fromal description of Transport and Session services and protocols), whether there should eventually be a standard on FDT, the relation of formal descriptions with testing and conformance issues, etc. It seems also important to come to a definite proposal for a common FDT to be used by ISO and CCITT for the descritpion of OSI protocols and services.

Within Subgroup B an important issue to resolve is the zero-queue option, for which certain difficulties were identified during the Catania meeting. There was not enough time to resolve this issue in Catania. Another item is the leaboration of an example specification of the Transport protocol.

We believe that Canada should continue its participation in the work on FDT's.

Title: Delegate's Report of the CCITT Rapporteurs meeting
on Q39/VII (FDT) in Geneva, Dec. 1982.

From: G.v. Bochmann

The meeting was attended by only 7 delegates (see attachment). The author could not attend the meeting during the second and third days because of other commitments. There were relatively few contributions, including a relatively large number of contributions from Australia on Numerical Petri nets. No proposal for the text of the Draft Recommendation planned during the last meeting in Melbourne was submitted. The chairman of the group could not attend due to personal reason. The meeting was chaired by J. Park from Australia.

Work was done in continuation of the general direction determined during the last meeting in Melbourne. Most work was in relation with the liaison report from the SDL group in SG XI, which deals with extensions of basic SDL for incorporating features that were identified in Melbourne as requirements for a FDT for protocols. A two hour's joint meeting with SWP XI/3-1 was held on the fourth day of the meeting.

A proposal of text for the Draft Recommendation was presented by the author at the morning of the last day of the meeting. This includes text on several sections of the Recommendation, and it is largely based on the ISO Subgroup B working document and on certain sections of the new Draft Recommendations on SDL. There was not enough time to discuss this text in detail. It is submitted for consideration at the next meeting, which will be held in Geneva, June 1983.

Concerning the development of a common FDT for use by ISO and CCITT, the idea of a "common semantic model" was further persued. A basic semantic model was identified, with a certain number of options (such as SAVE, zero-queue, etc.) to cater for particular features of SDL or the Subgroup B FDT.

It is the author's opinion that relatively little progress was made during this meeting due to small attendance. Since the next meeting will be the last occasion within this study period for the elaboration of a Draft Recommendation, it seems important to prepare proposals for text of this Recommendation prior to the meeting. This would include the first sections of the Recommendation for which relevant sections of existing documents were identified during the Melbourne meeting.

Delegate's Report of the ISO TC97/SC16/WG1 Meeting in Paris, February 1983

by G.v. Bochmann

(the report below only covers the topic of formal description techniques (FDT))

The following lines outline the discussions and conclusions in the WG1 meeting concerning formal description techniques (FDT) for protocols and services. This topic was handled (together with conformance questions) by the ad hoc group E during the meeting. A review of the work on FDT performed so far by the FDT Rapporteurs group was given to the planary during the first day of the meeting.

The previous work on FDT was endorsed. The need for closer collaboration with other working groups, in particular WG6 and WG5 was pointed out. A detailed program of work until the next WG1 meeting in October was worked out, which respects the following recommendations which were passed by the WG1 plenary (the content of the Recommendations is summerized here):

Recommendation 19: A joint meeting between WG6 experts and experts on FDT will be held around mid 1983 in order to develop a trial specification of the Transport protocol in the Subgroup B language.

Recommendation 24: The evaluation criteria (N84, slightly revised from the Canadian contribution) is approved by WG1; comments are invited.

Recommendation 21: Liaison with CCITT SG VII and XI: A certain number of documents on FDT are transmitted.

Recommendation 17: The question of an FDT standard is an issue for consideration at the next WG1 meeting in October.

Recommendation 18: The FDT subgroup will meet around July 1983 in order to progress the technical work.

In addition, Subgroup B of the FDT group will have a meeting in May or June in order to prepare for the joint meeting with the WG6 experts. The elaboration of a trial Transport specification in the Subgroup B FDT is given priority over the finalisation of certain remaining issues in the FDT definition, some of which are related to the coordination with the FDT developments in CCITT.

The Canadian proposal of an FDT standard was discussed during the meeting. There seemed general agreement that either a Technical Report (type 2) or a Standard should be aimed at. In both cases the present tutorial documents must be complemented with a definition of the FDT suitable for the Technical Report / Standard document. Some consideration on this issue were assembled into the document N85.

The results of the discussions on conformance are included in the documents N87 ("Comments on whether or not product standards are needed"; the need for product standards was pointed out by the UK) and N88 ("Proposal for unambiguous drafting of protocols"). It is proposed to establish a separate subgroup on conformance; comments from the member bodies are requested on this question.

ANNEX 12

A PARSER FOR AN FDT LANGUAGE

George Gerber and Gregor v. Bochmann

March 1983

TABLE OF CONTENTS

1. Introduction

The parser outlined herein accepts source input pertaining to a language based on the Formal Description Technique (FDT) for the specification of communication protocols, known as "Subgroup B language", as developed by the standardization committee of ISO TC97/SC16/WG1. Originated within the framework of a compiler project for automated implementation, in Pascal, of FDT specifications, the parser performs lexical and syntactical analysis, as well as certain semantic checks particular to the features of FDT that transcend Pascal.

2. FDT model and language survey

The following discussion assumes that the reader is reasonably familiar with the proposals put forward by ISO in the references [1] and [2]. Some concepts, notably the refinement method and the separation of process declarations from module declarations, have been borrowed from the present CCITT proposal for this language [3]. Other influences reflected in syntax updates stem from implementation considerations and exposure to modern Pascal derivates like Modula-2, Portal and ADA.

An FDT system's architecture is defined by a collection of interacting modules and a set of channels that embody the interconnections through which the modules communicate. Modules interact with their environment, i.e. other modules they are connected to, by exchanging messages, called "signals" in FDT terminology, over their channels. Regarding the internal contents of a module, it can either be given a process, in which case it is atomic, or it can be refined into another set of modules that are again appropriately interconnected by channels. A process represents the actual computing activity of its module, in the form of an extended finite state machine, and can execute - at least conceptually - in parallel with other processes. In the case of a refinement, the one-to-one correspondance between the channels of the enclosing module and their local aliases must be established.

Accordingly, four basic language elements are provided for specifying and implementing a system: the type declarations for channels, modules, processes and refinements.

The channel (type) declaration defines and restricts the kinds of signals that can be exchanged between two modules it joins, in compliance with the ISO proposal.

A module (type) declaration only defines its interface, i.e. occurences of external channels together with their types. Thus we have a typical black box formalism, where all but the nature of the interactions with the environment is concealed. If such a module interface specification suits a variety of module instances, it only has to be declared once, whence the appropriateness of the designation "module type".

A process (type) always belongs to a given module (type). This relationship must be explicitly stated in the process header. Otherwise, the declaration of a process closely resembles the module body description suggested by ISO, very similar, for that matter, to CCITT's process definition.

The refinement (type) relies to a great extent on CCITT's refinement part. As for processes, the refinement type header must contain a reference to the module type it is further detailing. An instantiation pattern for sub-module internal structure is given by furnishing each submodule occurrence with a process or refinement type. (A more flexible - but not carried out - approach would allow the body of submodules to remain undefined, and require an explicit system instantiation part). Connections between submodules and assignment of local channels to channels of the enclosing module are defined in the manner put forward by CCITT. Another purpose refinements serve for is the nesting of specification parts, providing thus the means for hiding local information (e.g. local channel types) and for neater system structuring.

3. Parser overview

The syntactic description of the FDT language can be made in terms of a context-free grammar with the LL(1) property (except, of course, for the well-known Pascal flaw for embedded if-then-else statements; the usual precedence rule applies to resolve the conflict). A Backus-Naur Form description of the language syntax appears in appendix B.

Since our grammar is essentially LL(1), a standard parsing-table representation can be used, upon which a language independent driver routine will operate. The parsing table is produced automatically from the BNF description.

Although the lexical and syntactic analysis could, in principle, be performed in two separate passes, this method was avoided in order to prevent the need of intermediate files. Therefore, the lexical scanner is embodied by a procedure which is called upon by the parser when an input symbol is required.

Relying solely on syntax information, the parser takes as input the source file, and produces as output a parse tree, generated top-down from the root, and possibly error messages. The resulting parse tree is, in general, a sub-tree of a tree that represents a syntactically correct FDT program: some "branches" may have been truncated because of syntactic errors. Each node is associated to a symbol of the language and possibly to a set of attributes. These attributes are evaluated in a subsequent pass over the tree, when semantic checks are handled.

Three kinds of error messages can be distinguished: lexical, syntactic, and semantic errors. A lexical error indicates that an illegal character or a malformed input symbol was encountered. A syntactic error message is issued when an input symbol is ignored because it couldn't, in spite of a certain effort, be placed in the current tree context, or when the parser assumes that some preliminary input is missing before accepting the given symbol. In the latter situation, the inserted string is displayed. The set of expected symbols at the point where the error occured is given in both cases. (Actually, it is omitted if it hasn't changed since the previous error message). All other error conditions tested - incidentally those that least lend themselves to systematic processing - produce error messages in the semantic verification phase. However, many possible semantic errors are not tested for; they would usually be found when the resulting Pascal program is compiled by a Pascal compiler.

The FDT language parser has been implemented on VAX/VMS and CYBER/NOS BE computer systems and should be easily transferable to other computing environments supporting Pascal.

4. Modifications in relation to ISO's FDT

The following list gives an informal overview of the main deviations and enhancements applying to the current ISO FDT standard proposal.

- There's no special construct for overall system structure specification. System isn't a reserved word; use module instead.

- Channel, Module, Process and Refinement declarations are terminated with "end <name>" clause in the style of the following example:

```
channel timer_interface(...);

    ...

end timer_interface;
```

- A module declaration only defines its external interface:

```
module p_module;
   U:  user_access_point (provider);
   N:  network_access_point (user);
   T:  timer_access_point (user);
end p_module;
```

- Processes and Refinements always refer to a module:

```
process alternating_bit_protocol for protocol_module; ...
refinement system_refinement for system; ...
```

- Process and Refinement declarations can be "empty", meaning that the actual declarations are specified in a separate compilation unit (program): refinement R for M; end R;
- Processes and refinements can have formal parameters to convey initial values to process state variables:

```
process terminal (hw_addr:integer) for terminal_module;
   ...
   var addr:integer;
   initialize begin addr:=hw_addr end;
   ...
end terminal;
```

- The default signal input queue discipline is "not queued"; if the discipline is "queued", this must be stated in the process declaration for the pertinent channels:

  ```
  process p for p_module
     queued U, N; ...
  ```

- Major state and state set initialization in processes:

  ```
  initialize stateset_1 = [running,idle,blocked]; ...
  begin state := idle; ... end;
  ```

- Continuous output functions aren't currently recognized.
- The delay clause isn't currently recognized.
- Output statements are prefixed by the reserved word "OUT"
- Any and when clauses should not be used for the same transition ; they're incompatible.
- All transition clauses are optional.
- A transition block can be tagged with an identifier that will be used in the code generated from this transition. This is especially useful to keep track of spontaneous transitions, because their triggering by means of pseudo-signals must be custom-programmed for a given system implementation.
- Refinements are supported, similar to the proposal in [3].
- The "(/" and "/)" enclosure for informal specification elements isn't (yet) recognized.
- Comments can be embedded.
- The proposed facility for describing optional or undefined parameters or fields isn't supported.
- Pascal pointers, files, goto jumps and labels are allowed.

References

[1] ISO TC97/SC16/N, Subgroup A of ad-hoc group on FDT "Concepts for describing the OSI architecture", November 1982.

[2] ISO TC97/SC16/WG1, Subgroup B of ad-hoc group on FDT of WG1, "An FDT based on an extended transition model", November 1982.

[3] "An FDT/LPR Syntax based on Pascal", Annex 8 of meeting on Q39/VII, CCITT, March 1982.

Acknowledgements:

The approach to the syntax analysis realized in the parser is based on the concepts of an experimental system built by Michel Gagné. We thank Michel for his support by providing his syntax table generator for this project.

APPENDIX A

A sample specification processed by the parser

```
   1 (*example of system specification with fdt*)
   2
   3 module system; (*outermost module*)
   4 end system;
   5
   6 refinement system_refinement for system;
   7
   8   const maxseq=1; maxlin=80;
   9
  10   type seq_type=0..maxseq;
  11     data_type=
  12       record line:array[1..maxlin] of char; ind:0..maxlin end;
  13     msg_type=record nr,ns:seq_type; d:data_type end;
  14
  15   channel medium_access_point(all);
  16     by all: msg(m:msg_type); ack(nr:seq_type);
  17   end medium_access_point;
  18
  19   module medium;
  20     port: array[zero..1] of medium_access_point(all);
****                ^unknown const
  21   end medium;
  22
  23   process ex_medium for medium;
  24     trans
  25       when port[i].msg
  26         begin out port[1-i].msg(d) end;
  27       when port[i].ack
  28         begin out port[1-i].ack(nr) end;
  29   end ex_medium;
  30
  31   module syde;
  32     port:medium_access_point(all);
  33   end side;
****       ^wrong ident
  34
  35   refinement side_refinement(choice:boolean) for side;
****                                                  ^unknown module
  36
  37     channel terminal_access_point(all);
  38       by all: data(d:data_type);
  39     end terminal_access_point;
  40
  41     module terminal;
  42       port:terminal_access_point(all);
  43     end terminal;
  44
```

```
     process ex_terminal for terminal;

       var this_side:boolean; ch:char; x:data_type;

       initialize
         begin this_side:=choice (*refinement parameter*);
           x.ind:=0
         end;

       procedure getch(choice:boolean; var ch:char); primitive;
       procedure write_data(choice:boolean; data:data_type);
         primitive;

       trans
         when port.data
           begin write_data(this_side,d) end;
       trans (*spontaneous*) keyboard:
         begin getch(this_side,ch);
           if ord(ch)>0 do (*character available*)
****                    ^"then" expected; symbol ignored
             if ch in ['a'..'z','0'..'9'] then (*store character*)
****         ^insertion:"then"
               begin x.ind:=x.ind+1; x.line[x.ind]:=ch;
                 if x.ind>=maxlin then
                   begin out port.data(x); x.ind:=0 end
               end
             else if x.ind>0 then
               begin out port.data(x); x.ind:=0 end
         end;

     end ex_terminal;

     channel clock_access_point(user,provider);
       by user: set_clock(delay:integer); disable_clock;
       by provider,sneaky: time_out;
****               ^undeclared role
     end clock_access_point;

     module clock;
       port: clock_access_point(provider);
     end clock;

     process ex_clock for clock;

       var state:(running,idle); this_side:boolean;

       initialize
         begin state:=idle;
           this_side:=choice (*refinement parameter*)
         end;

       procedure settimer(choice:boolean; delay:integer);
         primitive;
       procedure resettimer(choice:boolean); primitive;
```

```
        function timeout(choice:boolean):boolean; primitive;

        trans
          when port.set_clock to running
            begin settimer(this_side,delay) end;
          when port.disable_clock from running to idle
            begin resettimer(this_side) end;
        trans (*spontaneous*)
          provided timeout(this_side) from running to idle
            begin out port.time_out end;

      end ex_clock;

      module protocol;
        t_port: terminal_ackcess_point(all);
****              ^unknown channel
        c_port: clock_access_point(user);
        m_port: medium_access_point(all);
      end protocol;

      process ex_protocol for protocol;

        queued t_port,m_port;

        type bufelptr=^bufel;
          bufel=record next:bufelptr; info:data_type end;

        var p:bufelptr; next_frame_to_send,frame_expected:seq_type;

        initialize
          begin p:=nil;
            next_frame_to_send:=0; frame_expected:=0
          end;

        procedure sendmsg;
          var x:msg_type;
        begin x.nr:=1-frame_expected; x.ns:=next_frame_to_send;
          x.d:=p^.info;
          out m_port.msg(x); out c_port.set_clock(10);
        end;

        procedure putbuf(d:data_type);
          var q,r:bufelptr;
        begin new(r]; r^.info:=d; r^.next:=nil;
****               ^")" expected; symbol ignored
****                ^insertion:")"
          if p=nil then begin p:=r; sendmsg end
          else
            begin q:=p;
              while q^.next<>nil do q:=q^.next;
              q^.next:=r
            end
        end;

      procedure getbuf;
        var q:bufelptr;
      begin q:=p; p:=p^.next; dispose(q);
        out c_port.disable_clock;
        next_frame_to_send:=1-next_frame_to_send
      end;

      trans
        when t_port.data
          begin putbuf(d) end;
      trans
        when m_port.msg any k:integer do when m_port.ack
****                    ^incompatible clauses
****                                     ^doubly used clause
          begin
            if m.ns=frame_expected then
              begin out t_port.data(m.d);
                frame_expected:=1-frame_expected
              end;
            if m.nr=next_frame_to_send then getbuf;
            if p=nil then out m_port.ack(1-frame_expected)
            else sendmsg
          end;
      trans
        when m_port.ack
          begin
            if nr=next_frame_to_send then getbuf;
            if p<>nil then sendmsg
          end;
      trans
        when c_port.time_out
          begin sendmsg end;

    end ex_protocol;

    (*instantiation of modules in side_refinement*)
    t: terminal with (*process*) ex_terminal;
    c: clock with (*process*) ex_clock;
    p: protocol with (*process*) ex_protocol;

    internal connection t.port=p.t_port; c.port=p.c_port;
    external connection side_refinement.port=p.m_port;

  end side_refinement;

  (*instantiation of modules for system_refinement*)
  s1: side with (*refinement*) side_refinement(true);
  s2: side with (*refinement*) side_refinement(false);
  m: medium with (*process*) ex_medium;

  internal connection s1.port=m.port[0]; s2.port=m.port[1];

end system_refinement;
```

```
 199 process incomplete
 200
**** ^"("/";"/"for" expected; insertion:"for" ident ";" "end" ident ";"
```

APPENDIX B

Syntax accepted by the parser

```
<axiom> = <seqsect>.

<seqsect> = <section> ";" <seqsect> / empty.

<section> = <channel> / <module> / <process> / <refinemt>.

<channel> = <constd> <typed> "channel" <ident>
            "(" <rolelist> ")" ";" <byclause> "end" <ident>.

<rolelist> =  <ident> <seqident>.

<seqident> = "," <rolelist> / empty.

<byclause> = "by" <rolelist> ":" <signal> <byclause> / empty.

<signal> = <ident> <signalpara> ";" <signal> / empty.

<signalpara> = "(" <paradef> ")" / empty.

<seqparadef> = ";" <paradef> / empty.

<paradef> = <rolelist> ":" <ident> <seqparadef>.

<module> = "module" <ident> ";" <portlist> "end" <ident>.

<portlist> = <rolelist> ":" <array> <ident> "(" <ident> ")" ";"
             <portlist> / empty.

<array> = "array" "[" <indextype> <seqindext> "]" "of" / empty.

<indextype> = <simpletype>.

<seqindext> = "," <indextype> <seqindext> / empty.

<refinemt> = "refinement" <ident> <signalpara> "for" <ident> ";"
             <refbody> "end" <ident>.

<refbody> = <seqsect> <instance> <intconnec> <extconnec> / empty.

<instance> = <rolelist> ":" <ident> "with" <ident> <lparacint> ";"
             <seqinst>.

<seqinst> = <instance> / empty.

<intconnec> = "internal" "connection" <connectn> / empty.

<extconnec> = "external" "connection" <connectn> / empty.
```

```
<portspec> = <ident> "." <ident> <optindex>.

<connectn> = <portspec> "=" <portspec> ";" <seqconnectn>.

<seqconnectn> = <connectn> / empty.

<optindex> = "[" <constant> <listconst> "]" / empty.

<process> = "process" <ident> <signalpara> "for" <ident> ";"
            <procbody> "end" <ident>.

<qchannel> = "queued" <rolelist> ";" / empty.

<procbody> = <qchannel> <constd> <typed> <pvard> <init> <procfuncd>
             <trans> / empty.

<pvard> = "var" <procvar> / empty.

<procvar> = "state" ":" "(" <rolelist> ")" ";" <seqvardecl>
            / <vardecl>.

<stateset> = <ident> "=" "[" <seqsetint> "]" ";" <stateset>
             / empty.

<init> = "initialize" <stateset> "begin" <initstatmt> <seqstatmt>
         "end" ";" / empty.

<initstatmt> = "state" ":=" <ident> / <plainstatmt>.

<trans> = "trans" <seqclause> <opttrans>.

<opttrans> = <trans> / empty.

<seqclause> = <clause> <seqclause>
              / <opttag> <block> ";" <seqtrans>.

<clause> = "any" <paradef> "do" / "with" <variable> "do"
           / "when" <ident> <vparam> "." <ident>
           / "from" <rolelist> / "to" <nextmstate>
           / "save" <ident> <vparam> "." <ident>
           / "proemptyd" <expression> / "priority" <idorint>.

<seqtrans> = <seqclause> / empty.

<opttag> = <ident> ":" / empty.

<vparam> = "[" <constant> <listconst> "]" / empty.

<listvariable> = "," <variable> / empty.

<nextmstate> = <rolelist> / "same".

<idorint> = <ident> / <integer>.
```

```
<block> = <labeld> <constd> <typed> <vard> <procfuncd>
          "begin" <statmt> <seqstatmt> "end".

<labeld> = "label" <integer> <seqinteger> ";" / empty.

<seqinteger> = "," <integer> <seqinteger> / empty.

<constd> = "const" <defconst> / empty.

<defconst> = <ident> "=" <constant> ";" <seqdefconst>.

<seqdefconst> = <defconst> / empty.

<constant> = <optsign> <numconst> / <string>.

<sign> = "+" / "-".

<optsign> = <sign> / empty.

<numconst> = <integer> / <real> / <ident>.

<typed> = "type" <deftype> / empty.

<deftype> = <ident> "=" <type> ";" <seqdeftype>.

<seqdeftype> = <deftype> / empty.

<type> = <simpletype> / <optpack> <typstruct> / "^" <ident>.

<simpletype> = "(" <rolelist> ")"
               / <sign> <numconst> ".." <constant>
               / <string> ".." <constant>
               / <integer> ".." <constant>
               / <ident> <optconst>.

<optconst> = ".." <constant> / empty.

<optpack> = "packed" / empty.

<typstruct> = "array" "[" <simpletype> <seqsimplet> "]" "of" <type>
              /"record" <field> "end"
              /"set" "of" <simpletype>
              /"file" "of" <type>.

<seqsimplet> = "," <simpletype> <seqsimplet> / empty.

<field> = <fixedpart> <seqfield>
          / "case" <ident> <typselect> "of" <variant>.

<fixedpart> = <rolelist> ":" <type> / empty.

<seqfield> = ";" <field> / empty.
```

```
<typselect> = ":" <ident> / empty.

<variant> = <constant> <listconst> ":" "(" <field> ")" <seqvariant>
            / empty.

<seqvariant> = ";" <variant> / empty.

<listconst> = "," <constant> <listconst> / empty.

<vard> = "var" <vardecl> / empty.

<vardecl> = <rolelist> ":" <type> ";" <seqvardecl>.

<seqvardecl> = <vardecl> / empty.

<procfuncd> = <pfheader> ";" <pfbody> ";" <procfuncd> / empty.

<pfheader> = "procedure" <ident> <lpara>
             / "predicate" <ident> <lpara>
             / "function" <ident> <lpara> ":" <ident>.

<pfbody> = <block> / "external" / "forward" / "primitive" / "...".

<lpara> = "(" <spara> <seqspara> ")" / empty.

<seqspara> = ";" <spara> <seqspara> / empty.

<spara> = <rolelist> ":" <ident>
          / "var" <rolelist> ":" <ident>
          / "procedure" <ident> <lpara>
          / "function" <ident> <lpara> ":" <ident>.

<factor> = <real> / <string> / <integer> / "..." / "nil"
           / "[" <seqsetint> "]" / "(" <expression> ")"
           / "not" <factor> / <ident> <seqfactid>.

<seqfactid> = <lseqvaria> / "(" <index> ")".

<index> = <expression> <seqindex>.

<seqindex> = "," <index> / empty.

<lseqvaria> = "[" <index> "]" <lseqvaria>
              / "." <ident> <lseqvaria>
              / "^" <lseqvaria>
              / empty.

<seqsetint> = <setint> <lseqset> / empty.

<lseqset> = "," <setint> <lseqset> / empty.

<setint> = <expression> <seqxpset>.

<seqxpset> = ".." <expression> / empty.
```

```
<term> = <factor> <seqfact>.

<seqfact> = <opermult> <term> / empty.

<opermult> = "*" / "/" / "div" / "mod" / "and".

<simplexp> = <optsign> <term> <seqterm>.

<seqterm> = <operadd> <term> <seqterm> / empty.

<operadd> = "+" / "-" / "or".

<expression> = <simplexp> <seqsimplexp>.

<seqsimplexp> = <operel> <simplexp>.

<operel> = "=" / "<>" / "<" / ">" / "<=" / ">=" / "in".

<statmt> = <integer> ":" <plainstatmt> / <plainstatmt>.

<plainstatmt> = "goto" <integer> / <ident> <appendix>
               / "out" <ident> <seqind> "." <ident> <lparacint>
               / "nextstate" <newmstate>
               / "begin" <statmt> <seqstatmt> "end"
               / "if" <expression> "then" <statmt> <optelse>
               / "case" <expression> "of" <case> <seqcase> "end"
               / "repeat" <statmt> <seqstatmt>
               "until" <expression>
               / "while" <expression> "do" <statmt>
               / "for" <ident> ":=" <expression> <direction>
               <expression> "do" <statmt>
               / "with" <variable> "do" <statmt>
               / empty.

<lparacint> = "(" <index> ")" / empty.

<newmstate> = <ident> / "same".

<seqstatmt> = ";" <statmt> <seqstatmt> / empty.

<optelse> = "else" <statmt> / empty.

<seqcase> = ";" <case> <seqcase> / empty.

<case> = <constant> <listconst> ":" <statmt> / empty.

<direction> = "to" / "downto".

<variable> =  <ident> <lseqvaria> <listvariable>.

<appendix> = <lparacint> / <lseqvaria> ":=" <expression>.

<seqind> = "[" <index> "]" <seqind> / empty.
```

APPENDIX C

Calling sequence for using the parser

The FDT parser is currently available on two of the University of Montreal's computer systems, and can be incited by the following calling sequences:

- On CYBER/NOS BE through TELUM interaction facility:

```
SO FDT u=1394 P1=n P2=source
```

The parameters n and source are provided by the user. n (octal) is the maximum central memory space, in words, required to run the parser. This value depends on the length of the FDT source program. The example in appendix A, for instance, requires about 61000 (octal) memory words. The generated program listing is routed to the standard output file.

- On VAX/VMS at the Department of Computer Science and Operations Research, type

```
@SYS$DISK:[GERBER]FDT
```

and answer the subsequent questions.