TECHNICAL REPORT

ISO/IEC TR 10167

First edition
1991-11-15

Information technology — Open Systems Interconnection — Guidelines for the application of Estelle, LOTOS and SDL

Technologies de l'information — Interconnexion de systèmes ouverts — Principes directeurs pour l'application d'Estelle, LOTOS et SDL

Reference number
ISO/IEC TR 10167:1991(E)

Contents

List of Figures

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The main task of technical committees is to prepare International Standards, but in exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC TR 10167, which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

Annexes A, B, C and D of this Technical Report are for information only.

The formal descriptions in this Technical Report have been prepared with care, and have been checked by tools wherever practicable. However, the aim of this Technical Report is to be tutorial rather than defin-

itive in nature. The emphasis has therefore been on giving timely guidance on the use of FDTs.

It is therefore possible that some errors remain in the formal descriptions. Readers are encouraged to report these. Errors in SDL descriptions should be reported to:

CCITT Secretariat
(SG X Question X/1 - FDT)
Rue Varembé 2
GENEVA
Switzerland

Errors in Estelle or LOTOS descriptions should be reported to:

ISO/IEC JTC 1/SC 21 Secretariat
(Project 1.21.45)
1430 Broadway
NEW YORK
NY 10018
USA

via the appropriate National Standards Body.

Introduction

Formal Description Techniques have been developed to be used in the formal specification of OSI and other telecommunications services and protocols. SDL in particular has been developed for application in the wider field of telecommunications systems, but in this Technical Report the focus is on the specification of OSI services and protocols.

The purpose of this Technical Report is:

a) to aid the users of the FDTs *(Formal Description Techniques)* Estelle, LOTOS, and SDL; and

b) to assist and encourage the use of the FDTs for specifying OSI services and protocols; and

c) to introduce the FDTs through a carefully chosen set of graded examples; and

d) to assist and encourage the use of the FDTs for defining unambiguous requirements for implementation and conformance testing; and

e) to illustrate the errors and ambiguities which can arise with natural language descriptions; and

f) to illustrate how basic architectural ideas may be represented using FDTs.

Information technology — Open Systems Interconnection — Guidelines for the application of Estelle, LOTOS and SDL

1 Scope

This Technical Report provides:

a) an introduction to the nature and purpose of FDTs; and

b) formal descriptions in each of the FDTs for selected examples described in natural language; and

c) guidance to the FDT users as to how to judge and improve the quality of formal descriptions; and

d) management guidance to FDT users as to how to handle the relationship between informal and formal descriptions, and between formal descriptions; and

e) an implicit basis for comparison of the FDTs.

This Technical Report does not provide:

a) tutorials on the FDTs and the OSI architecture; and

b) a means of formally mapping between descriptions produced using different FDTs; and

c) an explicit comparison of the FDTs.

The definition of each FDT, a tutorial on its usage, and the definition of the OSI architecture are indicated in clause 2.

The intended audience for this Technical Report is those who require to develop formal descriptions, and those who require to use FDTs generally.

2 References

The following standards contain provisions that, through reference in this text, constitute provisions of this Technical Report. At the time of publication, the editions indicated were valid. All standards are subject to revision, so parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

A bibliography of related documents is given in Annex A.

ISO 7498 : 1987, *Information processing systems — Open Systems Interconnection — Basic Reference Model.*

ISO/TR 8509 : 1987, *Information processing systems — Open Systems Interconnection — Service conventions.*

ISO 8807 : 1989, *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour.*

ISO 9074 : 1989, *Information processing systems — Open Systems Interconnection — Estelle — A formal description technique based on an extended state transition model.*

ISO/TR 10023 : —[1], *Information processing systems — Open Systems Interconnection — Formal description of ISO 8072 (transport service definition) in LOTOS.*

CCITT T.70, *Network-Independent Basic Transport Service for the Telematic Services* (Red Book).

CCITT Z.100, SDL, *Specification and Description Language* (Blue Book).

CCITT Z.100 Annex D, *SDL User Guidelines* (Blue Book).

CCITT Z.100 Annex F, *SDL Formal Definition* (Blue Book).

CCITT Z.200, *Open Systems Interconnection Basic Reference Model* (Red Book).

3 Terminology

The following terms are referenced within the Technical Report. Where a term is used with a particular meaning (e.g. an FDT concept, keyword, or variable) it is given in **bold face** type.

[1] To be published

3.1 Architectural Terms

The following terms are used in accordance with the definitions in ISO 7498 and CCITT X.200:

a) (N)-association
b) concatenation
c) (N)-connection
d) (N)-connection-endpoint
e) (N)-entity
f) (N)-facility
g) flow-control
h) (N)-function
i) multiplexing
j) (N)-protocol
k) (N)-protocol-data-unit
l) segmentation
m) (N)-service
n) (N)-service-access-point
o) (N)-service-data-unit
p) splitting.

The following terms are used in accordance with the definitions in *ISO/TR 8509*:

q) confirm
r) indication
s) request
t) response
u) (N)-primitive
v) (N)-service-provider
w) (N)-service-user.

For brevity, the above terms are referred to in this Technical Report without the **(N)-** prefix (e.g. **Service Access Point**).

3.2 FDT Terms

The following FDT terms are referenced within this Technical Report. For those unfamiliar with FDT terminology, a tutorial introduction is given in Annex B.

a) abstract, abstraction
b) action
c) composition, decomposition
d) constructive, non-constructive
e) description
f) formal, formalisation
g) implementation
h) information
i) interaction
j) interpretation
k) model
l) non-determinism
m) specification.

4 FDT General Characteristics

4.1 Introduction

Formal Description Techniques exhibit different strengths with respect to their location on the range from abstract to implementation-oriented descriptions. All FDTs offer the means for producing unambiguous descriptions of OSI Services and Protocols in a more precise and comprehensive way than natural language descriptions.

FDTs provide a foundation for analysis and verification of a description. The target of analysis and verification may vary from abstract properties to concrete properties.

Natural language descriptions remain an essential adjunct to formal descriptions, enabling an unfamiliar reader to gain rapid insight into the structure and function of Services and Protocols.

4.2 The Nature and Purpose of FDTs

4.2.1 The Purpose of FDTs

ISO/TC97/SC21 has developed International Standards for two FDTs:

a) **Estelle** (based on Pascal, with extensions to describe finite state machines); and
b) **LOTOS** (based on the mathematical techniques **CCS (Calculus of Communicating systems)**, **CSP (Communicating Sequential Processes)**, and **ACT ONE**).

CCITT/SG/X has already developed and issued a Recommendation for the FDT:

c) **SDL** (based on an extended finite state machine model with two concrete syntaxes, one graphical and one textual).

All three FDTs share a common basis, namely labelled transition systems, for expressing dynamic behaviour. Estelle uses Pascal data types for data description, while LOTOS and SDL use Abstract Data Types.

The objectives of FDTs are (in brief):

a) *unambiguous*, *clear* and *concise* specifications; and
b) a basis for determining *completeness* of specifications; and
c) a foundation for *analysing* specifications for correctness, efficiency, etc.; and

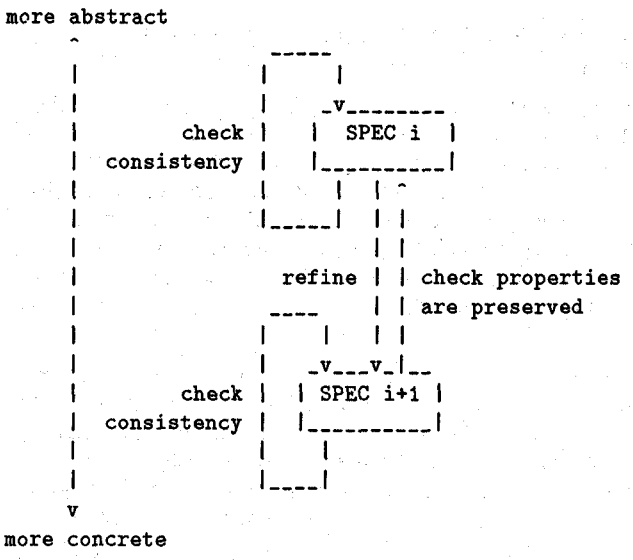


Figure 4.1: Development through Refinement

d) a basis for determining *conformance* of implementations to specifications; and

e) a basis for determining *consistency* of specifications relative to each other; and

f) a basis for *implementation* support

The use of an FDT imposes a discipline of attending to relevant details, thus increasing confidence in the resultant description. Although the development of tools was not an explicit objective of FDTs, the rigorous nature of FDTs makes it possible to develop tools which assist in the creation, analysis, and refinement of formal descriptions.

4.2.2 Use in Development

Each stage in the software (or hardware) development process can be pictured as in Figure 4.1.

The development process is a succession of such activities, beginning with informal requirements and ending up with an implementation. Different languages, appearing at different points in the specification-implementation spectrum, may therefore be appropriate at different stages in the development process.

4.2.3 Assessment of FDTs

FDTs are used to represent basic concepts such as abstraction, modularity, information-hiding, structuring, and synchronisation, as well as more complex architectural concepts. Later clauses illustrate these.

It is difficult to assess FDTs, since they differ in their technical aspects and in the goals of their application. However, tutorial Annexes are provided to assist in this:

A) bibliography; and

B) characteristics of FDTs; and

C) criteria for evaluating FDTs; and

D) criteria for evaluating formal descriptions.

4.3 Estelle

Estelle is a formally-defined specification language for describing distributed or concurrent processing systems, in particular those which implement OSI Services and Protocols. The language is based on widely used and accepted concepts of communicating non-deterministic state machines (automata). An Estelle specification defines a system of hierarchically-structured state machines. The machines communicate by exchanging messages through bi-directional channels that connect their communication ports. These messages are queued at either end of the channel. The actions of machines are specified in (extended) Pascal, hence familiarity with Pascal makes Estelle specifications easily readable.

Estelle mechanisms allow modelling of synchronous and asynchronous parallelism between the state machines of a specified system. They also permit dynamic development of the system configuration.

Estelle specifications can be prepared at different levels of abstraction, from abstract to quite implementation-oriented. The latter may be derived from the former with the aid of supporting tools. Since all Estelle concepts are rigorously defined, Estelle tools which accurately reflect the language can be developed.

4.4 LOTOS

LOTOS is a mathematically-defined FDT, developed from a large, well-established body of theory based on CCS, CSP, and ACT ONE.

Having a well-defined mathematical foundation, it provides a solid basis for both analysis and development of reliable tools, including simulation, compilation, and test sequence derivation.

The basic constructs of LOTOS allow modelling of sequencing, choice, concurrency, and non-determinism in an entirely unambiguous way. In addition, LOTOS permits modelling of both synchronous and asynchronous communication.

LOTOS may be applied to produce a specification of the allowed behaviours of a system, i.e the set of all behaviours which may be observed of a conforming implementation.

Furthermore, LOTOS permits the description of allowed behaviours without describing how this may be achieved, or by describing particular mechanisms which achieve the required behaviour.

4.5 SDL

SDL is based on the extended finite state machine model supplemented by capabilities for Abstract Data Types based on the initial algebra model (the same as used in the ACT ONE part of LOTOS). This combination is supported by a

well-defined formal semantics.

SDL provides constructs for representing structures, behaviours, interfaces, and communication links. In addition, it provides constructs for abstraction, module encapsulation, and refinement. All of these constructs were designed to assist the representation of a variety of telecommunications system specifications, including aspects of protocols and services.

SDL is quite widely used in the telecommunications community, and is well supported by a variety of tools, some of which are generally available.

4.6 Benefits of FDTs

With a specification written in an FDT, an accurate description of (system) behaviour can be given. Descriptions written so far with the FDTs mostly deal with the behaviour of data communications services and protocols, and with telecommunications switching systems. Other kinds of behaviour can be considered for specification, for instance the description of the dialogue between a user and a system.

A high-level description says exactly how a system should behave. It describes only its behaviour, not the realisation of that behaviour (implementation details are excluded). The description is also exact: there are no loose ends or specification gaps, and behaviour is described for all possible system inputs. When an input should give an undefined behaviour, this should be described as well.

The use of an FDT enforces a discipline on the specifier as to what information should be given and how it should be presented. Although it might be felt by the specifier that this discipline is very strict, the result is a much better quality description than would have resulted if natural language had been used. This benefit is obtained in addition to the benefits from using automated analysis tools.

Another problem with natural language descriptions is that they often jump from one abstraction level to another. An FDT gives better possibilities for structuring descriptions, and distinguishing between different abstraction levels.

In a development process which uses a (formal) specification language, several specification levels are generally used. Usually, the starting point is a rough idea of what a system should do. This idea is then written down in natural language. From this informal functional description a formal description is derived (written in a specification language). In this process, exact requirements must be captured, and loose ends or fuzzy areas must be detected and resolved. The resulting high-level description is an accurate representation of the functions of the system.

Getting a complete and unambiguous high-level description before most of the design decisions are made is one of the most important benefits of using an FDT. Even if the natural language requirements are written very carefully, as for the examples in this Technical Report, errors and omissions are found. The problems with the ambiguity of natural language descriptions is best illustrated by the fact that specialists who agree on the natural language description may not agree on a formal description: they may interpret the informal description differently.

A high-level description is the basis for further development of the system. An advantage of using an FDT is that, since the specification language is defined exactly, computers can be used in the development process. From the high-level description more detailed descriptions can be derived, leading finally to an implementation. At each step in the design process, implementation decisions and restrictions are made more explicit, being postponed as late as possible.

Each detailed description or the implementation can be verified (with computer assistance) for conformance to the specification it is derived from. A further advantage is the possibility of deriving test sequences with expected responses for the final implementation; in principle, these can be derived from the original specification automatically.

A formal description gives the precise relationship between components of a system. If it is a description of a system to be built, the formal description is a good basis for project planning (allocation of resources, scheduling of component design, unit and integration tests, etc.).

An advantage of using FDTs is that the appropriate level of abstraction may be used. For example, a high-level description may be given of what a system should do (its functionality); such an implementation-independent description would be appropriate in a definitive International Standard. Implementation-independence can also make descriptions re-usable for future systems (in a different environment or with different constraints) and can facilitate description of re-usable software components. However, it may also be advantageous to give a more implementation-oriented description so as to assist the production of conforming implementations.

More detailed tutorial guidance on FDTs and their application can be found in the Annexes.

4.7 Tools for FDTs

The availability of tools for FDTs can be an important factor in their application. The nature of these tools follows from the aspects of the application of FDTs described in the previous section.

There have to exist good tools for writing and changing descriptions. These tools could incorporate some kind of source version-control system to keep track of different versions. Furthermore, there should exist verification tools: a description which is not checked for correctness loses much of its value as an exact description of a system or an International Standard. When descriptions are written using a computer system, it would be a waste not to use that computer system for verification of correctness.

Several tools for FDTs can be considered. These tools can be divided into two categories, static and dynamic.

Static tools deal only with language aspects of the FDT used for a description. Into this category fall (graphical) (syntax-driven) editors, checkers for correct use of the FDT,

teaching tools, and static report-generators (reporting on use of language constructs).

When descriptions are written, a style must be chosen that is appropriate for the system to be described. The 'best' style is rarely found at the first attempt. Static tools facilitate modifications of style, as well as modifications and extensions of the description.

Dynamic tools deal with the behaviour of the specified system. They can be used to verify whether the specified system will behave as the specifier wants (e.g. freedom from (unwanted) deadlock or livelock). Useful tools in this area are simulators, prototype generators, symbolic executors, and tools for computer-assisted proving of properties like absence of deadlock and livelock.

The benefit of these tools is that it is possible, at an early stage, to experiment with the system, thus giving confidence in the correct behaviour of the system. If the description is of a system to be built, dynamic tools help to discover misunderstandings between the client and the specifier.

There should also be tools to verify whether the implementation conforms to the specification. It is hard to take large steps in the design process (e.g. deriving a low-level implementation from a high-level specification), but when stepwise refinement techniques are used in the process of going from the specification to an implementation, verification can be done for each refinement step. The tools in this category include interpreters, simulators/animators, validators, and test sequence generators. The benefit of these tools is that they help to avoid the introduction of errors during the design process.

Other possible tools include (interactive) generators of an implementation or prototype of the specified system. These tools ease the burden of straightforward but error-prone coding, giving the opportunity to concentrate on important implementation decisions.

The kinds of prototype tools for FDTs that are available or are being developed include:

a) specialised editors to help produce or modify formal descriptions; and

b) formatters to produce pretty-printed text or graphical representations of formal descriptions; and

c) verifiers and theorem-provers to analyse specification properties; and

d) parsers to detect lexical and syntactic errors, and to perform checks on static semantics (e.g. type-checking); and

e) simulators to aid interactive analysis of formal descriptions; and

f) compilers to generate executable code in some target language; the output of a compiler would vary according to the intended use of the code (simulation, debugging, implementation, etc.); and

g) test sequence generators, for checking implementations against a formal description.

The benefits of using such tools fall into three categories: increasing confidence in the description of a system, reducing the costs of implementing a formally-described system, and producing an implementation in a systematic way.

The current trend in tools development for FDTs shows that particular kinds of tools are likely to be developed for each FDT. For example, the emphasis with Estelle has been on the early development of compilers. In the case of LOTOS, simulators have been developed first. For SDL, graphical editors and top-down design aids have received priority.

5 Guide to the Examples

5.1 Explanation of the Examples

A careful choice has been made of a graded series of examples, given in each FDT. The examples have been chosen with the following aims.

5.1.1 Examples of Basic FDT Concepts

These are examples of concepts which are represented by all FDTs. They are neutral with respect to any particular FDT and to architectural concepts. They illustrate how the FDTs capture some basic ideas of Information Theory.

5.1.2 Examples of Basic Architectural Concepts

These are more complex examples, drawn from concepts defined in the OSI Basic Reference Model and elsewhere. They illustrate more specifically how FDTs can be used to represent OSI concepts.

5.1.3 Daemon Game

This illustrates a small self-contained system. Although not presented as a Service or Protocol example, this is a gentle lead-in to later, more realistic examples.

5.1.4 Sliding Window Protocol

This illustrates an important flow-control and error-recovery technique which is present in many real Protocols. In addition, it illustrates the description of a Protocol in relation to its underlying Medium.

5.1.5 Abracadabra Service and Protocol

This illustrates the familiar Alternating Bit Protocol, which is a precursor to some real Protocols. It also illustrates the extra features in connection-oriented Protocols. The example presents the description of a Protocol in relation to the Service it provides.

5.1.6 A Transport Protocol

This is based on the CCITT T.70 Transport Protocol in order to illustrate how real Protocols may be formally described.

It is, however, only an example and is not definitive with regard to T.70 as to either the informal or the formal descriptions.

5.2 How to read the Examples

The examples in each FDT have been carefully prepared by experts in the appropriate FDTs. As far as practicable, the formal descriptions have been checked with automated tools. The formal descriptions have also been checked by their authors for top-level consistency with each other and with the informal descriptions. This has lead to deficiencies in the informal descriptions being discovered and corrected. As a matter of policy, these deficiencies have been noted separately rather than correcting the original informal descriptions. The reason for this was to point out the kinds of errors which can arise in writing informal descriptions. The examples therefore serve a secondary purpose of justifying the use of FDTs. A reference to the offending clause in the informal description is given for each deficiency.

However, it should not be forgotten that the examples are just that. They are illustrative of good style in each FDT, but of course it would be possible to produce different and equally valid formal descriptions. The examples also reflect the individual style of their authors, and are thus not necessarily completely uniform in their approach. The examples illustrate the preferences of experienced specifiers, although in some cases the examples were modified in order to ensure commonality with companion descriptions in other FDTs. Alternative styles would be valid, and may be considered superior according to the subjective judgment of the reader.

The examples may be read in different ways:

by FDT	all the examples in one FDT could be read in order to gain insight into how that FDT may be used; or
by example	all the formal descriptions of one example could be read, in order to gain insight into how the FDTs differ in their approach; or
for problems	the informal descriptions and deficiencies of one example could be read in order to gain insight into the kinds of errors and ambiguities which can easily be introduced when writing informal descriptions.

To facilitate the above, the examples are generally presented in the form shown in Figure 5.1.

For the smaller examples, this structure is simplified. For some of the larger examples, the formal description is given in two parts. In the case of SDL, alternative graphical (GR) and textual (PR) representations are possible. The graphical representation is given in all cases, but for brevity the textual representation is given only for the Daemon Game example.

1. Informal Description

2. Deficiencies in the Informal Description
   - 2.1 Deficiency A
     - 2.1.1 Deficiency
     - 2.1.2 Resolution

   etc.

3. Estelle Description
   - 3.1 Architecture of the Formal Description
   - 3.2 Explanation of Approach
   - 3.3 Formal Description
   - 3.4 Subjective Assessment

4. LOTOS Description
   - 4.1 Architecture of the Formal Description
   - 4.2 Explanation of Approach
   - 4.3 Formal Description
   - 4.4 Subjective Assessment

5. SDL Description
   - 5.1 Architecture of the Formal Description
   - 5.2 Explanation of Approach
   - 5.3 Formal Description
   - 5.4 Subjective Assessment

6. Assessment of the Application of FDTs

Figure 5.1: Typical Layout of an Example

6 Examples of Basic FDT Concepts

These are examples of concepts of FDTs or their application. They are neutral with respect to any particular FDT and to architectural concepts. They illustrate how the FDTs capture some basic ideas of Information Theory. Definitions of FDT terms are given in clause B.4.

6.1 Abstraction

6.1.1 Estelle Representation

By design, Estelle allows the writing of descriptions in a style that closely mimics the way communications protocols are described.

Estelle descriptions may be written at various levels of **Abstraction**, ranging from abstract to concrete. Using appropriate support tools, it is possible to move from one level to another. Details may be deferred using **external module bodies**, **primitive** functions and procedures, and the type '...'.

Top-down design is supported in various ways: the usual mechanisms of a modern programming language (Pascal) are augmented by the ability to structure Estelle modules into submodules.

6.1.2 LOTOS Representation

It is an important strength of LOTOS that it may be used with an appropriate level of **Abstraction**. Although LOTOS is a constructive language, it can be used in a constraint-oriented, almost assertional style. That is, it is possible to write LOTOS descriptions which satisfy a 'separation of concerns'. Each such concern, or constraint, may be written as a separate behaviour expression in LOTOS. The constraints may then be combined by the appropriate LOTOS operators (e.g. sequence, choice, interleaving, or synchronisation).

6.1.3 SDL Representation

SDL provides the means to give a system description at any level of detail, and from several viewpoints. Thus it is possible to abstract and to represent only those aspects of a system that matter in a given context. For example, it is possible to neglect implementation detail, maintenance issues, etc.

Moreover, SDL provides the means to describe a system using a sequence of levels, each tied to the previous one and providing more details. In this sense, SDL supports both top-down design and representations of virtual system. This is achieved through structuring, partitioning, and refinement (top-down approach) and through channel structuring (virtual system approach). **Abstraction** is made possible by the use of **Abstract Data Types**, in which objects are described in terms of their properties rather than in terms of their implementation.

6.2 Information

6.2.1 Estelle Representation

Information is represented by Pascal data types. The usual types 'integer' and 'real' are interpreted to be the actual mathematical objects, not computer-dependent approximations to them. As does Pascal, Estelle allows the creation of new data types.

Information that constitutes implementation detail may be indicated using the **any** construct (e.g. **MaxSize** = **any** integer). Details about data types that are not important at a particular level of description may be deferred.

6.2.2 LOTOS Representation

LOTOS represents Information using the ACT ONE **Abstract Data Type** (ADT) language. The emphasis is on the structure of data objects and the operations on them, rather than their representation in a particular implementation (or class of implementations). **Information** may be established when synchronisation on a LOTOS event occurs, or may be transferred when processes are instantiated.

6.2.3 SDL Representation

SDL has a set of pre-defined data types (e.g. 'Real', 'Integer', and 'Charstring'). SDL also provides the means to define structures such as arrays and matrixes. SDL, like LOTOS, uses **Abstract Data Types**; indeed the same Abstract Data Type kernel is shared between SDL and the ACT ONE part of LOTOS.

6.3 Action

6.3.1 Estelle Representation

An action in Estelle corresponds to a transition of a module. A transition can be enabled either through an external event or through conditions local to the module. Transitions are atomic.

6.3.2 LOTOS Representation

The notion of an **Action** corresponds to the notion of an event in LOTOS. Events in LOTOS are atomic; there is no concept of events overlapping in time or happening simultaneously, so all events are fully interleaved. Events in LOTOS are associated with a list of values which corresponds to the **Information** associated with the **Action**. An event has an associated gate name, at which the event occurs, and a list of zero or more values.

6.3.3 SDL Representation

In SDL, the establishment of **Information** (including the manipulation of existing **Information**) is always tied to a state-transition pair. **Information** can be implicit (i.e. the state of the process) or explicit (i.e. contained in a data object). The activation of a transition is the only means through which an **Action** (or sequence of **Actions**) can

take place.

6.4 Interaction

6.4.1 Estelle Representation

Normally, **Interactions** in Estelle correspond to the inputs and outputs of modules. Indeed, these are called **interactions** in Estelle. In addition, Estelle also supports sharing of variables between modules, subject to certain restrictions that serve to eliminate race conditions.

The allowable **Interactions** at an interaction point are described by a channel description. Interactions are queued by the receiving module.

6.4.2 LOTOS Representation

The idea of an **Interaction** is built into the synchronisation mechanism of LOTOS. Two (or more) behaviour expressions may synchronise on an event. Such events may refer to only gate names (pure synchronisation), to matching **!** and **?** expressions (value-passing), and to matching **?** and **?** expressions (value-establishment). Examples of all these forms of synchronisation are found in OSI Standards.

6.4.3 SDL Representation

The idea of an Interaction is supported in SDL by three constructs:

a) **signal interchange**; and

b) **internal signals**; and

c) **shared data**.

The principal means of **Interaction** in SDL is through **signal interchange**. This allows asynchronous **Interactions** to be modelled, wherein the process sending the signal is not aware of when the signal will be received by the receiver. Signals are never lost, i.e. the receiver will always receive them. If no receiver exists, a dynamic interpretation error will occur, but the receiver might be in a state where that particular signal is not awaited, so that it will be discarded. The sender is not notified of the reception of the signal. If this is required, an explicit acknowledge signal should be sent from the receiver.

Through **internal signals**, the **Interaction** between SDL Services can be modelled. Note that the SDL Service construct models activities which are split into separate sequences, where only one sequence can be active at any time. Thus, this type of **Interaction** resembles a transfer of control. It is not the same control transfer as in a procedure call (i.e. immediate transfer, with return at a later point to the caller), nor a process activation (because the Service activation will occur according to the queued internal signals).

Through **shared data** (i.e. variables), two SDL Services or processes may interchange **Information**, and may also be activated upon the occurrence of a certain value using the **PROVIDED** construct. Two processes which are both in a state with no signals waiting, and with a **PROVIDED** clause with the same value for a variable, will be activated at the same time, i.e. when the variable assumes that value. This is the sole means of synchronisation existing in SDL. Note that the variable value should be set by a third process.

```
PROCESS A;
  ...
  OUTPUT WaitingForSync;
  NEXTSTATE WaitSync;

  STATE WaitSync;
    SAVE *;
    PROVIDED VIEW (Activate, Synchroniser);
  ...
ENDPROCESS A;

PROCESS B;
  ...
  OUTPUT WaitingForSync;
  NEXTSTATE WaitSync;

  STATE WaitSync;
    SAVE *;
    PROVIDED VIEW (Activate, Synchroniser);
  ...
ENDPROCESS B;

PROCESS Synchroniser;
  DCL  REVEALED Activate  BOOLEAN;
  ...
  DECISION BothSyncReqReceived;
  (true):  TASK Activate := TRUE;
  ...
ENDPROCESS Synchroniser;
```

6.5 Interaction Point

6.5.1 Estelle Representation

In Estelle, an **Interaction Point** corresponds to the construct of the same name. An Estelle interaction point is an abstract, bi-directional interface through which a module may send and receive interactions. It has three attributes: the corresponding channel identifier; a rôle identifier which describes the interactions a module may send and receive; and a queueing discipline to be used for interactions received through the interaction point. Estelle interaction points may be external or internal.

6.5.2 LOTOS Representation

In LOTOS, an **Interaction Point** corresponds most closely to an event gate. The gate name certainly distinguishes events from those at other gates. However, events are often tagged with identifiers which further distinguish different behaviours. For example, events at a particular endpoint might have the form:

```
Gate ! EndpointId ! Value1 ! ... ! ValueN
```

where **EndpointId** distinguishes that endpoint, and **Value1** to **ValueN** represent the relevant Information.

6.5.3 SDL Representation

In SDL, activities (i.e. SDL Services or processes) interact through an interface. This interface may consist of an implicit part (shared data) and/or an explicit part.

When the **Interaction** is achieved through **signal interchange** between different blocks, an **Interaction Point** can be represented by means of a uni-directional or bi-directional channel. Also, the graphic representation (**Functional Block Interaction Diagram**) is well suited to show the **Interaction Points**.

The channel, as a means of showing an **Interaction Point**, can be further decomposed into sub-channels, each one showing a sub-**Interaction Point**. Each of the sub-**Interaction Points** is connected to the parent **Interaction Point** using the **CONNECT** construct.

The refinement construct applied to the signals associated with a channel provides a means of representing a specification at several levels of detail.

The **Interaction Point**, in the case of **signal interchange**, can also be represented by the **SIGNALSET** construct, contained in the two interacting processes. However, the use of the channel (or signal route) makes the **Interaction Point** explicit and easier to handle from the viewpoint of a human reader.

When the **Interaction** is achieved through internal signals, the **Interaction Points** are defined by means of the **SIGNALROUTE** construct. The corresponding graphic representation may be used, in which lines connect the various SDL services.

When using shared data as an implicit interface, it is advisable to group the data into sets, each set corresponding to a certain **Interaction** between two activities. For example:

```
DCL  REVEALED
  n1, n2 type1,
  n3     type2 /* shared values with P1, P4 */;

DCL  REVEALED
  n3     type2,
  n4     type3 /* shared values with P2, P4 */;
```

and similarly for the **EXPORTED** mechanism. In the sharing processes, it is also advisable to group together in the **VIEWED** declaration, those variables whose values are in common with a group of processes. There is no rule in the language requesting such a grouping, but such grouping simplifies reading.

7 Examples of Basic Architectural Concepts

These are more complex examples, drawn from concepts defined in the OSI Basic Reference Model and elsewhere. They illustrate more specifically how FDTs can be used to represent OSI concepts. However, only *examples* of *particular interpretations* of the concepts are given; the concepts may be specified in other ways. The examples are given in a bottom-up fashion so that more elementary examples are given first.

Each example in this clause is explained in general terms, but a pointer to a specific use in later clauses is given. In many cases, the example could be given in a data-oriented or behaviour-oriented style. The most appropriate choice of style has been made according to the example and the FDT used.

7.1 Service Access Point

For a specific example, see the **Interaction Point** between a player and the system in the **Daemon Game** descriptions.

7.1.1 Estelle Representation

In the simplest case, a Service Access Point is represented in Estelle as an external interaction point. More generally, however, a Service Access Point may contain many End-points; this is represented by an array of interaction points.

External interaction points are indicated as part of a module header. Such an example might be:

```
module M process;
    ip SAP : SAP(provider);
end;
```

This declares **M** to be a process-type module with a single interaction point **SAP** which plays the role of Service Provider. **SAP** and **provider** refer to a channel definition (not given here) and serve to describe the allowable inputs and outputs through the interaction point **SAP**.

For a specific example, see the Daemon Game: interaction point **P**.

7.1.2 LOTOS Representation

A Service Access Point appears in LOTOS in events of the form:

```
S ! Sap ! ...
```

where:

a) **S** is the gate at which communication with the Service takes place; and

b) **Sap** is the identifier of the Service Access Point; it is usually the same as the Address of the Service Access Point.

At the most abstract level, all one can say about **Sap** values is that they are distinct:

```
type SapType is
```

```
    sorts SapSort

    opns  BaseSap : -> SapSort
          NextSap : SapSort -> SapSort

  endtype
```

This says that there is some base identifier for **Sap** values, from which other identifiers are constructed by repeated application of the **NextSap** function. In the absence of equations these values are all distinct. (The **Sap** sort is obviously isomorphic to the natural numbers.)

For a specific example, see the Daemon Game: gate **P**.

7.1.3 SDL Representation

A Service Access Point may be represented in SDL in the form of:

```
CHANNEL Sap

  FROM ServiceUser TO ProtocolEntity
  WITH  Event1, Event2, ...;

  FROM ProtocolEntity TO ServiceUser
  WITH  Event3, Event2, ...;

ENDCHANNEL;
```

where:

a) **Sap** is the name of the channel, and can be used to address the signals (events) using the **VIA** construct in an **OUTPUT** action; and
b) **Event1**, etc. are the names of the signals interchanged at the Service Access Point; and
c) **ServiceUser** is the User of the Service Access Point; and
d) **ProtocolEntity** identifies the supporting Protocol Entity.

Note that events are associated with a particular direction of the interaction.

For a specific example, see the Daemon Game: channel **Gameserver**.

7.2 Endpoint

For a specific example, see the **Transport Connection Endpoint** in the **Transport Protocol** descriptions.

7.2.1 Estelle Representation

As noted above, in the simplest case, an Endpoint corresponds to a Service Access Point, and thus is represented in Estelle as an external interaction point. More generally, however, a Service Access Point may contain many endpoints. In this case, a single Endpoint corresponds to a single element in an array of interaction points.

A slight modification of the Estelle example above shows an array of Endpoints.

```
module M process;
    ip SAP : array [1..MaxSize] of
                NSAP(provider);
end;
```

This defines **SAP** to be an array of Endpoints; the usual Pascal syntax is used to select a single Endpoint in the array. For example, output of an interaction called **ConReq** with address parameters **Addr1** and **Addr2** and Quality of Service parameter **QOS** through the first of the Endpoints would be written as:

```
output SAP[1].ConReq(Addr1, Addr2, QOS)
```

For a specific example, see the Transport Protocol: interaction points **TCEP**.

7.2.2 LOTOS Representation

An Endpoint is represented in LOTOS in events of the form:

```
S ! Sap ! Ep ! ...
```

where:

a) **Ep** is the identifier of an Endpoint, and would be described just as for **Sap**:

```
type EpType is

  sorts EpSort

  opns  BaseEp : -> EpSort
        NextEp : EpSort -> EpSort

endtype
```

For a specific example, see the Transport Protocol: type **TCEndpointIdentifier**.

7.2.3 SDL Representation

In SDL, the representation of an Endpoint can be hidden in the Service Access Point, since the **channel** construct supports several instances.

If with Endpoint it is necessary to connect different SDL Services (or processes) each representing a given set of communications, then the Endpoint is represented using the **SIGNALROUTE** construct:

```
SIGNALROUTE Endpoint1

  FROM ENV TO ServiceEntity1
  WITH Event1, ...;

  FROM ServiceEntity1 TO ENV
  WITH EventN, ...;

SIGNALROUTE Endpoint2
```

```
    FROM ENV TO ServiceEntity2
    WITH EventX, ...;

    FROM ServiceEntity2 TO ENV
    WITH EventM, ...;

  CONNECT Endpoint1 AND Sap;
  CONNECT Endpoint2 AND Sap;
```

Depending on the choice in representing a Service (or Protocol) Entity as a process or as an SDL Service, it is necessary to connect the signalroute to a channel (when using a process) or to a signalroute that is the continuation of the channel to the block embedding the SDL Services.

For a specific example, see the Transport Protocol: variable **CEP_ID**.

7.3 Service Primitive Parameter

The example used is a set of **Quality of Service** (QoS) requirements. For a specific example, see the **Connection Request Called Address** parameter in the **Transport Protocol** descriptions.

7.3.1 Estelle Representation

Service Primitive Parameters are described in Estelle as parameters of interactions. In the Estelle example in 7.4.1, **Addr1**, **Addr2**, and **QOS** are Service Primitive Parameters.

For a specific example, see the Transport Protocol: type **TADDRESS**, as parameter of interaction **TCON_REQ** of channel **TS_INTERFACE**.

7.3.2 LOTOS Representation

A Service Primitive parameter in LOTOS is simply a value of some sort, for example:

```
type QoSSetFormalType is Set renamedby

  sortnames QoSSetSort for Set

endtype

type QoSSetType is QoSSetFormalType

  actualizedby QoSType, Boolean using

    sortnames QoSSort for Element
              Bool    for Fbool

endtype
```

where:

a) **Set** is the standard **library data type**; and

b) **QoS** is the **type** which defines Quality of Service values.

In the above, a set of Quality of Service values is first named by renaming the standard sort **Set**, then the formal parameters **Element** and **Fbool** are instantiated.

For a specific example, see the Transport Protocol: type **TransportAddress**, as used in type **BasicTSP** with operation **TCONReq**.

7.3.3 SDL Representation

Service Primitive Parameters are variables of a given type defined, for example, as:

```
NEWTYPE Typename

ENDNEWTYPE;
```

For example:

```
NEWTYPE QoSSet Set (QoS)

ENDNEWTYPE;
```

where:

a) **QoS** defines Quality of Service values; and

b) **Set** is defined in the usual fashion.

For a specific example, see the Transport Protocol: type **TADDRESS**, as used in type **TPDU** with operator **BUILD_TCR**.

7.4 Service Data Unit

For a specific example, see the **Medium Data Request** parameter in the **Sliding Window Protocol** descriptions.

7.4.1 Estelle Representation

A Service Data Unit is represented in Estelle as one or more interaction parameters. Consider the following example:

```
channel SAP(user, provider);
    by user:
        CONNECTrequest(Addr1, Addr2, QOS);
        CONNECTresponse(Addr1, Addr2, QOS);
        DATArequest(UserData : UserDataType);
        DISCONNECTrequest;
    by provider:
        CONNECTindication(Addr1, Addr2, QOS);
        CONNECTconfirm(Addr1, Addr2, QOS);
        DATAindication(UserData : UserDataType);
        DISCONNECTindication;
```

In this example, the **DATArequest** and **DATAindication** Service Primitives each have Service Data Units represented as the interaction parameter **UserData**, which is defined to be of type **UserDataType**.

For a specific example, see the Sliding Window Protocol: type **DTPDUType**, as parameter of interaction **DT** in channel **M**.

7.4.2 LOTOS Representation

A Service Data Unit is represented in LOTOS as a value of some particular sort which is considered to be Service User data. In a Service description, a Service Data Unit has only operations which construct values. See 7.12.2 and 7.13.2 for what is involved in a Protocol description. A Service Data Unit might be described as:

```
type SduType is OctetString

endtype
```

where:

a) **OctetString** is the standard library data type for a string of octets.

For a specific example, see the Sliding Window Protocol: type **PduType**, as used in type **MPType** with operation **MReq**.

7.4.3 SDL Representation

A Service Data Unit is represented in SDL as a variable of a type which is considered to be Service User Data. For example, a Service Data Unit might be described as:

```
NEWTYPE Sdu

  INHERITS OctetString ALL;

ENDNEWTYPE;
```

where:

a) **OctetString** is defined in the usual fashion for octet strings.

For a specific example, see the Sliding Window Protocol: type **DataType**, as used in signal **MDTreq**.

7.5 Service Primitive

The general example used is a **Data Indication**. For a specific example, see the **Score** signal in the **Daemon Game** descriptions.

7.5.1 Estelle Representation

A Service Primitive is described in Estelle as an interaction, given in a channel definition. In the example given in 7.4.1, **CONNECTrequest**, **CONNECTresponse**, etc. are all Service Primitives.

For a specific example, see the Daemon Game: interaction **Score** in channel **Gameserver**.

7.5.2 LOTOS Representation

The occurrence of a Service Primitive corresponds to a LOTOS event of the form:

```
S ! Sap ! Ep ! Sp (...)
```

where **Sp** is an operation which constructs some value of the sort corresponding to the Service Primitive, for example:

```
DatInd (UserData)
```

where:

a) **DatInd** constructs a Data Indication; and

b) **UserData** holds the value corresponding to the data to be delivered.

For a specific example, see the Daemon Game: operation **Score** in type **SignalType**.

7.5.3 SDL Representation

A Service Primitive can be represented in SDL as a creation of a signal instance e.g.:

```
OUTPUT Sp (Sap, Ep, DatInd (...));
```

However, it is usual to simplify the form of the signal so that in the specific case it might be:

```
OUTPUT DatInd (UserData);
```

where:

a) **UserData** holds the value corresponding to the data to be delivered.

The corresponding reception by the Service User would be with the statement:

```
INPUT DatInd (Var1);
```

where:

a) **Var1** would have to be the same sort as **UserData**.

The time elapsing between the issuing of a Service Primitive and its reception is not determined. However, there are means to explicitly indicate that a Service Primitive must be received within a certain time to be effective. This can be done by adding to the output the parameter **NOW**. This parameter will take the value of the system time at the moment the **OUTPUT** clause is interpreted; such a value can be check against the current time at the reception point.

For a specific example, see the Daemon Game: signal **Score** in system **Daemongame**.

7.6 Protocol Entity

The example used is a **Protocol Entity** which is composed with other Protocol Entities and the underlying Service to yield the required Service. For a specific example, see the transmitter **Protocol Entity** in the **Sliding Window Protocol** descriptions.

7.6.1 Estelle Representation

A Protocol Entity usually corresponds to an Estelle module. Such a module may be refined into sub-modules. A module is described in two parts, a **header** and a **body**. The header is introduced with the keyword **module** and indicates the external visibility of the module, by listing the exported (shared) variables, the parameters, and the external interaction points of the module. Each associated body describes an allowable behaviour of the module. As an example, consider the following:

```
module M process;
    ip SAP : NSAP(provider);
end;

body MBody for M;
end;
```

This describes a module **M** with a single interaction point **SAP** and the most uninteresting body possible, **MBody**.

For a specific example, see the Sliding Window Protocol: module **Transmitter**.

7.6.2 LOTOS Representation

A Protocol Entity is modelled in LOTOS by giving the constraints on behaviour at the boundaries of its upper and lower Services, and the relationship it maintains between behaviour at these. For example:

```
process ProtocolEntity [U, L]
 (USapSet : USapSetSort, LSapSet : LSapSetSort)
  : noexit :=

  UConstraints  [U] (USapSet)
|[U]|
  ULConstraints [U, L]
|[L]|
  LConstraints  [L] (LSapSet)

endproc
```

where:

a) **U** and **L** are the gates for communication at the Upper and Lower Service boundaries of the Protocol Entity; and

b) **USapSet** and **LSapSet** are the sets of identifiers for the Upper and Lower Service Access Points supported by the Protocol Entity; and

c) **UConstraints**, **LConstraints**, and **ULConstraints** constrain the Upper and Lower Services, and the mapping between them.

At a lower level of description, each constraint would be expanded to deal with individual Endpoints.

For a specific example, see the Sliding Window Protocol: process **TransmitterEntity**.

7.6.3 SDL Representation

A Protocol Entity is represented in SDL by a block containing one or more processes. For example:

```
BLOCK ProtocolEntity;
  SIGNALROUTE RLSap
    FROM ProtocolProc TO ENV WITH LReq;
    FROM ENV TO ProtocolProc WITH LInd;

  SIGNALROUTE RUSap
    FROM ProtocolProc TO ENV WITH UInd;
    FROM ENV TO ProtocolProc WITH UReq;

  CONNECT RUSap AND USap;
  CONNECT RLSap AND LSap;

  PROCESS ProtocolProc REFERENCED;

ENDBLOCK ProtocolEntity;
```

where:

a) **USap** is the channel representing the Service Access Point of the Layer; and

b) **LSap** is the channel representing the Service Access Point of the underlying Layer; and

c) **RLSap** and **RUSap** are signalroutes connecting the channels **LSap** and **USap** respectively to the process **ProtocolProc**, and conveying the signals **LReq**, **LInd**, **UReq**, and **UInd**.

For a specific example, see the Sliding Window Protocol: block **sender_entity**.

7.7 Protocol

The example used is a set of **Protocol Entities** which support a Service using two underlying Services. For a specific example, see the **Abracadabra Protocol** descriptions.

7.7.1 Estelle Representation

In Estelle, the rules, procedures, and data structures needed to represent a Protocol are contained in the description of one or more modules and their associated bodies.

```
specification ProtocolExample;

default individual queue;

const
    MaxSap1 = any integer;
    MaxSap2 = any integer;

channel L1chan(user, provider);
    by user:
        U1request;
    by provider:
        U1response;
```

```
channel L2chan(user, provider);
    by user:
        U2request;
    by provider:
        U2response;

module L1 process;
    ip L1Sap: array [1..MaxSap1] of
                    L1chan(provider);
end;

module L2 process;
    ip L2Sap: array [1..MaxSap2] of
                    L2chan(provider);
end;

module U process;
    ip U1Sap: array [1..MaxSap1] of
                    L1chan(user);
       U2Sap: array [1..MaxSap2] of
                    L2chan(user);
end;

body L1body for L1;
end;

body L2body for L2;
end;

body Ubody for U;
end;

var i : integer;

modvar
    L1instance: L1;
    L2instance: L2;
    Uinstance:  U;

initialize
  begin
      init L1instance with L1body;
      init L2instance with L2body;
      init Uinstance with Ubody;
      for i := 1 to MaxSap1 do
          connect L1instance.L1Sap[i] to
              Uinstance.U1Sap[i];
      for i := 1 to MaxSap2 do
          connect L2instance.L2Sap[i] to
              Uinstance.U2Sap[i];
  end;
end.
```

For a specific example, see the Abracadabra Protocol: module **Abra**.

7.7.2 LOTOS Representation

A Protocol is modelled in LOTOS as a set of Protocol Entities composed with a set of underlying Services. For example:

```
process Protocol [U, L1, L2]
 (USapSet : USapSetSort,
  L1SapSet : L1SapSetSort,
   L2SapSet : L2SapSetSort)
  : noexit :=

  ProtocolEntities [U, L1, L2]
   (USapSet, L1SapSet, L2SapSet)
|[L1, L2]|
  (
    UService [L1] (L1SapSet)
  |||
    UService [L2] (L2SapSet)
  )

endproc
```

where:

a) **U**, **L1**, and **L2** are the gates for communication at the supported Service and the two underlying Services respectively; and

b) **USapSet**, **L1SapSet**, and **L2SapSet** are the sets of Service Access point identifiers for the corresponding Services; and

c) **ProtocolEntities** is the composed behaviour of the individual Protocol Entities; and

d) **UService** is the Service derived from either of the underlying Services.

For a specific example, see the Abracadabra Protocol: overall behaviour as represented by the composition of processes **Service**, **Protocol**, and **CMService**.

7.7.3 SDL Representation

A Protocol is represented in SDL by a set of blocks, each block representing a Protocol Entity. The blocks are connected to each other indirectly via the underlying Service. Normally a Protocol contains two blocks which are mirror images of each other. In this case it is sufficient to provide the description of one block only. For example:

```
SYSTEM Protocol;

  NEWTYPE UserDataType ... ENDNEWTYPE;

  SIGNAL ConReq, DatReq (UserDataType), ... ;

  SIGNALLIST ToUser = ConInd, ... ;

  SIGNALLIST FromUser = ConReq, ... ;

  BLOCK ProtocolEntity;

    CHANNEL USap
      FROM ENV TO ProtocolEntity
        WITH (FromUser);
      FROM ProtocolEntity TO ENV
        WITH (ToUser);
```

```
    ENDCHANNEL USap;

    CHANNEL MSap
      FROM ENV TO ProtocolEntity WITH UnitInd;
      FROM ProtocolEntity to ENV UnitReq;
    ENDCHANNEL MSap;

  ENDBLOCK ProtocolEntity;

ENDSYSTEM Protocol;
```

where:

a) **USap** is the channel representing the Service Access Point of the Layer, and conveying the signals contained in the signallists **ToUser** and **FromUser**; and

b) **MSap** is the channel representing the Service Access Point of the underlying Layer, and conveying the signals **UnitReq** and **UnitInd**; and

c) **ProtocolEntity** is a block representing a Protocol Entity.

For a specific example, see the Abracadabra Protocol: system diagram **Abracadabra**.

7.8 Protocol Data Unit

The example used is a **Data Transfer Protocol Data Unit** (DT TPDU), with End of Transfer (EOT) and User Data parameters. For a specific example, see the **Protocol Data Unit** handling in the **Transport Protocol** descriptions.

7.8.1 Estelle Representation

In Estelle, a Protocol Data Unit is realised as an encoded piece of information contained in a Service Data Unit. A Protocol Data Unit may be described using the **User Data Management** procedures and functions described in Annex B of ISO 9074. The following example is based on this:

```
const MaxData = any integer;

type octet    = 0 .. 255;
     LenType  = 0 .. MaxData;
     IdType   = 1 .. MaxData;
     DataType =
       record
         l : LenType;
         d : array [IdType] of octet
       end;
```

For a specific example, see the Transport Protocol: type **TDATA**.

7.8.2 LOTOS Representation

A Protocol Data Unit is represented in LOTOS as a value of some sort which is used to construct underlying Service Data Units. For example:

```
type PduType is Boolean, OctetString

  sorts  PduSort

  opns   Dt     : Bool, OctetString -> PduSort
         DtEot  : PduSort -> Bool
         DtData : PduSort -> OctetString

  eqns forall Eot : Bool, Ud : OctetString

    ofsort Bool

      DtEot  (Dt (Eot, Ud)) = Eot

    ofsort OctetString

      DtData (Dt (Eot, Ud)) = Ud

endtype
```

where:

a) **Boolean** and **OctetString** are the standard library data types; and

b) **DtPdu** represents a Data Transfer Protocol Data Unit; and

c) **Dt** is used to construct these Data Transfer Protocol Data Units; and

d) **DtEot** selects the End of Transfer flag; and

e) **DtData** selects the User Data field.

This description gives the abstract encoding of the Protocol Data Unit. A lower level description would be given if the concrete encoding were needed (i.e. field order, field sizes, bit patterns, etc.).

For a specific example, see the Transport Protocol: type **BasicBlock**.

7.8.3 SDL Representation

A Protocol Data Unit is simply a value of some sort, for example:

```
NEWTYPE Dt STRUCT
  DtEot BOOLEAN;
  DtData OctetString;
ENDTYPE DtPdu;
```

which defines a sort of Data transfer Protocol Data Units with constructor **Dt** and selectors for the various parameters. Such a description gives the abstract encoding of a Protocol Data Unit.

For a specific example, see the Transport Protocol: type **TPDU**.

7.9 Connection

The example used is a single connection between two Service Users. For a specific example of the single-connection case, see the **Connection** handling in the **Abracadabra Protocol** descriptions. For a specific example of the multiple-connection case, see the Transport Protocol descriptions.

7.9.1 Estelle Representation

As used in communications protocols, a Connection between peer Protocol Entities is established through the exchange of Protocol Data Units, as defined for a given Protocol. These Protocol Data Units are encoded in Service Data Units and exchanged through the Service provided by lower Layers. This meaning is distinct from the **connect** keyword of Estelle, which is used to associate interaction points of modules.

Typically, a Connection, as viewed by the Protocol Entities involved, goes through various phases such as: opening, data transfer, and closing. Each of these phases is usually represented by in Estelle as a state of the module that represents the Protocol Entity. The number of states used depends on the complexity of the Protocol, the level of abstraction required, etc.

For specific examples, see the Abracadabra Protocol and the Transport Protocol: the states and transitions of modules **Station** and **PARENT**, respectively.

7.9.2 LOTOS Representation

The phases of a Connection are described in LOTOS in the following general form:

```
Connect >> (Data [> Disconnect)
```

In such a description, connection refusal is properly handled as part of Connect and not Disconnect.

The behaviour of a Connection is decomposed into the independent behaviour of its Endpoints (**Ep**) and its End-to-End behaviour (**Ee**). For example:

```
process Connection [S]
 (Sap1, Sap2 : SapSort, Ep1,  Ep2  : EpSort)
  : noexit :=

  (
    EpConstraints [S] (Sap1, Ep1)
  |||
    EpConstraints [S] (Sap2, Ep2)
  )
||
  EeConstraints [S] (Sap1, Ep1, Sap2, Ep2)

endproc
```

where:

a) **S** is the gate for communication at the Service; and

b) **Sap1**, **Sap2**, **Ep1**, and **Ep2** are the identifiers of the pair of interconnected Service Access Points and Endpoints; and

c) **EpConstraints** and **EeConstraints** constrain activities at one Endpoint and end-to-end between two Endpoints.

For specific examples, see the Abracadabra Protocol and the Transport Protocol: processes **Connection** and **TCEP-Connections**, respectively.

7.9.3 SDL Representation

A Connection is represented in SDL by the combined behaviour of two process instances, each representing the active component of a Protocol Entity. When the Layer can provide several simultaneous Connections, then a process instance is created for each Connection in each Protocol Entity. These process instances are created by a dispatcher process that exists permanently from system start-up time.

For specific examples, see the Abracadabra Protocol and the Transport Protocol: the states and transitions of processes **SenderReceiver** and **T_MANAGER**, respectively.

7.10 Multiplexing

The example used is a Multiplexing/Demultiplexing Function which multiplexes data from one Service onto an underlying Service.

7.10.1 Estelle Representation

In Estelle, Multiplexing applies to interactions of two or more Users who request a Service (by Service Primitives) at two or more interaction points. These interactions are mapped by a Protocol Entity onto one underlying interaction point of the underlying Service Provider.

In the simplest case, multiplexing data from several Users onto a single underlying Service is accomplished by adding User identification to User interactions, and then sending the result via the underlying Service. Demultiplexing works in the opposite direction by removing User identification, and then sending the remaining information to that User.

The following example assumes that the necessary channels, types, variables, etc. have been defined. This Estelle fragment gives two transitions, one to multiplex from any of **NUsers** Users onto the underlying **Server**, and the other to demultiplex from the **Server** onto the Users.

```
trans {multiplex}
    any id : 1 .. NUsers do
        when User[id].UDataReq(UserData)
            begin
                output
                  Server.SDataReq(id, UserData)
            end;

trans {demultiplex}
    when Server.SDataInd(id, UserData)
```

```
        begin
            output
              User[id].UDataInd(UserData)
        end;
```

7.10.2 LOTOS Representation

A simple multiplexer/demultiplexer accepts data from different sources, distinguished by their identifier, and forwards it with a tag indicating its source. For example:

```
process MuxDemux [U, L]
 (USap : USapSort, LSap : LSapSort)
  : noexit :=

    Mux [U, L] (USap, LSap)
  |||
    Demux [U, L] (USap, LSap)

  where

  process Mux [U, L]
   (USap : USapSort, LSap : LSapSort)
    : noexit :=

    choice UEp : UEpSort, Ud : OctetString []
      (
        U ! USap ! UEp ! UDataReq (Ud);
        L ! LSap ! LDataReq (UEp, Ud);
        Mux [U, L] (USap, LSap)
      )

  endproc (* Mux *)

  process Demux [U, L]
   (USap : USapSort, LSap : LSapSort)
    : noexit :=

    choice UEp : UEpSort, Ud : OctetString []
      (
        L ! LSap ! LDataInd (UEp, Ud);
        U ! USap ! UEp ! UDataInd (Ud);
        Demux [U, L] (USap, LSap)
      )

  endproc (* Demux *)

endproc (* MuxDemux *)
```

where:

a) **U** and **L** are the gates for communication at the Upper and Lower Services; and

b) **USap** refers to the Upper Service Access Point identifier, and **LSap** refers to the Lower Service Access Point identifier; and

c) **UEp** refers to the Upper Service Endpoint identifier; and

d) **OctetString** is the standard library sort, used for Service Data Units.

7.10.3 SDL Representation

Multiplexing can be handled in SDL by mapping Protocol Data Units from different Associations onto a single Association (channel and receiver identity), tagging each PDU with a final destination identifier.

```
PROCESS Multiplexing;
  ...
  INPUT PDU (Par1, Par2, Par3);
  OUTPUT MediumPDU (Par1, Par2, Par3, RefNo);

PROCESS Demultiplexing;
  ...
  INPUT MediumPDU (Par1, Par2, Par3, RefNo);
  DECISION (RefNo);
    (PathA) :
      OUTPUT PDU (Par1, Par2, Par3) VIA PATHA;
    ...
    (PathN) :
      OUTPUT PDU (Par1, Par2, Par3) VIA PATHN;
  ENDDECISION
```

7.11 Splitting

The example used is a Splitting/Recombining Function which splits data over a number of Service Access Points in an underlying connectionless Service.

7.11.1 Estelle Representation

In the simplest case, splitting data from one User onto several interactions points of the underlying Service is accomplished by accepting interactions from a User, and then sending them through one of several interaction points of the underlying Service. Recombining data in the opposite direction is accomplished by sending the incoming interactions to the User.

The following example assumes that the necessary channels, types, variables, etc. have been defined. This Estelle fragment gives two transitions, one to split the data via one of **NServers** underlying **Servers**, and the other to recombine it the data. It is further assumed that the function **select** chooses an underlying interaction point on which to output the data.

```
trans {split}
    when User.UDataReq(UserData)
        begin
            output
              Server[select].SDataReq(UserData)
        end;

trans {recombine}
    any id : 1 .. NServers
        when Server[id].SDataInd(UserData)
            begin
                output
                  User.UDataInd(UserData)
            end;
```

7.11.2 LOTOS Representation

A simple splitter/recombiner accepts data and forwards it via different routes. For example:

```
process SplitRecombine [U, L]
 (USap1 : USapSort, LSaps : LSapSetSort)
  : noexit :=

  Split [U, L] (USap1, LSaps)
|||
  Recombine [U, L] (USap1, LSaps)

where

process Split [U, L]
 (USap1 : USapSort, LSaps : LSapSetSort)
  : noexit :=

  choice USap2 : USapSort, LSap : LSapSort,
   Ud : OctetString []
    [LSap IsIn LSaps] ->
      (
        U ! USap1 ! UDataReq (USap2, Ud);
        L ! LSap !
         LDataReq (Dt (USap1, USap2, Ud));
        Split [U, L] (USap1, LSaps)
      )

endproc (* Split *)

process Recombine [U, L]
 (USap1 : USapSort, LSaps : LSapSetSort)
  : noexit :=

  choice USap2 : USapSort, LSap : LSapSort,
   Ud : OctetString []
    [LSap IsIn LSaps] ->
      (
        L ! LSap !
         LDataInd (Dt (USap2, USap1, Ud));
        U ! USap1 ! UDataInd (USap2, Ud);
        Recombine [U, L] (USap1, LSaps)
      )

  endproc (* Recombine *)

endproc (* SplitRecombine *)
```

where:

a) **U** and **L** are the gates for communication at the Upper and Lower Services; and

b) **USap1** and **USap2** are the source and destination Upper Service Access Point identifiers, and **LSaps** gives the identifiers of the Lower Service Access Points which may be used for splitting; and

c) **Dt** is used to construct a datagram from the source address, destination address, and user data; and

d) **OctetString** is the standard library sort, used for Service Data Units.

7.11.3 SDL Representation

Splitting in SDL can be represented as follows:

```
PROCESS Splitting;
  ...
  DECISION Split;
    (1): OUTPUT A VIA SAP_1;
    (2): OUTPUT B VIA SAP_2;
  ...
```

Recombining is represented by converging channels (and signalroutes) to a process.

7.12 Concatenation

The example used is a Concatenation/Separation Function which concatenates Protocol Data Units into one Service Data Unit of the underlying Service.

7.12.1 Estelle Representation

In Estelle, Concatenation is carried out by a Protocol Entity combining a set of Protocol Data Units into a single interaction sent to an underlying Service Provider. The following example is based on Annex B of ISO 9074, using the **DataType** definition in 7.8.1:

```
{ append the data contained in "addition" to the
  variable "base", setting "addition" to the
  null data type }

procedure assemble
 (var base : DataType; var addition : DataType);
 var TotLength : integer;
     Index : LenType;

 begin
   TotLength := base.l + addition.l;
   if TotLength > MaxData
     then TotLength := MaxData;
   for index := 1 to TotLength - base.l do
     base.d [index + base.l] :=
       addition.d [index];
   base.l := TotLength;
   null (addition)
 end
```

7.12.2 LOTOS Representation

Concatenation and Separation correspond in LOTOS to operations which relate Protocol Data Units to Service Data Units of the underlying Service. For example:

```
type SduType is OctetString

  opns
    ConcatenateSdu : OctetString, OctetString
                     -> OctetString
    SeparatePdu    : OctetString -> OctetString
```

```
    SeparateSdu     : OctetString -> OctetString

  eqns
   forall Pdu, Pdu1, Pdu2, Sdu : OctetString

    ofsort OctetString

      SeparatePdu (<>) = <>;
      SeparatePdu (ConcatenateSdu (Pdu, <>)) =
        Pdu;
      SeparatePdu (ConcatenateSdu (Pdu1,
       ConcatenateSdu (Pdu2, Sdu))) =
         SeparatePdu (
          ConcatenateSdu (Pdu2, Sdu))

      SeparateSdu (<>) = <>;
      SeparateSdu (ConcatenateSdu (Pdu, <>)) =
        <>;
      SeparateSdu (ConcatenateSdu (Pdu1,
       ConcatenateSdu (Pdu2, Sdu))) =
         ConcatenateSdu (Pdu1, SeparateSdu (
          Concatenate (Pdu2, Sdu)))

endtype
```

where:

a) **SduType** defines a Service Data Unit of the underlying Service; and

b) **OctetString** is the standard library sort, used for Protocol Data Units and Service Data Units of the underlying Service; and

c) <> is the representation of an empty octet string; and

d) **ConcatenateSdu** appends a Protocol Data Unit to a Service Data Unit, yielding a new Service Data Unit; and

e) **SeparatePdu** and **SeparateSdu** yield the last Protocol Data Unit and the remainder of the Service Data Unit respectively, after separation of the original Service Data Unit.

This defines Concatenation and Separation to be inverse operations: Protocol Data Units are separated in the same order as they were concatenated.

7.12.3 SDL Representation

The concatenation of Protocol Data Units can be represented in SDL by operators defined on those Protocol Data Units.

```
NEWTYPE SduType

  LITERALS NullSdu;

  OPERATORS
   ConcatenateSdu : OctetString, OctetString
                    -> OctetString;
   SeparatePdu    : OctetString -> OctetString;
   SeparateSdu    : OctetString -> OctetString;

  AXIOMS
    FOR ALL Pdu, Pdu1, Pdu2, Sdu IN OctetString
      (
        SeparatePdu (NullSdu) == NullPdu;
        SeparatePdu (ConcatenateSdu (
         Pdu, NullSdu) ==
           Pdu;
        SeparatePdu (ConcatenateSdu (Pdu1,
          ConcatenateSdu (Pdu2, Sdu))) ==
            SeparatePdu (
             ConcatenateSdu (Pdu2, Sdu));

        SeparateSdu (NullSdu) == NullSdu;
        SeparateSdu (ConcatenateSdu (Pdu,
         NullSdu)) ==
           NullSdu;
        SeparateSdu (ConcatenateSdu (Pdu1,
         ConcatenateSdu (Pdu2, Sdu))) ==
           ConcatenateSdu (Pdu1, SeparateSdu
            (ConcatenateSdu (Pdu2, Sdu)));
      )
ENDNEWTYPE SduType;
```

where the operations and sorts are much as in the LOTOS example.

7.13 Segmentation

The example used is a Segmentation/Reassembly Function which segments one Service Data Unit into multiple Protocol Data Units of the supporting Protocol. For a specific example, see **Segmentation** and **Reassembly Functions** in the **Transport Protocol** descriptions.

7.13.1 Estelle Representation

In Estelle, Segmentation is realised by decomposing one Service Data Unit into two or more Protocol Data Units, and sending them as two or more interactions via an underlying Service Provider. The following example is based on Annex B of ISO 9074, using the **DataType** definition in 7.8.1:

```
{ segment the data in "old" by moving the first
    "len" octets to "head", leaving the tail (of
    length "length (old) - len") in "old"
}

procedure segment
 (var head : DataType; var old : DataType;
   len : LenType);
  var index, length : LenType;
  begin
    if len > old.l
      then length := old.l
      else length := len;
    create (head, length);
    if length > 0
      then
        begin
```

```
        for index :=
          1 to length do
            head.d [index] := old.d [index];
        for index :=
          1 to old.l - length do
            old.d [index] :=
              old.d [index + length];
        for index :=
          old.l - length + 1 to old.l do
            old.d [index] := 0;
        old.l := old.l - length
      end
end;
```

For a specific example, see the Transport Protocol: procedure **D_FRAGMENT** as used in **trans TC22** and procedure **D_ASSEMBLE**, as used in **trans TC28**.

7.13.2 LOTOS Representation

Segmentation and Reassembly correspond in LOTOS to operations which relate Service Data Units to Protocol Data Units of the supporting Protocol. For example:

```
type PduType is SduType

  opns
    SegmentPdu     : OctetString -> OctetString
    SegmentSdu     : OctetString -> OctetString
    ReassembleSdu  : OctetString, OctetString
                       -> OctetString

  eqns
   forall Pdu, Pdu1, Pdu2, Sdu : OctetString

    ofsort OctetString

      SegmentPdu (<>) = <>;
      SegmentPdu (ReassembleSdu (Pdu, <>)) =
        Pdu;
      SegmentPdu (ReassembleSdu (Pdu1,
       ReassembleSdu (Pdu2, Sdu))) =
         SegmentPdu (ReassembleSdu (Pdu2, Sdu))

      SegmentSdu (<>) = <>;
      SegmentSdu (ReassembleSdu (Pdu, <>)) =
        <>;
      SegmentSdu (ReassembleSdu (Pdu1,
       ReassembleSdu (Pdu2, Sdu))) =
         ReassembleSdu (Pdu1, SegmentSdu (
          ReassembleSdu (Pdu2, Sdu)))

endtype
```

where:

a) **SduType** defines a Service Data Unit; and

b) **PduType** defines a Protocol Data Unit of the supporting Protocol; and

c) **OctetString** is the standard library sort, used for Service Data Units and Protocol Data Units of the supporting Protocol; and

d) <> is the representation of an empty octet string; and

e) **SegmentPdu** and **SegmentSdu** yield a Protocol Data Unit and the remainder of the Service Data Unit respectively, after segmentation of the original Service Data Unit; and

f) **ReassembleSdu** appends a Protocol Data Unit to a Service Data Unit.

This defines Segmentation and Reassembly to be inverse operations: Service Data Units are reassembled in the same order as they were segmented.

For a specific example, see the Transport Protocol: operations **ReplaceTop** and **AddSegment** in type **TSDUS**.

7.13.3 SDL Representation

The segmentation of a Service Data Unit can be represented by *ad hoc* operators defined for the Service Data Unit.

```
NEWTYPE PduType

LITERALS NullPdu;

OPERATORS
  SegmentPdu     : OctetString -> OctetString
  SegmentSdu     : OctetString -> OctetString
  ReassembleSdu  : OctetString, OctetString
                     -> OctetString

AXIOMS
  FOR ALL Pdu, Pdu1, Pdu2, Sdu IN OctetString
    (
      SegmentPdu (NullSdu) == NullPdu;
      SegmentPdu (ReassembleSdu (Pdu,
       NullSdu)) ==
        Pdu;
      SegmentPdu (ReassembleSdu (Pdu1,
       ReassembleSdu (Pdu2, Sdu))) ==
         SegmentPdu (
          ReassembleSdu (Pdu2, Sdu));

      SegmentSdu (NullSdu) == NullSdu;
      SegmentSdu (ReassembleSdu (Pdu,
       NullSdu)) ==
        NullSdu;
      SegmentSdu (ReassembleSdu (Pdu1,
       ReassembleSdu (Pdu2, Sdu))) ==
         ReassembleSdu (Pdu1, SegmentSdu (
          ReassembleSdu (Pdu2, Sdu)));
    )
ENDNEWTYPE PduType;
```

where the operations and sorts are much as in the LOTOS example.

For a specific example, see the Transport Protocol: procedure **DATA_PHASE**.

7.14 Service

The example used is a Service which hides its supporting Protocol and the underlying Service. For a specific example of a Service viewed as a black box, see the **Abracadabra Service** descriptions. For a specific example of a Service viewed as a Protocol combined with the underlying Service, see the **Sliding Window Protocol** plus **Medium** descriptions.

7.14.1 Estelle Representation

In Estelle, a Service is represented by one or more module definitions whose inputs and outputs are Service Primitives for the Layer described. Module behaviour, as defined in relevant module bodies, maps Service Primitives sent by one Service User to Service Primitives received by the corresponding peer Service User. Services that are unique to a Layer are realized by algorithms described within the module bodies.

For specific examples, see the Abracadabra Service module **AbraService**, and Sliding Window Protocol plus Medium modules **TransmitterUser**, **ReceiverUser**, **Timer** plus **Cms**.

7.14.2 LOTOS Representation

A Service description in LOTOS may be a Protocol description with the internal details hidden (i.e. the communication at gates corresponding to the underlying Services). For example:

```
process Service [U]
 (USapSet : USapSetSort, LSapSet : LSapSetSort)
  : noexit :=

  hide L in Protocol [U, L] (USapSet, LSapSet)

endproc
```

where:

a) **U** and **L** are the gates for communication at the Upper and Lower Services; and

b) **USapSet** and **LSapSet** define which Upper and Lower Service Access Points are supported by the supporting Protocol.

However, a Service may also be described in LOTOS without reference to a Protocol description. The behaviour of a Service is decomposed into the independent behaviour of its Endpoints (**Ep**) and its End-to-End behaviour (**Ee**). For example:

```
process Service [U]
 (USapSet : USapSetSort) : noexit :=

  EpConstraints [U] (USapSet)
||
  EeConstraints [U] (USapSet)

endproc
```

where:

a) **EpConstraints** and **EeConstraints** constrain activities independently at each Endpoint and end-to-end between two Endpoints.

For specific examples, see the Abracadabra Service and Sliding Window Protocol plus Medium: overall behaviour as represented by the composition of the processes **Connection** and **Backpressure**; and overall behaviour as represented by the composition of the processes **TransmitterEntity**, **ReceiverEntity**, and **Medium**, respectively.

7.14.3 SDL Representation

A Service can be represented in SDL by means of two interworking processes, which are mirror images of each other. Each represents the behaviour of the Service Provider as seen by a Service User. The processes interact by means of internal signals, conveyed on a bi-directional signalroute and mapping the Service User Primitives. Using two processes instead of one single process is essential in order to model faithfully the time delay in the Service Provider between a Request at one side and the corresponding Indication at the other.

For specific examples, see the Abracadabra Service and Sliding Window Protocol plus Medium: system diagrams **AbraService** and **SlidingWindowProtocol**, respectively.

8 Daemon Game Example

This illustrates a small self-contained system. Although not presented as a Service or Protocol example, this is a gentle lead-in to later, more realistic examples.

8.1 Informal Description

This is a simple game having several players. The game is the system that is to be defined in a chosen specification language. The players belong to the environment of this system.

In the system there is a daemon that generates **Bump** signals randomly. A player has to guess whether the number of generated **Bump** signals is odd or even. The guess is made by sending a **Probe** signal to the system. The system replies by sending the signal **Win** if the number of the generated **Bump** signals is odd, otherwise by the signal **Lose**.

The system keeps track of the score of each player. The score is initially zero. It is increased by 1 for each successful guess (signal **Win** is sent), and reduced by 1 for each unsuccessful guess (signal **Lose** is sent). A player can ask for the current value of the score by the signal **Result**, which is answered by the system with the signal **Score**.

Before a player can start playing, the player must log in. This is accomplished by the signal **Newgame**. A player logs out by the signal **Endgame**. The system allocates a player a unique identifier on logging in, and de-allocates it on logging out. The system cannot tell whether different identifiers are being used by the same player.

8.2 Deficiencies in the Informal Description

8.2.1 Presence of Daemon (Clause 8.1)

8.2.1.1 Deficiency

Should the daemon be an integral part of the description, or is it an artefact of the informal explanation?

8.2.1.2 Resolution

It was not intended that the daemon be part of the system description.

8.2.2 Login to a Current Game (Clause 8.1)

8.2.2.1 Deficiency

What should happen if a player who is already logged in tries to issue **Newgame** again? The informal description does not clearly cover this case.

8.2.2.2 Resolution

The intention was to treat games like 'Bingo' game panels with buttons for input and indicators for output. **Newgame** should therefore be allowed to happen in a current game, but should be ignored.

8.2.3 Attempt to play before Login (Clause 8.1)

8.2.3.1 Deficiency

What should happen if a player issues any signal other than **Newgame** before logging into a game? The informal description says that a player must first login, but does not say what happens if **Newgame** is not the first signal.

8.2.3.2 Resolution

The intention was to allow **Probe**, **Result**, or **Endgame** when a game is not current, but to ignore these signals.

8.2.4 Identification of Players and Games (Clause 8.1)

8.2.4.1 Deficiency

The informal description precludes the case of logging into a current game, because it implies that a further login will result in a new game. This contradicts the intended behaviour as described in 8.2.2. Presumably some identifiers are needed, but how should they be allocated and what should they distinguish?

8.2.4.2 Resolution

The intended behaviour was that each game should be distinguished from the system's point of view by some identifier. The system was not intended to be able to tell which player (or even players) were issuing signals for a game. A player should therefore be able to play multiple games simultaneously without the system knowing: the players should be an anonymous part of the environment of the system.

8.2.5 Player Use of System Signals (Clause 8.1)

8.2.5.1 Deficiency

What should happen if the player issues **Win**, **Lose**, or **Score** signals?

8.2.5.2 Resolution

The intention was to disallow such behaviour: it simply must not happen, as opposed to happening but be ignored.

8.2.6 Interruption of Probe or Result (Clause 8.1)

8.2.6.1 Deficiency

Should it be allowable for another signal to be processed by the system between **Probe** and **Win/Lose**, or between **Result** and **Score**?

8.2.6.2 Resolution

The intention was that **Probe** or **Result** should be followed by their respective responses before any other signal is processed.

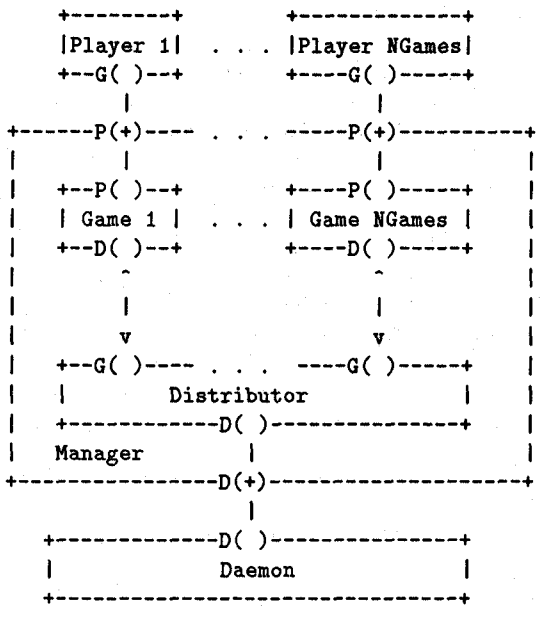


Figure 8.1: Architecture of the Daemon Game in Estelle

8.2.7 Counting of 'Bump' Signals (Clause 8.1)

8.2.7.1 Deficiency

In deciding whether a player wins or loses, is it necessary to use the count of **Bump** signals since the system started or since that game started?

8.2.7.2 Resolution

The intention was to count the number of **Bump** signals since a game started.

8.3 Estelle Description

8.3.1 Architecture of the Formal Description

Figure 8.1 shows the architecture of the Daemon Game description in Estelle using an explicit daemon; a description without an explicit daemon is given in 8.3.4. Interaction points are indicated by parentheses and labelled by their names. Those marked with a + symbol are bound both by a **connect** and an **attach** and thus logically continue the path of interactions to or from a child module. The specification of **ManagerBody** illustrates the dynamic structuring capabilities of Estelle. Declaring the **Manager** module to be a **systemprocess** is a way to guarantee fairness in the service provided to the players, because the **Game** modules will be synchronised.

8.3.2 Explanation of Approach

Definitions of bodies for the **Player** and the **Daemon** are left unspecified, as these are not relevant to the description: each may behave in any way consistent with the corresponding channel definition.

The **Manager** handles the instantiation and eventual removal of game instances, as the players initiate and terminate them. Once it has established the game, the **Manager attaches** the **Gameserver** channel to the newly established games, so that the remaining interactions between the player and the game require no mediation on the part of the **Manager**. The shared variable **Done** is used by the game to indicate to its manager-parent that its user has finished play, so the game should be removed (**released**). This approach is one of several possible; it was chosen to show the use of shared variables.

The game modules simply implement the rules of the game as given in the informal description. The **Distributor** module distributes the daemon's **Bump** to each of the games.

8.3.3 Formal Description

```
specification Daemongame;

const NGames = any integer;

channel Daemonserver (User, Provider);
    by Provider:
        Bump;

channel Gameserver (Player, Machine);
    by Player:
        Probe;   { Player takes a turn }
        Result;  { Player requests score }
        Newgame; { Player initiates game }
        Endgame; { Player terminates game }
    by Machine:
        Win;      { tells Player he won }
        Lose;     { tells Player he lost }
        Score (nwon: integer); { in response to
          Result, tells Player his score }

module Daemon systemprocess;
    ip D: Daemonserver (Provider)
            individual queue;
end; { Daemon }

body DaemonBody for Daemon; external;

module Player systemprocess;
    ip G: Gameserver (Player)
            individual queue;
end; { Player }

body PlayerBody for Player; external;

module Manager systemprocess;
    ip P: array [1..NGames] of
            Gameserver (Machine)
              common queue;
          D: Daemonserver (user)
              common queue;
end; { Manager }

body ManagerBody for Manager;
```

```
module Distributor process;
    ip  G: array [1..NGames] of
             Daemonserver (provider)
               common queue;
        D: Daemonserver (user)
             common queue;
end; { Distributor }

body DistributorBody for Distributor;
trans
    when D.Bump
        begin
        { distribute bump to all games }
            all i: 1 .. NGames do
              output G[i].Bump
        end;
end; { DistributorBody }


module Game process;
    ip P: Gameserver (machine) common queue;
       D: Daemonserver (user) common queue;
    export
        Done: boolean;
end; { Game }

body GameBody for Game;
    var NCorrect: integer;
    state EVEN, ODD; { records parity of
      bumps }
    stateset EITHER = [EVEN, ODD];

    initialize
        to EVEN
            begin
                NCorrect := 0;
                Done := false;
            end;

    trans
        { *** Player makes guess *** }
        when P.Probe
            from EVEN to EVEN
                begin
                    NCorrect := NCorrect - 1;
                    output P.Lose
                end;
            from ODD to ODD
                begin
                    NCorrect := NCorrect + 1;
                    output P.Win
                end;

        { *** Player wants score *** }
        when P.Result
            from EITHER to same
                begin
                    output
                      P.Score(NCorrect)
                end;

        { *** Player is done *** }
        when P.Endgame
            from EITHER to same
                begin
                    Done := true
                end;

        { *** Player requests new game *** }
        when P.Newgame
            from EITHER to same
                begin
                { ignore Player's error }
                end;

        { *** Daemon bumped *** }
        when D.Bump
            from EVEN to ODD
                begin
                end;
            from ODD to EVEN
                begin
                end;
end; { GameBody }

{ Actual Manager description begins here }
modvar
    GameInstance: Game;
    DistributorInstance: Distributor;

state MANAGING;

initialize
    to MANAGING
        begin
            init DistributorInstance
              with DistributorBody;
            attach D
              to DistributorInstance.D;
        end;

trans
    any GameNumber: 1..NGames do

        { *** Player requests game *** }
        when P[GameNumber].Newgame
            begin
                init GameInstance
                  with GameBody;
                attach P[GameNumber]
                  to GameInstance.P;
                connect GameInstance.D
                  to DistributorInstance.G
                    [GameNumber];
            end;

        { *** Ignore Player's errors *** }
        when P[GameNumber].Probe
            begin
            end;
        when P[GameNumber].Result
```

```
                begin
                end;
            when P[GameNumber].Endgame
                begin
                end;
    trans
        { *** Clean up after game *** }
        provided exist GameBody: Game
          suchthat GameBody.Done
            begin
                all GameBody: Game do
                    if GameBody.Done then
                      release GameBody
            end;
end; { Manager }

{ here is the body of the specification
    itself }
modvar
    DaemonInstance:  Daemon;
    ManagerInstance: Manager;
    PlayerInstance:  array [1..NGames]
                       of Player;

initialize
    begin
        init DaemonInstance
          with DaemonBody;
        init ManagerInstance
   with ManagerBody;
        all i: 1 .. NGames do
            begin
                init PlayerInstance[i]
                  with PlayerBody;
                connect ManagerInstance.P[i]
                  to PlayerInstance[i].G
            end;
        connect DaemonInstance.D
          to ManagerInstance.D;
    end;
end. { specification Daemongame }
```

8.3.4 Alternative Formal Description

This alternative approach of specifying the Daemon Game avoids explicit representation of the daemon. The description given earlier was written to reflect the informal description more naturally. However, it was recognized that there was no way a player could distinguish between a system that had a central daemon and a system where the effect of the daemon was purely non-determinism. The architecture of the alternative description without the daemon is shown in Figure 8.2.

```
specification Daemongame;

const NGames = any integer;

channel Gameserver (Player, Machine);
    by Player:
        Probe;  { Player takes a turn }
```

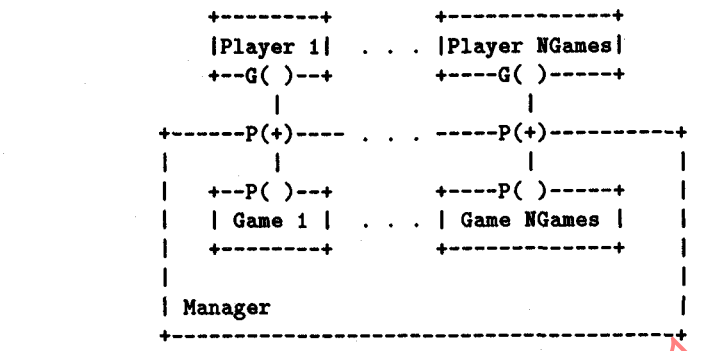


Figure 8.2: Alternative Architecture of the Daemon Game in Estelle

```
        Result;  { Player requests score }
        Newgame; { Player initiates game }
        Endgame; { Player terminates game }
    by Machine:
        Win;     { tells Player he won }
        Lose;    { tells Player he lost }
        Score (nwon: integer); { in response to
          Result, tells Player his score }

module Player systemprocess;
    ip G: Gameserver (Player) individual queue;
end; { Player }

body PlayerBody for Player; external;

module Manager systemprocess;
    ip P: array [1..NGames] of
            Gameserver (Machine) common queue;
end; { Manager }

body ManagerBody for Manager;

    module Game process;
        ip P: Gameserver (machine) common queue;
        export
            Done: boolean;
    end; { Game }

    body GameBody for Game;
        var NCorrect: integer;
        state EVENorODD; { records parity of
          bumps }

        initialize
            to EVENorODD
                begin
                    NCorrect := 0;
                    Done := false;
                end;

        trans
            { *** Player makes guess *** }
            when P.Probe
```

```
            begin
                NCorrect := NCorrect - 1;
                output P.Lose
            end;

        when P.Probe
            begin
                NCorrect := NCorrect + 1;
                output P.Win
            end;

        { *** Player wants score *** }
        when P.Result
            begin
                output P.Score(NCorrect)
            end;

        { *** Player is done *** }
        when P.Endgame
            begin
                Done := true
            end;

        { *** Player requests new game *** }
        when P.Newgame
            begin
            { ignore Player's error }
            end;

end; { GameBody }

{ Actual Manager description begins here }
modvar
    GameInstance: Game;

state MANAGING;

initialize
    to MANAGING
        begin
            init DistributorInstance
              with DistributorBody;
            attach D
              to DistributorInstance.D;
        end;

trans
    any GameNumber: 1..NGames do

        { *** Player requests game *** }
        when P[GameNumber].Newgame
            begin
                init GameInstance
                  with GameBody;
                attach P[GameNumber]
                  to GameInstance.P;
            end;

        { *** Ignore Player's errors *** }
        when P[GameNumber].Probe
            begin
                end;
            when P[GameNumber].Result
                begin
                end;
            when P[GameNumber].Endgame
                begin
                end;
    trans
        { *** Clean up after game *** }
        provided exist GameBody: Game
          suchthat GameBody.Done
            begin
                all GameBody: Game do
                    if GameBody.Done then
                      release GameBody
            end;
end; { Manager }

{ here is the body of the specification
    itself }
modvar
    ManagerInstance: Manager;
    PlayerInstance:  array [1..NGames] of
                       Player;

initialize
    begin
        init ManagerInstance
          with ManagerBody;
        all i: 1 .. NGames do
            begin
                init PlayerInstance[i]
                  with PlayerBody;
                connect ManagerInstance.P[i]
                  to PlayerInstance[i].G
            end;
        connect DaemonInstance.D
          to ManagerInstance.D;
    end;
end. { specification Daemongame }
```

8.3.5 Subjective Assessment

The Daemon Game was originally invented as a graded series of examples, each more complex than the next, to explain Estelle. In its original, simplest version, the game had no beginning and no end: it allowed one player, and it did not report a score. In this case, it is unnecessary to have the complex structure given here since there are only empty daemon and player modules, and a game module that has states and no variables. Each version of the game in the series forced the use of more complex Estelle constructs until the most complete version of the game (approximating the one given here) made use of a fairly large subset of Estelle. The informal description of the game given here was initially written by augmenting the old, original informal description; perhaps that affected some of the design choices.

8.4 LOTOS Description

```
(*-----------------------------------------
```

8.4.1 Architecture of the Formal Description

The top-level structure of the LOTOS description is as follows:

8.4.1.1 Gates

P for all communication between players and the system; interactions are tagged with the identifier of a game in order to distinguish them

D an internal gate for communication between the daemon and the games; a **Bump** signal is represented purely by synchronisation at **D**, i.e. there is no **Bump** value in the event

8.4.1.2 Data Types

Identifier for distinguishing games

IdentifierSet for indicating which games may be used

Integer for scoring

Signal for interactions between players and the system

8.4.1.3 Processes

System for explaining the top-level specification behaviour; this is decomposed into the independent constraints on permitted games

NoGame for describing the behaviour of a game which is not current (i.e. not logged into)

Game for describing the behaviour of a current game (i.e. logged into)

Daemon for describing the behaviour of the daemon

8.4.2 Explanation of Approach

The major decision taken in the writing of the description was whether to explicitly represent the daemon. In the following description, the daemon is explicitly represented as a process which interacts with game processes. The daemon process is responsible for generating **Bump** signals. The description was written this way in order to reflect the informal description more naturally. However, the philosophy of LOTOS is to describe only *observable* behaviour, so this style of description is unnatural in LOTOS. An alternative description without an explicit daemon is therefore given in 8.4.4.

8.4.3 Formal Description

The whole description of the system is parameterised by the gate at which external communication occurs with players (**P**), and by the set of game identifiers which may be used (**ids**).

```
-----------------------------------------------*)

specification Daemongame [P]
 (ids : IdSetSort) : noexit

  library
    Boolean, Set       (* use standard library *)
  endlib

(*-----------------------------------------------
```

The following type defines game identifiers. The only formal property which identifiers have is that they are distinct. This is explained by giving a base value (**BaseId**) and an operation for reaching **all** other identifier values (**NextId**). Equality (**eq**) and inequality (**ne**) are defined for game identifiers.

```
-----------------------------------------------*)

  type IdentifierType is Boolean
    sorts IdSort
    opns
      BaseId      : -> IdSort
      NextId      : IdSort -> IdSort
      _eq_, _ne_ : IdSort, IdSort -> Bool
    eqns
     forall Id, Id1, Id2 : IdSort
      ofsort Bool
        BaseId eq BaseId = true;
        BaseId eq NextId (Id) = false;
        NextId (Id) eq BaseId = false;
        NextId (Id1) eq NextId (Id2) =
          Id1 eq Id2;
        Id1 ne Id2 = not (Id1 eq Id2)
  endtype

(*-----------------------------------------------
```

The following type renames the standard library data type **Set**, still with formal sorts **Element** and **FBool**

```
-----------------------------------------------*)

  type IdentifierSetFormalType is Set renamedby
    sortnames IdSetSort for Set
  endtype

(*-----------------------------------------------
```

The following type defines sets of game identifiers as an actualisation of the parameterised type **IdentifierSetFormalType**. A set of game identifiers is a parameter to the overall description.

```
-----------------------------------------------*)

  type IdentifierSetType is
   IdentifierSetFormalType
    actualizedby IdentifierType, Boolean using
      sortnames
        IdSort for Element
        Bool for FBool
  endtype

(*-----------------------------------------------
```

The following type defines the integers (..., -1, 0, 1, ...) in terms of a zero value, an 'add one' operation (**inc**), and a 'subtract one' operation (**dec**).

```
-----------------------------------------------*)

  type IntegerType is
    sorts IntSort
    opns
     0        : -> IntSort
     inc, dec : IntSort -> IntSort
    eqns
     forall n : IntSort
      ofsort IntSort
        inc (dec (n)) = n;
        dec (inc (n)) = n
  endtype

(*-----------------------------------------------
```

The following type defines the signals between the players and the system. With the exception of **Score**, these signals are constants.

```
-----------------------------------------------*)

  type SignalType is IntegerType
    sorts SigSort
    opns
      Newgame, Endgame, Probe,  Win, Lose,
       Result : -> SigSort
      Score : IntSort -> SigSort
  endtype

(*-----------------------------------------------
```

The following behaviour expression specifies the entire game. It is parameterised by the given gate and set of identifiers. The internal gate **D** is used for communication between the daemon and system processes.

```
-----------------------------------------------*)

  behaviour
    hide D in
      System [P, D] (ids) |[D]| Daemon [D]

  where
(*-----------------------------------------------
```

The following process specifies the overall behaviour of the system. It sets up games independently in parallel one by one, assigning each of them a unique identifier. However, the games must all synchronise on signals from the daemon at gate **D**. The process is non-terminating, since all the games are.

```
-----------------------------------------------*)

process System [P, D]
 (ids : IdSetSort) : noexit :=
  choice id : IdSort []
    [(Card (ids) eq Succ (0)) and
     (id IsIn ids)] ->              (* one id *)
      NoGame [P, D] (id)
   []
    [(Card (ids) gt Succ (0)) and
     (id IsIn ids)] ->          (* several ids *)
      (
        NoGame [P, D] (id)
       |[D]|
        System [P, D] (Remove (id, ids))
      )

  where

(*-----------------------------------------------
```

The following process specifies the behaviour of a game when it is not current (i.e. logged into). The process is non-terminating, since on completion of a game it offers to start a new game. Unwanted signals from the player or the daemon are discarded while a game is not in progress.

```
-----------------------------------------------*)

  process NoGame [P, D]
   (id : IdSort) : noexit :=
    P ! id ! Newgame;
    (                    (* score 0, even Bumps *)
      Game [P, D]
       (id, 0 of IntSort, false)
    >>
      NoGame [P, D] (id)
    )
  []
    P ! id ! Probe; NoGame [P, D] (id)
  []
    P ! id ! Result; NoGame [P, D] (id)
  []
    P ! id ! Endgame; NoGame [P, D] (id)
  []
    D;                           (* Bump signal *)
    NoGame [P, D] (id)

    where

(*-----------------------------------------------
```

The following process specifies the behaviour of a current game. Only the parity of the number of **Bump** signals is relevant, so the actual number of the signals is not stored. The process is entered after **Newgame**, and terminates once **Endgame** is received.

```
-----------------------------------------------*)

      process Game [P, D]
```

```
      (id : IdSort, total : IntSort, odd : Bool)
      : exit :=
       P ! id ! Newgame;
       Game [P, D] (id, total, odd)
     []
       P ! id ! Probe;
       (
         [odd] ->
           P ! id ! Win;
           Game [P, D] (id, inc (total), odd)
       []
         [not (odd)] ->
           P ! id ! Lose;
           Game [P, D] (id, dec (total), odd)
       )
     []
       P ! id ! Result;
       P ! id ! Score (total);
       Game [P, D] (id, total, odd)
     []
       P ! id ! Endgame; exit
     []
       D;                          (* Bump signal *)
       Game [P, D] (id, total, not (odd))
   endproc (* Game *)

endproc (* NoGame *)

endproc (* System *)

(*-------------------------------------------
```

The following process specifies the behaviour of the daemon. It simply generates an endless series of event offers at the **D** gate, corresponding to **Bump** signals.

```
-------------------------------------------*)

process Daemon [D] : noexit :=
  D; Daemon [D]
endproc (* Daemon *)

endspec (* Daemongame *)

(*-------------------------------------------
```

8.4.4 Alternative Formal Description

It was recognised that there was no way a player could distinguish between a system that had a central daemon and a system that had one independent daemon per game process. Since there is no concept of absolute time or simultaneity in LOTOS, a description could not differentiate between the behaviour of these two systems. If two players sent **Probe** at almost the same time and one received **Win** while the other received **Lose**, they would conclude that the system had internally generated **Bump** in between the two signals. This would be true no matter how close in time the two **Probe** signals were. Since the two **Probe** signals could never be simultaneous in LOTOS, and could not be determined to be simultaneous in the real world, the players could not observe whether there were one or many daemons in the system. This illustrates a deep difference found between some FDTs. FDTs such as LOTOS model concurrency by interleaving of events, whereas others model simultaneity using the concept of 'true concurrency'.

An alternative formal description has therefore been provided with one daemon per game process. However, since the daemon is simply a source of non-determinism, it can be dispensed with altogether in the LOTOS description. The manifestation of the daemon is that a player receives a **Win** or **Lose** signal after a **Probe**. It is therefore not necessary to model the **Bump** signals (which are, after all, invisible from the outside), nor to count whether an odd or even number has occurred. Such non-determinism is simply hidden as an internal event in the LOTOS description. This description therefore dispenses with the internal gate **D** and the **Daemon** process.

```
-------------------------------------------*)

specification Daemongame [P]
 (ids : IdSetSort) : noexit

  library
    Boolean, Set       (* use standard library *)
  endlib

(*-------------------------------------------
```

The following type defines game identifiers. The only formal property which identifiers have is that they are distinct. This is explained by giving a base value (**BaseId**) and an operation for reaching all other identifier values (**NextId**). Equality (**eq**) and inequality (**ne**) are defined for game identifiers.

```
-------------------------------------------*)

  type IdentifierType is Boolean
    sorts IdSort
    opns
      BaseId      : -> IdSort
      NextId      : IdSort -> IdSort
      _eq_, _ne_ : IdSort, IdSort -> Bool
    eqns
     forall Id, Id1, Id2 : IdSort
      ofsort Bool
        BaseId eq BaseId = true;
        BaseId eq NextId (Id) = false;
        NextId (Id) eq BaseId = false;
        NextId (Id1) eq NextId (Id2) =
          Id1 eq Id2;
        Id1 ne Id2 = not (Id1 eq Id2)
  endtype

(*-------------------------------------------
```

The following type renames the standard library data type **Set**, still with formal sorts **Element** and **FBool**

```
-----------------------------------------------*)

  type IdentifierSetFormalType is Set renamedby
    sortnames IdSetSort for Set
  endtype

(*-----------------------------------------------
```

The following type defines sets of game identifiers as an actualisation of the parameterised type **IdentifierSetFormalType**. A set of game identifiers is a parameter to the overall description.

```
-----------------------------------------------*)

  type IdentifierSetType is
   IdentifierSetFormalType
    actualizedby IdentifierType, Boolean using
      sortnames
        IdSort for Element
        Bool for FBool
  endtype

(*-----------------------------------------------
```

The following type defines the integers (..., -1, 0, 1, ...) in terms of a zero value, an 'add one' operation (**inc**), and a 'subtract one' operation (**dec**).

```
-----------------------------------------------*)

  type IntegerType is
    sorts IntSort
    opns
      0        : -> IntSort
      inc, dec : IntSort -> IntSort
    eqns forall n : IntSort
      ofsort IntSort
        inc (dec (n)) = n;
        dec (inc (n)) = n
  endtype

(*-----------------------------------------------
```

The following type defines the signals between the players and the system. With the exception of **Score**, these signals are constants.

```
-----------------------------------------------*)

  type SignalType is IntegerType
    sorts SigSort
    opns
      Newgame, Endgame, Probe,  Win, Lose,
      Result : -> SigSort
      Score : IntSort -> SigSort
  endtype

(*-----------------------------------------------
```

The following behaviour expression specifies the entire game. It is parameterised by the given gate and set of identifiers.

```
-----------------------------------------------*)

  behaviour System [P] (ids)

  where

(*-----------------------------------------------
```

The following process specifies the overall behaviour of the system. It sets up games independently in parallel one by one, assigning each of them a unique identifier. The process is non-terminating, since all the games are.

```
-----------------------------------------------*)

  process System [P]
   (ids : IdSetSort) : noexit :=
    choice id : IdSort []
      [id IsIn ids] ->
        (
          NoGame [P] (id)
          |||
          System [P] (Remove (id, ids))
        )

    where

(*-----------------------------------------------
```

The following process specifies the behaviour of a game when it is not current (i.e. logged into). The process is non-terminating, since on completion of a game it offers to start a new game. Unwanted signals from the player are discarded while a game is not in progress.

```
-----------------------------------------------*)

    process NoGame [P]
     (id : IdSort) : noexit :=
      P ! id ! Newgame;
      (                                (* score 0 *)
        Game [P] (id, 0 of IntSort)
      >>
        NoGame [P] (id)
      )
    []
      P ! id ! Probe; NoGame [P] (id)
    []
      P ! id ! Result; NoGame [P] (id)
    []
      P ! id ! Endgame; NoGame [P] (id)

      where

(*-----------------------------------------------
```

The following process specifies the behaviour of a current game. The **Bump** actions of the daemon are not explicitly modelled since their external effect is only non-determinism. For this reason, there is no central daemon which sends **Bump** signals to game processes. The process is entered after **Newgame**, and terminates once **Endgame** is received.

```
-----------------------------------------------*)

      process Game [P]
       (id : IdSort, total : IntSort)
        : exit :=
        P ! id ! Newgame; Game [P] (id, total)
      []
        P ! id ! Probe;
        (
          i ; P ! id ! Win;
          Game [P] (id, inc (total))
        []
          i ; P ! id ! Lose;
          Game [P] (id, dec (total))
        )
      []
        P ! id ! Result; P ! id ! Score (total);
        Game [P] (id, total)
      []
        P ! id ! Endgame; exit
      endproc (* Game *)

    endproc (* NoGame *)

  endproc (* System *)

endspec (* Daemongame *)

(*-----------------------------------------------
```

8.4.5 Subjective Assessment

The LOTOS description shows a clear separation between static aspects (the data typing) and dynamic aspects (the behaviour). The data typing draws on already established data types, which are defined in an Annex to the LOTOS Standard. The description of the data types concerns itself with implementation-independent aspects; for example, scores are described as mathematical integers, not bit patterns.

The behaviour description illustrates the 'constraint-oriented' style in which LOTOS can be used. In this style, behaviour is decomposed into largely separate constraints which are then combined using the appropriate LOTOS operators. In this specification, the overall system behaviour is expressed in terms of game behaviours. These in turn are expressed in terms of the login/logout behaviour and game-playing behaviour. The data typing also shows a similar modularity, whereby more complex data types (e.g. **IdentifierSetType**) are built out of simpler ones.

The fact that there should be no central daemon process, or for that matter any daemon processes at all, reflects the emphasis in LOTOS on observational behaviour. A well-written LOTOS description will focus on the sequences of interactions which can be externally observed, and will avoid unnecessary and implementation-dependent detail. To this extent, the informal description is weak because it describes a particular mechanism for implementing the system, not the externally required behaviour. The informal description is an example of over-specification, which must be carefully avoided in International Standards.

```
-----------------------------------------------*)
```

8.5 SDL Description

8.5.1 Architecture of the Formal Description

The **Daemongame** system contains only one block, called **Blockgame**. This block has two process types, called **Monitor** and **Game**. Of **Monitor** there is one single process that is created at the same time when the system is created (initial process). Of **Game** a dynamic process is created for each player.

A player is regarded as a process in the environment of the system. Each process in SDL is given a unique identity (of the sort **Pid**), and each signal carries the identity of the sending process. Thus, when a player logs in by the signal **Newgame**, his identity is known to the system. In the system a **Game** process is created for him. The process 'presents itself' by sending the signal **Gameid** to the player and takes care of the rest of the game session.

The **Monitor** process has the task of creating **Game** processes and distributing **Bump** signals to all the **Game** processes.

Note that an SDL system may ignore some possible sequences of signals coming from the environment, for instance a **Probe** signal coming from a player who has not logged in is ignored by **Daemongame**. In other words, the allowed behaviour of the environment is specified indirectly by the SDL system description.

8.5.2 Explanation of Approach

The architectural approach above is rather natural. A unique **Monitor** process is necessary to receive signals from the environment (**Newgame** and **Bump**) which cannot be addressed to a specific process (these signals are sent without an address).

The relation between the **Monitor** and **Game** processes could be simplified by addressing the **Endgame** signal to **Monitor**, which would then update its record of players and **Game** processes, passing the signal to the **Game** process in question. However, this would require a coupling between a player and the corresponding **Game** process in the **Monitor** process.

A main feature of this game is the non-deterministic reply to a **Probe** signal. Since an SDL system behaves in a deterministic way, this non-determinism must be modelled by signals sent (from a daemon in the environment) to the system.

8.5.3 Formal Description

The SDL description is given in both concrete representations, SDL/GR and SDL/PR. In both representations, use is made of **remote definitions**, allowing the SDL description to be structured as an overview part and a set of remote definitions giving the detailed description. Actually it is only the overview part which is specific for a representation; the remote definitions in SDL/GR and SDL/PR can be used interchangeably (e.g. a **remote block definition** (in SDL/PR) can be 'called' from a system diagram (in SDL/GR).

The structure of the **Daemongame** system is described in SDL/GR by a **system diagram** and a **remote block diagram**. The behaviour of the system is described in SDL/GR by a **remote process diagram** for each process type.

In SDL/PR there is a corresponding **system definition**, **block definition** and **process definition**. However, the **macro definition** for **Datatypedef** is common to both SDL/GR and SDL/PR.

The following SDL description is in SDL/PR:

```
/*********************************************/
/**/            SYSTEM Daemongame;            /**/
/*********************************************/

  SIGNAL
    Newgame, Probe, Result, Endgame, Gameid,
    Win, Lose, Score(Integer), Bump;

  CHANNEL Gameserver.in
    FROM ENV TO Blockgame
    WITH Newgame, Probe, Result, Endgame;
  ENDCHANNEL Gameserver.in;

  CHANNEL Gameserver.out
    FROM Blockgame TO ENV
    WITH Gameid, Win, Lose, Score;
  ENDCHANNEL Gameserver.out;

  CHANNEL Daemonserver
    FROM ENV TO Blockgame
    WITH Bump;
  ENDCHANNEL Daemonserver;

  BLOCK Blockgame REFERENCED;

ENDSYSTEM Daemongame;

/*********************************************/
/**/              BLOCK Blockgame;            /**/
/*********************************************/

  SIGNAL Gameover, Gameoverack;

  SIGNALROUTE R1
    FROM ENV TO Monitor
    WITH Newgame;

  SIGNALROUTE R2
    FROM ENV TO Game
    WITH Probe, Result, Endgame;

  SIGNALROUTE R3
    FROM Game TO ENV
    WITH Gameid, Win, Lose, Score;

  SIGNALROUTE R4
    FROM ENV TO Monitor
    WITH Bump;

  SIGNALROUTE R5
    FROM Monitor TO Game
    WITH Bump, Gameoverack;
    FROM Game TO Monitor
    WITH Gameover;

  CONNECT Gameserver.in AND R1, R2;
  CONNECT Gameserver.out AND R3;
  CONNECT Daemonserver AND R4;

  PROCESS Monitor (1, 1) REFERENCED;
  PROCESS Game (0, ) REFERENCED;

ENDBLOCK Blockgame;

/*********************************************/
/**/          PROCESS Monitor (1, 1);         /**/
/*********************************************/

/* This process registers new players, creates a Game process for each of them, and distributes Bump signals to all the Game processes. If a registered player tries to 'log in', then no action is taken. Note that no record is kept for the coupling between a player and the corresponding Game process. */

  DCL player PId; /* The identity of the
        corresponding player is stored
          temporarily in this variable */

  DCL userset, /* Keeps record of the players */
      gameset, /* Keeps record of the Game
        processes */
      copygameset Pidset;

  MACRO Datatypedef;

  START;
    NEXTSTATE Idle;

  STATE Idle;
    INPUT Newgame;
      DECISION (SENDER IN userset);
        (True): NEXTSTATE -;
        (False): CREATE Game(SENDER);
                 TASK gameset :=
                   Incl(OFFSPRING, gameset);
                 TASK userset :=
                   (SENDER, userset);
                 NEXTSTATE -;
      ENDDECISION
```

```
    INPUT Gameover(player);
      TASK gameset := Del(SENDER, gameset);
      TASK userset := Del(player, userset);
      OUTPUT Gameoverack TO SENDER;
      NEXTSTATE -;
    INPUT Bump;
      TASK copygameset := gameset;
      1: DECISION (copygameset = empty);
           (True): NEXTSTATE -;
           (False): OUTPUT Bump TO
                      take(copygameset);
                    TASK copygameset :=
                      Del(take(copygameset),
                        copygameset);
                    JOIN 1;
         ENDDECISION

ENDPROCESS Monitor;

/*********************************************/
/**/            PROCESS Game (0, );          /**/
/*********************************************/
```

/* This process is created for a new player, and takes care of the rest of the game session. The identity of the player is given as the formal parameter **player**. When a player 'logs out', the **Monitor** process must be informed (in order not to send **Bump** signals to the process) before the process can terminate. */

```
  FPAR
    player PId;

  DCL
    count Integer := 0;  /* Counter to keep track
      of the score */

  START;
    OUTPUT Gameid TO player;
    NEXTSTATE Even;

  STATE Even;
    INPUT Probe;
      OUTPUT Lose TO player;
      TASK count := count - 1;
      NEXTSTATE -;
    INPUT Bump;
      NEXTSTATE Odd;

  STATE Odd;
    INPUT Bump;
      NEXTSTATE Even;
    INPUT Probe;
      OUTPUT Win TO player;
      TASK count := count + 1;
      NEXTSTATE -;

  STATE *(Wait.for.ack);
    INPUT Result;
      OUTPUT Score(count) TO player;
      NEXTSTATE -;
    INPUT Endgame;
      OUTPUT Gameover(player);
      NEXTSTATE Wait.for.ack;

  STATE Wait.for.ack;
    INPUT Gameoverack;
      STOP;

ENDPROCESS Game;
```

The corresponding SDL/GR representation is given in figure 8.3.

8.5.4 Subjective Assessment

The SDL description is separated into a static description (represented by a **system diagram** and a **block diagram**) and a dynamic description (represented by **process diagrams**). The static description gives a clear structure of the SDL system, reducing its overall complexity, so that it can be studied one part at a time. This feature of SDL is reinforced by the use **remote definitions**. The graphical representation greatly improves the readability of the formal description.

SDL does not have constructs to express non-determinism. The use of state machines to express behaviour contributes to user friendliness, but at the same time this may lead to over-specification.

8.6 Assessment of the Application of the FDTs

It is remarkable that such an apparently simple example should result in so many different interpretations. It took several iterations among the authors of the informal and formal descriptions to determine exactly what the original intentions were. The conclusions from this small example are:

a) How difficult it is to be precise about even simple thing.

b) How easy it is to forget to specify all error cases. Failure to do so often results in problems of incompatibility between implementations of a complex description.

c) How easy it is to be unclear about the responsibilities of different parts of a system, and how these parts should view each other.

d) How easy it is to colour the description of a system with unnecessary implementation detail which excludes valid implementations.

These conclusions apply even more so to complex descriptions (e.g. International Standards). The application of the FDTs to this small example has shown how the writing of a formal description can identify ambiguities, errors, and over-specification.

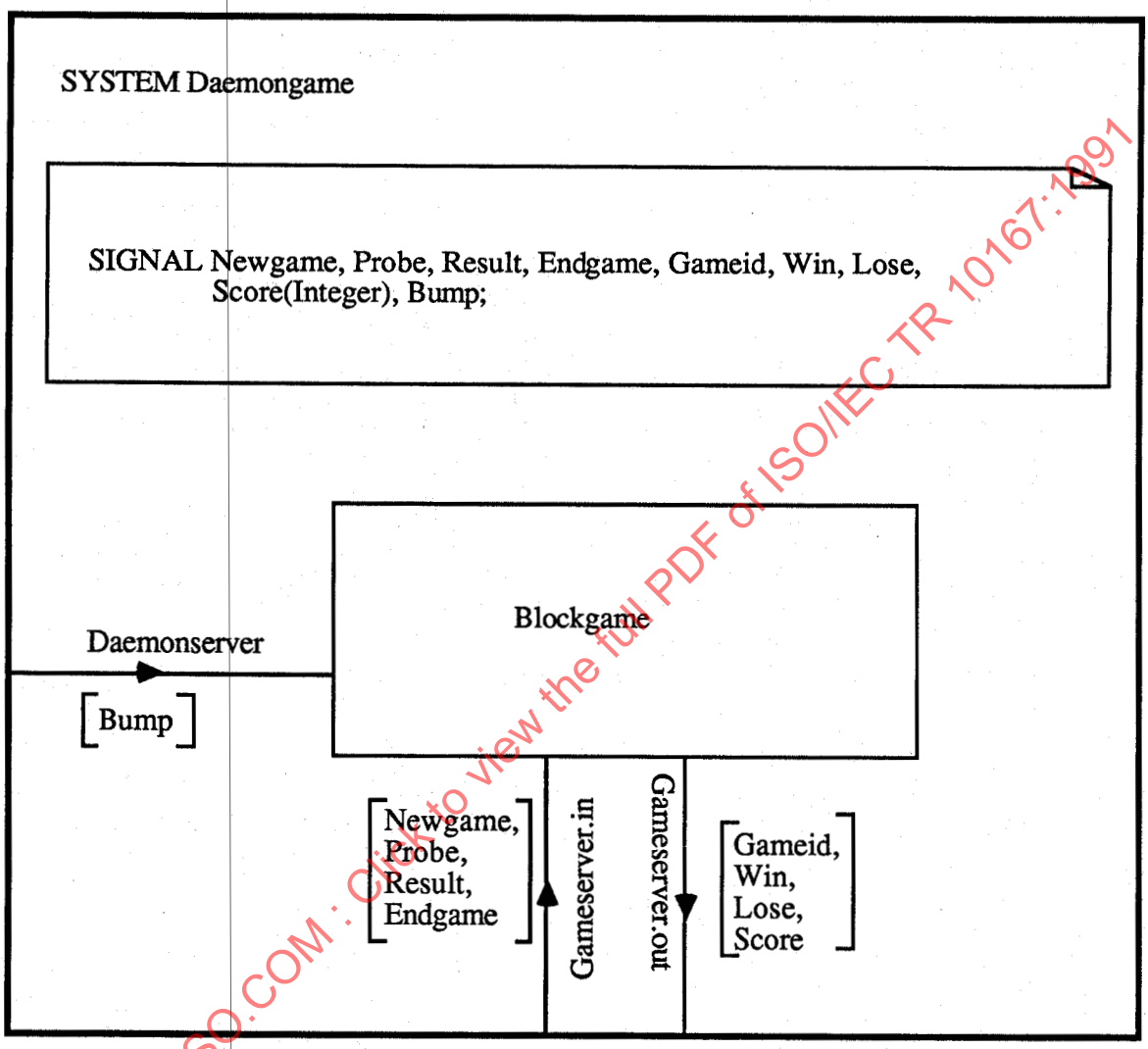
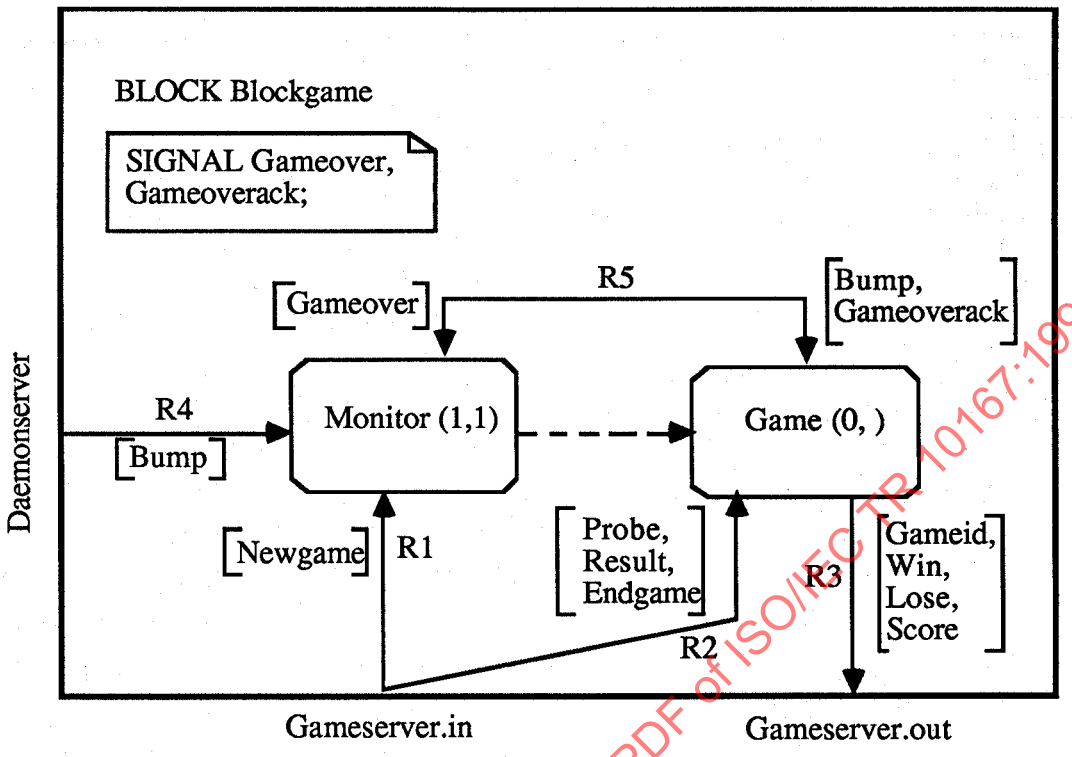


Figure 8.3: SDL Specification of Daemon Game




```
MACRODEFINITION Datatypedef

  NEWTYPE Pidset Powerset (PId)

    ADDING

      OPERATORS
        take! : Pidset PId -> PId;
        take  : Pidset -> PId;

      AXIOMS
        take(empty) == Error;
        take(Pidset) == take!(Pidset, null);
        take!(empty,PId) == Error;
        take!(Pidset,PId) == IF PId IN Pidset
                             then PId
                             else take!(Pidset,unique!(PId));
        /* The take operator returns an element of the Pidset. */

    DEFAULT empty;

  ENDNEWTYPE Pidset;

ENDMACRO Datatypedef;
```

Figure 8.3 *(continued)*

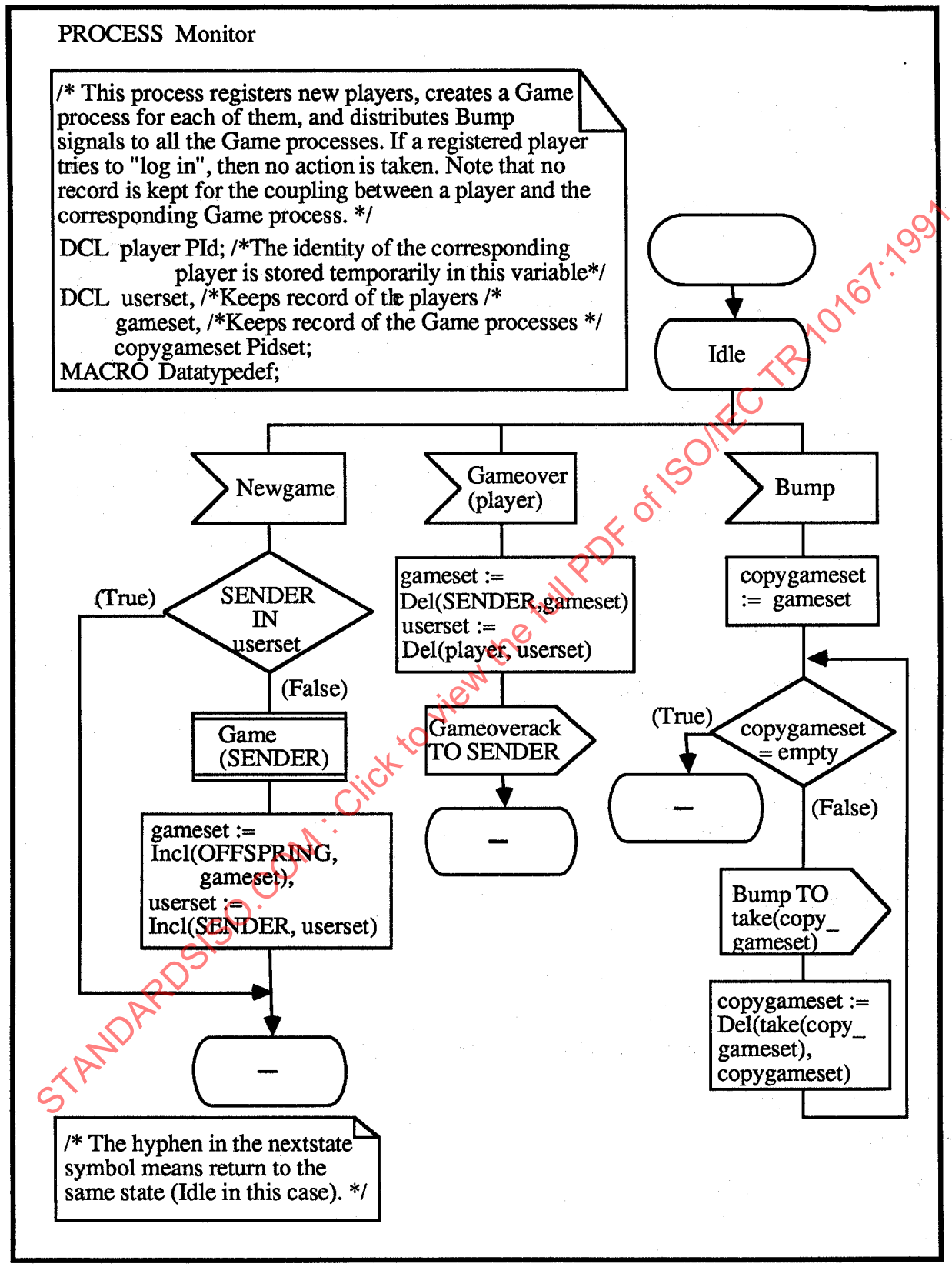


Figure 8.3 *(continued)*

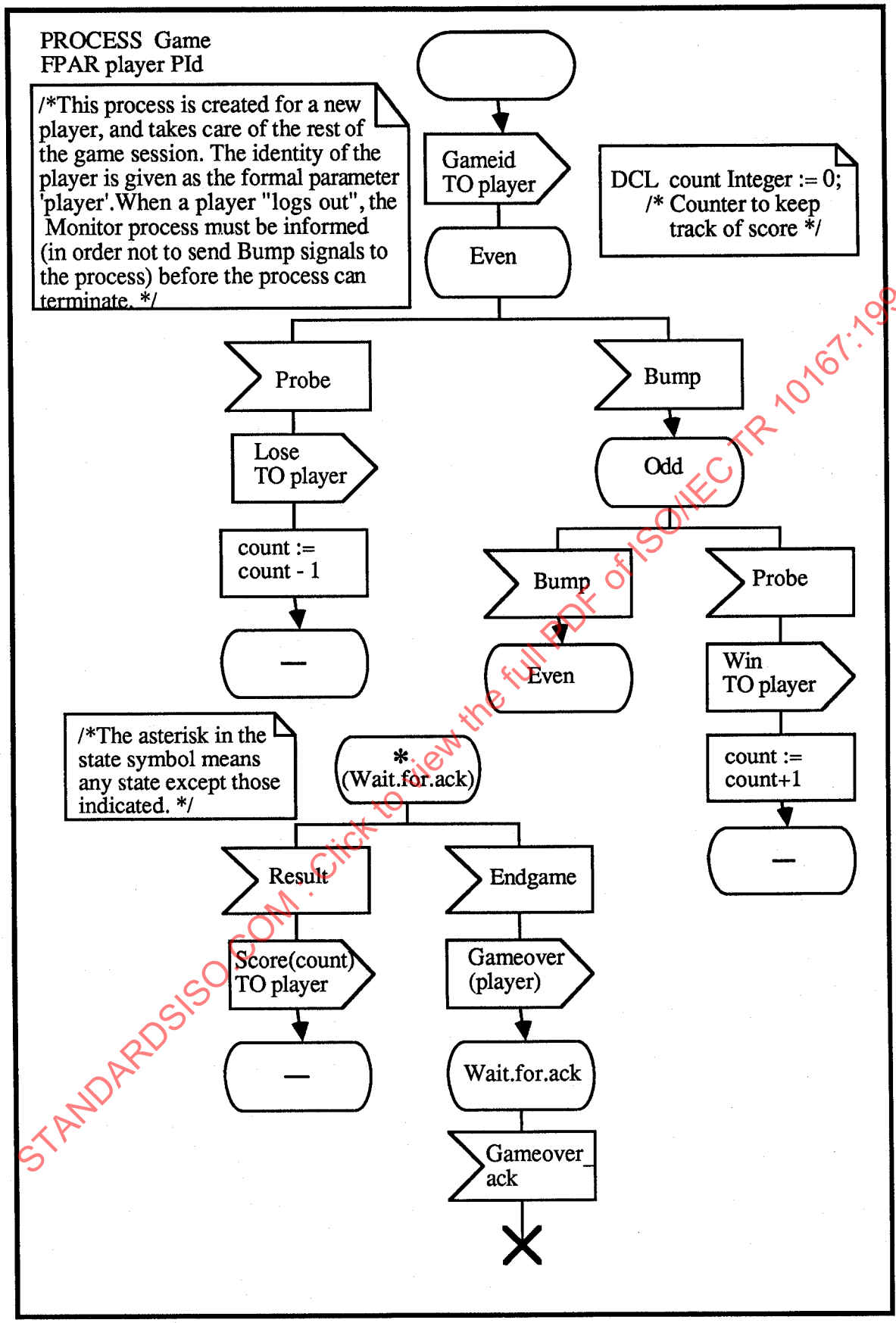


Figure 8.3 *(continued)*

9 Sliding Window Protocol Example

This illustrates an important flow-control and error recovery technique which is present in many real Protocols. In addition, it illustrates the description of a Protocol in relation to its underlying Service.

9.1 Informal Description

9.1.1 Overview

The following prose description was derived from the initial work by Stenning [Stenning 1976]. The description was produced from the narrative and Pascal-like programs in the paper.

The Sliding Window Protocol supports a unidirectional flow of data with a positive handshake on each transfer, and use of an acknowledgement window for flow control. The protocol operates over a medium which may lose, duplicate, or re-order messages. It is assumed that the corruption of messages can be reliably detected. Connection and disconnection procedures are not described by the protocol.

9.1.2 Sequence Numbering

The transmitter sends a sequence number with each message. A sequence number is unbounded and is incremented for each new message. The first message transmitted is given sequence number 1.

The receiver sends an Acknowledgement when it receives a message. The Acknowledgement carries a sequence number which refers to the last message successfully transferred to the receiving user. If an Acknowledgement has to be sent before a successful reception (e.g. the first message was corrupted), it is given sequence number 0.

9.1.3 Transmitter Behaviour

The transmitter maintains a window of sequence numbers as shown in Figure 9.1.

This gives the lowest sequence number for which an Acknowledgement is awaited, and the highest sequence number so far used. The window size is limited to the value **TWS**.

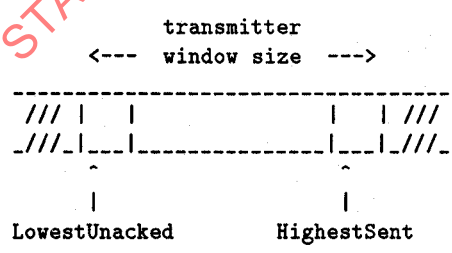


Figure 9.1: Transmitter Window Parameters

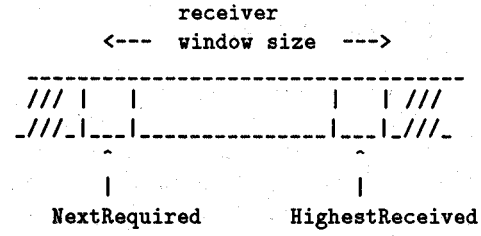


Figure 9.2: Receiver Window Parameters

The transmitter behaves initially as a) below, and then loops doing b), c) and d) where possible:

a) **LowestUnacked** is set to 1, **HighestSent** to 0 and **TWS** to an implementation-defined value (≥ 1).

b) If the current window size (i.e. $\mathbf{HighestSent} - \mathbf{LowestUnacked}$) is less than **TWS**, then a message with the next sequence number (i.e. $\mathbf{HighestSent}+1$) may be transmitted. In this case, **HighestSent** is incremented and a timer for that message is started.

c) If an Acknowledgement is received which is not corrupted and whose sequence number is not less than **LowestUnacked**, then all timers for messages up to and including that sequence number are cancelled. In this case, **LowestUnacked** is set to the sequence number following the acknowledged one.

d) If a time-out occurs, then the timers for all messages transmitted after the timed-out one are cancelled. All these timed-out messages are re-transmitted (in sequence, starting with the earliest) and have timers started for them.

9.1.4 Receiver Behaviour

The receiver maintains a window of sequence numbers as shown in Figure 9.2.

This gives the lowest sequence number which is awaited and the highest sequence number which has been received. The window size is limited to the value **RWS**.

The receiver behaves initially as a) below, and then loops doing b) and c) where possible.

a) **NextRequired** is initialised to 1, and **TWS** to an implementation-defined value (≥ 1)

b) If a message is received which is not corrupted, which has not already been received, and which lies within the maximum receive window (defined by **NextRequired** and **RWS**), then all messages from **NextRequired** up to but not including the first unreceived message are delivered to the receiving user. (There may be no such messages if there is a gap due to misordering). In this case, **NextRequired** is set to the sequence number of the next message to be delivered to the receiving User.

c) If a message is received under any circumstances, an Acknowledgement giving the last delivered sequence number (i.e. $\mathbf{NextRequired} - 1$) is returned.

9.2 Deficiencies in the Informal Description

9.2.1 Underlying Medium (Clause 9.1.1)

9.2.1.1 Deficiency

Is the description of the underlying medium an integral part of the description of the Sliding Window Protocol?

9.2.1.2 Resolution

The description of the underlying medium is provided as technical justification for the design of the protocol, and so is not an integral part of its description. However, it should be included so as to illustrate the relationship between a Protocol and its underlying Service.

9.2.2 Window Size (Clauses 9.1.3 and 9.1.4)

9.2.2.1 Deficiency

Do the Transmit and Receive Window Sizes (**TWS**, **RWS**) have to be the same? What should happen if these parameters are not greater than 0?

9.2.2.2 Resolution

The window sizes are intentionally allowed to be different. If the window size is not greater than 0, the protocol should simply fail to transmit ($\mathbf{TWS} \leq 0$) or receive ($\mathbf{RWS} \leq 0$) any messages.

9.2.3 Flow Control (Clause 9.1.4)

9.2.3.1 Deficiency

The informal description is unclear as to what 'delivery' means. Does it mean dispatch by the receiver to its user, or receipt by its user? These may not be the same if there is buffering or delay between the receiver and its user.

9.2.3.2 Resolution

Since the interface between the receiver and its user is an implementation-dependent matter, it is not reasonable to restrict the meaning of 'delivery' in the informal description. Similarly, the concept of 'delivery' in the formal descriptions depends on the most natural style in the FDT used.

9.2.4 Delivery of Corrupted Messages (Clause 9.1.1)

9.2.4.1 Deficiency

Does the medium deliver corrupted messages, or are they discarded within the medium?

9.2.4.2 Resolution

The medium was intended to deliver corrupted messages, and the Protocol to detect this by some unspecified means.

9.2.5 Value of Time-Out Period (Clause 9.1.3)

9.2.5.1 Deficiency

Is the time-out period fixed for all implementations, fixed for one implementation, or dynamically variable?

9.2.5.2 Resolution

It was the intention that the time-out period be left unspecified (i.e. to be specified at a lower level of description).

9.2.6 Consistent Use of NextRequired (Clause 9.1.4)

9.2.6.1 Deficiency

Is the 'next lowest sequence number which is awaited' the same as **NextRequired**?

9.2.6.2 Resolution

NextRequired should have been used consistently throughout the informal description.

9.2.7 Receive Window Size (Clause 9.1.4 a))

9.2.7.1 Deficiency

Should the receiver initialise **RWS**, or **TWS** as stated?

9.2.7.2 Resolution

The use of **TWS** was a typographical error: **RWS** was intended.

9.2.8 Sequence of Operations (Clauses 9.1.3 and 9.1.4)

9.2.8.1 Deficiency

Do the phrases 'b), c) and d)' and 'b) and c)' mean a sequence in time, or a set of operations which may be carried out in parallel?

9.2.8.2 Resolution

A sequence in time was intended.

9.2.9 Transmit Window Size (Clause 9.1.3)

9.2.9.1 Deficiency

The diagram and the definition of 'current window size' are inconsistent.

9.2.9.2 Resolution

The value '$\mathbf{HighestSent} - \mathbf{LowestUnacked} + 1$' should have been used for the current window size.

9.2.10 Receive Window Size (Clause 9.1.4 b))

9.2.10.1 Deficiency

Processing of a message is said to be allowed if its sequence number lies within the 'maximum receive window'. Is the upper bound of this (i.e. $\mathbf{NextRequired} + \mathbf{RWS}$)

included in this range?

9.2.10.2 Resolution

The upper bound is not included. The text should have read 'within the current receive window ($NextRequired + RWS - 1$)'.

9.2.11 Corruption of Messages (Clause 9.1.1)

9.2.11.1 Deficiency

Can Acknowledgements as well as Data messages be corrupted in the medium?

9.2.11.2 Resolution

The intention was that this could happen.

9.2.12 Transfer of Data and Acknowledgements (Clause 9.1.1)

9.2.12.1 Deficiency

Does the medium support 'Data' and 'Ack' Service Primitives, or does the Protocol have to encode this information in Protocol Data Units?

9.2.12.2 Resolution

The intention was that 'Data' and 'Ack' be distinguished by the medium.

9.2.13 Retransmission on Timeout (Clause 9.1.3 d))

9.2.13.1 Deficiency

It is unclear what 'all these timed-out messages' are; only one message has in fact timed out. The phrase might also mean all the messages following, but not including, the timed-out one.

9.2.13.2 Resolution

The intention was that the timed-out message and all later messages be retransmitted.

9.3 Estelle Description

9.3.1 Architecture of the Formal Descriptions

The architecture of the formal descriptions is shown in Figure 9.3. The Protocol description is found in 9.3.3 and the Medium description is found in 9.3.4. All the modules of the description are **systemprocesses**, and so run asynchronously. As these modules are not refined into submodules, the global behaviour would not change if they were designated **systemactivities**. The crucial point is that they are distinct systems. An explicit **Timer** module was chosen for two reasons:

a) it shows a way to manage timeouts that does not depend directly on the use of delay clauses, although an Estelle description of a timer module would obviously use these; and

b) it seemed to model more closely the informal requirements of the Protocol.

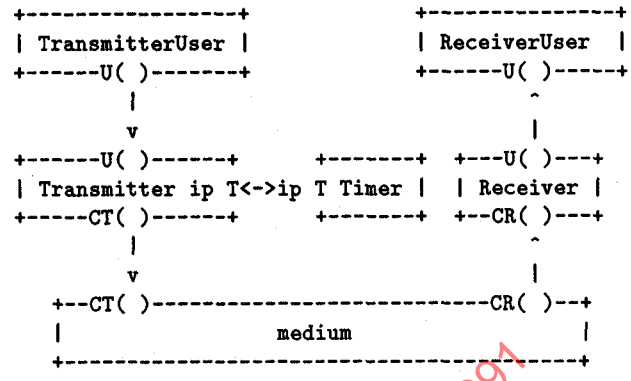


Figure 9.3: Architecture of the Sliding Window Protocol in Estelle

9.3.2 Explanation of Approach

The Sliding Window Protocol is unusual in several ways. For example, its data flow is uni-directional, leading to a few peculiarities in the architecture of the formal description, such as having distinct and different modules acting as peers.

Unless cancelled, the timer module generates an interaction for each DT interaction which arises, in order to ensure retransmission.

The Communications Medium is described quite simply as a single module. Its unreliable behaviour is modelled by the procedure **mung**, which is defined only informally. (See the description of procedure **mung** for the meaning and supposed etymology of this word.)

9.3.3 Formal Description of the Protocol

```
specification SlidingWindowProtocol;

default individual queue;

type SeqType = integer; { sequence number type;
  will always be >= 0 }
    UserDataType = ...;
    DTPDUType = record
        Seq: SeqType;
        Msg: UserDataType;
    end;
    AKPDUType = record
        Seq: SeqType;
    end;

channel UT(user, provider);
    by user:
        DataRequest(Data : UserDataType);
```

```
channel UR(user, provider);
    by provider:
        DataIndication(Data : UserDataType);

{ this one channel, M, takes the place of the
    two separate channels, MT and MR, used in
      other specifications }
channel M(DTSender, DTReceiver);
    by DTSender:
        DT(PDU : DTPDUType);
    by DTReceiver:
        AK(PDU : AKPDUType);

channel TimerChan(user, provider);
    by user:
        TimerRequest(Seq : SeqType);
        TimerCancel(Seq : SeqType);
    by provider:
        TimerResponse(Seq : SeqType);

module TransmitterUser systemprocess;
    ip U : UT(user);
end;

body TransmitterUserBody for TransmitterUser;
external;

module ReceiverUser systemprocess;
    ip U : UR(user);
end;

body ReceiverUserBody for ReceiverUser;
external;

module Cms systemprocess;
    ip CT : M(DTReceiver);
       CR : M(DTSender);
end;

{ the body for the Cms module is given in the
    following clause}
body CmsBody for Cms;
external;

module Timer systemprocess;
    ip T : TimerChan(provider);
end;

body TimerBody for Timer;
external;

module Transmitter systemprocess;
    ip U  : UT(provider);
       CT : M(DTSender);
       T  : TimerChan(user);
end;

{ Transmitter module body }
body TransmitterBody for Transmitter;

    const TransmitterWindowSize = any integer;

state SENDING;

{ save user data in buffer until
    Acknowledegment }
procedure BufSave(
  s : SeqType; d : UserDataType);
primitive;

{ free user data buffer entry after
    Acknowledegment }
procedure BufFree(s : SeqType);
primitive;

{ retrieve user data entry from buffer }
function BufRetrieve(s : SeqType)
  : UserDataType;
primitive;

{ returns true if the PDU is corrupted }
function corrupted(PDU : AKPDUType) :
  boolean;
primitive;

{ construct a DT PDU from the user data and
    sequence number }
function
  PDUDT(s : SeqType; d : UserDataType) :
    DTPDUType;
primitive;

var
    LowestUnacked : SeqType;
    HighestSent : SeqType;
    TWS : integer;

initialize
    to SENDING
        provided (TransmitterWindowSize >
          0)
            begin
                LowestUnacked := 1;
                HighestSent := 0;
                TWS :=
                  TransmitterWindowSize;
            end;

trans

    { transmit while window not full }
    from SENDING to same
        when U.DataRequest
            provided HighestSent -
              LowestUnacked + 1 < TWS
                begin
                    HighestSent :=
                      HighestSent + 1;
                    output T.TimerRequest(
                      HighestSent);
                    output CT.DT(PDUDT(
                      HighestSent, Data));
```

```
                    BufSave(HighestSent,
                      Data);
                end;

        { receive Acknowledgement }
        from SENDING to same
            when CT.AK
                provided (PDU.Seq >=
                  LowestUnacked) and
                    (PDU.Seq <= HighestSent)
                      and not corrupted(PDU)
                    var S : SeqType;
                    begin
                      for S := LowestUnacked
                        to PDU.Seq do
                          begin
                            output
                              T.TimerCancel(S);
                            BufFree(S);
                          end;
                      LowestUnacked :=
                        PDU.Seq + 1;
                    end;

        { receive ack not in window }
        provided otherwise
            begin
                { ignore this ack }
            end;

        { Timer response }
        from SENDING to same
            when T.TimerResponse
                provided (Seq >= LowestUnacked)
                  and (Seq <= HighestSent)
                    var S : SeqType;
                    begin
                      for S := Seq
                        to HighestSent do
                          begin
                            output
                              T.TimerCancel(S);
                            output CT.DT(
                              PDUDT(S,
                                BufRetrieve(
                                  S)));
                            output T.
                              TimerRequest(S);
                          end;
                    end;

                provided otherwise
                    begin
                    { ignore timer response for
                      sequence number outside
                      window; can happen when
                      AK arrives just as timer
                      responds }
                    end;

end; { TransmitterBody }
```

```
module Receiver systemprocess;
    ip U  : UR(provider);
       CR : M(DTReceiver);
end;

{ Receiver module body }
body ReceiverBody for Receiver;

    const ReceiverWindowSize = any integer;

    state RECEIVING;

    { construct an AK PDU, given the sequence
        number }
    function PDUAK(S : SeqType) : AKPDUType;
    primitive;

    { retrieve the PDU of sequence number S
      from buffer.  If it is not in the buffer,
       return a PDU with seq number set to 0 }
    function PDURetrieve(S : SeqType) :
      DTPDUType;
    primitive;

    { Save the PDU in the buffer }
    procedure PDUSave(PDU : DTPDUType);
    primitive;

    { returns true if the PDU is corrupted }
    function corrupted(PDU : DTPDUType) :
      boolean;
    primitive;

    { Return the user data from the given PDU }
    function UserData(
      p : DTPDUType) : UserDataType;
    primitive;

    var
        NextRequired : SeqType;
        HighestReceived : SeqType;
        RWS : integer;

    initialize
        to RECEIVING
            provided ReceiverWindowSize > 0
                begin
                    NextRequired := 1;
                    HighestReceived := 0;
                    RWS := ReceiverWindowSize;
            end;

    trans

        { receive message in window }
        from RECEIVING to same
            when CR.DT
                provided (PDU.Seq >=
                  NextRequired) and
                    (PDU.Seq <
```

```
                NextRequired + RWS) and
                 not corrupted(PDU)
              var
                  S    : SeqType;
                  TPDU : DTPDUType;
                  Done : boolean;
              begin
                PDUSave(PDU);
                S := NextRequired;
                Done := false;
                { Retrieve each PDU from
                  buffer and send it to
                  user.  Stop at first
                  gap in buffer, i.e.
                  the first PDU not
                  received.  PDURetrieve
                  returns a PDU with
                  sequence number 0 if
                  desired PDU is not in
                  buffer. }

                repeat
                  TPDU :=
                    PDURetrieve(S);
                  if TPDU.Seq = S then
                    begin
                      { extract user data
                        from PDU and send
                        to user }
                      output U.
                        DataIndication(
                         UserData(TPDU));
                      S := S + 1;
                    end
                  else
                    { reached gap in
                      buffer }
                    Done := true;
                until Done;
                NextRequired := S;
                output CR.AK(PDUAK(
                    NextRequired - 1));
              end;
          { receive message not in window
            or is corrupted }
          provided otherwise
              begin
                  output CR.AK(PDUAK(
                    NextRequired - 1));
              end;

end; { ReceiverBody }

{ main body for Sliding Window specification }
modvar
    TransmitterInstance : Transmitter;
    ReceiverInstance : Receiver;
    TransmitterUserInstance : TransmitterUser;
    ReceiverUserInstance : ReceiverUser;
    CmsInstance : Cms;
    TimerInstance : Timer;

initialize
    begin
        init TransmitterUserInstance
          with TransmitterUserBody;
        init ReceiverUserInstance
          with ReceiverUserBody;
        init TransmitterInstance
          with TransmitterBody;
        init ReceiverInstance
          with ReceiverBody;
        init CmsInstance with CmsBody;
        init TimerInstance with TimerBody;

        connect TransmitterUserInstance.U
          to TransmitterInstance.U;
        connect ReceiverUserInstance.U
          to ReceiverInstance.U;
        connect TransmitterInstance.CT
          to CmsInstance.CT;
        connect ReceiverInstance.CR
          to CmsInstance.CR;
        connect TransmitterInstance.T
          to TimerInstance.T;
    end;
end. { specification SlidingWindowProtocol }
```

9.3.4 Formal Description of the Medium

```
body CmsBody for Cms;
  const MaxDelay = any integer; { maximum
    delay }
    QueueData = record
        Seq: SeqType;
        Msg: UserDataType
    end;

    { The next several procedures and
      functions manipulate queues in the
      usual fashion. The details are left to
      the reader. }

    procedure initqueue(var q: QueueType);
    primitive;

    procedure enqueue(
      Data: QueueData; var q: QueueType);
    primitive;

    procedure dequeue(
      var Data: QueueData; var q: QueueType);
    primitive;

    function isempty(q: QueueType): boolean;
    primitive;

    { The procedure "mung" is invoked to
      model the unreliability of the medium.
      Each time it is invoked, it may drop,
      reorder, duplicate, or corrupt some of
      the entries of the queue, q. Of course,
```

```
        it may also leave the queue unaltered.
        Again, details are left to the reader.
        It is reputed that the acronym "mung"
        comes from the phrase "modify until no
        good". }
    procedure mung(var q: QueueType);
    primitive;

var
    TtoR: QueueType; { queue for data from
      Transmitter to Receiver }
    RtoT: QueueType; { queue for data from
      Receiver to Transmitter }

initialize
    provided ( MaxDelay > 0 )
        begin { 1 }
            initqueue(TtoR);
            initqueue(RtoT);
        end;

trans
    when CT.DT
        var QueueElement: QueueData;
        begin { 2 }
            QueueElement.Seq := PDU.Seq;
            QueueElement.Msg := PDU.Msg;
            enqueue(QueueElement, TtoR);
        end;
    when CR.AK
        var QueueElement: QueueData;
        begin { 3 }
            QueueElement.Seq := PDU.Seq;
            enqueue(QueueElement, RtoT);
        end;

trans
    provided not isempty(TtoR)
        delay(0, MaxDelay)
            var PDUtoSend: DTPDUType;
                QueueElement: QueueData;
            begin { 4 }
                mung(TtoR);
                if not isempty(TtoR) then
                    begin
                        dequeue(QueueElement,
                          TtoR);
                        PDUtoSend.Seq :=
                          QueueElement.Seq;
                        PDUtoSend.Msg :=
                          QueueElement.Msg;
                        output
                          CR.DT(PDUtoSend);
                    end
            end;

    provided not isempty(RtoT)
        delay(0, MaxDelay)
            var AKtoSend: AKPDUType;
                QueueElement: QueueData;
            begin { 5 }
                mung(RtoT);
                if not isempty(RtoT) then
                    begin
                        dequeue(QueueElement,
                          RtoT);
                        AKtoSend.Seq :=
                          QueueElement.Seq;
                        output
                          CT.AK(AKtoSend)
                    end
            end;
end; { CmsBody }
```

9.3.5 Subjective Assessment

In the description of the Sliding Window Protocol, the **Timer** module was not described because it was felt that it would make the text longer without really adding much information to the example.

One could argue that the Communications Medium need not be described in order to understand the workings of the Sliding Window Protocol; it suffices to know its properties. However, it was decided to include the description of the medium for completeness.

Nevertheless, the medium description was written so as to avoid irrelevant details such as how to re-order the messages in the medium, how to lose a message in the medium, etc. This is all hidden inside the procedure **mung**. To understand the workings of the system, it is necessary to understand the workings of **mung**. Indeed, in general an Estelle description is parameterised in terms of its **primitive** procedures and functions. For example, if there were no guarantee that **mung** would eventually allow something to exit the queue unaltered, then there would be no way for the Protocol to work.

9.4 LOTOS Description

```
(*-----------------------------------------
```

9.4.1 Architecture of the Formal Descriptions

The formal description of the Protocol is given in 9.4.3. The architecture of the Protocol is decomposed into three major entities, as reflected in Figure 9.4. This structure is reflected in LOTOS as shown in Figure 9.5.

The top-level structure of the Protocol description is as follows:

9.4.1.1 Gates

ut for interactions between the sending user and the transmitter

ur for interactions between the receiving user and the receiver

mt for interactions between the transmitter and the medium

mr for interactions between the receiver and the medium.

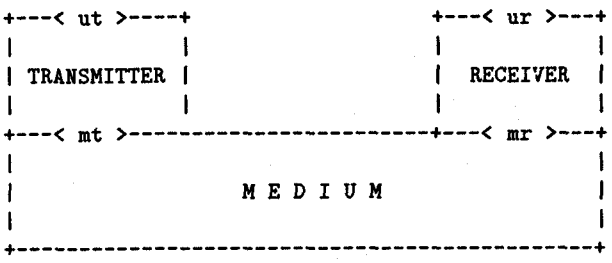


Figure 9.4: Architecture of the Sliding Window Protocol in LOTOS

```
specification SlidingWindowProtocol
 [ut, ur, mt, mr] (...) : noexit

  behaviour

    TransmitterEntity [ut, mt] (...)
  |||
    ReceiverEntity [ur, mr] (...)

    where

    process TransmitterEntity
     [ut, mt] (...) : noexit :=
      ...
    endproc

    process ReceiverEntity
     [ur, mr] (...) : noexit :=
      ...
    endproc
endspec
```

Figure 9.5: Outline Decomposition of the Sliding Window Protocol in LOTOS

9.4.1.2 Data types

UserData	Service User Data
SP	Service Primitive
MP	Medium Service Primitive
Pdu	Protocol Data Unit
PduQueue	First-in First-out queue of PDUs
SeqNumberSet	Set of Sequence Numbers
PduSet	Set of PDUs
TimerSignal	Signals to communicate with the timers
EnrichedNat	Enrichment of the Natural numbers with the *1* operation
NatMod	Enrichment of the Natural numbers with the *Mod* operation (modulo).

The outline structure shown in Figure 9.5 is further decomposed into processes as shown in Figure 9.6.

9.4.1.3 Medium

The Medium of the Sliding Window Protocol is described in 9.4.4. Only those parts of the formal description which are additional to the formal description of the Protocol are given for the Medium.

The place of the Medium in the overall architecture of the medium is shown in Figure 9.4. The two entities, **TransmitterEntity** and **ReceiverEntity** communicate through the Medium. To the user, the only observable gates are those at top of the sliding window protocol. So, gates **ut** and **ur** provide a Service to the users of the Sliding Window Protocol. The underlying Service upon which this Protocol operates is irrelevant and not observable by the Sliding Window Service Users. In LOTOS this fact is described as shown in Figure 9.7. Notice that the gates through which the Medium is accessed are **hidden** to reflect the fact of non-observability.

This view of the Sliding Window Protocol can be more convenient since it is an asymmetrical Protocol. The underlying Service of any Protocol must be described somewhere, although it may be considered a non-integral part of the Protocol description itself.

In Figure 9.8 the process decomposition for the Medium description is shown. Only processes which are additional to those of the Protocol are included.

9.4.1.4 Abbreviations

The following abbreviations are used in the descriptions:

tws	transmitter window size
rws	receiver window size
lu	lowest unacknowledged message
hs	highest sent message
nr	next required message
sn	sequence number
rq	queue for re-transmissions

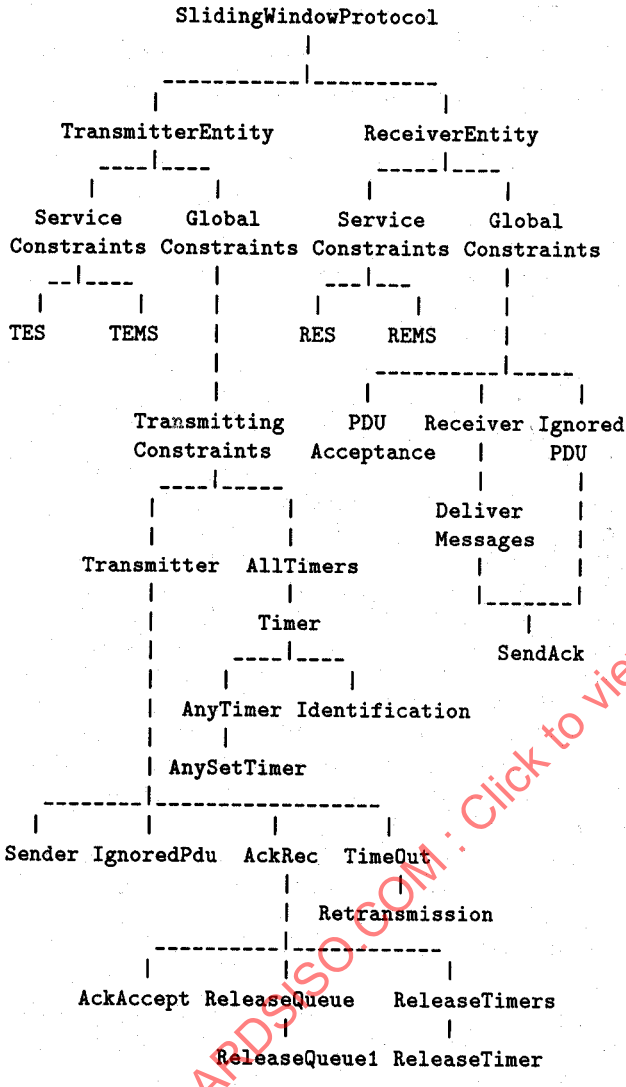


Figure 9.6: Processes of the Sliding Window Protocol in LOTOS

```
specification SlidingWindowProtocol
 [ut, ur] (...) : noexit

  behaviour

    hide mt, mr in
      (
        TransmitterEntity [ut, mt] (...)
      |||
        ReceiverEntity [ur, mr] (...)
      )
    ||
      Medium [mt, mr] (...)

  where

  process TransmitterEntity
   [ut, mt] (...) : noexit :=
    ...
  endproc

  process ReceiverEntity
   [ur, mr] (...) : noexit :=
    ...
  endproc

  process Medium [mt, mr] (...) : noexit :=
    ...
  endproc

endspec
```

Figure 9.7: Outline Decomposition of Sliding Window Medium in LOTOS

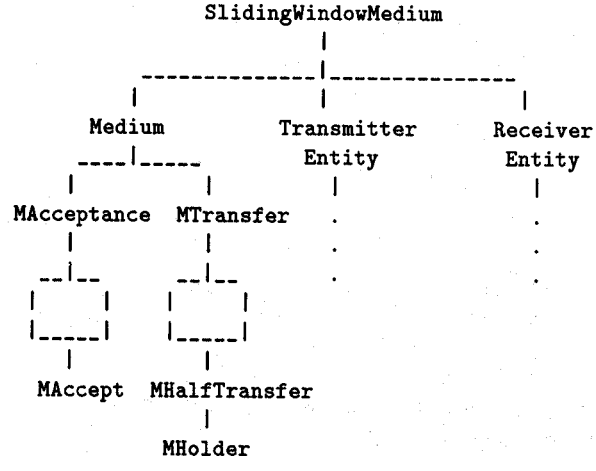


Figure 9.8: Processes of Sliding Window Medium in LOTOS

sns sequence numbers set
ps pdu set
up User Service Primitive
mp Medium Service Primitive
pdu Protocol Data Unit.

9.4.2 Explanation of Approach

These are 'constraint-oriented' descriptions. This structuring method has been extensively used with LOTOS. It provides a well-structured description, having several advantages, including:

a) the description is clearer and better structured; and
b) the description is separated into different constraints, logically related ones being grouped in a single process; and
c) further constraints may be added with little effort; and
d) the description is relatively abstract because no internal structure is defined.

The queues are used in such a way that constructor-selector problems with the data types are avoided. That is, in order to prevent the proliferation of error values in sorts, queues are always referred to by explicit construction rather than by the usual 'head' and 'tail' operations (which yield error values with an empty queue). This decision was not extended to other data types, such as Service Primitives, in order not to jeopardize the description.

9.4.3 Formal Description of the Protocol

At the highest level, the protocol is accessed through gates **ut** and **ur**. Gates **mt** and **mr** are for communication between these two entities through the Medium. The specification is parameterised by the transmitter and receiver window sizes, respectively **tws** and **rws**.

```
(*-------------------------------------------*)

specification SlidingWindowProtocol
 [ut, ur, mt, mr] (tws : Nat, rws : Nat) : noexit

(*-------------------------------------------
```

Abstract Data Types: imported from the Standard Library.

```
-------------------------------------------*)

  library
    NaturalNumber, Element, String, OctetString,
    Boolean, Set
  endlib

(*-------------------------------------------
```

Sliding Window Service Data: a definition of user data; based on the standard **BitString** type.

```
-------------------------------------------*)

  type UserDataType is OctetString renamedby
      sortnames UserData for OctetString
      opnnames No_Octets for <>
  endtype (* UserDataType *)

(*-------------------------------------------
```

Service Primitive: a definition of Service Primitives; this type has the usual structure for OSI Services specified in LOTOS.

```
-------------------------------------------*)

  type SPType is UserDataType, Boolean
    sorts SP
    opns
      Sreq, Sind      : UserData  -> SP
      IsSreq, IsSind  : SP        -> Bool
      Data            : SP        -> UserData
    eqns
      forall sp : SP, udata   : UserData
        ofsort Bool
          IsSreq (Sreq (udata)) = true;
          IsSreq (Sind (udata)) = false;
          IsSind (sp) = not (IsSreq (sp));
        ofsort UserData
          Data (Sreq (udata)) = udata;
          Data (Sind (udata)) = udata;
  endtype (* SPType *)

(*-------------------------------------------
```

Medium Service Primitive: description of the Service Primitives of the **Medium**; this is also a common structure in OSI Services, especially for Connectionless-Mode.

```
-------------------------------------------*)

  type MPType is PduType, Boolean
    sorts MP
    opns
      Mreq, Mind      : Pdu -> MP
      IsMreq, IsMind : MP  -> Bool
      Pdu             : MP  -> Pdu
    eqns
      forall mp : MP, pdu : Pdu
      ofsort Bool
        IsMreq (Mreq (pdu)) = true;
        IsMreq (Mind (pdu)) = false;
        IsMind (mp)         = Not (IsMreq (mp));
      ofsort Pdu
        Pdu (Mreq (pdu))    = pdu;
        Pdu (Mind (pdu))    = pdu;
  endtype (* MPType *)

(*-------------------------------------------
```

PDU Type: the Medium Service Primitives as PDUs instead of normal data. The only thing which can be described is

the PDU types and parameters. The actual encodings of these PDUs have not been defined in the informal text.

```
-----------------------------------------------*)

  type PduType is UserDataType, NaturalNumber,
   Boolean
    sorts Pdu
    opns
      MakeDTPdu    : UserData, Nat -> Pdu
      MakeAKPdu    : Nat           -> Pdu
      Data         : Pdu           -> UserData
      SN           : Pdu           -> Nat
      IsDTPdu      : Pdu           -> Bool
      IsAKPdu      : Pdu           -> Bool
      _eq_         : Pdu, Pdu      -> Bool
      _ne_         : Pdu, Pdu      -> Bool
    eqns
      forall
        sn, sn1, sn2      : Nat,
        data, dat1, dat2 : UserData,
        pdu, pdu1, pdu2   : Pdu
      ofsort Bool
        IsDTPdu (MakeDTPdu (data, sn))    = true;
        IsAKPdu (MakeAKPdu (sn))          = true;
        IsDTPdu (MakeAKPdu (sn))          = false;
        IsAKPdu (MakeDTPdu (data, sn))    = false;
        IsDTPdu (pdu1), IsAKPdu (pdu2) =>
          pdu1 eq pdu2                    = false;
        MakeDTPdu (dat1, sn1) eq
         MakeDTPdu (dat2, sn2) =
          (dat1 eq dat2) and (sn1 eq sn2);
        MakeAKPdu (sn1) eq
         MakeAKPdu (sn2) = sn1 eq sn2;
        pdu1 ne pdu2 = not (pdu1 eq pdu2);
      ofsort Nat
        SN (MakeDTPdu (data, sn)) = sn;
        SN (MakeAKPdu (sn))       = sn;
      ofsort UserData
        Data (MakeDTPdu (data, sn)) = data;
  endtype (* PduType *)

(*-----------------------------------------------
```

Queue of PDUs: used when re-transmission of PDUs is necessary. The type is an instantiation of the standard library type **String**.

```
-----------------------------------------------*)

  type PduQueueType is String actualizedby
   PduType using
    sortnames
      Pdu for Element
      Bool for FBool
      PduQueue for String
    opnnames
      No_Pdus for <>
      Pdu for String
  endtype (* PduQueueType *)

(*-----------------------------------------------
```

Set of Sequence Numbers: used by the receiver. A variable of this type will hold the sequence numbers of the PDUs already received. It is based on the standard types **Set** and **Nat**.

```
-----------------------------------------------*)

  type SeqNumberSetType is Set actualizedby
   NaturalNumber using
    sortnames
      Nat for Element
      Bool for FBool
      SeqNumberSet for Set
  endtype (* SeqNumberSetType *)

(*-----------------------------------------------
```

Set of PDUs: also used by the receiver. It holds PDUs already received.

```
-----------------------------------------------*)

  type PduSetType is Set actualizedby
   PduType using
    sortnames
      Pdu for Element
      Bool for FBool
      PduSet for Set
  endtype (* PduSetType *)

(*-----------------------------------------------
```

TimerSignal: the signals used to communicate with the timers via the **t** gate.

```
-----------------------------------------------*)

  type TimerSignalType is
    sorts TimerSignal
    opns
      set     : -> TimerSignal
      cancel  : -> TimerSignal
      expired : -> TimerSignal
  endtype (* TimerSignalType *)

(*-----------------------------------------------
```

NatModType: natural numbers enriched with the **Mod** operation; used for the identification of timers. The auxiliary operation '-' is introduced to help in the definition of **Mod**.

```
-----------------------------------------------*)

  type NatModType is NaturalNumber
    opns
      _-_, _Mod_   : Nat, Nat -> Nat
```

```
    eqns
      forall m, n : Nat
      ofsort Nat
        0 - n                 = 0;
        Succ (m) - 0          = Succ (m);
        Succ (m) - Succ (n)   = m - n;
        m Mod 0               = 0;
        m lt n =>
          m Mod n             = m;
        m ge n, n gt 0 =>
          m Mod n             = (m - n) Mod n
endtype (* NatModType *)

(*-----------------------------------------------
```

Protocol Behaviour: the general constraint that both the transmitter and the receiver window sizes must be greater than zero. The general decomposition into **Transmitter** and **Receiver** is also expressed here.

```
-----------------------------------------------*)

  behaviour
    [(tws gt 0) and (rws gt 0)] ->
      (
        TransmitterEntity [ut, mt] (tws)
      |||
        ReceiverEntity [ur, mr] (rws)
      )

  where

(*-----------------------------------------------
```

TransmitterEntity: the Transmitter Entity decomposed into the following constraints:

a1) the service constraints at gate **ut**; and

a2) the service constraints at gate **mt**.

a3) the protocol constraints relating events at gates **ut** and **mt**.

```
-----------------------------------------------*)

  process TransmitterEntity [ut, mt] (tws : Nat)
   : noexit :=
    (
      TES [ut]
    |||
      TEMS [mt]
    )
  ||
    TransmittingConstraints [ut, mt] (tws)

    where

(*-----------------------------------------------
```

Protocol Entity-Medium Constraint: expressing the temporal ordering of Service Primitives that the protocol imposes on the acceptance of Service Primitives from the Medium at gate **mt** (i.e. by the transmitter).

The constraint is that a Data PDU will be sent first; after that, more Data PDUs may be sent, or Indications may be accepted in any order.

```
-----------------------------------------------*)

    process TEMS [mt] : noexit :=
      mt ? mp : MP [IsMreq (mp) and
       IsDTPdu (Pdu (mp))];
      TEMS1 [mt]

      where

      process TEMS1 [mt] : noexit :=
        mt ? mp : MP [IsMreq (mp) and
         IsDTPdu (Pdu (mp))];
        TEMS1 [mt]
      []
        mt ? mp : MP [IsMind (mp)];
        TEMS1 [mt]
      endproc (* TEMS1 *)

    endproc (* TEMS *)

(*-----------------------------------------------
```

Transmitter gate constraint: acceptance of Service Requests at all times.

```
-----------------------------------------------*)

    process TES [ut] : noexit :=

      ut ? up : SP [IsSreq (up)];
      TES [ut]

    endproc (* TES *)

(*-----------------------------------------------
```

TransmittingConstraints: decomposition of the transmitter into two processes: an actual transmitter process and all the timers needed for the time-outs. These two processes synchronise through the **t** gate.

```
-----------------------------------------------*)

    process TransmittingConstraints [ut, mt]
     (tws : Nat) : noexit :=
      hide t in
        Transmitter [ut, mt, t]
         (tws, 0, Succ (0), No_Pdus)
      |[t]|
        AllTimers [t] (tws, 0)

      where

(*-----------------------------------------------
```

AllTimers: forking into all possible timers. The number of timers needed is **tws**.

```
-------------------------------------------*)

      process AllTimers [t]
       (MaxId : Nat, TimerId : Nat) : noexit :=
        [TimerId lt MaxId] ->
          (
            AllTimers [t] (MaxId, Succ (TimerId))
          |||
           Timer [t] (MaxId, TimerId)
          )

        where

(*-------------------------------------------
```

Timer: decomposed into the following constraints:

AnyTimer the behaviour of a generic timer; and

Identification a constraint that uniquely identifies each particular timer.

```
-------------------------------------------*)

        process Timer [t]
         (MaxTimer : Nat, TimerId  : Nat)
          : noexit :=
          AnyTimer [t]
        ||
          Identification [t] (MaxTimer, TimerId)

          where

(*-------------------------------------------
```

Identification: selection of the signals sent to the timer **TimerId**.

```
-------------------------------------------*)

          process Identification [t]
           (MaxTimer: Nat, TimerId: Nat)
            : noexit :=
            t ? Identifier : Nat ?
              AnySignal : TimerSignal [TimerId =
               (Identifier Mod MaxTimer)] ;
              Identification [t]
               (MaxTimer, TimerId)
          endproc (* Identification *)

(*-------------------------------------------
```

AnyTimer: triggered initially with a **set** event, after which it behaves like **AnySetTimer**. While set, a timer may be cancelled (if re-transmission becomes necessary before expiry) or may expire. The elapsed time between setting and expiry is not defined since LOTOS abstracts away from absolute time; a LOTOS internal event is used. When **AnySetTimer** exits (due to cancellation or expiry), **AnyTimer** repeats its behaviour.

```
-------------------------------------------*)

        process AnyTimer [t] : noexit :=
          t ? AnyId : Nat ! set;
          AnySetTimer [t] (AnyId)
        >>
          AnyTimer [t]

          where

          process AnySetTimer [t]
           (AnyId : Nat) : exit :=
            t ! AnyId ! cancel;
            exit
          []
            i;
            t ! AnyId ! expired;
            exit
          endproc (* AnySetTimer *)

        endproc (* AnyTimer *)

      endproc (* Timer *)

    endproc (* AllTimers *)

(*-------------------------------------------
```

Transmitter: decomposed into the following constraints:

Sender how the protocol sends data, modifying the parameters **HighestSent** and **Retransmissions Queue**; and

AckRec receipt of an Acknowledgement, modifying the parameters **HighestUnacked** and **Retransmissions Queue**; and

TimeOut behaviour when a time out occurs; and

IgnoredPdu ignoring out-of-order PDUs.

```
-------------------------------------------*)

    process Transmitter [ut, mt, t]
     (tws : Nat, hs : Nat, lu : Nat,
      rq : PduQueue) :  noexit :=
      (
        Sender [ut, mt, t] (tws, hs, lu, rq)
      >>
        accept hs : Nat, rq : PduQueue in
          Transmitter [ut, mt, t]
           (tws, hs, lu, rq)
      )
    []
      (
        AckRec [mt, t] (hs, lu, rq)
      >>
```

```
      accept lu : Nat, rq : PduQueue in
        Transmitter [ut, mt, t]
         (tws, hs, lu, rq)
    )
  []
    (
      TimeOut [mt, t] (hs, lu, rq)
    >>
      Transmitter [ut, mt, t]
       (tws, hs, lu, rq)
    )
  []
    (
      IgnoredPdu [mt] (hs, lu)
    >>
      Transmitter [ut, mt, t]
       (tws, hs, lu, rq)
    )

    where
```

```
(*-----------------------------------------------
```

Sender: if there is room enough (the maximum window size **tws** is not fully used), a new Service Primitive from the user is accepted and its corresponding PDU is sent. The PDU is added to the re-transmissions queue. The **HighestSent** value is incremented. A timer is started.

```
-----------------------------------------------*)

      process Sender [ut, mt, t]
       (tws : Nat, hs : Nat, lu  : Nat,
        rq  : PduQueue)
         : exit (Nat, PduQueue) :=
        [(lu + tws) gt Succ (hs)] ->
          ut ? up : SP;
          (
            let pdu : Pdu = MakeDTPdu (
             Data (up), Succ (hs)) in
              mt ! Mreq (pdu);
              t ! Succ (hs) ! set;
              exit (Succ (hs), pdu + rq)
          )
      endproc (* Sender *)
```

```
(*-----------------------------------------------
```

AckRec: reception of an Acknowledgement decomposed into the following constraints:

a) acceptance of Acknowledgements; and

b) release PDUs from the re-transmissions queue; and

c) release of timers.

```
-----------------------------------------------*)

      process AckRec [mt, t]
       (hs : Nat, lu : Nat, rq : PduQueue)
         : exit (Nat, PduQueue) :=
          AckAccept [mt] (hs, lu, rq)
        |[mt]|
          ReleaseQueue [mt] (rq)
        |[mt]|
          ReleaseTimers [mt, t] (lu)
        >>
          accept lu : Nat, rq : PduQueue in
            exit (lu, rq)

        where
```

```
(*-----------------------------------------------
```

AckAccept: acceptance of an Acknowledgement, provided that its sequence number is between the values **LowestUnacked** and **HighestSent**.

```
-----------------------------------------------*)

      process AckAccept [mt]
       (hs : Nat, lu : Nat, rq : PduQueue)
        : exit (Nat, PduQueue) :=
        mt ? mp : MP [(SN (Pdu (mp)) ge lu)
         and (SN (Pdu (mp)) le hs)];
         exit (any Nat, any PduQueue)
       endproc (* AckAccept *)
```

```
(*-----------------------------------------------
```

ReleaseQueue: release from the queue of all PDUs from the received sequence number to **LowestUnacked (lu)**. The new value of **LowestUnacked** and the queue itself are results.

```
-----------------------------------------------*)

      process ReleaseQueue [mt]
       (rq : PduQueue)
        : exit (Nat, PduQueue) :=
        mt ? mp : MP;
        (
          ReleaseQueue1 (SN (Pdu (mp)), rq)
        >> accept rq : PduQueue in
             exit (Succ (SN (Pdu (mp))), rq)
        )

        where
```

```
(*-----------------------------------------------
```

ReleaseQueue1: release from the queue of all the PDUs up to the sequence number received. The mechanism for releasing is by a **choice** of exactly that queue which is equal to the tail of the current queue. This may be hard to understand, but it avoids the constructor-selector problem.

```
-----------------------------------------------*)
```

```
        process ReleaseQueue1
        (sn : Nat, rq : PduQueue)
         : exit (PduQueue) :=
          choice newrq : PduQueue,
           pdu : Pdu []
            [rq eq (newrq + pdu)] ->
              (
                [SN (pdu) lt sn] ->
                   ReleaseQueue1 (sn, newrq)
                 []
                   [SN (pdu) eq sn] ->
                     exit (newrq)
              )
        endproc (* ReleaseQueue1 *)

      endproc (* ReleaseQueue *)

(*-------------------------------------------
```

ReleaseTimers: release of all timers from that for **LowestUnacked** to that for the received sequence number.

```
-------------------------------------------*)

      process ReleaseTimers [mt, t]
       (lu : Nat) : exit (Nat, PduQueue) :=
        mt ? mp : MP;
        ReleaseTimer [t] (SN (Pdu (mp)), lu)
      >>
        exit (any Nat, any PduQueue)

        where

        process ReleaseTimer [t]
         (Last : Nat, First : Nat) : exit :=
          [First le Last] ->
            (
              t ! first ! cancel;
              exit
            |||
              ReleaseTimer [t]
               (Last, Succ (First))
            )
        []
          [First gt Last] ->
            exit
        endproc (* ReleaseTimer *)

      endproc (* ReleaseTimers *)

    endproc (* AckRec *)

(*-------------------------------------------
```

TimeOut: a time out for the sequence number **sn** has expired. The timer for the first sequence number has to be set.

```
-------------------------------------------*)

    process TimeOut [mt, t]
     (hs : Nat, lu : Nat, rq : PduQueue)
      : exit :=
      t ? sn : Nat ! expired [(sn le hs) and
       (sn ge lu)];
      Retransmission [mt, t] (sn, rq, set)

      where

(*-------------------------------------------
```

Retransmission: re-transmission of all timed out PDUs. The elements of the queue are processed in the same way, except that the timer does not need to be cancelled for the first one. Re-transmission is not undertaken for earlier messages whose timers have not yet expired.

```
-------------------------------------------*)

      process Retransmission [mt, t]
       (sn : Nat, rq : PduQueue,
        sig : TimerSignal) : exit :=
          choice newrq : PduQueue,
           pdu : Pdu []
            [rq eq (newrq + pdu)] ->
              (
                [sn gt SN (pdu) ] ->
                  Retransmission [mt, t]
                   (sn, newrq, sig)
                []
                  [sn le SN (pdu)] ->
                    (
                      [sig = set] ->
                       mt ! Mreq (pdu);
                       t ! SN (pdu) ! set;
                       Retransmission [mt, t]
                        (sn, newrq, cancel)
                      []
                      [sig = cancel] ->
                       t ! SN (pdu) ! cancel;
                       mt ! Mreq (pdu);
                       t ! SN (pdu) ! set;
                       Retransmission [mt, t]
                        (sn, newrq, cancel)
                    )
                )
          )
        []
          [rq eq No_Pdus] ->
            exit
        endproc (* Retransmission *)

      endproc (* TimeOut *)

(*-------------------------------------------
```

IgnoredPdu: ignoring the incoming PDUs not accepted by the **AckRec** process. The discarding of corrupted PDUs is modelled by non-deterministically forcing a PDU to be accepted but ignored.

```
-------------------------------------------*)

        process IgnoredPdu [mt]
         (hs, lu : Nat) : exit :=
          mt ? mp : MP [(SN (Pdu (mp)) lt lu) or
           (SN (Pdu (mp)) gt hs)];
          exit
        []
          i; (* PDU assumed to be corrupted *)
          mt ? mp : MP;
          exit
        endproc (* IgnoredPdu *)

      endproc (* Transmitter *)

    endproc (* TransmittingConstraints *)

  endproc (* TransmitterEntity *)

(*-------------------------------------------
```

ReceiverEntity: decomposed into the constraints at each one of the gates, and those at both gates.

```
-------------------------------------------*)

  process ReceiverEntity [ur, mr]
   (rws : Nat) : noexit :=
    (
      RES [ur]
    |||
      REMS [mr]
    )
  ||
    Receiver [ur, mr] (rws, Succ (0),
     {} of SeqNumberSet, {} of PduSet)

    where

(*-------------------------------------------
```

Receiver Entity-Medium Constraint: transmission of an Acknowledgement after any Indication is sent.

```
-------------------------------------------*)

    process REMS [mr] : noexit :=
      mr ? mp : MP [IsMind (mp)];
      mr ? mp : MP [IsMreq (mp) and
       IsAKPdu (Pdu (mp))];
      REMS [mr]
    endproc (* REMS *)

(*-------------------------------------------
```

Receiving User Constraint: transmission of Indications to the user at any time.

```
-------------------------------------------*)

    process RES [ur] : noexit :=
      ur ? up : SP [IsSind (up)];
      RES [ur]
    endproc (* RES *)

(*-------------------------------------------
```

Receiver: decomposition of the constraints at both gates into the acceptance of valid PDUs, the constraints on incoming valid PDUs, and the constraint on ignoring invalid PDUs.

```
-------------------------------------------*)

    process Receiver [ur, mr]
     (rws : Nat, nr : Nat, sns : SeqNumberSet,
      ps : PduSet) : noexit :=
      (
        Receiver1 [ur, mr] (rws, nr, sns, ps)
      |[mr]|
        PduAcceptance [mr] (rws, nr, sns)
      )
    []
      (
        IgnoredPdu [mr] (rws, nr, sns)
      >>
        Receiver [ur, mr] (rws, nr, sns, ps)
      )

      where

(*-------------------------------------------
```

Receiver1: acceptance by the receiver of a valid PDU. The sequence number is inserted into a set, and the pdu itself is inserted in another set. These sets are given to a process which delivers data to the user.

```
-------------------------------------------*)

      process Receiver1 [ur, mr]
       (rws : Nat, nr : Nat, sns : SeqNumberSet,
        ps : PduSet) : noexit :=
        mr ? mp : MP;
        DeliverMessages [ur, mr]
         (nr, Insert (SN (Pdu (mp)), sns),
          Insert (Pdu (mp), ps))
      >>
        accept nr : Nat, sns : SeqNumberSet,
         ps : PduSet in
          Receiver [ur, mr] (rws, nr, sns, ps)

        where

(*-------------------------------------------
```

DeliverMessages: delivery of all the received messages to the user if they are in order. All ordered messages are delivered. An Acknowledgement is issued after delivery even if no message has been indicated to the user.

```
-----------------------------------------------*)

        process DeliverMessages [ur, mr]
         (nr : Nat, sns : SeqNumberSet,
          ps : PduSet)
           : exit (Nat, SeqNumberSet, PduSet) :=
          [nr IsIn sns] ->
            (
              choice pdu : Pdu []
                [(SN (pdu) eq nr) and
                 (pdu IsIn ps)] ->
                  ur ! Sind (Data (pdu));
                  DeliverMessages [ur, mr]
                   (Succ (nr), Remove (nr, sns),
                    Remove (pdu, ps))
            )
          []
            [nr NotIn sns] ->
              (
                SendAck [mr] (nr)
              >>
                exit (nr, sns, ps)
              )
        endproc (* DeliverMessages *)

      endproc (* Receiver1 *)

(*-----------------------------------------------
```

SendAck: transmission of an Acknowledgement of the next required minus one.

```
-----------------------------------------------*)

      process SendAck [mr] (sn : Nat) : exit :=
        choice ld : Nat []
          [sn eq Succ (ld)] ->
            mr ! Mreq (MakeAKPdu (ld));
            exit
      endproc (* SendAck *)

(*-----------------------------------------------
```

PduAcceptance: acceptance of an incoming PDU if its sequence number is within the window and if it has not been already received.

```
-----------------------------------------------*)

      process PduAcceptance [mr]
       (rws : Nat, nr : Nat, sns : SeqNumberSet)
        : noexit :=
        mr ? mp : MP [IsMReq(mp) or (IsMInd (mp)
         and ((SN (Pdu (mp)) lt (nr + rws)) and
          ((SN (Pdu (mp)) ge nr) and
           (SN (Pdu (mp)) NotIn sns))))];
        PduAcceptance [mr] (rws, nr, sns)
      endproc (* PduAcceptance *)

(*-----------------------------------------------
```

IgnoredPdu: ignoring PDUs which are outside the window, duplicated, or corrupted. An Acknowledgement of **NextRequired** minus one is sent. (See process **SendAck**). The discarding of corrupted PDUs is modelled by non-deterministically forcing a PDU to be accepted but ignored; in this case, no Acknowledgement is sent.

```
-----------------------------------------------*)

      process IgnoredPdu [mr]
       (rws : Nat, nr : Nat, sns : SeqNumberSet)
        : exit :=
        mr ? mp : MP [(SN (Pdu (mp)) ge
         (nr + rws)) or ((SN (Pdu (mp)) lt nr) or
          (SN (Pdu (mp)) IsIn sns))];
        SendAck [mr] (nr)
      []
        i;      (* PDU assumed to be corrupted *)
        mr ? mp : MP;
        exit
      endproc (* IgnoredPdu *)

    endproc (* Receiver *)

  endproc (* ReceiverEntity *)

endspec (* SlidingWindowProtocol *)
```

9.4.4 Formal Description of the Medium

```
(*-----------------------------------------------
```

This is not a self-standing description of the Medium. It relies on definitions which are given in the description of the Protocol, but which, for brevity, have not been copied into the Medium description. At the highest level, the protocol is accessed through gates **ut** and **ur**. Only the user gates are now visible. The specification is parameterised by the transmitter and receiver window sizes, respectively **tws** and **rws**.

```
-----------------------------------------------*)

specification SlidingWindowProtocol [ut, ur]
 (tws : Nat, rws : Nat) : noexit

(*-----------------------------------------------
```

Medium Objects: common in OSI Service descriptions, as well. The Medium transfers **objects** from one Service Access Point to another. The relationship between these objects and the Medium Service Primitives is expressed by means of the functions **Object** and **Indication**. The equations for **Indication** state that a medium object corresponds to a **Medium Request** or **Medium Indication** with the same PDU. The second equation for **Indication** is also required so that the operation **Object** is specified as total.

```
-----------------------------------------------*)
```

```
type MOType is MPType, PduType
  sorts MO
  opns
    Object     : MP -> MO
    Indication : MO -> MP
  eqns
    forall mp : MP, mo : MO, pdu : Pdu
    ofsort MP
      Indication (Object (Mreq (pdu))) =
        Mind (pdu);
      Indication (Object (Mind (pdu))) =
        Mind (pdu);
endtype (* MOType *)

(*-----------------------------------------------
```

Protocol Behaviour: decomposed in a new way. The Medium and the Service gates to it are hidden from the Service User.

The Protocol has the general constraint that both the transmitter and the receiver window sizes must be greater than zero.

```
-----------------------------------------------*)

  behaviour
    [(tws gt 0) and (rws gt 0)] ->
      (
        hide mt, mr in
          (
            TransmitterEntity [ut, mt] (tws)
          |||
            ReceiverEntity [ur, mr] (rws)
          )
        ||
          Medium [mt, mr]
      )

  where

(*-----------------------------------------------
```

Medium: decomposed into the constraints related to the acceptance of Service Primitives and the constraints related to the actual transfer of data.

```
-----------------------------------------------*)

  process Medium [mt, mr] : noexit :=
    MAcceptance [mt, mr]
  ||
    MTransfer [mt, mr]

    where

(*-----------------------------------------------
```

Acceptance: decomposed into the constraints at each of the gates.

```
-----------------------------------------------*)

    process MAcceptance [mt, mr] : noexit :=
      MAccept [mt]
    |||
      MAccept [mr]

      where

(*-----------------------------------------------
```

Accept: acceptance of Indications or Requests at all times.

```
-----------------------------------------------*)

      process MAccept [m] : noexit :=
        m ? mp : MP [IsMreq (mp)];
        MAccept [m]
      []
        m ? mp : MP [IsMind (mp)];
        MAccept [m]
      endproc (* MAccept *)

    endproc (* MAcceptance *)

(*-----------------------------------------------
```

Transfer: decomposition of the constraints related to the transfer of data into two identical halves.

```
-----------------------------------------------*)

    process MTransfer [mt, mr] : noexit :=
      MHalfTransfer [mt, mr]
    |||
      MHalfTransfer [mr, mt]

      where

(*-----------------------------------------------
```

HalfTransfer: acceptance of a Medium Service Primitive. It transforms the Service Primitive into an object to be transferred, then creates a process to hold this object.

```
-----------------------------------------------*)

      process MHalfTransfer [t, r] : noexit :=
        t ? mp : MP [IsMreq (mp)];
        (
          MHalfTransfer [t, r]
        |||
          MHolder [r] (Object (mp))
        )

        where

(*-----------------------------------------------
```

Holder: discarding, duplication, corruption, or delivery of an object. Corruption is modelled by non-deterministically replacing the object by another one.

```
-----------------------------------------------*)

        process MHolder [r]
         (obj : MO) : noexit :=
          (
            i;                (* Object discarded *)
            stop
          []
            i;          (* Duplication of objects *)
            r ! Indication (obj);
            MHolder [r] (obj)
          []
            i;                  (* PDU corruption *)
            (
              choice corrupted_obj : MO []
                MHolder [r] (corrupted_obj)
            )
          []
            i;                (* Object delivery *)
            r ! Indication (obj);
            stop
            )
        endproc (* MHolder *)

      endproc (* MHalfTransfer *)

    endproc (* MTransfer *)

  endproc (* Medium *)

endspec (* SlidingWindowProtocol *)
```

9.4.5 Subjective Assessment

The LOTOS description of the Sliding Window Protocol is a good example of how the 'constraint-oriented' style can be applied to a large and complex problem, breaking it down into many small and manageable pieces. This allows a description to be understood and analysed in a highly modular fashion. However, it should be said that by such economical means it is possible to construct from simple parts some very complex behaviours which may be difficult to understand in their entirety. The constraint-oriented style is therefore appropriate to a component-engineering approach reminiscent of that used in the Engineering disciplines. Although a constraint-oriented description is designed top-down, it may be necessary to understand it bottom-up!

The LOTOS style used in the descriptions is typical of that which has evolved through a large amount of experience in describing OSI Standards in LOTOS. This style is a distillation of many debates among LOTOS and OSI experts. Although many other approaches are possible, and have been tried, the style used in these examples is recommended to future specifiers.

As shown by the descriptions, it is quite straightforward in LOTOS to decompose a system into a number of parts and to describe these individually. Two separate ideas make this possible. For the data typing, the mechanisms of **enrichment** (building on existing data types), **renaming** (copying existing data types), and **actualization** (instantiating a parameterised data type) make it possible to build complex data types out of simpler ones. For the behaviour description, the process combinators (notably $[], ||, >>, and [>$) make it possible to build complex behaviours our of simpler ones.

9.5 SDL Description

9.5.1 Architecture of the Formal Descriptions

The **SlidingWindowProtocol** system is modelled as the composition of the following three blocks: **sender_entity**, **receiver_entity**, and **medium**. The sending and receiving users are located in the environment: they interact with the system via two Service Access Points, modelled by means of two channels **ut** (from the environment to the **sender_entity**) and **ur** (from the **receiver_entity** to the environment). The channels **ut** and **ur** carry the signals **UDTreq** and **UDTind** respectively, which model the simplest interaction imaginable between the User and the Provider of a uni-directional data transfer Service.

The **sender_entity** puts Data (**MDTreq** signal) on the Medium and gets Acknowledgements (**MAKind** signal) from it by using a bi-directional channel **mt**. Conversely, the **receiver_entity** gets Data from the medium (**MDTind** signal) and puts Acknowledgements (**MAKreq** signal) onto it by using a bi-directional channel **mr**.

The **sender_entity** block consists of a process type **transmitter**, instantiated just once at system start-up time. The **receiver_entity** block consists of a process type **receiver**, instantiated just at once at system start-up time. The **medium** block consists of two 'queue manager' processes (one for the Data and the other for the Acknowledgements) and two 'hazard' processes (to model abnormal behaviour in manipulating objects). The description of the Protocol is given in 9.5.3. The description of the Medium is given separately in 9.5.4 because it is not an integral part of the Protocol.

9.5.2 Explanation of Approach

The architecture described in the previous clause is quite a natural mapping between static SDL semantics and some layering concepts of OSI.

9.5.2.1 Medium description

Although the formal description of the Protocol does not require a formal description of the Medium, the description of the Medium is useful for understanding some of the features of the Protocol, and would be essential in order to simulate the Protocol or to validate it against the required Service. Indeed, the Protocol features are based on the assumption that the Medium may lose, re-order, corrupt, and duplicate objects.

9.5.2.2 Signal definition

The informal description of the Protocol suggests no names for Protocol Data Units. In the SDL description the following identifiers are used: **MDT** (Medium Data) and **MAK** (Medium Acknowledgement).

In a pure OSI approach, the **sender_entity** and the **receiver_entity** would normally interact with the Medium via two Service Primitives (say, **data_req** and **data_ind**) in order to convey Protocol Data Units. For the sake of simplicity, the following short-hand notations have been used: **MDTreq** (standing for **data_req (MDT)**), **MDTind** (standing for **data_ind (MDT)**, **MAKreq** (standing for **data_req (MAK)**), and **MAKind** (standing for **data_ind (MAK)**).

Four corresponding signals have been defined. From this point of view, the Medium can be thought of as just a block performing transfer and re-naming of signals (Requests becoming Indications in both directions).

9.5.2.3 Timer management

The informal description of the Sliding Window Protocol requires an individual timer to be set for each message sent. This is managed in SDL by using a timer **tim** with multiple instances referred to by a value in the range [0 .. **tws**]. The indexing value is used either to set/reset a given instance or to detect which timer instance has expired. When timer primitives **SET** and **RESET** are used, a duration value should be specified. If specifying a value is undesirable, such primitives cannot be used; alternatively, three external signals (say **set_timer**, **reset_timer**, and **timer_expiry**) could be used between the process **sender** and the environment. The timing mechanism would then be located in the environment. Unfortunately, this solution would introduce an unacceptable level of detail into the overall system block interaction diagram, and was therefore not adopted. However, it should be noticed that in the Sliding Window Protocol descriptions using the other FDTs, timers are described without giving any fixed delay.

9.5.3 Formal Description of the Protocol

The SDL/GR description of the Protocol and some supporting macro and type definitions are shown in figure 9.9.

9.5.4 Formal Description of the Medium

The SDL/GR description of the Medium is shown in figure 9.10. From the viewpoint of SDL syntax and semantics, the description cannot be considered in isolation, but as a part of the previous system description. Nevertheless, it is given in a separate clause in order to emphasise the fact that it is not part of the Protocol description. The SDL description of the Medium could be avoided simply by locating it in the environment.

9.5.5 Subjective Assessment

The SDL description consists of a static part, represented by the system diagram and the block diagrams (which faithfully describe the essential architecture of the real system), and a dynamic part, represented by the process diagrams (which algorithmically describe the behaviour of the active components of the system). This distinction greatly helps to ease understanding of far more complex systems than the Sliding Window Protocol.

The dynamic description is, to some extent, oriented towards an implementation; consider, for example, the use of concrete data structures such as arrays and queues. This implementation-oriented bias can hardly be avoided with SDL. However, it does not seem to add an undesirable amount of detail, at least as far the Protocol description is concerned. The Medium description, however, has perhaps too much bias towards procedural details.

Timer management is performed in a natural and elegant way. This is not necessarily true of every FDT and should therefore be considered an important feature of SDL in describing real Protocols, where timers are used extensively.

Finally, it can be stated that the SDL description is quite effective for the purpose of clearly understanding the system in question. In addition, it is also friendly and self-explanatory; very few comments are needed within the formal text in order to assist comprehension.

9.6 Assessment of the Application of the FDTs

This example is fairly typical of the style of Protocol descriptions. The deficiencies found in the informal description included the usual straight errors or lack of information. However, some interesting types of errors were found:

a) It is easy to be imprecise in natural language about whether the bounds of a range are included or excluded.

b) The word 'and' can be ambiguous in natural language. For example, it is commonplace in restaurant menus to see that a meal is followed by 'coffee and tea'!

c) It is easy in natural language to lapse into 'elegant variation' ([Fowler 1968]). For example, the same thing may be called a 'unit', a 'component', a 'sub-system', and a 'module'. Although this is acceptable in a literary work, such a style leads to imprecision in a specification.

d) A much deeper problem was to how to interpret the 'time-out' parameter. The informal description refers to 'a timer' being started. Does the word 'a' reflect the fact that each message has an individual timer, that some particular value be used for each message timer, or that any timer value (perhaps different from timer values used on other occasions) be used?

e) This issue is also tied up with the distinction between a non-deterministic specification and a partial specification. A non-deterministic specification of such a timer could say that a timer value would be chosen (by means which could not be determined). A partial specification could indicate that a single timer value would be used, but that the precise timer value would be defined when the specification was made total (i.e. at a later stage in the design).

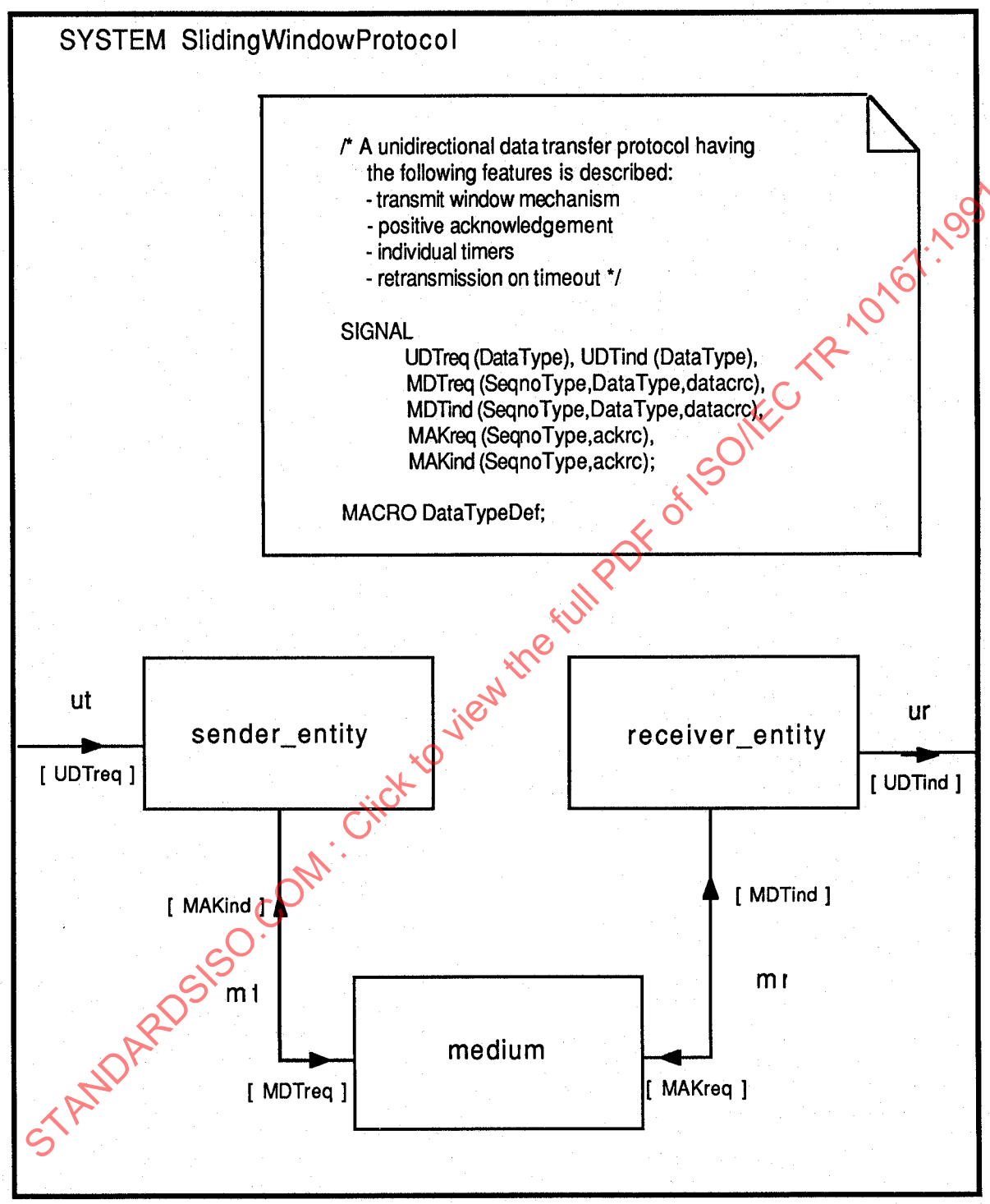


Figure 9.9: SDL Specification of Sliding Window Protocol

```
MACRODEFINITION DataTypeDef;
GENERATOR  queue (TYPE item);

LITERALS  qnew;

OPERATORS
      qadd: item,queue -> queue;
      qfirst: queue -> item;
      qrest: queue -> queue;
      qconcat: queue,queue -> queue;
      qdelete: integer,queue -> queue;
      qempty: queue -> boolean;

AXIOMS
      qfirst(qnew) == ERROR!;
      qfirst(qadd(x,qnew)) == x;
      qfirst(qadd(x1,qadd(x2,q))) == qfirst(qadd(x2,q));
      qrest(qnew) == qnew
      qrest(qadd(x,qnew)) == qnew;
      qrest(qadd(x1,qadd(x2,q))) == qadd(x1,qrest(qadd(x2,q)));
      qconcat(qnew,q) == q;
      qconcat(qadd(x1,q1),q2)) ==
                    qadd(x1,qconcat(q1,q2));
      qdelete(0,q) == q;
      FORALL i in NATURAL
        (qdelete(i,q) == qdelete(i-1,qrest(q)));
      qempty(qnew);
      NOT(qempty(qadd(x,q)));
ENDGENERATOR queue;

SYNTYPE positive=INTEGER;
        CONSTANTS > 0;
ENDSYNTYPE positive;

SYNTYPE index=INTEGER;
        CONSTANTS 1:lmax;
ENDSYNTYPE index;

SYNTYPE datacrcindex = INDEX;
        CONSTANTS 1:lendatacrc;
ENDSYNTYPE datacrcindex;

SYNTYPE ackcrcindex = INDEX;
        CONSTANTS 1:lenackcrc;
ENDSYNTYPE ackcrcindex;

SYNTYPE tsn = INTEGER /* integer in range 0..tws-1 */;
        CONSTANTS 0:tws-1
ENDSYNTYPE tsn;

SYNTYPE rsn = INTEGER /* integer in range 0..rws-1 */;
        CONSTANTS 0:rws-1
ENDSYNTYPE rsn;

SYNTYPE sequence_number = natural;
ENDSYNTYPE sequence_number;

NEWTYPE datacrc ARRAY(crcindex,bit)
```

Figure 9.9 *(continued)*

```
    OPERATORS dcheck: sequence_number, bitstring -> datacrc
     /* for a given pair of sequence_number and userdata to be
        inserted in a  MDT protocol data unit builds the crc
field */
ENDNEWTYPE datacrc;


NEWTYPE ackcrc ARRAY(crcindex,bit)
    OPERATORS acheck: sequence_number -> ackcrc
     /* for a  sequence_number to beinserted in a  MAK protocol
        data unit builds the crc field */

    AXIOMS /* undefined */
ENDNEWTYPE ackcrc;


NEWTYPE msgqueue queue(bitstring);
ENDNEWTYPE msgqueue;


NEWTYPE DataType STRING(bit,'')
ENDNEWTYPE bitstring;


NEWTYPE bit
     LITERALS 0,1;
ENDNEWTYPE bit;

ENDMACRO DataTypeDef;
```


Figure 9.9 *(continued)*

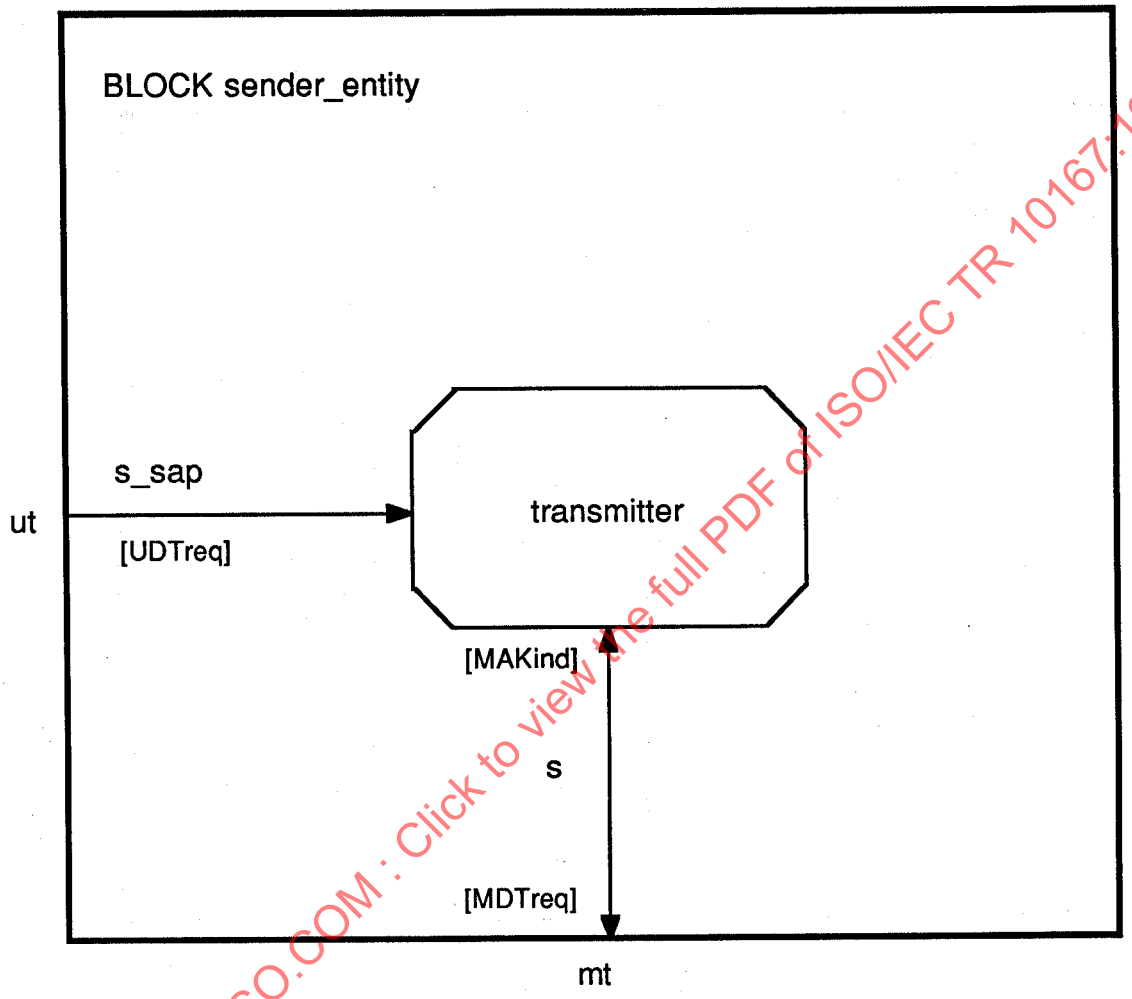


Figure 9.9 *(continued)*

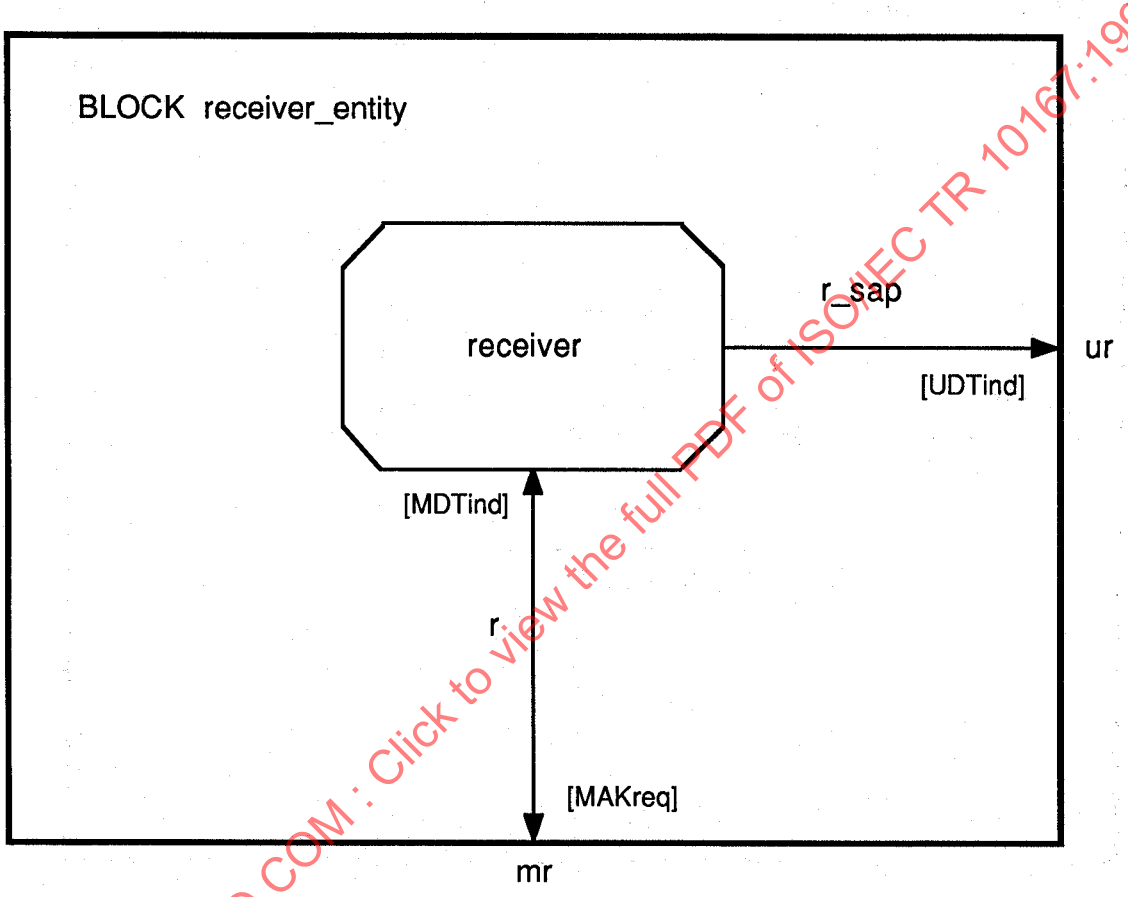


Figure 9.9 *(continued)*

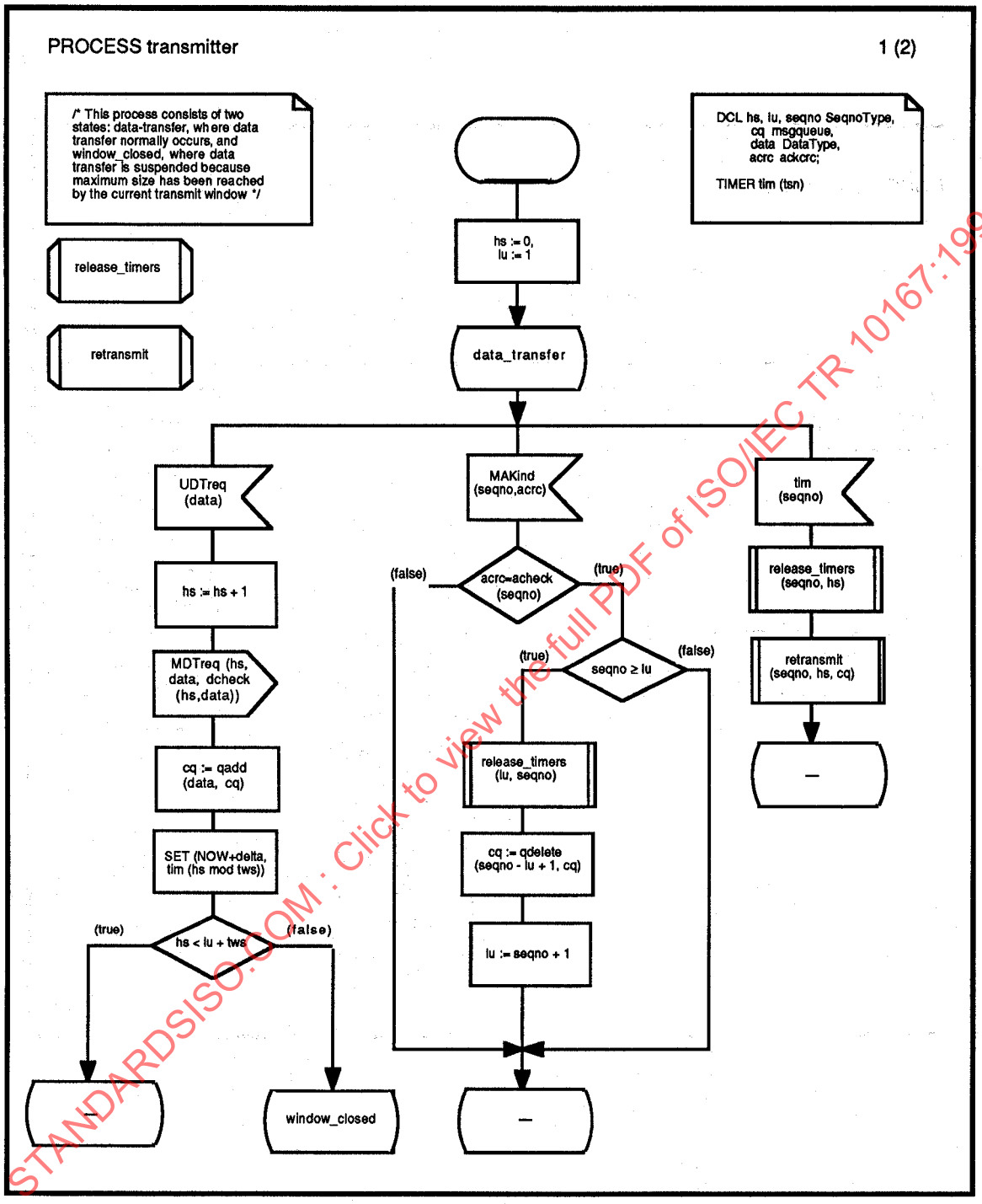


Figure 9.9 *(continued)*

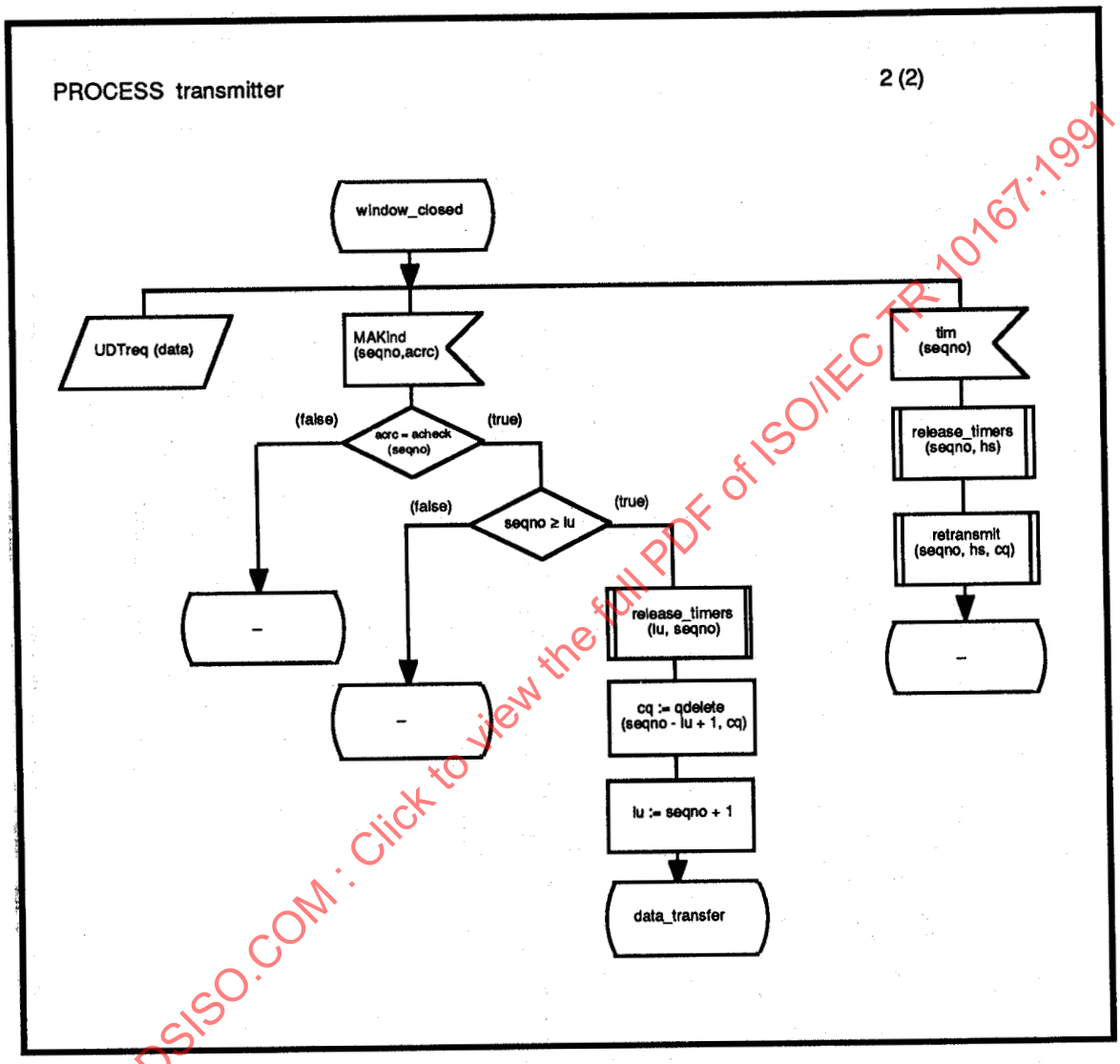


Figure 9.9 *(continued)*

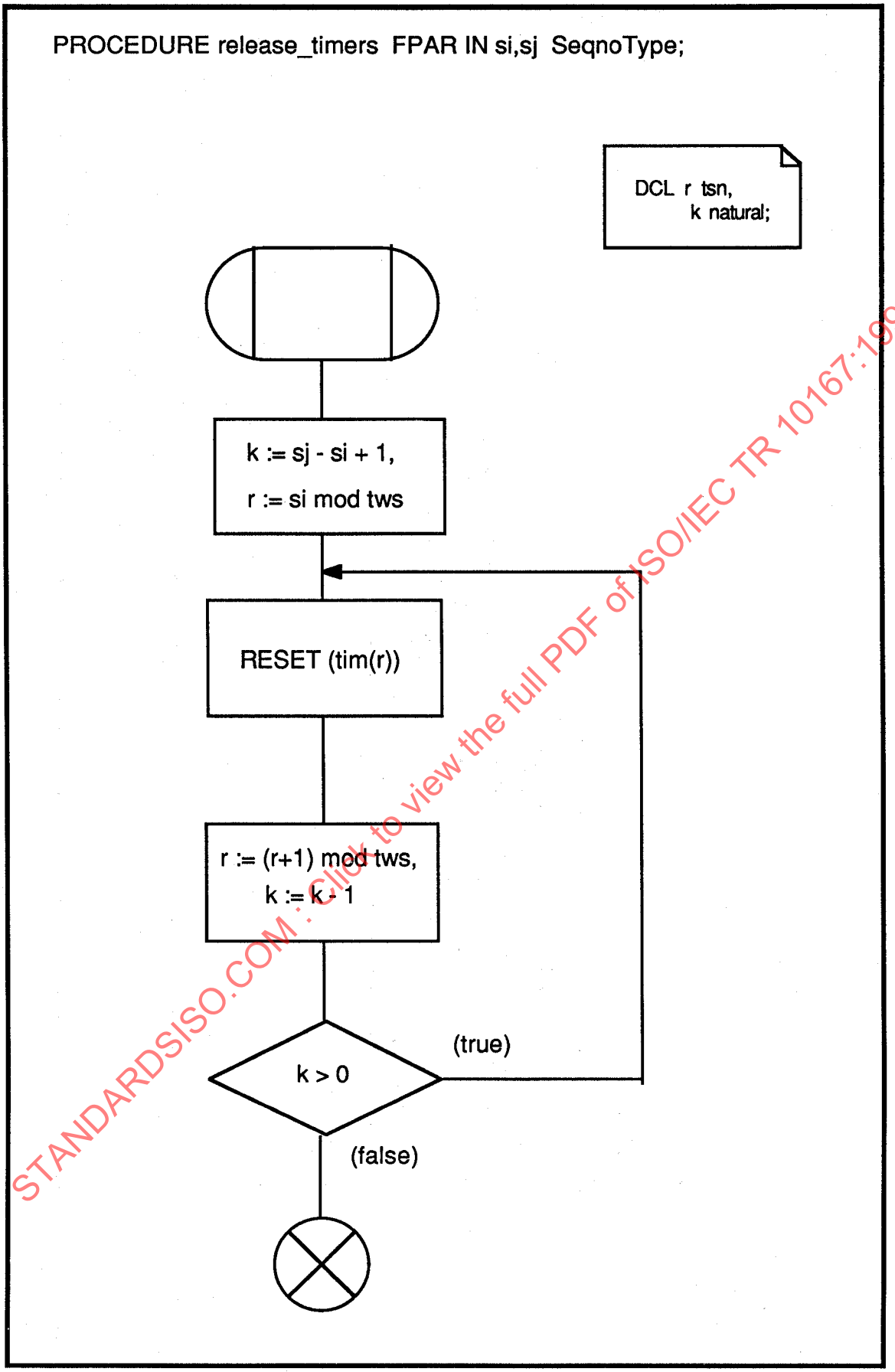


Figure 9.9 *(continued)*

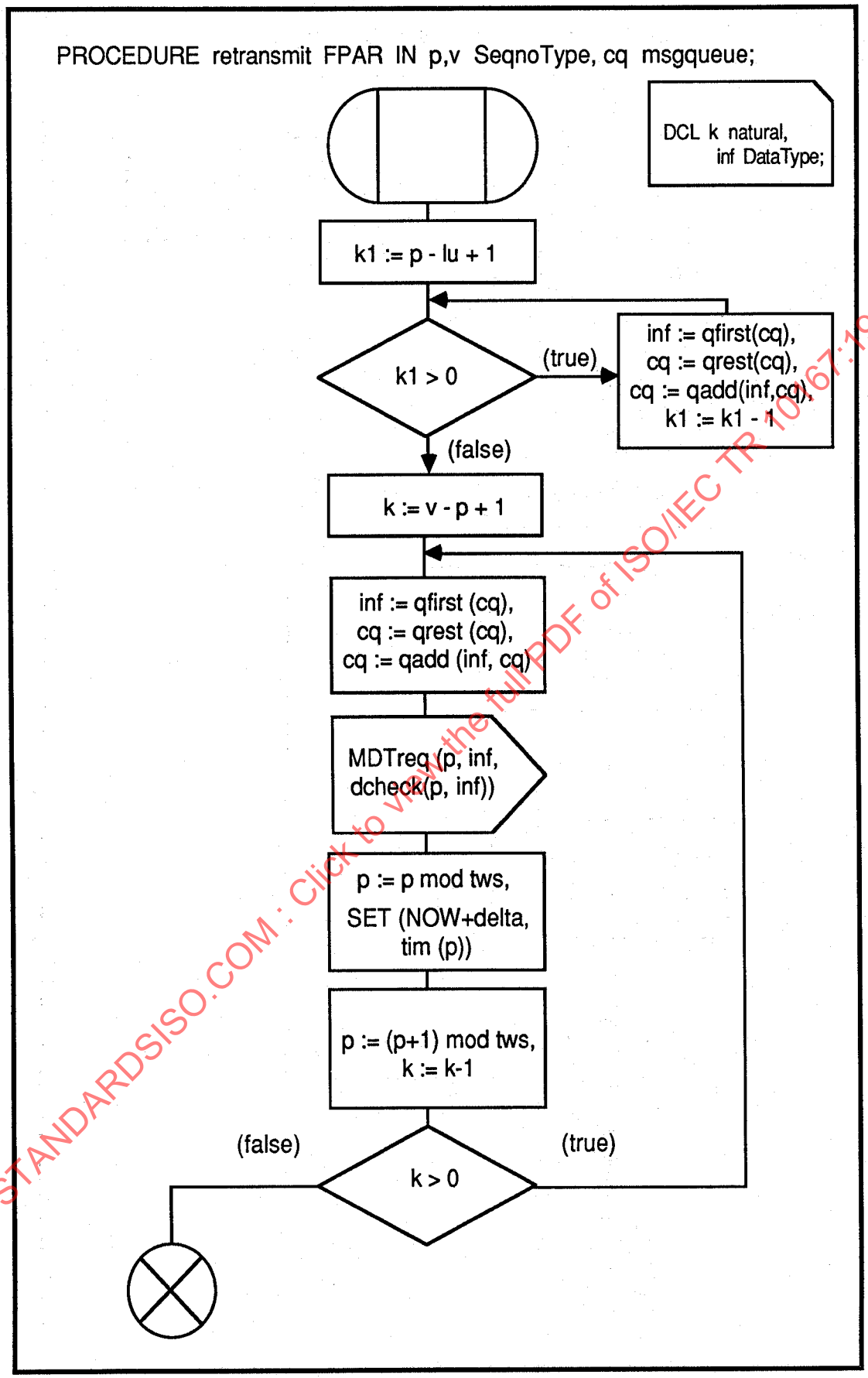


Figure 9.9 *(continued)*

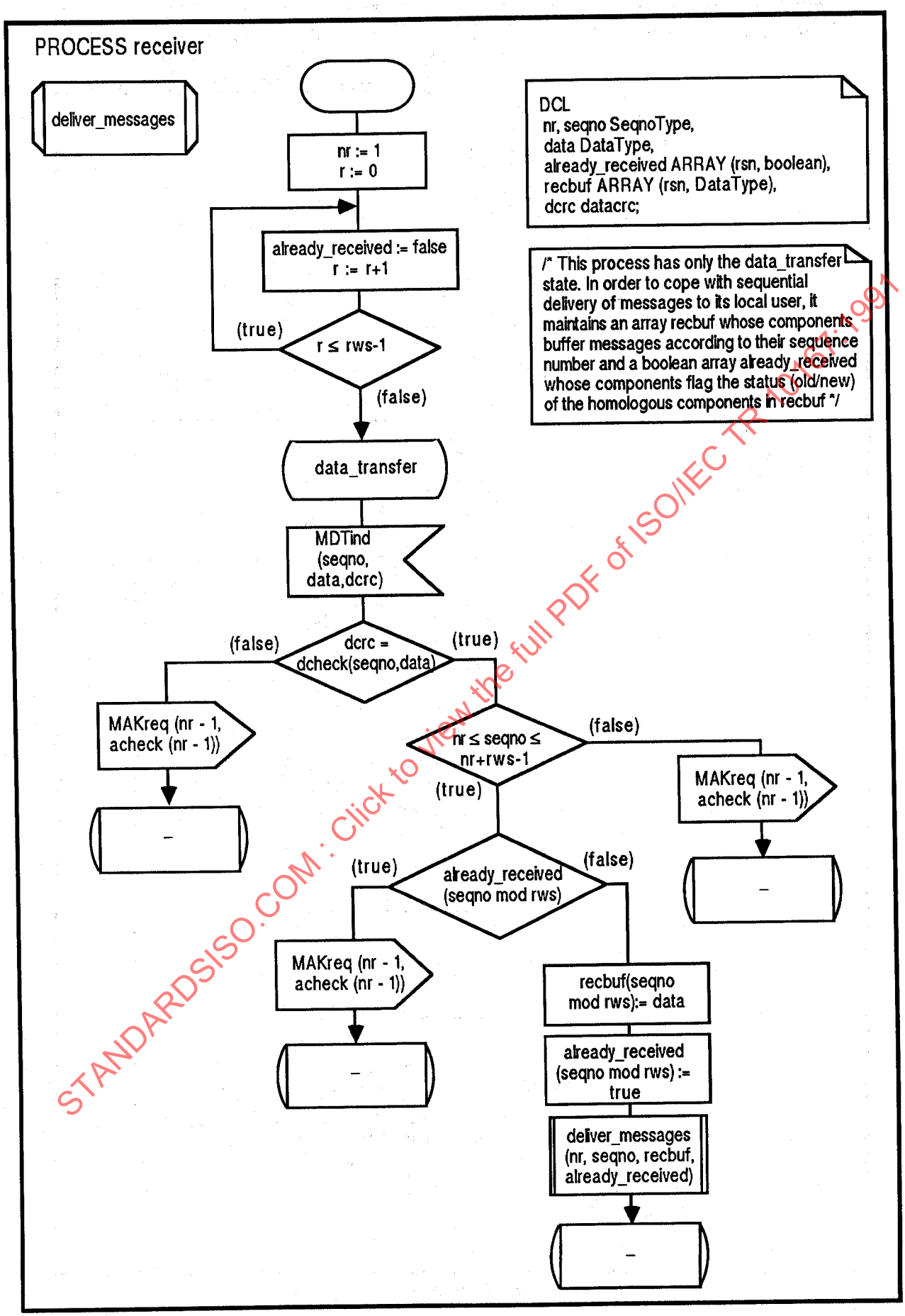


Figure 9.9 *(continued)*

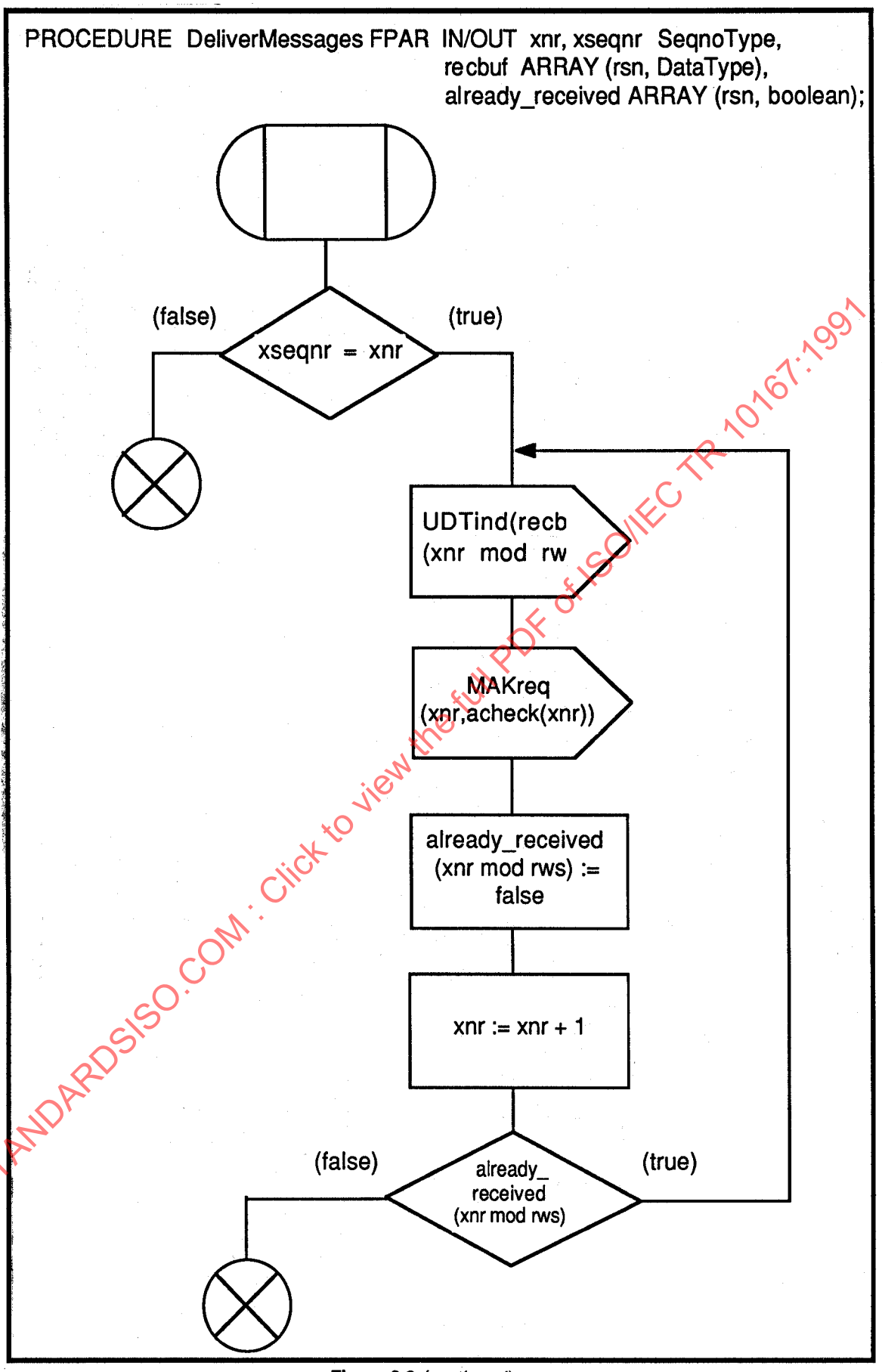


Figure 9.9 *(continued)*

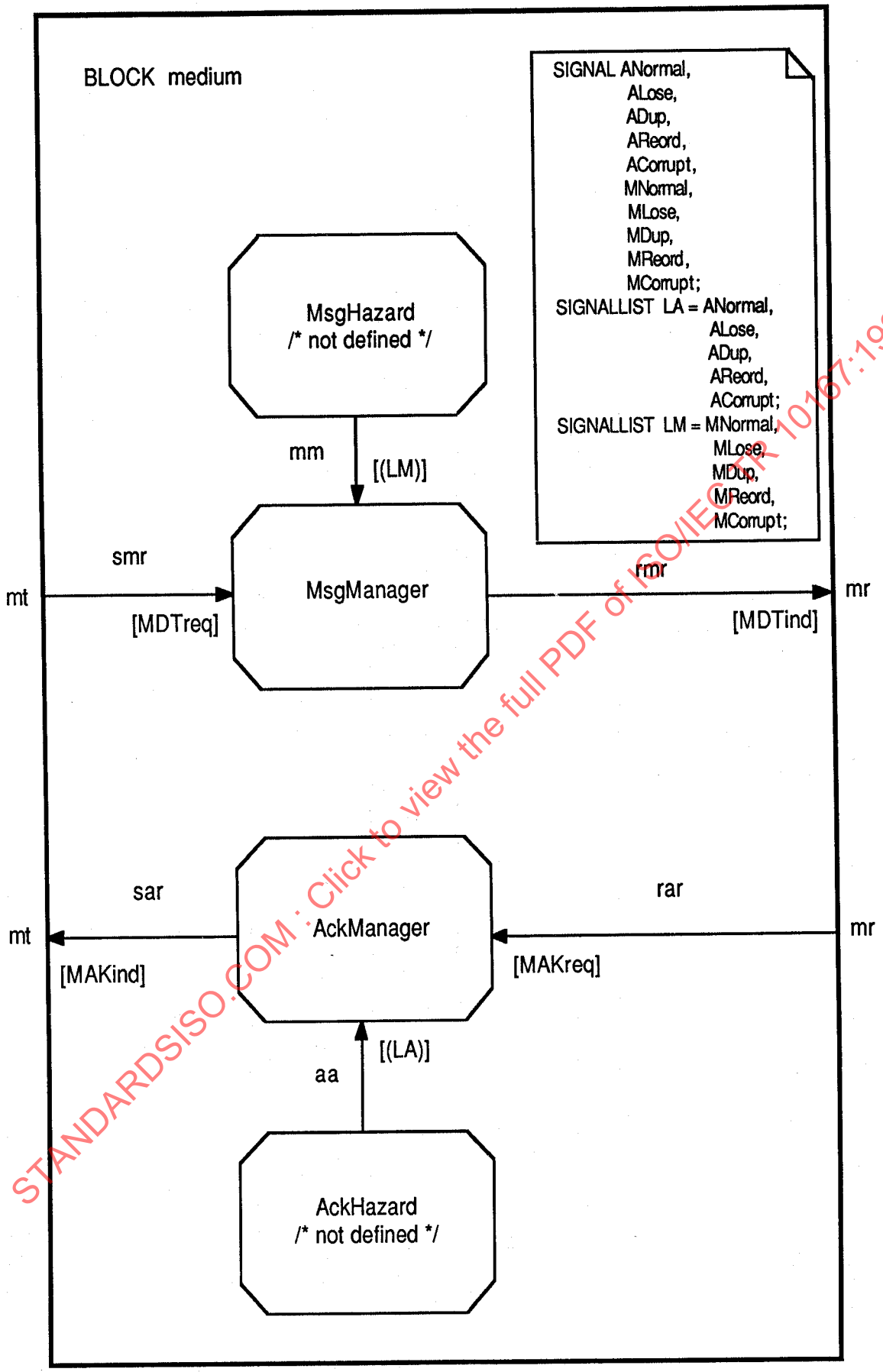


Figure 9.10: SDL Specification of Sliding Window Medium

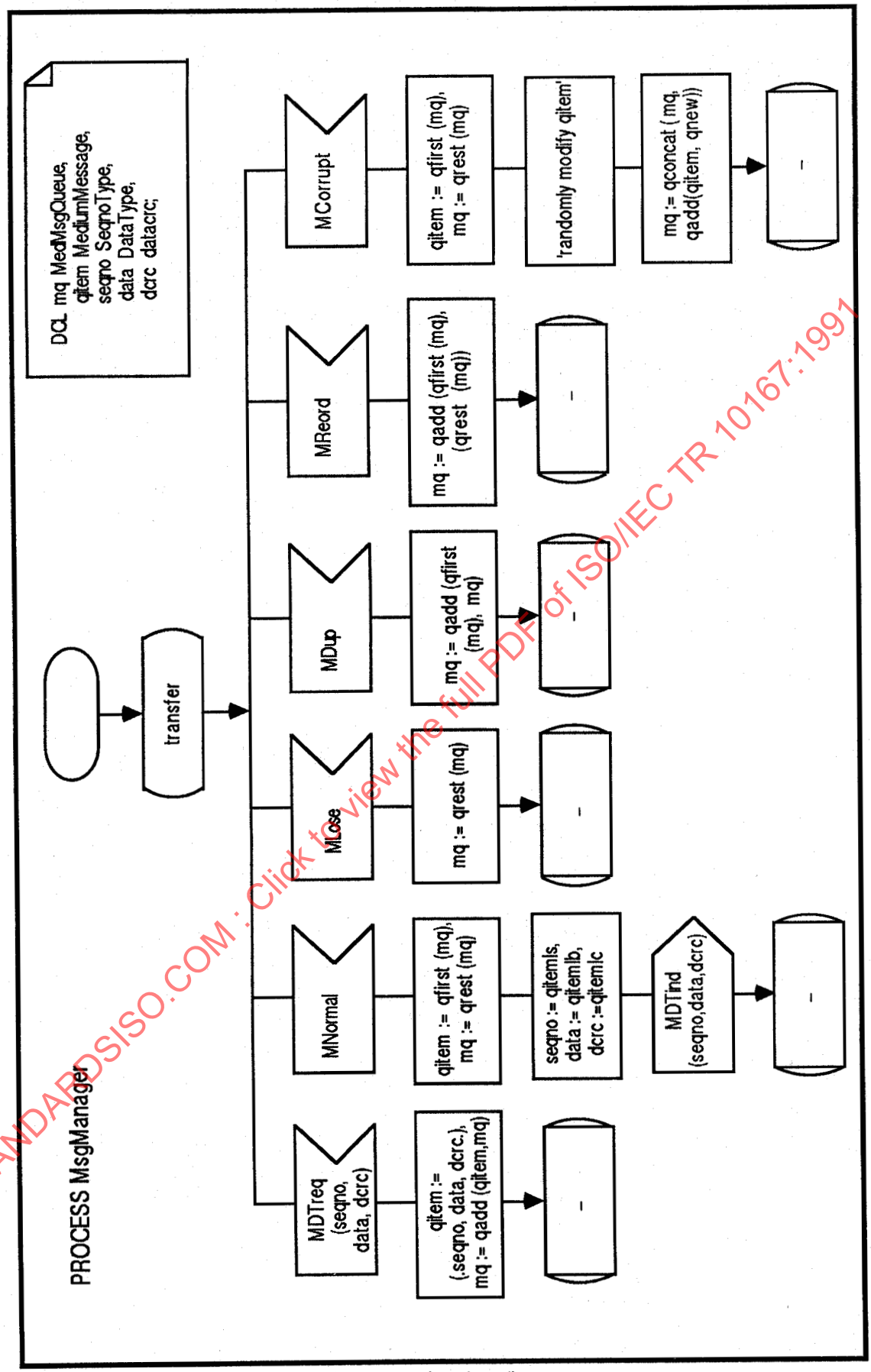


Figure 9.10 *(continued)*

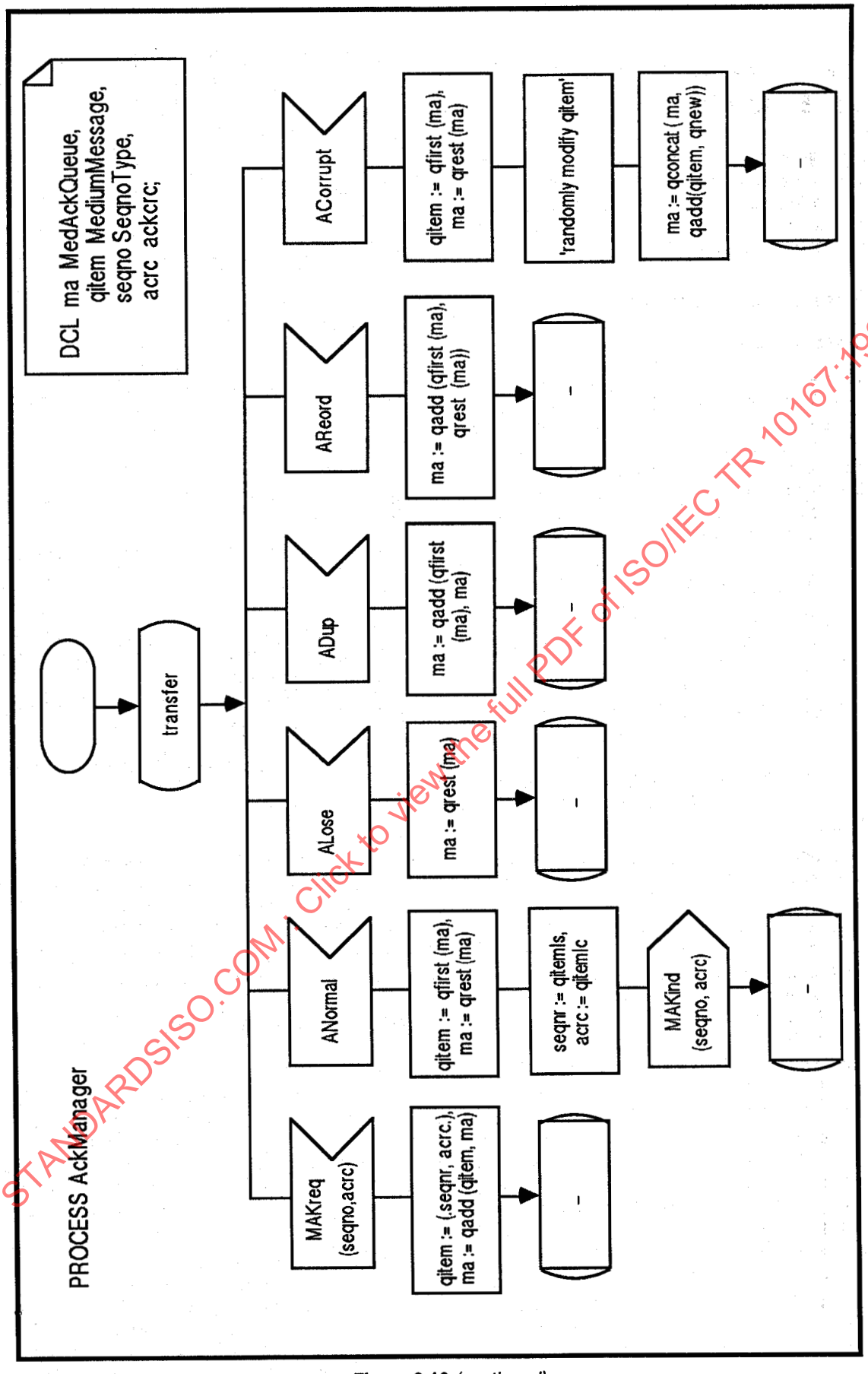


Figure 9.10 *(continued)*

10 Abracadabra Service and Protocol Example

This example illustrates the familiar Alternating Bit Protocol, which is a precursor to some real Protocols. It also illustrates extra features found in connection-oriented Protocols. The example presents the description of a Protocol in relation to the Service it provides.

10.1 Informal Description

10.1.1 Introduction

Many people have studied the Alternating Bit protocol which supports a unidirectional flow of information with a positive handshake on each transfer. This protocol is too simple to represent a number of complexities found in real communications protocols. This example describes a more realistic protocol which has **A**lternating **B**it sequence numbers, **R**etransmission on timeout, **A**cknowledgements, **C**onnection **A**nd **D**isconnection.

10.1.2 Service Description

The Abracadabra Service operates between a pair of stations, addressed as **A** and **B**. Each station is presumed to support a local user interface to the Protocol Entities. The service offered is a reliable, connection-oriented service between a pair of Service Users. The Service Primitives supported are:

ConReq/Ind	Connection Request/Indication
ConResp/Conf	Connection Response/Confirmation
DatReq/Ind	Data Request/Indication
DisReq/Ind	Disconnect Request/Indication

Only **DatReq** and **DatInd** carry a parameter, which is a Service Data Unit (**SDU**). The Service Primitives are related as shown in Figure 10.1.

A connection may be established through the Service by either station. The normal sequence of primitives is: **ConReq, ConInd, ConResp, ConConf**. However, if each station simultaneously initiates a connection then each end sees only **ConReq, ConConf**. A connection establishment attempt may be abandoned by the initiator by sending **DisReq**, before receiving **ConConf**. A connection establishment attempt may also be abandoned by the responder, sending a **DisReq** following **ConInd**.

Once a connection is established, either station may send a **DatReq** which will be delivered as **DatInd**. Data messages are preserved in sequence and content, except when a disconnection occurs. In this case, an undefined number of data messages already in the Service may be lost. Data transfer is subject to flow control by back-pressure.

Either station may terminate an established connection by issuing **DisReq**. This is normally matched by a **DisInd** at the other station, but if the other station issues **DisReq** in the meantime then the connection is terminated immediately.

The Service Provider itself may abandon a connection at-

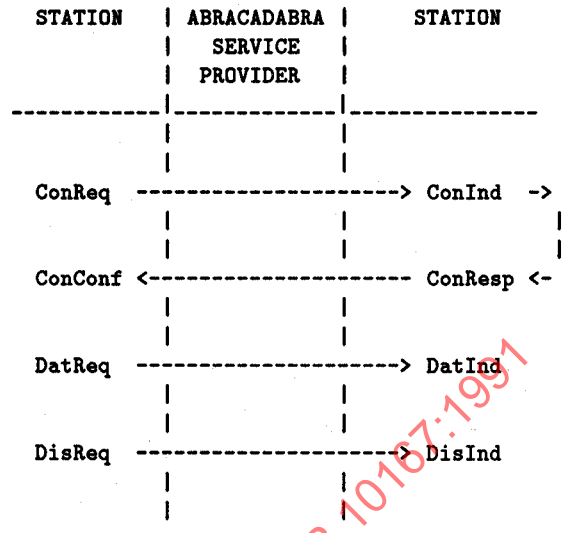


Figure 10.1: Relationship between Abracadabra Service Primitives

PDU	Meaning	Corresponding Primitives
CR	Connection Request	ConReq/Ind
CC	Connection Confirmation	ConResp/Conf
DT	Data Transfer	DatReq/Ind
AK	Acknowledgement	-
DR	Disconnection Request	DisReq/Ind
DC	Disconnection Confirmation	-

Figure 10.2: Abracadabra Protocol Data Units

tempt or may terminate the connection. Normally each station which knows of the connection (attempt) is informed of this by **DisInd**. However, if the station issues **DisReq** in the meantime then the **DisInd** is not delivered.

Once a connection has been terminated, either party may initiate a new connection with **ConReq**.

10.1.3 Protocol Description

10.1.3.1 General

The protocol operates over a full-duplex, unreliable communications medium between two stations. The two stations communicate by transfer of Protocol Data Units (**PDUs**). The communications protocol is two-way simultaneous, symmetrical and reliable. Communication takes place in various phases: Connection, Data Transfer, Disconnection, and Error.

Only the PDUs shown in Figure 10.2 are permitted. Only **DT** and **AK** carry parameters : both carry a one-bit sequence number, and **DT** carries a Service Data Unit. Each Abracadabra Service Data Unit is carried in one **DT** Protocol Data Unit. Each Abracadabra Protocol Data Unit is carried in one Service Data Unit of the underlying medium.

The Abracadabra Protocol is parameterised by two con-

stants. **N** (> 0) defines the maximum number of attempts to transmit a PDU without receiving an acknowledgement. **P** (which exceeds the round-trip transit delay) defines the time period which should elapse before attempting re-transmission.

10.1.3.2 Connection Phase

A connection attempt is made following **ConReq** by sending a **CR**. If a **CC** is received, a **ConConf** is issued and the Data Transfer Phase is entered; the same is true if **CR** is received instead. If **DR** is received or **DisReq** occurs, the Disconnection Phase is entered. If any PDU other than **CC**, **CR** or **DR** is received, it is ignored. If no response to **CR** is received within period **P**, the **CR** is re-transmitted. A maximum of **N** connection attempts (i.e. **N** periods of value **P**) are permitted. After this, the Error Phase is entered.

When no connection is set up, receipt of a **CR** causes a **ConInd**; any other PDU is ignored. If a **ConResp** follows, then **CC** is sent and the Data Transfer Phase is entered. If, however, the connection attempt is abandoned with **DisReq**, then the Disconnection Phase is entered.

10.1.3.3 Data Transfer Phase

A **DatReq** leads to a **DT** being sent. On receipt of the corresponding **AK**, a further **DatReq** may be accepted. If the corresponding **AK** is not received within period **P**, the **DT** is re-transmitted. A maximum of **N** transmission attempts (i.e. **N** periods of value **P**) are permitted. After this, the Error Phase is entered.

DTs and **AK**s carry a one-bit sequence number which is independent for each direction of transmission. The sequence number starts at 0 after connection. The correct acknowledgement to a **DT** bears the next (i.e. other) sequence number. If an **AK** with the wrong sequence number is received, then the Error Phase is entered.

When a **DT** is received, it is acknowledged with **AK** (with the next sequence number after the one in the **DT**). However, if a further **DT** is received before the **AK** is sent, the Error Phase is entered. If the **DT** bears the sequence number which is expected, a **DatInd** is issued. Otherwise the **DT** is not delivered to the User.

If a further **CR** is received before any **DT**s or **AK**s, a **CC** is sent. If a **DR** is received by either station the Disconnection Phase is entered. If any PDU apart from **DT**, **AK**, **CR** (initial re-transmission only), or **DR** is received, then the Error Phase is entered.

10.1.3.4 Disconnection Phase

A **DisReq** leads to a **DR** being sent. On receipt of **DC** the connection is terminated and a new connection may be attempted; the same is true if **DR** is received instead. If a further **DR** is received, then **DC** is sent. Any other kind of PDU is ignored. If no response to **DR** is received within period **P**, the **DR** is re-transmitted. A maximum of **N** disconnection attempts (i.e. **N** periods of value **P**) are permitted. After this, the connection is considered to have been terminated and a new connection may be attempted.

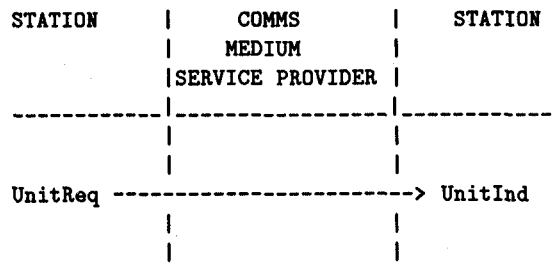


Figure 10.3: Communications Medium Service Primitives

When a **DR** is received, it is acknowledged with **DC**. If a connection is established, a **DisInd** is issued. After this, a new connection may be attempted. Any PDU other than **DR** or **CR** which arrives subsequently is ignored.

10.1.3.5 Error Phase

A Protocol error leads to the Error Phase being entered and **DR** being sent. This is identical to the Disconnection Phase except that the station which detected the error also issues **DisInd** before sending the **DR**.

10.1.4 Communications Medium Service Description

The Communication Medium Service operates between a pair of stations, addressed as **a** or **b**. The Communications Medium Service is connectionless, and is accessed by **UnitReq** and **UnitInd** (Unit Request and Indication) Service Primitives, which carry Service Data Units corresponding to Abracadabra Protocol Data Units. The communications medium is full-duplex and transparent, but does not guarantee delivery. Messages may be lost, but may not be misordered, corrupted, invented, or duplicated. The Service Primitives are related as shown in Figure 10.3.

Either station may issue **UnitReq**, which may be delivered as **UnitInd** or may be lost.

10.1.5 Model

The Service and Protocol should be modelled as shown in Figure 10.4.

10.2 Deficiencies in the Informal Description

10.2.1 Flow Control (Clause 10.1.2)

10.2.1.1 Deficiency

Is it reasonable that the informal description should stipulate that flow control by back-pressure be modelled?

10.2.1.2 Resolution

Flow control by back-pressure is an implicit feature of many OSI Service definitions, although it is not normally referred

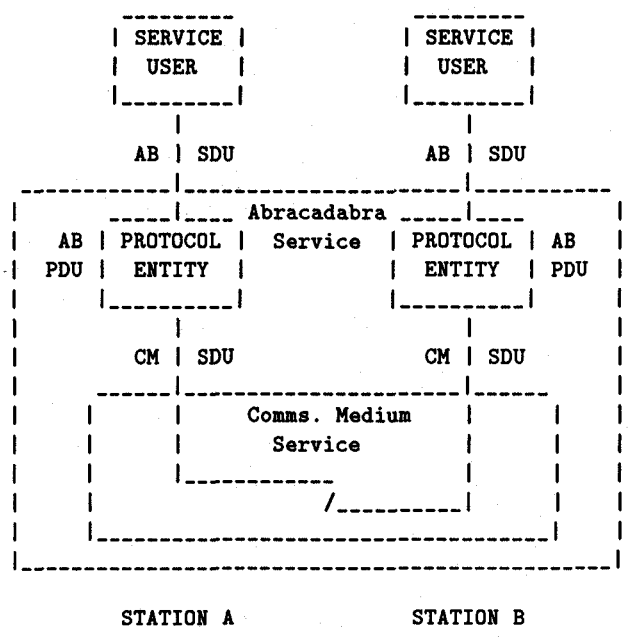


Figure 10.4: Abracadabra Service and Protocol Model

to by this explicit name. The mechanisms for realising flow control by back-pressure lie partly within the Service Provider (i.e. the Protocol) and partly in the inter-Layer interface between the Service Users and the Service Provider. Although the former is subject to standardisation, the latter is implementation-dependent and therefore not subject to standardisation. Therefore, although the precise mechanisms for achieving flow control by back-pressure would not normally be included in a Service description, it is permissible to refer to the end-to-end effect of these.

10.2.2 Premature Transmission of DT (Clause 10.1.3.3)

10.2.2.1 Deficiency

Is it reasonable that the informal description should regard the reception of a further **DT** before an **AK** can be transmitted as an error?

10.2.2.2 Resolution

The intention was to trap misuse of the protocol by the transmitter, or to detect that the timeout period was too short. However, this intention was probably misguided; this case should not have been regarded as an error.

10.2.3 Stopping Retransmission on Error (Clauses 10.1.3.2 and 10.1.3.3)

10.2.3.1 Deficiency

Should retransmission of a **CR** or **DT** be stopped if the Error Phase is entered?

10.2.3.2 Resolution

Any current retransmission on timeout of a PDU should cease on entry to the Error Phase.

10.2.4 Retransmission Limit and Period (Clause 10.1.3.1)

10.2.4.1 Deficiency

What should be the behaviour of the protocol if the parameters **N** and **P** are negative?

10.2.4.2 Resolution

The intention was that the protocol should refuse to accept or transmit any messages.

10.2.5 Repeated ConReq (Clause 10.1.2)

10.2.5.1 Deficiency

Should **ConReq** be accepted while a connection is being attempted or is current? More generally, should the behaviour of the Service under incorrect use by the Service User be described?

10.2.5.2 Resolution

The intention was that a **ConReq** should be issued only once to establish a connection. More generally, the actual FDT being used affects how Service User misbehaviour should be most naturally described.

10.2.6 DR when Disconnected (Clauses 10.1.3.2 and 10.1.3.4)

10.2.6.1 Deficiency

The informal description says that receipt of any PDU other than **CR** is ignored in the Connection Phase. However, in the Disconnection Phase it says that receipt of **DR** should result in **DC**. Which is right?

10.2.6.2 Resolution

The description of the Disconnection Phase is right. The intention was that receipt of **DR** when not connected should result in **DC**.

10.2.7 Connection Refusal (Clause 10.1.3.2)

10.2.7.1 Deficiency

If **DR** is received in response to **CR**, should a **DisInd** be given to the User?

10.2.7.2 Resolution

The intention was to inform the User by **DisInd** if the connection was refused by **DR**.

10.2.8 Connection Refusal (Clause 10.1.3.2)

10.2.8.1 Deficiency

Should the Disconnection Phase be entered if the connection is refused by the other party (i.e. is the sequence **CR**, **DR**, **DC** correct as stated)?

10.2.8.2 Resolution

Although the informal description is viable, it was intended that connection refusal be by **CR**, **DR** only. The informal description should refer to a **DR** being sent and the 'not connected' state being entered, rather than the Disconnection Phase being entered.

10.2.9 Ignoring Out-of-sequence Data (Clause 10.1.3.3)

10.2.9.1 Deficiency

Is it the **DatInd** rather than the **DT** which is not delivered to the User if the received sequence number is wrong?

10.2.9.2 Resolution

It should be the **DatInd**; the informal description is a bit loose.

10.3 Estelle Description

10.3.1 Architecture of the Formal Descriptions

The modules in the descriptions are **systemprocesses**, and so run asynchronously. As these modules are not refined into submodules, the global behaviour of these descriptions would not changed if they were designated **systemactivities**. The crucial point is that they are distinct systems.

10.3.1.1 Architecture of the Service Description

The modules and interaction points for the Abracadabra Service description are shown in Figure 10.5. The Abracadabra Service Provider in Estelle is modelled by two identical processes, one for each SAP. Of course, there are other solutions possible which only use one process. The reason for choosing two processes is that there may be a possible delay between the reception of a Service Primitive by the Service Provider and the sending of the corresponding Service Primitive to the respective Service User. This delay is modelled by the communication via channel **INTERNAL** between the two **SAPmanagers**. The Abracadabra Service is described in 10.3.3.

10.3.1.2 Architecture of the Protocol Description

The modules and interaction points for the Abracadabra Protocol description are shown in Figure 10.6. The Abracadabra Protocol is described in 10.3.4.

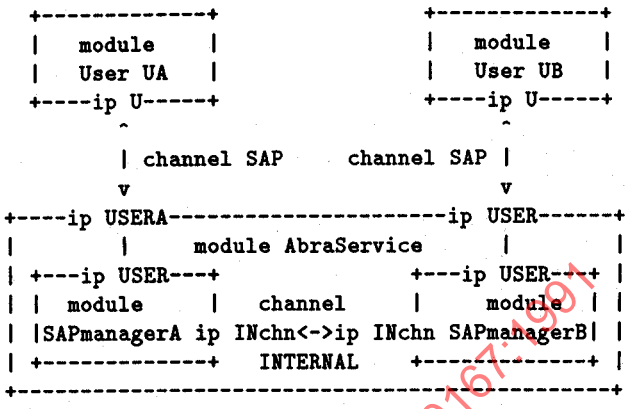


Figure 10.5: Architecture of the Abracadabra Service in Estelle

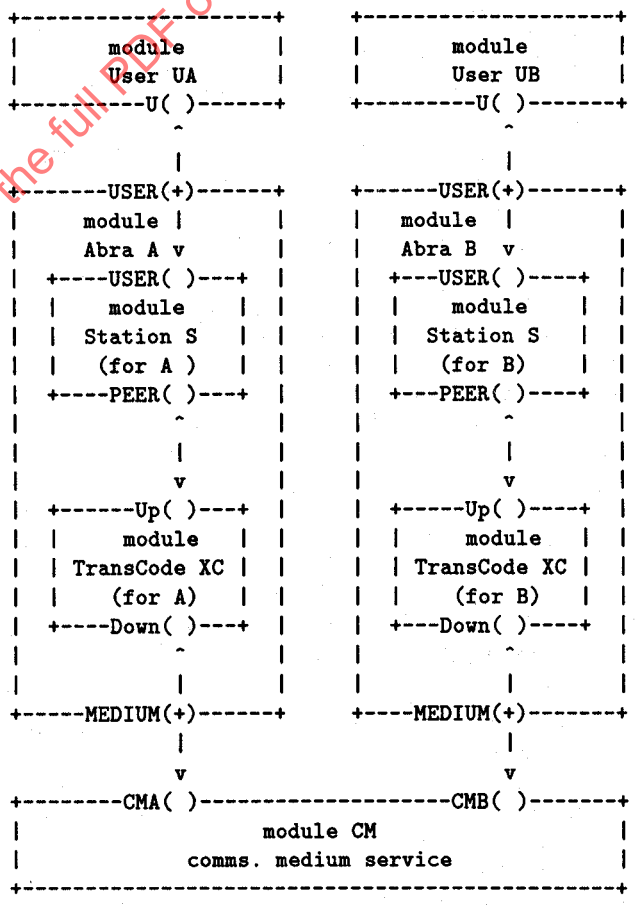


Figure 10.6: Architecture of the Abracadabra Protocol in Estelle

10.3.2 Explanation of Approach

10.3.2.1 Explanation of Service Approach

If an Abracadabra Service description were produced using only one process for the Service Provider, time constraints would have to be introduced for the sending of Service Primitives corresponding to received ones. The two-process solution has been used for the sake of simplicity.

When one **SAPmanager** receives a Service Primitive from a Service Access Point, it will send an internal message to the **SAPmanager** for the other Service Access Point, which will the result in a Service Primitive being sent to the User at this Service Access Point. For example, a **ConReq** from User **A** will invoke an **IConReqInd**, which will result in a **ConInd** to User **B**. The names for the internal messages are defined in a straightforward manner, and are believed to be self-explanatory.

The main difference between the Abracadabra Service and Abracadabra Protocol descriptions can be seen in the fact that the Service is an abstraction of the Protocol. The interface to the User is, of course, identical in both cases. In the Service description there are no re-transmissions and there is no alternating bit, because these things cannot be seen by the User. Therefore, it is possible to omit them in order to abstract from the Protocol. Furthermore, the processes of the Service Provider at the Service Access Points do not have to communicate through an unreliable medium, which is the Service Provider of the underlying Layer. Instead, the inability of the Service Provider to establish or maintain a connection in some cases (the cases where in the Protocol's **N** re-transmissions have failed) is modelled in the Service description by the use of non-determinism.

10.3.2.2 Explanation of Protocol Approach

The structure of the Abracadabra Protocol description is chosen to illustrate a way to solve the common problem of peer-to-peer communication. Specifically, the problem is this: normally, peers are considered to communicate with each other, but within the OSI reference model, they cannot. An (N)-entity cannot communicate directly with another (N)-entity; rather it communicates with an (N-1)-entity, which provides the necessary Service to convey the peer-to-peer communications. When describing Protocols, therefore, it is not possible for an (N)-entity to 'send a **CR**'. Instead, it must package a **CR** up into an (N-1) Service Data Unit, and its peer must unwrap the **CR** from the (N-1) Service Data Unit.

The description is broken up into several modules. The Protocol itself is described in the **Station** module. The reader primarily interested in the Protocol should concentrate on this module. The Abracadabra **Station** modules have been written as though they were communicating directly, so they can 'send a CR' or 'receive an AK'. To accommodate the OSI architecture, a **TransCode** module has been interposed between each Abracadabra **Station** and the communications medium. This **TransCode** module encodes ('packages') the peer-to-peer communication from the **Station** into the **UnitData** of a **UnitReq** that the communications medium requires. The **TransCode** module also decodes ('unwraps') the **UnitData** of a **UnitInd** that the communications medium provides into a peer-to-peer communication.

This structuring is done using features of Estelle that make the sub-structures 'invisible' to the rest of the system. Note that the module **Abra** serves simply to form the sub-structures and connect them together; it does not have any transitions. Note also that the **TransCode** module has only one (unnamed) state, thus **from** and **to** clauses are unnecessary for its transitions.

Although back-pressure flow-control has a global, end-to-end effect, its precise realisation is a local and implementation-dependent matter. The Protocol description given in 10.3.4 does not therefore deal with back-pressure flow-control. The changes to describe this in general terms are as follows.

The approach taken is based on the introduction of primitive predicates (i.e. boolean-valued functions). These functions are 'true' when the intended receiver of an interaction wishes to assert back-pressure flow-control. These primitive functions are given names like **ReceiverBlocked**. By their nature, these clearly show that back-pressure flow-control is indeed a local implementation issue which depends on the availability of local resources. It is necessary to make the firing of those transitions that have output on a channel subject to back-pressure flow-control depend on the value of these primitive functions.

Note that these primitive functions are necessarily primitive: it is unlikely that they could be written in Estelle. From the point of view of the formal semantics of Estelle, a description is technically incomplete until all primitive functions and procedures have been formally described in terms of the semantic model. However, for an implementation it is enough to have a description of the desired behaviour which is sufficiently detailed for an implementation to be made. The intended meaning of a predicate like **ReceiverBlocked** could be formalised in terms of the Estelle semantic model: it would be defined in terms of the queue in **ReceiverUserInstance**.

In the following changes it was decided to describe both parties which are subject to back-pressure, i.e. the transmitting and the receiving stations. Since the **TransCode** module is a simple translator, its actions are not made subject to flow-control; instead the **Station** module is made the locus of this activity. The effect on the sending User or on the Communications Medium has not been shown since the actions of these modules are left unspecified. However, the changes that would be required to describe back-pressure flow-control in these modules should be clear from the examples.

The specific changes to the Abracadabra Protocol description to implement back-pressure flow-control in this way are as follows:

- Two boolean-valued functions should be declared in the body **StationBody** for **Station**. The effect of these functions can be described in terms of the queues of the Communications Medium module (**CM**) and the Users (**UA** and **UB**).

```
function MediumBlocked : boolean;
primitive;

function ReceiverBlocked : boolean;
primitive;
```

- Two **provided** clauses in the body **StationBody** for **Station** should be modified as follows. Transitions 13 and 16 then become:

```
{ send data in DT PDU }
from ESTAB to same
    when USER.DatReq
        provided not Sending and
          not MediumBlocked
            begin { 13 }
                OldData := UserData;
                output
                  PEER.DT(SendSeq, OldData);
                OldSendSeq := SendSeq;
                SendSeq :=
                  (SendSeq + 1) mod 2;
                Sending := true;
                { turn on retransmission
                  timer }
                DTRetranRemaining := N-1;
            end;

{ receive data in DT PDU }
from ESTAB to same
    when PEER.DT
        provided not ReceiverBlocked
        begin { 16 }
            if Seq = RecvSeq then
                begin
                    output
                      USER.DatInd(UserData);
                    RecvSeq :=
                      (RecvSeq + 1) mod 2;
                end;
            { send AK with next expected
              sequence number }
            output PEER.AK(RecvSeq);
            DTorAK := true;
        end;
```

10.3.3 Formal Description of the Service

```
specification AbracadabraService;

  default individual queue;
  type UserDataType = ...;

  channel SAP (user, provider);
      by user:
            ConReq;
            ConResp;
            DatReq(UserData: UserDataType);
            DisReq;
      by provider:
            ConInd;
            ConConf;
            DatInd(UserData: UserDataType);
            DisInd;

channel INTERNAL (A, B);
    by A, B:
            IConReqInd;
            IConRespConf;
            IDat(UserData: UserDataType);
            IDis;

module User systemprocess;
    ip U: SAP(user);
end;

body UserBody for User;
end;

module AbraService;
    ip USERA: SAP(provider);
       USERB: SAP(provider);
end;

body AbraServiceBody for AbraService;

    module SAPmanagerA systemprocess;
        ip USER: SAP(provider);
           INchn: INTERNAL(A);
    end;

    body SAPmanagerBodyA for SAPmanagerA;

        state
           DISCONNECTED, CALLED, CALLING,
             CONNECTED;

        stateset
           DISoccurs =
             [CALLED, CALLING, CONNECTED];

        initialize
           to DISCONNECTED
               begin end; { no variables }

        trans

        { *** Connection Phase *** }

        from DISCONNECTED to same
               when USER.ConReq
                      begin { 1 }
                        output USER.DisInd;
                      end;
        from DISCONNECTED to same
               when INchn.IDat(UserData)
                      begin { 2 }
                        output INchn.IDis;
                      end;
        from DISCONNECTED to CALLING
               when USER.ConReq
                      begin { 3 }
                        output
```

```
                INchn.IConReqInd;
            end;
from DISCONNECTED to CALLED
      when INchn.IConReqInd
            begin { 4 }
              output USER.ConInd;
            end;
from CALLED to CONNECTED
      when USER.ConResp
            begin { 5 }
              output INchn.
                IConRespConf;
            end;
from CALLED to CONNECTED
      when USER.ConReq
        { collision situation }
            begin { 6 }
              output USER.ConConf;
              output INchn.
                IConRespConf;
            end;
from CALLING to CONNECTED
      when INchn.IConRespConf
            begin { 7 }
              output USER.ConConf;
            end;
from CALLING to CONNECTED
      when INchn.IConReqInd
        { collision situation }
            begin { 8 }
              output USER.ConConf;
              output INchn.
                IConRespConf;
            end;

{ *** Data Phase *** }

from CONNECTED to same
      when USER.DatReq(UserData)
            begin { 9 }
              output INchn.
                IDat(UserData);
            end;
from CONNECTED to same
      when INchn.IDat(UserData)
            begin { 10 }
              output USER.
                DatInd(UserData);
            end;
from CONNECTED to DISCONNECTED
      when USER.DatReq(UserData)
            begin { 11 }
              output USER.DisInd;
              output INchn.IDis;
            end;
from Connected to CALLED
      when INchn.IConReqInd
            begin { 12 }
              output USER.ConInd;
            end;
from DISoccurs to DISCONNECTED
            { spontaneous disconnection
              by the Provider }
                begin { 13 }
                  output USER.DisInd;
                  output INchn.IDis;
                end;
      from DISoccurs to DISCONNECTED
            when USER.DisReq
                begin { 14 }
                  output INchn.IDis;
                end;
      from DISoccurs to DISCONNECTED
            when INchn.IDis
                begin { 15 }
                  output USER.DisInd;
                end;
end;  { SAPmanagerBodyA }

module SAPmanagerB systemprocess;
      ip USER : SAP(provider);
         INchn: INTERNAL(B);
end;

body SAPmanagerBodyB for SAPmanagerB;

      state
          DISCONNECTED, CALLED, CALLING,
            CONNECTED;

      stateset
          DISoccurs =
            [CALLED, CALLING, CONNECTED];

      initialize
          to DISCONNECTED
              begin end; { no variables }

      trans

      { *** Connection Phase *** }

      from DISCONNECTED to same
            when USER.ConReq
                begin { 1 }
                  output USER.DisInd;
                end;
      from DISCONNECTED to same
            when INchn.IDat(UserData)
                begin { 2 }
                  output INchn.IDis;
                end;
      from DISCONNECTED to CALLING
            when USER.ConReq
                begin { 3 }
                  output INchn.
                    IConReqInd;
                end;
      from DISCONNECTED to CALLED
            when INchn.IConReqInd
                begin { 4 }
                  output USER.ConInd;
```

```
                end;
from CALLED to CONNECTED
        when USER.ConResp
                begin { 5 }
                  output INchn.
                    IConRespConf;
                end;
from CALLED to CONNECTED
        when USER.ConReq
          { collision situation }
                begin { 6 }
                  output USER.ConConf;
                output INchn.
                  IConRespConf;
                end;
from CALLING to CONNECTED
        when INchn.IConRespConf
                begin { 7 }
                  output USER.ConConf;
                end;
from CALLING to CONNECTED
        when INchn.IConReqInd
          { collision situation }
                begin { 8 }
                  output USER.ConConf;
                  output INchn.
                    IConRespConf;
                end;

{ *** Data Phase *** }

from CONNECTED to same
        when USER.DatReq(UserData)
                begin { 9 }
                  output INchn.
                    IDat(UserData);
                end;
from CONNECTED to same
        when INchn.IDat(UserData)
                begin { 10 }
                  output USER.
                    DatInd(UserData);
                end;
from CONNECTED to DISCONNECTED
        when USER.DatReq(UserData)
                begin { 11 }
                  output USER.DisInd;
                  output INchn.IDis;
                end;

from Connected to CALLED
        when INchn.IConReqInd
                begin { 12 }
                  output USER.ConInd;
                end;
from DISoccurs to DISCONNECTED
        { spontaneous disconnection
          by the Provider }
                begin { 13 }
                  output USER.DisInd;
                  output INchn.IDis;
                end;
        from DISoccurs to DISCONNECTED
                when USER.DisReq
                        begin { 14 }
                          output INchn.IDis;
                        end;
        from DISoccurs to DISCONNECTED
                when INchn.IDis
                        begin { 15 }
                          output USER.DisInd;
                        end;
    end;  { SAPmanagerBodyB }

    { main body for AbraServiceBody }

    modvar
            A: SAPmanagerA;
            B: SAPmanagerB;

    initialize
            begin
                    init A with SAPmanagerBodyA;
                    init B with SAPmanagerBodyB;
                     attach USERA to A.USER;
                     attach USERB to B.USER;
                    connect A.INchn to B.INchn;
            end;
    end; { AbraServiceBody }

{ main body for specification
  AbracadabraService }

  modvar
      UA, UB: User;
      AS    : AbraService;

  initialize
      begin
              init UA with UserBody;
              init UB with UserBody;
              init AS with AbraServiceBody;
              connect UA.U to AS.USERA;
              connect UB.U to AS.USERB;
      end;

end. { Specification AbracadabraService }
```

10.3.4 Formal Description of the Protocol

```
specification AbracadabraProtocol;

  default individual queue;
  timescale seconds;

  const
      N = any integer; { number of transmission
        attempts }
      P = any integer; { delay amount for
        timers }

  type
```

```
    SeqType = 0..1;  { sequence number type }
    UserDataType = ...;

    PduType = (CR, CC, DT, AK, DR, DC);

    UnitDataType = record
        Pdu    : PduType;
        SeqNo  : SeqType;
        UData  : UserDataType
    end;

channel USAP(user, provider);
    by user:
        ConReq;
        ConResp;
        DatReq(UserData : UserDataType);
        DisReq;
    by provider:
        ConInd;
        ConConf;
        DatInd(UserData : UserDataType);
        DisInd;

channel PeerCode(peer, coder);
    by peer, coder:
        CR;
        CC;
        DT(Seq : SeqType;
          UserData : UserDataType);
        AK(Seq : SeqType);
        DR;
        DC;

channel MSAP(user, provider);
    by user:
        UnitReq(UnitData: UnitDataType);
    by provider:
        UnitInd(UnitData: UnitDataType);

module User systemprocess;
    ip U : USAP(user);
end;

body UserBody for User;
end;

module Cms systemprocess;
    ip CMA, CMB : MSAP(provider);
end;

body CmsBody for Cms;
external;

module Abra systemprocess;
    ip USER   : USAP(provider);
       MEDIUM : MSAP(user);
end;

body AbraBody for Abra;

    module Station process;
        ip USER : USAP(provider);
           PEER : PeerCode(peer);
    end;

    body StationBody for Station;

        state
            CLOSED, CRSENT, CRRECV, ESTAB,
              DRSENT;

        stateset
            CRignore = [CRRECV];
            CCignore =
              [CLOSED, CRRECV, DRSENT];
            DTignore =
              [CLOSED, CRSENT, CRRECV, DRSENT];
            AKignore =
              [CLOSED, CRSENT, CRRECV, DRSENT];
            DCignore =
              [CLOSED, CRSENT, CRRECV];
            ConReqIgnore  =
              [CRSENT, CRRECV, ESTAB, DRSENT];
            ConRespIgnore =
              [CLOSED, CRSENT, ESTAB, DRSENT];
            DatReqIgnore  =
              [CLOSED, CRSENT, CRRECV, DRSENT];
            DisReqIgnore  =
              [CLOSED, DRSENT];

        var
            Sending : boolean;
            SendSeq, RecvSeq : SeqType;
            OldSendSeq : SeqType;
            CRRetranRemaining : integer;
            DTRetranRemaining : integer;
            DRRetranRemaining : integer;
            OldData : UserDataType;
            DTorAK : boolean;

            procedure InitVar;
                begin
                  Sending := false;
                  SendSeq := 0;
                  RecvSeq := 0;
                  { setting the following
                    counters to -1 guarantees
                    the predicates that check
                    them will fail. }
                  CRRetranRemaining := -1;
                  DTRetranRemaining := -1;
                  DRRetranRemaining := -1;
                  DTorAK := false;
                end;

        initialize
            to CLOSED
                begin { 1 }
                  { Variables are initialized
                    when leaving CLOSED state,
                    since the protocol module
                    may cycle through CLOSED
```

```
            repeatedly. }
        end;

trans

{ *** Connection Phase *** }

    { user requests connection }
    from CLOSED to CRSENT
        when USER.ConReq
            begin { 2 }
              { initialize module
                variables whenever
                leaving CLOSED }
              InitVar;
              output PEER.CR;
              CRRetranRemaining := N-1;
            end;
    { other user accepted connection }
    from CRSENT to ESTAB
        when PEER.CC
            begin { 3 }
              output USER.ConConf;
              CRRetranRemaining := -1;
            end;
    { colliding CRs }
    from CRSENT to ESTAB
        when PEER.CR
            begin { 4 }
              output USER.ConConf;
              CRRetranRemaining := -1;
            end;
    { other user rejected connection }
    from CRSENT to CLOSED
        when PEER.DR
            begin { 5 }
              output USER.DisInd;
              CRRetranRemaining := -1;
            end;
    { sender requests disconnection }
    from CRSENT to DRSENT
        when USER.DisReq
            begin { 6 }
              output PEER.DR;
              CRRetranRemaining := -1;
              DRRetranRemaining := N-1;
            end;
    { retransmission timer for CR
      fires }
    from CRSENT to same
        provided CRRetranRemaining > 0
          delay (P)
            begin { 7 }
              CRRetranRemaining :=
                CRRetranRemaining - 1;
              output PEER.CR;
            end;
    { terminate retransmission of CR }
    from CRSENT to DRSENT
        provided CRRetranRemaining = 0
          delay (P) { allow time for
                      last CR }
            begin { 8 }
              { enter error phase }
              output USER.DisInd;
              output PEER.DR;
              CRRetranRemaining :=
                -1;
              DRRetranRemaining :=
                N - 1;
            end;
    { receive connect request from
      peer entity }
    from CLOSED to CRRECV
        when PEER.CR
            begin { 9 }
              { initialize module
                variables whenever
                leaving CLOSED }
              InitVar;
              output USER.ConInd;
            end;
    { user accepts connection }
    from CRRECV to ESTAB
        when USER.ConResp
            begin { 10 }
              output PEER.CC;
            end;
    { user rejects connection }
    from CRRECV to CLOSED
        when USER.DisReq
            begin { 11 }
              output PEER.DR {just
                once }
            end;
    { other user disconnected }
    from CRRECV to CLOSED
        when PEER.DR
            begin { 12 }
              output USER.DisInd;
              output PEER.DC;
            end;

{ *** Data Transfer Phase *** }

    { send data in DT PDU }
    from ESTAB to same
        when USER.DatReq
            provided not Sending
              begin { 13 }
                OldData := UserData;
                output PEER.
                  DT(SendSeq, OldData);
                OldSendSeq := SendSeq;
                SendSeq :=
                  (SendSeq + 1) mod 2;
                Sending := true;
                { turn on retrans-
                  mission timer }
                DTRetranRemaining :=
                  N - 1;
              end;
```

```
{ receive ack with correct sequence
  number in AK PDU }
from ESTAB to same
    when PEER.AK
        provided Seq = SendSeq
            begin { 14 }
              Sending := false;
              { turn off retrans-
                mission timer }
              DTRetranRemaining :=
                -1;
              DTorAK := true;
            end;
{ receive acknowledgement with
  incorrect sequence number }
from ESTAB to DRSENT
    when PEER.AK
        provided Seq <> SendSeq
            begin { 15 }
              { enter error phase }
              output USER.DisInd;
              output PEER.DR;
              DTorAK := true;
              DTRetranRemaining :=
                -1;
              DRRetranRemaining :=
                N-1;
            end;
{ receive data in DT PDU }
from ESTAB to same
    when PEER.DT
        begin { 16 }
          if Seq = RecvSeq then
            begin
              output USER.
                DatInd(UserData);
              RecvSeq := (RecvSeq
                + 1) mod 2;
            end;
          { send AK with next
            expected sequence
            number }
          output PEER.AK(RecvSeq);
          DTorAK := true;
        end;
from ESTAB to same
    when PEER.CR
        provided not DTorAK
            begin { 17 }
              output PEER.CC;
            end;
from ESTAB to DRSENT
    when PEER.CR
        provided DTorAK
            begin { 18 }
              { enter error phase }
              output USER.DisInd;
              output PEER.DR;
              DTRetranRemaining :=
                -1;
              DRRetranRemaining :=
                N-1;
            end;
from ESTAB to DRSENT
    when PEER.CC
        begin { 19 }
          { enter error phase }
          output USER.DisInd;
          output PEER.DR;
          DTRetranRemaining := -1;
          DRRetranRemaining := N-1;
        end;
    when PEER.DC
        begin { 20 }
          { enter error phase }
          output USER.DisInd;
          output PEER.DR;
          DTRetranRemaining := -1;
          DRRetranRemaining := N-1;
        end;
{ retransmission timer for DT
  fires }
from ESTAB to same
    provided DTRetranRemaining > 0
      delay (P)
        begin { 21 }
          DTRetranRemaining :=
            DTRetranRemaining - 1;
          output PEER.
           DT(OldSendSeq, OldData);
        end;
{ terminate retransmission of DT }
from ESTAB to DRSENT
    provided DTRetranRemaining = 0
        delay (P)
            begin { 22 }
              { enter error phase }
              output USER.DisInd;
              output PEER.DR;
              DTRetranRemaining :=
                -1;
              DRRetranRemaining :=
                N - 1;
            end;

{ *** Disconnection Phase *** }

    { receive disconnect request from
      user }
    from ESTAB to DRSENT
        when USER.DisReq
            begin { 23 }
              output PEER.DR;
              DTRetranRemaining :=
                -1;
              DRRetranRemaining :=
                N - 1;
            end;
    { receive DC }
    from DRSENT to CLOSED
        when PEER.DC
            begin { 24 }
```

```
          DRRetranRemaining := -1;
        end;
{ receive DR }
from DRSENT to CLOSED
    when PEER.DR
        begin { 25 }
          DRRetranRemaining := -1;
        end;
{ receive DR }
from ESTAB to CLOSED
    when PEER.DR
        begin { 26 }
          output USER.DisInd;
          output PEER.DC;
          DTRetranRemaining := -1;
        end;
{ reply to retransmitted DR }
from CLOSED to same
    when PEER.DR
        begin { 27 }
          output PEER.DC;
        end;
{ retransmission timer for DR
  fires }
from DRSENT to same
    provided DRRetranRemaining > 0
      delay (P)
        begin { 28 }
          DRRetranRemaining :=
            DRRetranRemaining - 1;
          output PEER.DR;
        end;
{ terminate retransmission of DR }
from DRSENT to CLOSED
    provided DRRetranRemaining = 0
      delay (P)
        begin { 29 }
          { The connection is
            regarded as closed. }
          DRRetranRemaining := -1;
        end;
{ ignore other PDU's }
from CRignore to same
    when PEER.CR
        begin { 30 }
        end;
from CCignore to same
    when PEER.CC
        begin { 31 }
        end;
from DTignore to same
    when PEER.DT
        begin { 32 }
        end;
from AKignore to same
    when PEER.AK
        begin { 33 }
        end;
from DCignore to same
    when PEER.DC
        begin { 34 }
        end;
from ConReqIgnore to same
    when USER.ConReq
        begin { 35 }
        end;
from ConRespIgnore to same
    when USER.ConResp
        begin { 36 }
        end;
from DatReqIgnore to same
    when USER.DatReq
        begin { 37 }
        end;
from DisReqIgnore to same
    when USER.DisReq
        begin { 38 }
        end;
end; { StationBody }

{ **** The TransCode section  **** }

{ See the Explanation of Approach, describing
  the structure of the specification }

module TransCode process;
    ip Up   : PeerCode(coder);
       Down : MSAP(user);
end;


body TransCodeBody for TransCode;

    var SDU: UnitDataType;

    procedure BuildCR(
      var SDU: UnitDataType);
    begin
        SDU.PDU := CR
    end;

    procedure BuildCC(
      var SDU: UnitDataType);
    begin
        SDU.PDU := CC
    end;

    procedure BuildDT(
      Seq: SeqType; Data: UserDataType;
        var SDU: UnitDataType);
    begin
        SDU.PDU    := DT;
        SDU.SeqNo := Seq;
        SDU.UData := Data
    end;

    procedure BuildAK(
      Seq: SeqType; var SDU: UnitDataType);
    begin
        SDU.PDU    := AK;
        SDU.SeqNo := Seq
    end;
```

```
    procedure BuildDR(
      var SDU: UnitDataType);
    begin
        SDU.PDU := DR
    end;

    procedure BuildDC(
      var SDU: UnitDataType);
    begin
        SDU.PDU := DC
    end;

trans
    when Up.CC
        begin { 1 }
          BuildCC(SDU);
          output Down.UnitReq(SDU)
        end;
    when Up.CR
        begin { 2 }
          BuildCR(SDU);
          output Down.UnitReq(SDU)
        end;
    when Up.DT
        begin { 3 }
          BuildDT(Seq, UserData, SDU);
          output Down.UnitReq(SDU)
        end;
    when Up.AK
        begin { 4 }
          BuildAK(Seq, SDU);
          output Down.UnitReq(SDU)
        end;
    when Up.DR
        begin { 5 }
          BuildDR(SDU);
          output Down.UnitReq(SDU)
        end;
    when Up.DC
        begin { 6 }
          BuildDC(SDU);
          output Down.UnitReq(SDU)
        end;

    when Down.UnitInd
        provided (UnitData.PDU = CR)
            begin { 7 }
              output Up.CR
            end;
    when Down.UnitInd
        provided (UnitData.PDU = CC)
            begin { 8 }
              output Up.CC
            end;
    when Down.UnitInd
        provided (UnitData.PDU = DT)
            begin { 9 }
              output Up.DT(UnitData.SeqNo,
                UnitData.UData)
            end;
    when Down.UnitInd
        provided (UnitData.PDU = AK)
            begin { 10 }
              output Up.
                AK(UnitData.SeqNo)
            end;
    when Down.UnitInd
        provided (UnitData.PDU = DR)
            begin { 11 }
              output Up.DR
            end;
    when Down.UnitInd
        provided (UnitData.PDU = DC)
            begin { 12 }
              output Up.DC
            end;
  end; { TransCodeBody }

  { main body for AbraBody }
  modvar
      S : Station;
      XC : TransCode;
  initialize
      begin
          { instantiate the modules }
          init S with StationBody;
          init XC with TransCodeBody;

          { make connections }
          attach USER to S.USER;
          connect S.PEER to XC.Up;
          attach MEDIUM to XC.Down;
      end;
end; { AbraBody }

{ main body for specification
  AbracadabraProtocol }
  modvar
      A, B   : Abra;
      UA, UB : User;
      CM     : Cms;

  initialize
      provided (N > 0) and (P > 0)
          begin { 1 }
              init UA  with UserBody;
              init UB  with UserBody;
              init A   with AbraBody;
              init B   with AbraBody;
              init CM  with CmsBody;
              connect UA.U to A.USER;
              connect UB.U to B.USER;
              connect A.MEDIUM to CM.CMA;
              connect B.MEDIUM to CM.CMB;
          end;

end. { Specification AbracadabraProtocol }
```

```
specification AbracadabraService
  [a] (stationa, stationb : Address) : noexit

  (* type definitions *)

  behaviour
    Connection [a] (stationa, stationb : Address)
  ||
    Backpressure [a]

  where

  process Connection [a]
   (stationa, stationb : Address) : noexit :=
    ...
  endproc

  process Backpressure [a] : noexit :=
    ...
  endproc

endspec
```

Figure 10.7: Outline Decomposition of the Abracadabra Service in LOTOS

10.3.5 Subjective Assessment

The writing of the Abracadabra Protocol description in Estelle was fairly straightforward. The reader will note that the order of the transitions of the StationBody closely follows the order of the original description. Indeed, this portion of the specification was written almost as a translation of the text. In the course of writing the description, it was noted that there were contradictory requirements in the case where a dr is received and no connection is established; this lead to a *deficiency report*. In addition, it was noted that the phrase 'the disconnection phase is entered' was unclear, and it was necessary to guess the correct interpretation to write the formal description; clearly different interpretations were necessary at different points in the Protocol.

After the first version of the description was complete, it was analysed using automated tools, and additional deficiencies were found. Subsequent discrepancies were noted in the course of coordinating the three descriptions.

10.4 LOTOS description

```
(*-----------------------------------------------
```

10.4.1 Architecture of the Formal Descriptions

The description is divided into the description of the Service and of the Protocol. Both are self-contained and independent. The architecture of the descriptions follows that suggested in the informal description. The decomposition of the Service description is given in Figure 10.7, and that of the Protocol is given in Figure 10.8.

```
specification AbracadabraProtocolEntity [a, m]
 (N : Nat, P : Nat, station : Address) : noexit

  (* type definitions *)

  behaviour

    Service [a] (station)
  |[a]|
    Protocol [a, m] (N, P)
  |[m]|
    CMService [m]

  process Service [a]
   (station : Address) : noexit :=
    ...
  endproc

  process CMService [m] : noexit :=
    ...
  endproc

  process Protocol [a, m]
   (N : Nat, P : Nat) : noexit :=
    ...
  endproc

endspec
```

Figure 10.8: Outline Decomposition of the Abracadabra Protocol in LOTOS

The Abracadabra Service is described in 10.4.3. The top-level structure reveals one gate **a** for communication, with an address value to distinguish the stations. The behaviour of the description is described in two interleaved main processes: **Connection**, which controls the handling of connections; and **Backpressure**, which controls the flow of data.

The Abracadabra Protocol is described in 10.4.4. The top-level structure reveals two gates for communication with the User and with the underlying Medium:

- **a** this is the upper Service gate to the Abracadabra Service User
- **m** this is the Communications Medium gate

The structure of events at these gates is the usual one:

```
a ! Station ! ASP (...)

m ! User ! MSP (...)
```

where:

a) **Station** is the station identification; and

b) **ASP** constructs a value for the Abracadabra Service Primitive sort; and

c) **User** is the user identification; and

d) **MSP** constructs a value for the Medium Service Primitive sort.

The behaviour of the description is described in three interleaved main processes: **Service** and **CMService** describe the behaviour at the gates and **a** and **m** respectively; and **Protocol** describes the behaviour of the Protocol itself. The behaviour of **Protocol** is split into the following processes: **Connect** and **DataTransfer**, which describe their respective phases; and **Disconnect** and **TryDisconnect**, which cover the Disconnection Phase. The 'not connected' state is also described in the **Connect** process.

Although some of the data types are common both to the Service and the Protocol, the Protocol description is self-contained. There are very few differences in these data types, arising mainly from the necessity in the protocol to select some parameters from certain structures (e.g. the Service Primitives).

10.4.2 Explanation of Approach

The descriptions use a mixture of 'constraint-oriented' and 'construction-oriented' styles. Complex systems may be described as a collection of constraints that filter out only those action denotations may take place at a given gate at each moment. Requirements may be easily translated into constraints. This approach is thus a requirement-oriented philosophy. More formally, a constraint is a behaviour, i.e. list of actions imposed on a gate. It is a higher level of concern than that of the basic sequencing (;) and choice ([]) operators.

Constraints (i.e. requirements) on a gate may be put together using synchronised composition:

```
    constraint1 [g] ...
  ||[g]||
    constraint2 [g] ...
```

or by means of interleaving:

```
    constraint1 [g] ...
  |||
    constraint2 [g] ...
```

Notice the difference from this 'parallel or' and the 'just one or':

```
    constraint1 [g] ...
  []
    constraint2 [g] ...
```

The compact versions of the parallel operator are frequently used: || for 'and' composition on every gate of each behaviour expression, and ||| for 'or' composition with no synchronisation at all. This style of description is quite terse. Usually, very few gates are considered, although synchronisation often involve many behaviours. The multi-way synchronisation feature of LOTOS is essential to this.

The additional use of a construction-oriented style is justified as follows. Protocols are usually considered as state machines. The informal description, for instance, states facts according to the so-called Phase in which the entity is. This may be regarded as a macro state.

The general style of a construction-oriented description is to derive one process per phase (roughly) such that the Protocol Entity starts behaving according to the first phase, moves to the next one under certain conditions, etc. The new phase is a 'continuation' of the current one.

10.4.3 Formal Description of the Service

The description itself is parameterised by the addresses of both stations.

```
-----------------------------------------------------*)

specification AbracadabraService [a]
 (stationa, stationb : Address) : noexit

(*-----------------------------------------------
```

Standard Library: imports some data types from the Standard Library.

```
-----------------------------------------------*)

  library
    NaturalNumber, Boolean, Set, DecDigit,
     OctetString
  endlib

(*-----------------------------------------------
```

Abracadabra Service Addresses: defines the known addresses that the stations may have.

```
-----------------------------------------------*)

  type AddressType is Boolean
    sorts Address
    opns
      A, B       : -> Address
      _eq_, _ne_ : Address, Address -> Bool
    eqns forall a1, a2 : Address
      ofsort Bool
        A eq A = true;
        A eq B = false;
        B eq A = false;
        B eq B = true;
        a1 ne a2 = not (a1 eq a2)
  endtype (* AddressType *)

(*-----------------------------------------------
```

Abracadabra Service Data Units: defines Service User data in terms of the standard type **OctetString**.

```
-----------------------------------------------*)

  type UserDataType is OctetString renamedby
    sortnames UserData for OctetString
  endtype (* UserDataType *)

(*-----------------------------------------------
```

Set of Addresses: used by the **Backpressure** process.

```
-----------------------------------------------*)

  type SetOfAddressFormalType is Set renamedby
    sortnames SetofAddress for Set
  endtype (* SetOfAddressFormalType *)

  type SetOfAddressType is SetOfAddressFormalType
    actualizedby AddressType, Boolean using
      sortnames
        Address for Element
        Bool for FBool
  endtype (* SetOfAddressType *)

(*-----------------------------------------------
```

Abracadabra Service Primitives: defined as values of sort **SP**, with some operations to extract information from the Service Primitives. There is also a mapping to **DecDigit** in order to simplify the definition of the recognition predicates. Notice that any injective mapping would suffice.

```
-----------------------------------------------*)

  type SPType is Boolean, UserDataType, DecDigit
    sorts SP
    opns
      ConReq,  ConInd, ConResp, ConConf : -> SP
      DatReq,  DatInd  : UserData -> SP
      DisReq,  DisInd  :          -> SP
      IsConReq,  IsConInd, IsConResp, IsConConf,
      IsDatReq,  IsDatInd,
      IsDisReq,  IsDisInd : SP -> Bool
      IsReq,     IsInd    : SP -> Bool
      data : SP -> UserData
      _==_ : SP, SP -> Bool
      map  : SP -> DecDigit
    eqns forall d : UserData, sp : SP
      ofsort UserData
        data (ConReq)     = <>;
        data (ConInd)     = <>;
        data (ConResp)    = <>;
        data (ConConf)    = <>;
        data (DatReq (d)) = d ;
        data (DatInd (d)) = d ;
        data (DisReq)     = <>;
        data (DisInd)     = <>;
      ofsort DecDigit
        map (ConReq)     = 0 ;
        map (ConInd)     = 1 ;
        map (ConResp)    = 2 ;
        map (ConConf)    = 3 ;
        map (DatReq (d)) = 4 ;
        map (DatInd (d)) = 5 ;
        map (DisReq)     = 6 ;
        map (DisInd)     = 7 ;
      ofsort Bool
        IsConReq (sp)  = map (sp) eq 0 ;
        IsConInd (sp)  = map (sp) eq 1 ;
        IsConResp (sp) = map (sp) eq 2 ;
        IsConConf (sp) = map (sp) eq 3 ;
        IsDatReq (sp)  = map (sp) eq 4 ;
        IsDatInd (sp)  = map (sp) eq 5 ;
        IsDisReq (sp)  = map (sp) eq 6 ;
        IsDisInd (sp)  = map (sp) eq 7 ;
        IsReq (sp) =
          IsConReq (sp) or
           (IsConResp (sp) or (IsDatReq (sp) or
            IsDisReq (sp)));

        IsInd (sp) = not (IsReq (sp));
        sp == sp = true;
  endtype (* SPType *)

(*-----------------------------------------------
```

Abracadabra Service Objects: defines the **Objects** which are used to represent the information in transit on a connection between its entry to the Service as Requests/Responses and its delivery as Indications/Confirmations. A distinct form of **Object** is defined for each pair of Service Primitive Request and Indication (or, Response and Confirmation respectively).

```
-----------------------------------------------*)

  type ObjectType is SPType
    sorts Object
    opns
      object : SP -> Object
      indication, altindication : Object -> SP
      IsCon, IsCak, IsDat, IsDis : Object -> Bool
      _==_ : Object, Object -> Bool
    eqns
    forall sp : SP, obj : Object, data : UserData
      ofsort Bool
        IsCon (object (sp)) =
          IsConReq (sp)  or IsConInd (sp);
        IsCak (object (sp)) =
          IsConResp (sp) or IsConConf (sp);
        IsDat (object (sp)) =
          IsDatReq (sp)  or IsDatInd (sp);
        IsDis (object (sp)) =
          IsDisReq (sp)  or IsDisInd (sp);
        obj == obj = true;
      ofsort SP
        indication (object (ConReq)) = ConInd;
        indication (object (ConInd)) = ConInd;
        indication (object (ConResp)) = ConConf;
        indication (object (ConConf)) = ConConf;
        indication (object (DatReq (data))) =
          DatInd (data);
```

```
        indication (object (DatInd (data))) =
          DatInd (data);
        indication (object (DisReq)) = DisInd;
        indication (object (DisInd)) = DisInd;
        altindication (object (ConReq)) =
          ConConf;
        altindication (object (ConInd)) =
          ConConf;
        altindication (object (ConResp)) =
          ConInd;
        altindication (object (ConConf)) =
          ConInd;
        altindication (object (DatReq (data))) =
          DatInd (data);
        altindication (object (DatInd (data))) =
          DatInd (data);
        altindication (object (DisReq)) = DisInd;
        altindication (object (DisInd)) = DisInd;
  endtype (* ObjectType *)

(*-----------------------------------------------
```

Behaviour: decomposed into the following constraints:

a) the Connection that the Service Provider can offer; and

b) application of Service Provider backpressure flow control.

```
-----------------------------------------------*)

  behaviour
    Connection [a] (stationa, stationb)
  ||
    Backpressure [a]

  where

(*-----------------------------------------------
```

Connection: decomposed into the dependent conjunction of the following constraints:

a) the two associated Connection Endpoints; and

b) the correct bi-directional Service Primitive transfer.

```
-----------------------------------------------*)

  process Connection [a]
   (stationa, stationb : Address) : noexit :=
    CEPs [a] (stationa, stationb)
  ||
    Association [a] (stationa, stationb)

    where

(*-----------------------------------------------
```

CEPs: decomposed into the constraints on the sequence of Service Primitives and addressing possible at the Connection Endpoints of each station.

```
-----------------------------------------------*)

    process CEPs [a]
     (stationa, stationb : Address) : noexit :=
      CEP [a] (stationa) ||| CEP [a] (stationb)

      where

(*-----------------------------------------------
```

CEP: decomposed into the disjoint constraints on a single Connection Endpoint at a given address (station):

a) the order in which Service Primitives occur at the CEP; and

b) the constraint regarding the address at which the Service Primitives occur.

The ordering of Service Primitives is constrained as follows for a calling (and called) endpoint:

a) the first event may only be a **ConReq** (**ConInd**); and

b) the event following the initial **ConReq** (**ConInd**) may be a **ConConf** (**ConResp**); and

c) following occurrence of a **ConConf** (**ConResp**), any sequence of **DatReqs** and **DatInds** may occur; and

d) at any point after the initial **ConReq** (**ConInd**), a **DisReq** or a **DisInd** may occur; and

e) after either a **DisReq** or **DisInd**, the whole behaviour may be repeated.

The constraint on the address at which the Service Primitives occur is simply that:

a) all Service Primitives must occur at the address given in the parameter.

```
-----------------------------------------------*)

    process CEP [a]
     (stationx : Address) : noexit :=
      PrimitiveOrdering [a]
    ||
      Addressing [a] (stationx)

      where

      process PrimitiveOrdering [a] : noexit :=
        a ? station : Address ? sp1 : SP
         [IsConReq(sp1) or IsConInd(sp1)];
        (
          a ? station : Address ? sp2 : SP
            [(IsConReq(sp1) implies
               IsConConf(sp2)) and
                (IsConInd(sp1) implies
                 IsConResp(sp2))];
          DataTransfer [a]
        [>
          Disconnect [a]
```

```
    )

    where

    process DataTransfer [a] : noexit :=
      a ? station : Address ? sp : SP
        [IsDatReq(sp) or IsDatInd(sp)];
      DataTransfer [a]
    endproc (* DataTransfer *)

    process Disconnect [a] : noexit :=
      a ? station : Address ? sp : SP
       [IsDisReq(sp) or IsDisInd(sp)];
      PrimitiveOrdering [a]
    endproc (* Disconnect *)

  endproc (* PrimitiveOrdering *)

  process Addressing [a]
   (stationx : Address) : noexit :=
    a ? station : Address ? sp : SP
     [station eq stationx];
    Addressing [a] (stationx)
  endproc (* Addressing *)

endproc (* CEP *)

endproc (* CEPs *)

(*-----------------------------------------------
```

Association: represents the correct bi-directional transfer of Service Primitives. It is decomposed in two independent constraints, relating Requests (and Responses) at one End-point to Indications and Confirmations at the other, for each direction.

Each one of these Associations can be represented as the one-way transfer of Service Primitives through a Medium.

```
-----------------------------------------------*)

  process Association [a]
   (stationa, stationb : Address) : noexit :=
    Assoc [a] (stationa, stationb, empty)
  |||
    Assoc [a] (stationb, stationa, empty)

    where

(*-----------------------------------------------
```

Basic Abracadabra Service Medium: defines a simple *FIFO medium, used to relate the order of output Service* Primitives to input Service Primitives. The basic medium type allows expression of Medium objects as either:

a) an empty medium (**empty**); or

b) a medium, **asm**, to which a further object, **aso**, has been added (aso + – – asm).

In addition, a further constructor is defined for a medium in the form of a first object and the following medium. Note that all media of this form may also be expressed in form b) above.

Lastly, a Boolean equality function, ==, is defined for the Medium.

```
-----------------------------------------------*)

  type BasicMediumType is ObjectType, Boolean
    sorts Medium
    opns
      empty  :                  -> Medium
      _+--_  : Object, Medium -> Medium
      _--+_  : Medium, Object -> Medium

      _==_   : Medium, Medium -> Bool
    eqns
     forall sm, sm1, sm2 : Medium,
      obj, obj1, obj2 : Object
      ofsort Medium
        obj +-- empty = empty --+ obj;
        (obj2 +-- (sm --+ obj1)) =
          ((obj2 +-- sm) --+ obj1);
      ofsort Bool
        sm == sm = true;
        (obj2+--sm2) == empty = false;
        empty == (obj1+--sm1) = false;
        (obj2+--sm2) == (obj1+--sm1) =
          ((obj2 == obj1) and (sm2 == sm1));
  endtype (* BasicMediumType *)

(*-----------------------------------------------
```

Provider Disconnection of the Abracadabra Service Medium: contains the information relating to the Service Provider's ability to cause a (Provider) **DisInd** at any time during the lifetime of a connection.

```
-----------------------------------------------*)

  type DisconnectedMediumType is
   BasicMediumType
    opns
      _MayDisconnect_ :
       Medium, Medium -> Bool
    eqns
     forall obj, obj1 : Object,
      sm, sm1 : Medium
      ofsort Bool
        empty MayDisconnect empty = true;
        empty MayDisconnect (obj+--sm) =
          false;
        (obj1+--sm1) MayDisconnect sm =
          (((obj1+--sm1) == sm) or
           (IsDis(obj1) and
            (sm1 MayDisconnect empty)));
  endtype (* DisconnectedMediumType *)

(*-----------------------------------------------
```

Reorderings of the Abracadabra Service Medium: contains the information relating to the Service Provider's ability to reorder the messages in transit. An object (e.g. a Disconnect) may advance through the medium, destroying the objects it overtakes (operation **destroy**). In some circumstances (e.g. a Disconnect and a Connect), two objects may mutually destroy each other (operation **cancel**).

The Boolean function, **IsReorderingOf**, returns **true** if its first argument could have been derived from its second in accordance with the reordering rules of the Abracadabra Service, otherwise it returns **false**.

```
-----------------------------------------------*)

      type MediumType is DisconnectedMediumType
        opns
          _negates_, _destroys_ :
            Object, Object -> Bool
          _<<_, _IsReorderingOf_  :
            Medium, Medium -> Bool
        eqns
         forall m, m1, m2 : Medium,
          obj, obj0, obj1, obj0a, obj1a : Object
          ofsort Bool
            obj1 destroys obj0 =
              IsDis(obj1) and (not(IsCon(obj0)));
            obj1 negates obj0 =
              IsDis (obj1) and IsCon (obj0);
            empty << empty = true;
            empty << (obj0+--empty) = false;
            empty << (obj1+--(obj0+--m)) =
              (obj1 negates obj0) and
               (empty << m);
            (obj+--empty) << empty = false;
            (obj+--empty) << (obj0+--empty) =
              (obj == obj0);
            (obj+--empty) <<
             (obj1+--(obj0+--m)) =
              ((obj == obj1) and
               (empty << (obj0+--m))) or
                ((obj1 negates obj0) and
                 ((obj+--empty) << m)) or
                  ((obj1 destroys obj0) and
                   ((obj+--empty) <<
                    (obj1+--m)));
            (obj1a+--(obj0a+--m)) << empty =
             false;
            (obj1a+--(obj0a+--m)) <<
             (obj0+--empty) = false;
            (obj1a+--(obj0a+--m1)) <<
             (obj1+--(obj0+--m2)) =
              ((obj1a == obj1) and
               ((obj0a+--m1) << (obj0+--m2))) or
                ((obj1 negates obj0) and
                 (((obj1a+--(obj0a+--m1)) <<
                  m2))) or ((obj1 destroys obj0)
                   and
                    (((obj1a+--(obj0a+--m1)) <<
                     (obj1+--m2))));
          m1 IsReorderingOf m2 = m1 << m2;

      endtype (* MediumType *)

(*-----------------------------------------------
```

Uni-directional Primitive Transfer: defines the behaviour as accepting an object to be transferred or delivering an object at the other end, plus the possible reorderings of the Medium itself.

```
-----------------------------------------------*)

      process Assoc [a]
       (stationa, stationb : Address,
        sm : Medium) : noexit :=
        (
        transferin  [a] (stationa, sm)
      []
        transferout [a] (stationb, sm)
        )
      >>
        accept sm1 : Medium in
          (
            choice sm2, sm3: Medium []
              [(sm2 MayDisconnect sm1) and
               (sm3 IsReorderingOf sm2)] ->
                (
                  choice sm4 : Medium,
                   obj : Object []
                    [sm3 = (sm4 --+ obj)] ->
                      (
                        Assoc [a] (stationa,
                         stationb, sm3)
                      []
                        [IsDis (obj)] ->
                          i;
                          Assoc [a] (stationa,
                           stationb, sm4)
                      )
                )
          )

        where

(*-----------------------------------------------
```

Acceptance of Requests and Confirmations: defines the constraint associated with the acceptance of a Request or Response Service Primitive as follows:

a) A Service Primitive may occur provided that it is a Request (or Response), and the corresponding object is sent over the Medium associated with the Connection.

```
-----------------------------------------------*)

      process transferin [a]
       (station : Address, sm : Medium)
        : exit (Medium) :=
        a ! station ? sp : SP [IsReq(sp)];
        exit ((object(sp)) +-- sm)
```

```
        endproc (* transferin *)

(*-----------------------------------------
```

Delivery of Indications and Confirmations: defined as the constraint on the delivery of Service Primitives to the receiving User:

a) Whatever object, if any, is foremost in the Medium may be delivered to the User in the form of the appropriate *Indication (or Confirmation)*. *The foremost object* is removed from the Medium as it is delivered.

```
-----------------------------------------*)

        process transferout [a]
         (station : Address, sm : Medium)
          : exit (Medium) :=
          choice undelivered : Medium,
           deliver : Object []
            [sm = (undelivered --+ deliver)] ->
              a ! station ? sp : SP
                [(sp == indication(deliver)) or
                  (sp ==
                   altindication(deliver))];
              exit (undelivered)
        endproc (* transferout *)

      endproc (* Assoc *)

    endproc (* Association *)

  endproc (* Connection *)


(*-----------------------------------------
```

Backpressure: defined as the constraint which is associated with *provider backpressure flow control*:

a) a **DatReq** primitive may be refused at either Connection Endpoint at any time.

The Service User has no control over whether **DatReq** Service Primitives are refused.

```
-----------------------------------------*)

  process Backpressure [a] : noexit :=
    choice RefuseDatReq : SetofAddress []
      i;
      a ? station : Address ? sp : SP
        [(IsDatReq(sp) implies
          (station NotIn RefuseDatReq))];
      Backpressure [a]
  endproc (* Backpressure *)

endspec (* AbracadabraService *)
```

10.4.4 Formal Description of the Protocol

There are two parameters to the description: the retransmission limit, and the period for timeouts. The unit for timeouts is not defined since LOTOS cannot describe absolute time. These two parameters are of sort **Nat** from the standard library. A further parameter is used to identify the Protocol Entity as an Abracadabra station.

Medium (Endpoint) identification is not directly described, but is dynamically decided on the first event at the **m** gate.

```
-----------------------------------------*)

specification AbracadabraProtocolEntity [a, m]
 (N : Nat, P : Nat, station : Address) : noexit

(*-----------------------------------------
```

Standard Library: imports some data types from the Standard Library. **Boolean** is needed everywhere, **NaturalNumber** is for the specification parameters, **DecDigit** is used to simplify the description of objects by mapping them onto digits, and **OctetString** is used for Service User data.

```
-----------------------------------------*)

  library
    Boolean, NaturalNumber, DecDigit, OctetString
  endlib

(*-----------------------------------------
```

Abracadabra Service Addresses: defines the known addresses that the stations may have.

```
-----------------------------------------*)

  type AddressType is Boolean
    sorts Address
    opns
      A, B        : -> Address
      _eq_, _ne_ : Address, Address -> Bool
    eqns forall a1, a2 : Address
      ofsort Bool
        A eq A = true;
        A eq B = false;
        B eq A = false;
        B eq B = true;
        a1 ne a2 = not (a1 eq a2)
  endtype (* AddressType *)

(*-----------------------------------------
```

Behaviour: considers the constraints introduced by the Abracadabra Service, plus those of the Communications Medium Service and the internal ordering imposed by the Abracadabra Protocol.

```
-----------------------------------------*)
```

```
behaviour
  Service [a] (station)
|[a]|
  Protocol [a, m] (N, P)
|[m]|
  CMService [m]

where
```

```
(*-----------------------------------------------
```

Abracadabra Service Data Units: defines Service User data in terms of the standard type **OctetString**.

```
-----------------------------------------------*)

  type UserDataType is OctetString renamedby
    sortnames UserData for OctetString
  endtype (* UserDataType *)

(*-----------------------------------------------
```

Abracadabra Service Primitives: defined as values of sort **SP**, with some operations to extract information from the Service Primitives. There is also a mapping to **DecDigit** in order to simplify the definition of the recognition predicates. Notice that any injective mapping would suffice.

```
-----------------------------------------------*)

  type SPType is Boolean, UserDataType, DecDigit
    sorts SP
    opns
      ConReq,  ConInd, ConResp, ConConf : -> SP
      DatReq,  DatInd  : UserData -> SP
      DisReq,  DisInd  :          -> SP
      IsConReq,  IsConInd, IsConResp, IsConConf,
      IsDatReq,  IsDatInd,
      IsDisReq,  IsDisInd : SP -> Bool
      IsReq,     IsInd    : SP -> Bool
      data : SP -> UserData
      _==_ : SP, SP -> Bool
      map : SP  -> DecDigit
    eqns forall d : UserData, sp : SP
      ofsort UserData
        data (ConReq)     = <>;
        data (ConInd)     = <>;
        data (ConResp)    = <>;
        data (ConConf)    = <>;
        data (DatReq (d)) = d ;
        data (DatInd (d)) = d ;
        data (DisReq)     = <>;
        data (DisInd)     = <>;
      ofsort DecDigit
        map (ConReq)     = 0 ;
        map (ConInd)     = 1 ;
        map (ConResp)    = 2 ;
        map (ConConf)    = 3 ;
        map (DatReq (d)) = 4 ;
        map (DatInd (d)) = 5 ;
        map (DisReq)     = 6 ;
        map (DisInd)     = 7 ;
      ofsort Bool
        IsConReq (sp)  = map (sp) eq 0 ;
        IsConInd (sp)  = map (sp) eq 1 ;
        IsConResp (sp) = map (sp) eq 2 ;
        IsConConf (sp) = map (sp) eq 3 ;
        IsDatReq (sp)  = map (sp) eq 4 ;
        IsDatInd (sp)  = map (sp) eq 5 ;
        IsDisReq (sp)  = map (sp) eq 6 ;
        IsDisInd (sp)  = map (sp) eq 7 ;
        sp == sp       = true;
  endtype (* SPType *)

(*-----------------------------------------------
```

Abracadabra Protocol Data Units: defined rather like the Service Data Units.

```
-----------------------------------------------*)

  type PDUType is UserDataType, Boolean, DecDigit
    sorts PDU
    opns
      CR, CC  :                -> PDU
      DT      : UserData, Bool -> PDU
      AK      :  Bool          -> PDU
      DR, DC  :                -> PDU
      data    : PDU            -> UserData
      bool    : PDU            -> Bool
      IsCR, IsCC, IsDT, IsAK, IsDR, IsDC :
        PDU -> Bool
      map : PDU -> DecDigit
    eqns forall d : UserData, b : Bool, pdu : PDU
      ofsort UserData
        data (CR)        = <>;
        data (CC)        = <>;
        data (DT (d, b)) = d ;
        data (AK (b))    = <>;
        data (DR)        = <>;
        data (DC)        = <>;
      ofsort Bool
        bool (CR)        = false;
        bool (CC)        = false;
        bool (DT (d, b)) = b ;
        bool (AK (b))    = b ;
        bool (DR)        = false;
        bool (DC)        = false;
      ofsort DecDigit
        map (CR)         = 0 ;
        map (CC)         = 1 ;
        map (DT (d, b))  = 2 ;
        map (AK (b))     = 3 ;
        map (DR)         = 4 ;
        map (DC)         = 5 ;
      ofsort Bool
        IsCR (pdu)  = map (pdu) eq 0 ;
        IsCC (pdu)  = map (pdu) eq 1 ;
        IsDT (pdu)  = map (pdu) eq 2 ;
        IsAK (pdu)  = map (pdu) eq 3 ;
```

```
        IsDR (pdu)  = map (pdu) eq 4 ;
        IsDC (pdu)  = map (pdu) eq 5 ;
  endtype (* PDUType *)

(*-----------------------------------------------
```

Communications Medium Service Primitives: defined explicitly without use of any mapping to **DecDigit**, since this type is simpler than that for Abracadabra Service Primitives.

```
-----------------------------------------------*)

  type CMSPType is PDUType, Boolean
    sorts CMSP
    opns
      UnitReq, UnitInd : PDU -> CMSP
      pdu : CMSP -> PDU
      IsUnitReq, IsUnitInd : CMSP -> Bool
    eqns forall d : PDU, cmsp : CMSP
      ofsort PDU
        pdu (UnitReq (d)) = d ;
        pdu (UnitInd (d)) = d ;
      ofsort Bool
        IsUnitReq (UnitReq (d)) = true  ;
        IsUnitReq (UnitInd (d)) = false ;
        IsUnitInd (UnitReq (d)) = false ;
        IsUnitInd (UnitInd (d)) = true  ;
  endtype (* CMSPType *)

(*-----------------------------------------------
```

Abracadabra Service Constraints: decomposed into constraints on the station addressing, as well as on the ordering of Service Primitives. Unlike the Service description, only the addressing constraint is considered. The ordering of Service Primitives is to be deduced from the Protocol. The constraint on the addressing is to accept only transactions for this station. In the remainder of the description, it is then possible to forget about which station is involved.

```
-----------------------------------------------*)

  process Service [a]
   (station : Address) : noexit :=
    a ! station ? sp: SP ;
    Service [a] (station)
  endproc (* Service *)

(*-----------------------------------------------
```

Connections Medium Service Constraints: obtains a User (Connection Endpoint) identifier on the first event, and then sticks to it for all subsequent events.

```
-----------------------------------------------*)

  process CMService [m] : noexit :=
    m ? user : Address ? cmsp : CMSP;
    StickTo [m] (user)
  where

    process StickTo [m]
     (user : Address) : noexit :=
      m ! user ? cmsp: CMSP;
      StickTo [m] (user)
    endproc (* StickTo *)

  endproc (* CMService *)

(*-----------------------------------------------
```

Abracadabra Protocol Constraints: defines the Protocol behaviour in a more constructive manner, i.e. instead of a composition of constraints, there is a traversal of states. The natural language description clearly distinguishes four phases. The entity starts in the Connect Phase and moves to other phases according to interactions with the environment.

```
-----------------------------------------------*)

  process Protocol [a, m]
   (N, P : Nat) : noexit :=
    Connect [a, m] (N, P)
  where

(*-----------------------------------------------
```

Connection: may be started by either station. Normally, the Protocol Entity proceeds to data transfer, where it remains for ever, unless a disconnection is requested or an error rises. As processes give control to each other, a reason is given to explain the cause of the disconnection.

```
-----------------------------------------------*)

    process Connect [a, m]
     (N, P : Nat) : noexit :=
      (* this station starts *)
      a ? s : Address ! ConReq;
      (
        TryConnect [a, m] (0 of Nat, N, P)
      [>
        (
          m ? u : Address ! UnitInd (DR);
          a ? s : Address ! DisInd;
          Connect [a, m] (N, P)
        []
          a ? s : Address ! DisReq;
          GiveUp [a, m] (N, P, UserDisc)
        )
      )
    []
      (* the other station starts *)
      m ? u : Address ! UnitInd (CR);
      a ? s : Address ! ConInd;
      (
```

```
      a ? s : Address ! ConResp;
      m ? u : Address ! UnitReq (CC);
      DataTransfer [a, m] (N, P)
    [>
      Disconnect [a, m]
    >>
      accept r: reason in
        GiveUp [a, m] (N, P, r)
    )
  []
    (* any other incoming message is ignored *)
    m ? u : Address ? cmsp : CMSP
      [IsUnitInd (cmsp) and (IsCC (pdu (cmsp))
       or IsDT (pdu (cmsp)) or
        IsAK (pdu (cmsp)) or
         IsDC (pdu (cmsp)))];
    Connect [a, m] (N, P)
  []
    (* however, DRs are replayed *)
    m ? u : Address ! UnitInd (DR) ;
    m ? u : Address ! UnitReq (DC) ;
    Connect [a, m] (N, P)

    where

(*-----------------------------------------------
```

Reasons to release a Connection: defines the several reasons for Connection Release that are passed between the processes of each phase.

```
-----------------------------------------------*)

    type ReasonType is
      sorts reason
      opns
        UserDisc,    (* from Abracadabra user *)
        CMDisc,         (* from Comms. Medium *)
        error : -> reason
              (* unexpected incoming message *)
    endtype (* ReasonType *)

(*-----------------------------------------------
```

Try to Connect: attempts a connection up to **N** times, when this station starts up. The number of connection attempts so far is held in **This**. The timeout period is **P**.

```
-----------------------------------------------*)

    process TryConnect [a, m]
     (This, N, P : Nat) : noexit :=
      [This lt N] ->
        m ? u : Address ! UnitReq (CR);
        (
          (* loop *)
          StandBy [m]
        [>
          (
            m ? u: Address ? cmsp : CMSP
              [IsUnitInd (cmsp) and
               (IsCC (pdu (cmsp)) or
                IsCR (pdu (cmsp)))];
            a ? s : Address ! ConConf;
            DataTransfer [a, m] (N, P)
          )
        [>
            Disconnect [a, m]
          >>
            accept r : reason in
              GiveUp [a, m] (N, P, r)
        []
          (* timeout *)
          (
            Wait (P)
          >>
            TryConnect [a, m]
             (Succ (This), N, P)
          )
        )
    []
      [This ge N] ->
        GiveUp [a, m] (N, P, error)

      where

        process StandBy [m] : noexit :=
          m ? u: Address ? cmsp: CMSP
            [IsUnitInd (cmsp) and
              (IsDT (pdu (cmsp)) or
               IsAK (pdu (cmsp)) or
                IsDC (pdu (cmsp)))];
          StandBy [m]
        endproc (* StandBy *)

    endproc (* TryConnect *)

(*-----------------------------------------------
```

Data Transfer: decomposed into three constraints:

a) the transmission of data (downward flow); and
b) the reception of data (upward flow); and
c) the acceptance of **CR** only before any **DT** or **AK**.

which are composed in the obvious way:

```
  CRFlow [m] ...
|[m]|
  (
    DownwardFlow [a, m] ...
  |||
    UpwardFlow [a, m] ...
  )
```

where the first is an 'and' composition and the second a 'parallel or' one. This normal transfer may be disrupted by unexpected incoming messages, leading to the next phase.

Each of the Protocol Data Units that may arrive in **UnitInds** is 'captured' by a separate part of the description:

CR **CRFlow** and **UpwardFlow**

CC unexpected incoming messages

DT **UpwardFlow** and **CRFlow**

AK **DownwardFlow** and **CRFlow**

DR **Disconnect**

DC unexpected incoming messages.

UnitReqs are controlled by **DownwardFlow**, **UpwardFlow** and **CRFlow**.

It is necessary to have three processes running in parallel to model the composition of the constraints. Any of these processes may find an error and, consequently, may want to **exit**. But LOTOS requires that *every* process in a parallel composition must synchronise on the **exit**. It would be helpful if LOTOS had an abrupt termination of parallel composition, i.e. a non-synchronised **exit**: if any of the processes **exit**ed, the others would be disabled. But for the time being, there is no such a facility in LOTOS.

The following outline style of description has therefore been used to emulate the desired behaviour:

```
  (
    B1
  [>
    choice i : Nat []  [i ne 1] -> exit (i)
  )
|[...]|
  (
    B2
  [>
    choice i : Nat []  [i ne 2] -> exit (i)
  )
|[...]|
  (
    B3
  [>
    choice i : Nat []  [i ne 3] -> exit (i)
  )
|[...]|
   ...
```

(This is a schematic description, not syntactically correct LOTOS.) Each process **B** would **exit** with its proper code, i.e. 1 for **B1**, 2 for **B2**, etc. Using this approach, the whole Data Transfer phase may be modelled as follows.

```
-----------------------------------------------*)

      process DataTransfer [a, m]
       (N, P : Nat) : exit (reason) :=
        (
          (
            CRFlow [m] (true)
          [>
            (
              choice w : who []
                [w <> crflow] ->
                  exit (w, any reason)
            )
          )
        |[m]|
          (
            (
              DownwardFlow [a, m] (N, P, false)
            [>
              (
                choice w : who []
                  [w <> down] ->
                    exit (w, any reason)
              )
            )
          |||
            (
              UpwardFlow [a, m] (false)
            [>
              (
                choice w : who []
                  [w <> up] ->
                    exit (w, any reason)
              )
            )
          )
        )
      >>
        accept w : who, r : reason in
          exit (r)
      [>
        m ? u: Address ? cmsp: CMSP
          [IsUnitInd (cmsp) and
            (IsCC (pdu (cmsp)) or
             IsDC (pdu (cmsp)))];
        exit (error)

        where

(*-----------------------------------------------
```

Who: distinguishes who exits from the Data Transfer Phase.

```
-----------------------------------------------*)

        type WhoType is Boolean, DecDigit
          sorts who
          opns
            up, down, crflow : -> who
            map              : who -> DecDigit
            _ <> _           : who, who -> Bool
          eqns forall w1, w2 : who
            ofsort DecDigit
              map (up)       = 0 ;
              map (down)     = 1 ;
              map (crflow)   = 2 ;
            ofsort Bool
              w1 <> w2 = map (w1) ne map (w2) ;
        endtype (* WhoType *)

(*-----------------------------------------------
```

Constraint on Flow of CR PDUs: following the informal description, defines that **CRs** are accepted before the first **DT** or **AK**. **CRFlow** monitors gate **m** to fulfill this constraint. Any **UnitReq** is ignored.

```
-----------------------------------------------*)

        process CRFlow [m]
         (initial : Bool) : exit (who, reason) :=
          m ? u : Address ? cmsp : CMSP
           [IsUnitReq (cmsp)];
          CRFlow [m] (initial)
        []
          m ? u : Address ! UnitInd (CR);
          (
            [initial] ->
              m ? u : Address ! UnitReq (CC);
              CRFlow [m] (initial)
          []
            [not (initial)] ->
              exit (crflow, error)
          )
        []
          m ? u : Address ? cmsp : CMSP
            [IsUnitInd (cmsp) and
             (IsDT (pdu (cmsp)) or
              IsAK (pdu (cmsp)))];
          CRFlow [m] (false)
        endproc (* CRFlow *)

(*-----------------------------------------------
```

Constraint on Downward Flow: takes care of sending data to the Communications Medium. It mostly fits the usual timeout pattern, but there is no **Standby** process as elsewhere in this description to absorb unwanted Protocol Data Units.

```
-----------------------------------------------*)

        process DownwardFlow [a, m]
         (N, P : Nat, seq : Bool)
          : exit (who, reason) :=
          a ? s : Address ? sp : SP
           [IsDatReq (sp)];
          TryData [a, m]
           (0 of Nat, N, P, data (sp), seq)

          where

          process TryData [a, m]
            (This, N, P : Nat, d : UserData,
             seq : Bool) : exit (who, reason) :=

            [This lt N] ->
              m ? u : Address !
               UnitReq (DT (d, seq));
              (
                m ? u : Address !
                 UnitInd (AK (not (seq)));
                DownwardFlow [a, m]
                 (N, P, not (seq))
              []
                m ? u : Address !
                 UnitInd (AK (seq));
                exit (down, error)
              []
                (* timeout *)
                (
                  Wait (P)
                >>
                  TryData [a, m]
                   (Succ (This), N, P, d, seq)
                )
              )
            []
              [This ge N] ->
                exit (down, error)
          endproc (* TryData *)

        endproc (* DownwardFlow *)

(*-----------------------------------------------
```

Constraint on Upward Flow: takes care of incoming data from the Communications Medium. As an added activity, it must accept any **CR** received and treated by **CRFlow**, as well as any **CC** generated as an answer. Events with these are offered to avoid blocking, but nothing is done with them.

```
-----------------------------------------------*)

        process UpwardFlow [a, m]
         (seq : Bool) : exit (who, reason) :=

          m ? u : Address ? cmsp : CMSP
            [IsUnitInd (cmsp) and
             (IsDT (pdu (cmsp)))];
          (
            (* begin *)
            let dr : UserData =
             data (pdu (cmsp)), seqr : Bool =
              bool (pdu (cmsp)) in
              m ? u : Address !
               UnitReq (AK (not (seqr)));
              (
                [seqr eq seq] ->
                  a ? s : Address ! DatInd (dr);
                  UpwardFlow [a, m] (not (seq))
              []
                [seqr ne seq] ->
                  UpwardFlow [a, m] (seq)
              )
            []
              m ? u : Address ? cmsp : CMSP
                [IsUnitInd (cmsp) and
                 (IsDT (pdu (cmsp)))];
              exit (up, error)
            (* end *)
          )
        []
```

```
        (* absorb CRs *)
        m ? u : Address ! UnitInd (CR);
        UpwardFlow [a, m] (seq)
      []
        (* absorb CCs *)
        m ? u : Address ! UnitReq (CC) ;
        UpwardFlow [a, m] (seq)
      endproc (* UpwardFlow *)

    endproc (* DataTransfer *)

(*-------------------------------------------
```

Disconnection Phase: releases an established Connection on a Disconnection Request or due to an error. The Disconnection Requests are captured by **Disconnect**.

```
-------------------------------------------*)

    process Disconnect [a, m]
     : exit (reason) :=
      a ? s : Address ! DisReq;
      exit (UserDisc)
    []
      m ? u: Address ! UnitInd (DR);
      exit (CMDisc)
    endproc (* Disconnect *)

(*-------------------------------------------
```

GiveUp: performs the actual release.

```
-------------------------------------------*)

    process GiveUp [a, m]
     (N, P : Nat, r : reason) : noexit :=
      [r = UserDisc] ->
        TryDisconnect [a, m] (0 of Nat, N, P)
    []
      [r = CMDisc] ->
        m ? u : Address ! UnitInd (DC);
        a ? s : Address ! DisInd;
        Connect [a, m] (N, P)
    []
      [r = error] ->
        a ? s : Address ! DisInd;
        TryDisconnect [a, m] (0 of Nat, N, P)
      where

(*-------------------------------------------
```

TryDisconnect: this uses the familiar timeout pattern. **Standby** ignores unwanted Protocol Data Units during disconnection.

```
-------------------------------------------*)

      process TryDisconnect [a, m]
       (This, N, P : Nat) : noexit :=
        [This lt N] ->
          m ? u : Address ! UnitReq (DR) ;
          (
            StandBy [m]
          [>
            m ? u : Address ? cmsp : CMSP
              [IsUnitInd (cmsp) and
               (IsDR (pdu (cmsp)) or
                IsDC (pdu (cmsp)))];
            Connect [a, m] (N, P)
          []
            (* timeout *)
            (
              Wait (P)
            >>
              TryDisconnect [a, m]
               (Succ (This), N, P)
            )
          )
      []
        [This ge N] ->
          Connect [a, m] (N, P)
        where

        process StandBy [m] : noexit :=
          m ? u : Address ? cmsp : CMSP
           [IsUnitInd (cmsp) and
            (IsCR (pdu (cmsp)) or
             IsCC (pdu (cmsp)) or
              IsDT (pdu (cmsp)) or
               IsAK (pdu (cmsp)))];
          StandBy [m]
        endproc (* StandBy *)

      endproc (* TryDisconnect *)

    endproc (* GiveUp *)

(*-------------------------------------------
```

Wait: this models a timeout for period **P**. Because LOTOS abstracts away from time, the actual delay cannot be specified. The effect is only of an internal event.

```
-------------------------------------------*)

    process Wait (P : Nat) : exit :=
      exit
    endproc (* Wait *)

  endproc (* Connect *)

endproc (* Protocol *)

endspec (* AbracadabraProtocolEntity *)
```

10.4.5 Subjective Assessment

The LOTOS descriptions show a clear separation between those aspects modelled using Abstract Data Types and those aspects modelled using behaviour expressions. The data types given in the formal descriptions match those given in the informal description.

The constraint-oriented style in LOTOS has been amply demonstrated in earlier examples. However, there are many other styles open to the specifier, for example: 'data-oriented', in which the emphasis is on the description of data types and their operations; 'process-oriented', in which the emphasis is on the description of the dynamic behaviour; 'resource-oriented', in which the emphasis is on the description of resources in the system (Service Access Points, Protocol Entities, etc.); and 'verification-oriented', in which the emphasis is on structuring the description to ease formal verification.

A constraint-oriented style has been used for the Abracadabra Service description, and in part for the Abracadabra Protocol description. However, the Abracadabra Protocol description also uses a 'construction-oriented' style which emphasizes the construction of the system in terms of a number of states. Estelle, LOTOS, and SDL are all examples of **Labelled Transition Systems**. Estelle and SDL tend to emphasize the states of the systems, whereas LOTOS tends to emphasize the transitions between states of the system. In fact, state is generally implicit in a LOTOS description, being embodied in the currently permitted behaviour (loosely, the processes which are active). The LOTOS process operators are the means whereby sequences of potential events can be given a compact representation. This can lead to concise descriptions if the basic problem is not too state-oriented. Service definitions are often given in a black-box, requirements-oriented fashion, so a constraint-oriented style is appropriate. Protocol definitions are often given in a state-oriented and rule-oriented fashion, so a mixture of styles is appropriate. The construction-oriented style lends itself well to describing the division into phases of the overall Protocol behaviour, where the phases may be compared to states. The constraint-oriented style works well for describing the possible sequences of actions that may occur within a phase. These constraints are then combined using the LOTOS process operators.

LOTOS abstracts away from absolute time; this is the meaning of the term **Temporal Ordering** in the title of LOTOS. It is therefore not possible to express absolute timeouts in LOTOS. It is certainly possible to specify a timer process which accepts **set** or **cancel** events and which responds with **timeout** events. However, since such a timer process would be hidden from external view, the events it engages in would be hidden, thus turning into internal events. Rather than make the description too constructive by introducing and hiding such a timer process, timeouts are therefore generally described directly as internal events.

It is possible to 'simulate' the passage of time in LOTOS as a number of time-ticks. A clock process could be defined which distributed ticks to other parts of the system. This would give a measure of relative time. But the interval between ticks could not be specified, nor indeed that the ticks occurred at regular intervals.

A large majority of errors in Protocol design are logic errors, not timing errors. Therefore, the inability of LOTOS to describe absolute time is not a serious issue. Work, however, is already in hand on timed models of LOTOS which will allow the description of delays or time limits in a more meaningful fashion.

10.5 SDL Description

10.5.1 Architecture of the Formal Descriptions

10.5.1.1 Architecture of the Service Description

The Abracadabra Service is described in 10.5.3. It is modelled as a system **AbraService**, consisting of one single block **Serv**. This communicates with the environment by means channels **SAP A** and **SAP B**, which represent Service Access Points. Service Users are located in the environment. Service Primitives are represented by means of signals. According to the informal description, only the **DatReq** and **DatInd** carry parameters, namely a **UserDataType** parameter. The Service Provider may behave non-deterministically by refusing connection attempts and disrupting established connections on its own initiative, for some 'internal reason'. In order to model this, the channels **AServOnOff** and **BServOnOff** from the environment to block **Serv** are meant to convey, asynchronously and unpredictably, two possible signals **ServiceOn** and **ServiceOff**. These cause the Service to become available or unavailable respectively.

Block **Serv** includes two processes which are always active from system-startup. These mirror-image processes are **SAPManagerA** and **SAPManagerB**; they model the behaviour of the Service at the two user sides. Channels to or from the environment are mapped onto corresponding signalroutes within the block. Peer processes **SAPManagerA** and **SAPManagerB** communicate via the signalroute **Internal**. The implicit underlying queue mechanism models the delay it takes for a Service Primitive issued at one SAP to be converted into the corresponding Service Primitive at the remote SAP. Signalroute **Internal** may transfer four possible objects in both directions: Connection Request or Indication, Connection Response or Confirmation, Data, and Disconnection. The distinction between 'from-user' and 'to-user' Service interactions is obviously meaningless at the level of communication between the peer processes. Since **SAPManagerA** and **SAPManagerB** behave identically, they are described by means of an SDL macro **SAPManagerDef**.

10.5.1.2 Architecture of the Protocol Description

The Abracadabra Protocol is described in 10.5.4. It is modelled as a system **Abracadabra**, consisting of only one block **Station**, which represents a Protocol Entity. Accordingly, the boundary of the system is represented by the user Service Access Point (channel **USAP**) and by the Service Access Point to the Medium (channel **MSAP**).

Block **Station** is refined into two processes: **SenderReceiver** and **Transcode**. **SenderReceiver** describes the Protocol Entity as an extended finite state machine which relates input Service Primitives and/or Protocol Data Units to output Service Primitives and/or Protocol Data Units. **Transcode** describes the lowest level of functionality of the Protocol, i.e. PDU encoding and decoding. The **PduType** and **UserDataType** components of a Protocol Data Unit are encoded into the parameters of a **UnitReq**, or are decoded from the parameters of a **UnitInd**.

Process **SenderReceiver** has five states:

Closed the Protocol Entity is ready to accept a Connection Request

CRsent the Protocol Entity is waiting for a Connection Confirmation from the peer Protocol Entity

CRrecv the Protocol Entity is waiting for a Connection Response from its User

Send data transfer is permitted

Wait data transfer is delayed until previous data is acknowledged; in the meantime, data transfer requests from the User are buffered.

A timer is needed in order to count down the maximum delay for Connection Confirmation and for Data acknowledgement. The SDL built-in constructs for timer management are used with an instance called **Timer1**.

Process **Transcode** consists of just one state, **Transwait**. In fact, encoding or decoding is always enabled and the mapping function does not require any memory of past conditions.

10.5.2 Explanation of Approach

10.5.2.1 Explanation of Service Approach

A Service description must always cover two key aspects:

- local behaviour of the Service, i.e. the correct sequencing of Service Primitives at one Service Access Point
- end-to-end behaviour of the Service, i.e. the correct relationship between Service Primitives at different Service Access Points.

The SDL description expresses both aspects. Local behaviour is expressed independently by processes **SAPManagerA** and **SAPManagerB**. End-to-end behaviour is expressed by the mapping that each process performs between Service Primitives and objects on the inter-process signalroute. This communication structure implicitly models the intrinsic delay between related Service Primitive interactions at different Service Access Points.

Signals **ServiceOn** and **ServiceOff** may be independently received by either of the two processes. This explains why two distinct but equivalent channels have been chosen at the system level in order to model non-determinism in the Service.

10.5.2.2 Explanation of Protocol Approach

The approach taken consists of formally describing the Protocol by expressing the behaviour of just one party. This is sufficient due to the symmetry of the Abracadabra Protocol. It would not be the case for unsymmetrical protocols, where it would be necessary to describe the individual behaviour of the two parties.

Interactions with the Medium use only the signals **UnitReq** and **UnitInd**, meaning Protocol Data Unit transmission and reception respectively. The Medium is implicitly assumed to be ready for transmission or reception at all times.

Isolating low-level features of the Protocol such as encoding and decoding greatly improves readability of the formal description. The partitioning of block **Station** into processes **SenderReceiver** and **Transcode** serves this purpose, without necessarily imposing this structure on an actual implementation.

10.5.3 Formal Description of the Service

The formal description of the Service is shown in figure 10.9.

10.5.4 Formal Description of the Protocol

The formal description of the Protocol is shown in figure 10.10.

10.5.5 Subjective Assessment

The SDL description of the Abracadabra example shows how SDL can satisfactorily express both Services and Protocols. Nevertheless, it is interesting to note how the approaches taken in describing a Service and a Protocol may differ substantially. As far as the Service description is concerned, an end-to-end view was considered appropriate, whereas with the Protocol description a local view was chosen. Adopting a local description for the Service would have resulted in incomplete specification: the distributed nature of the Service Provider, with its property of delaying information exchange between Service Users, would have been left out. As a consequence, some legal sequences of Service Primitives would not have been modelled. Conversely, an end-to-end description for the Protocol, describing both Protocol Entities, would have caused duplication in the description due to the equivalent behaviour of both parties.

The approach chosen for the description of the Service had to rely on a higher degree of abstraction than was appropriate for the Protocol. For example, the solution of using signals from the environment in order to express non-determinism is the only one possible in SDL. Unfortunately it leads to a system interaction diagram where the intuitive mapping between channels and physical pathways for information is no longer valid. However, the approach chosen for the description of the Protocol turned out to be quite natural: all the SDL features which have been used are easily justified and intuitively comprehensible.

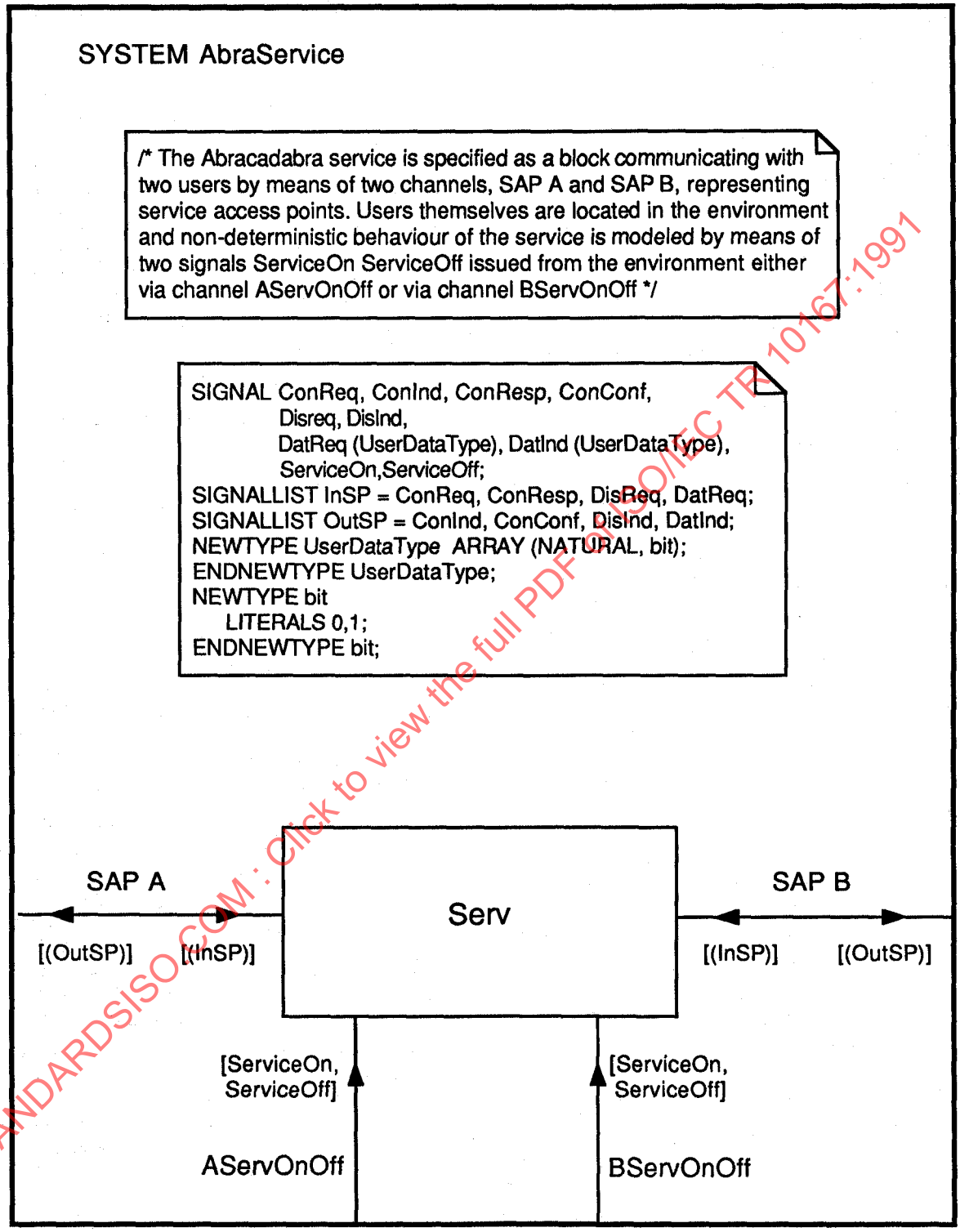


Figure 10.9: SDL Specification of Abracadabra Service

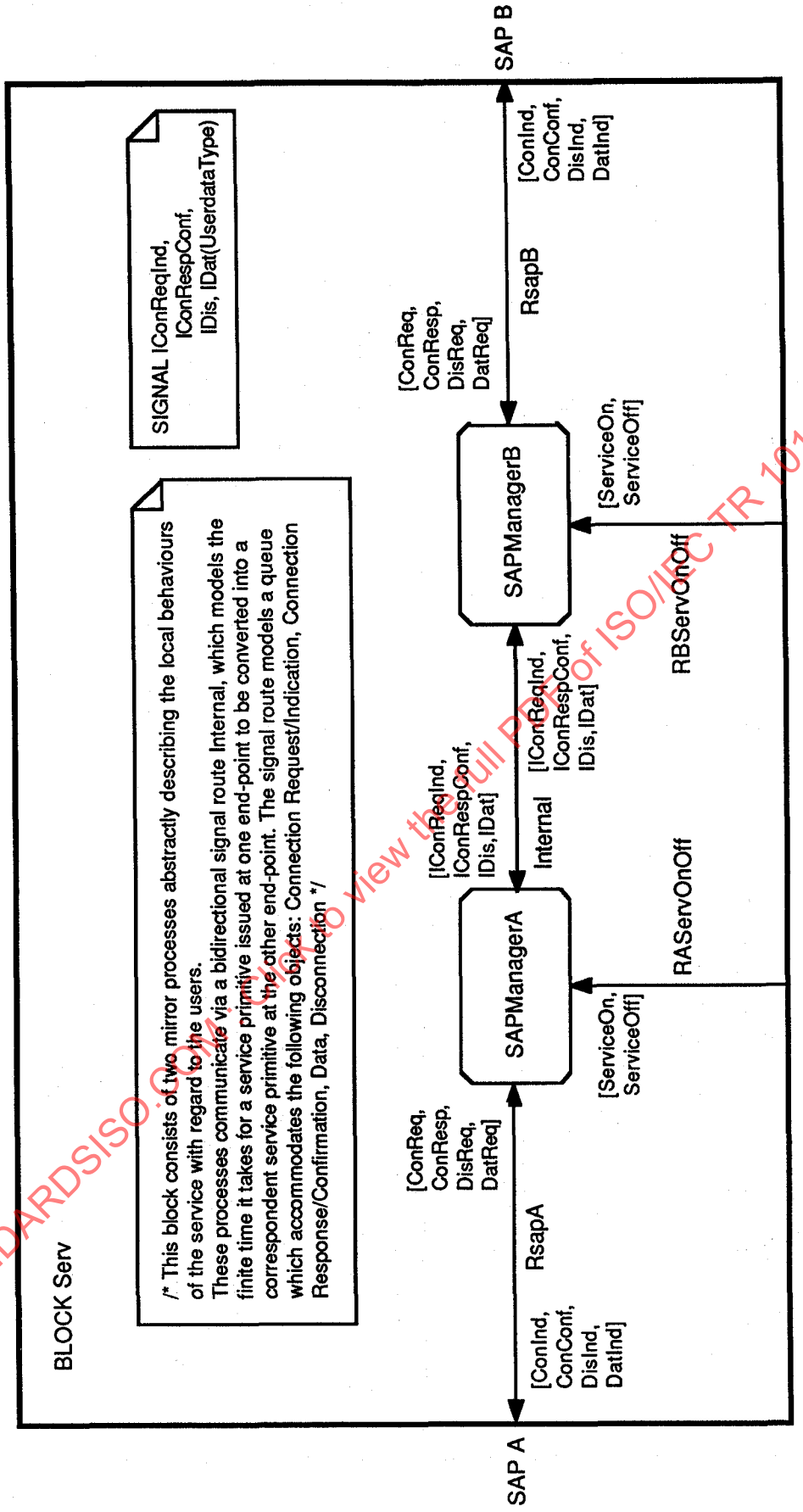


Figure 10.9 *(continued)*

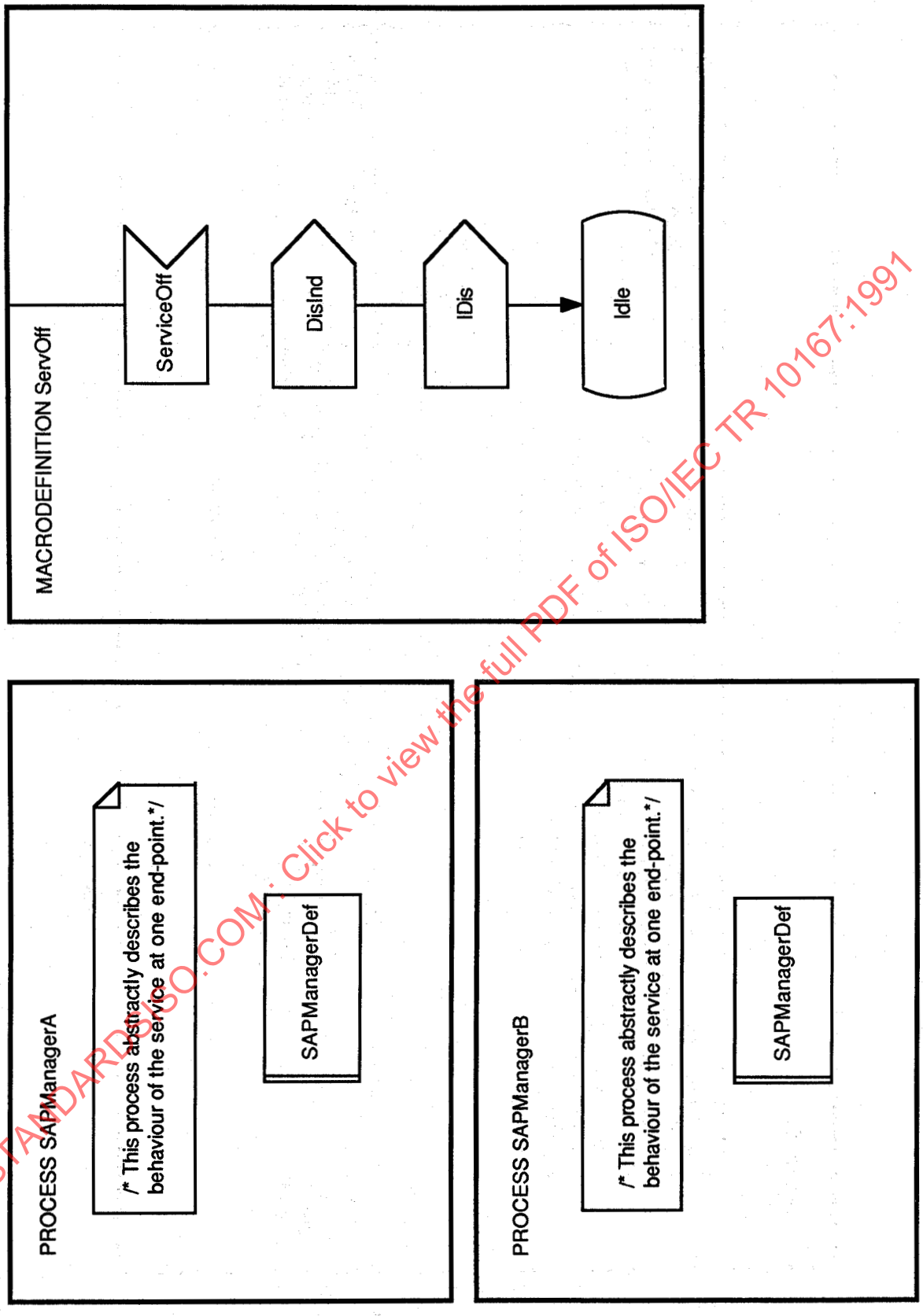


Figure 10.9 *(continued)*

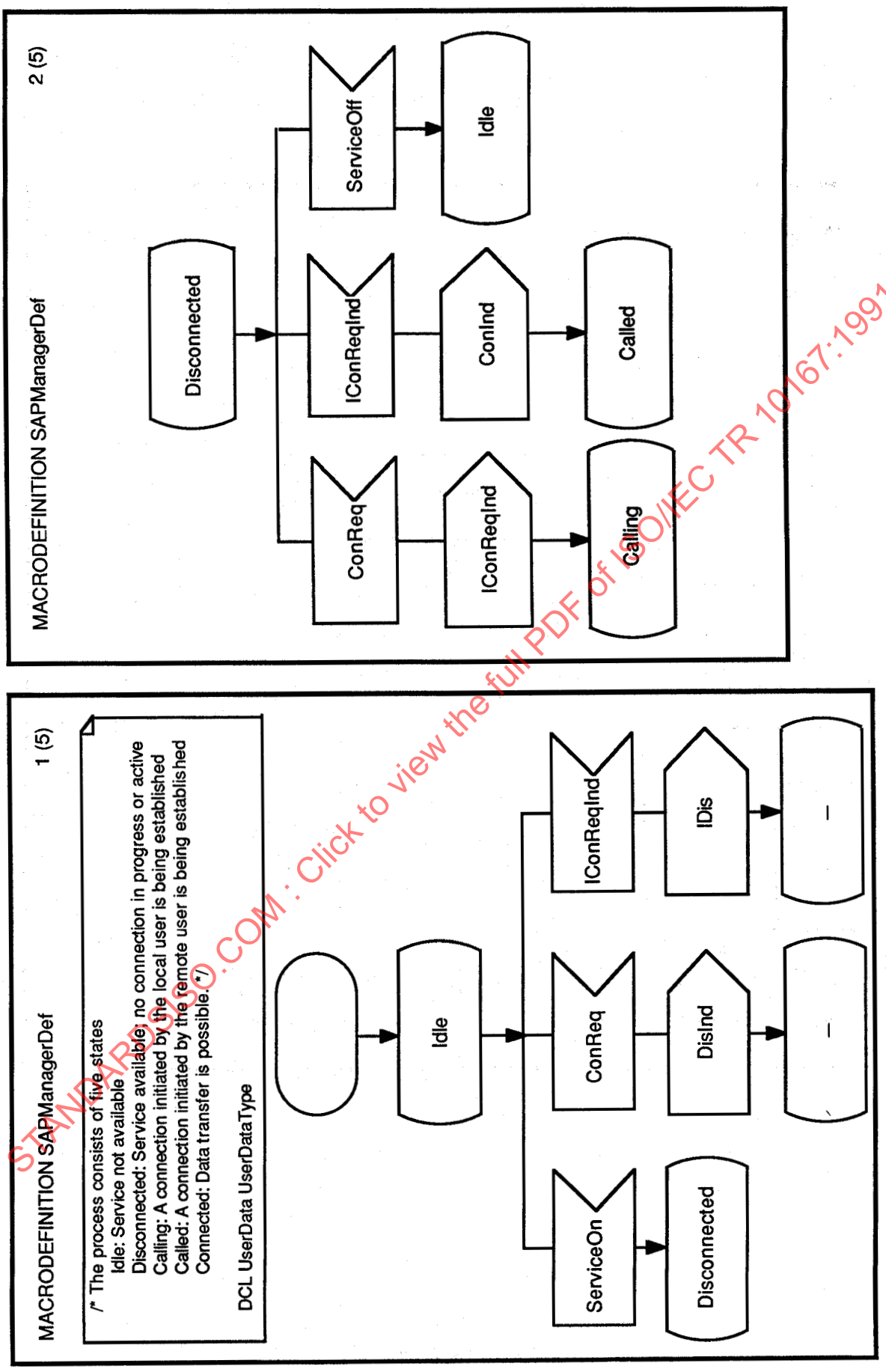


Figure 10.9 *(continued)*

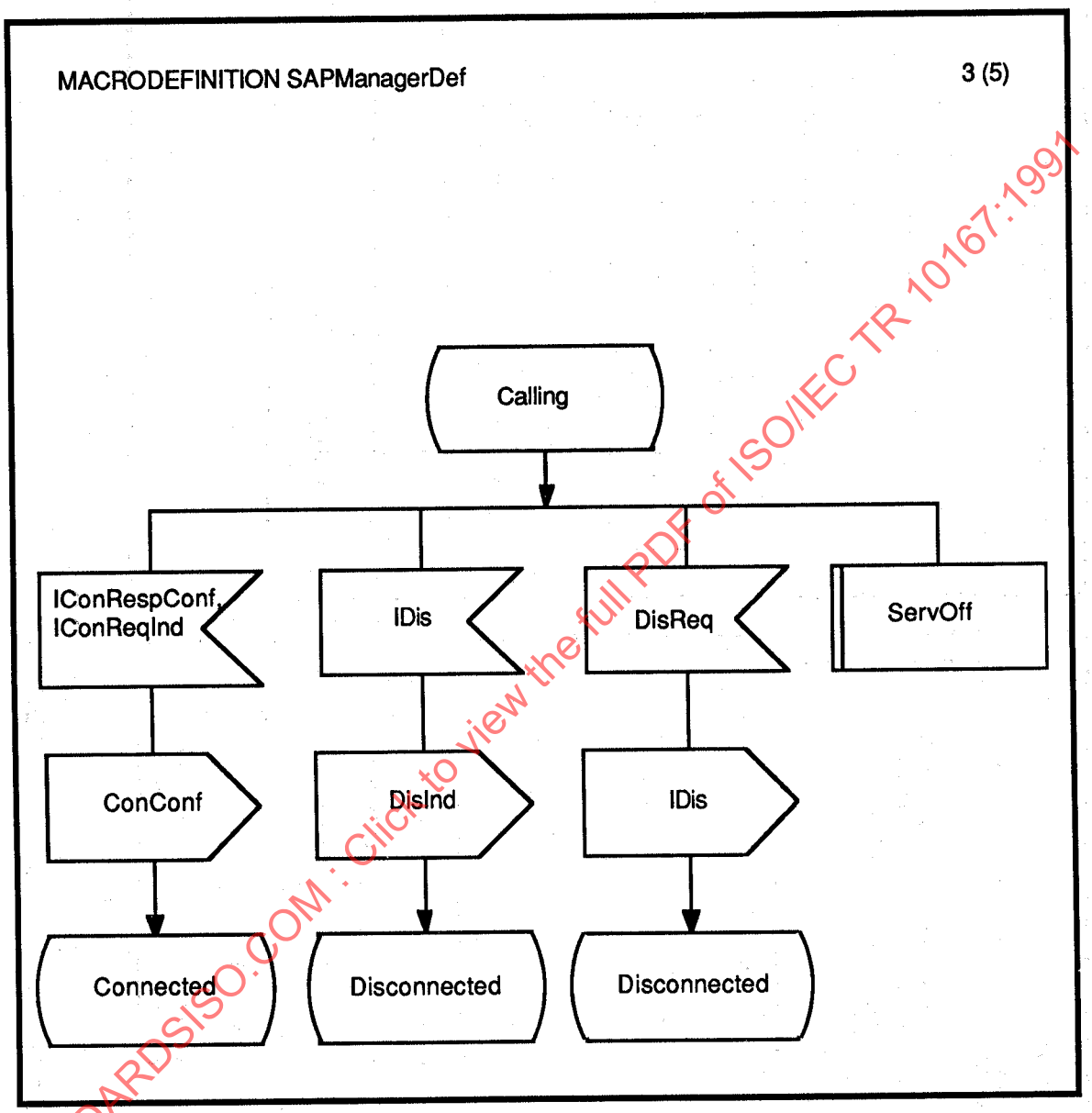


Figure 10.9 *(continued)*

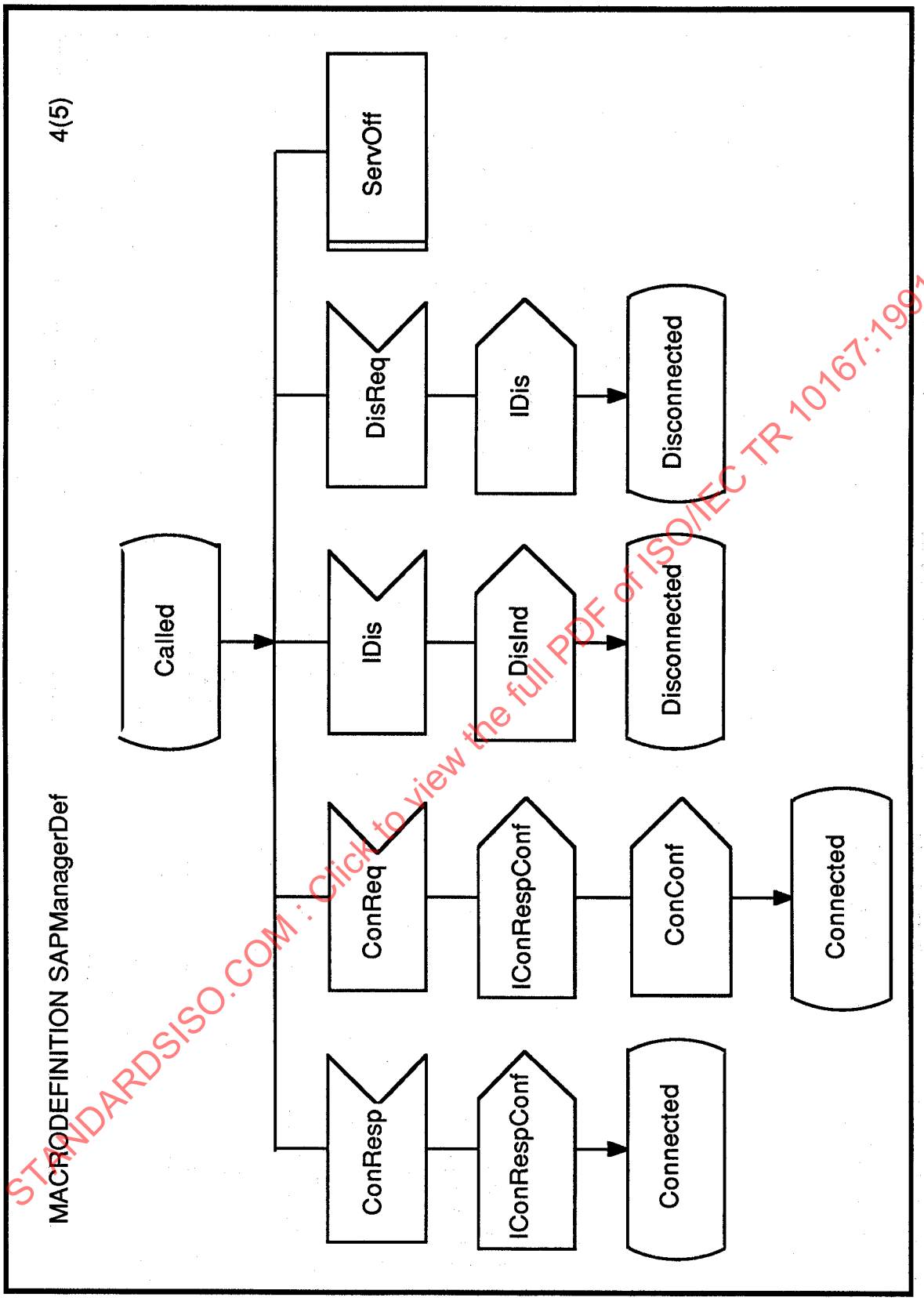


Figure 10.9 *(continued)*

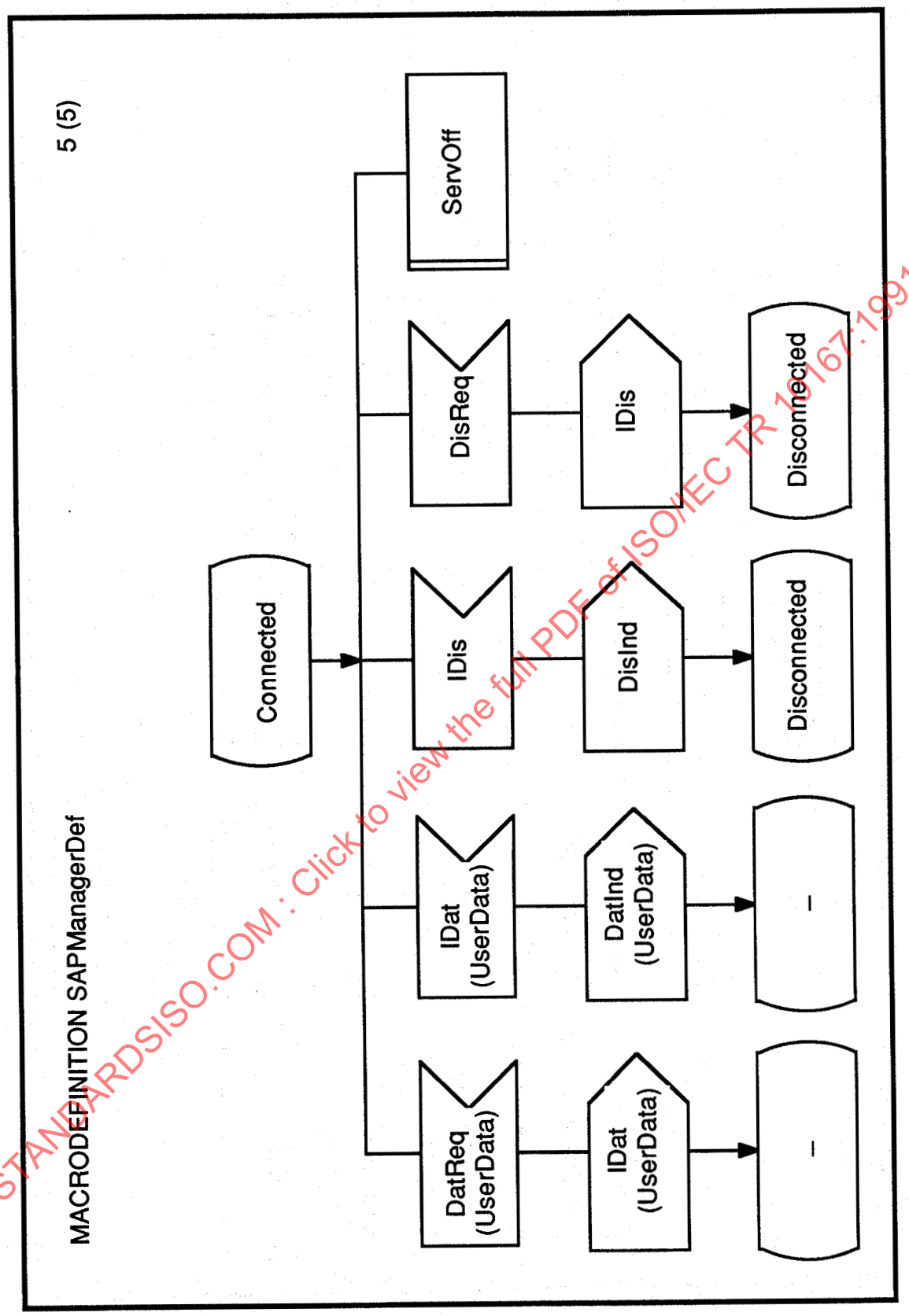
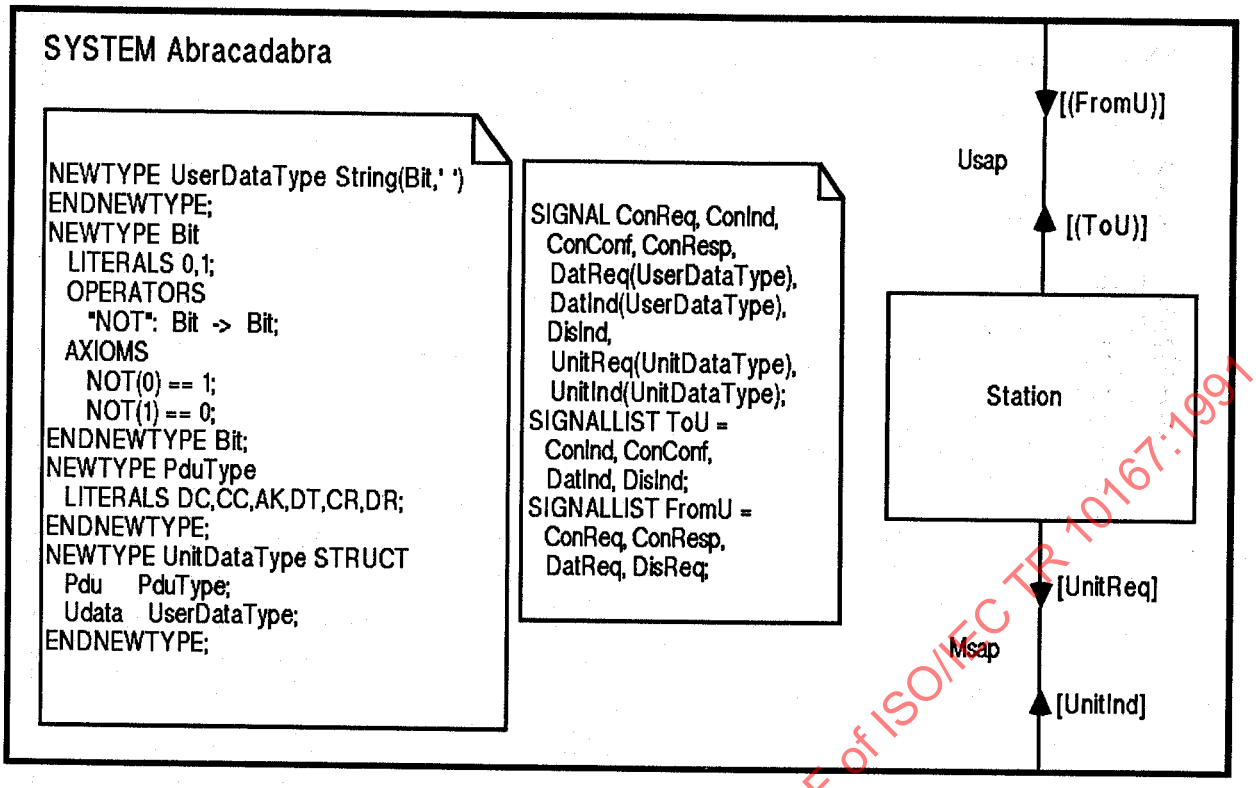


Figure 10.9 *(continued)*



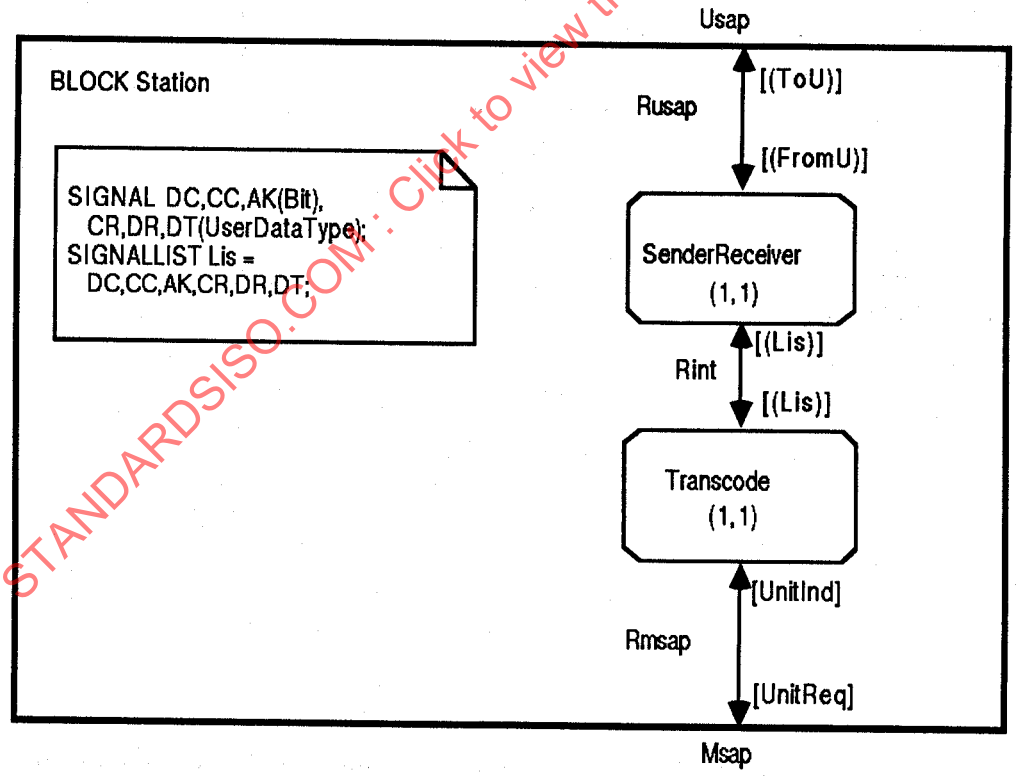


Figure 10.10: SDL Specification of Abracadabra Protocol

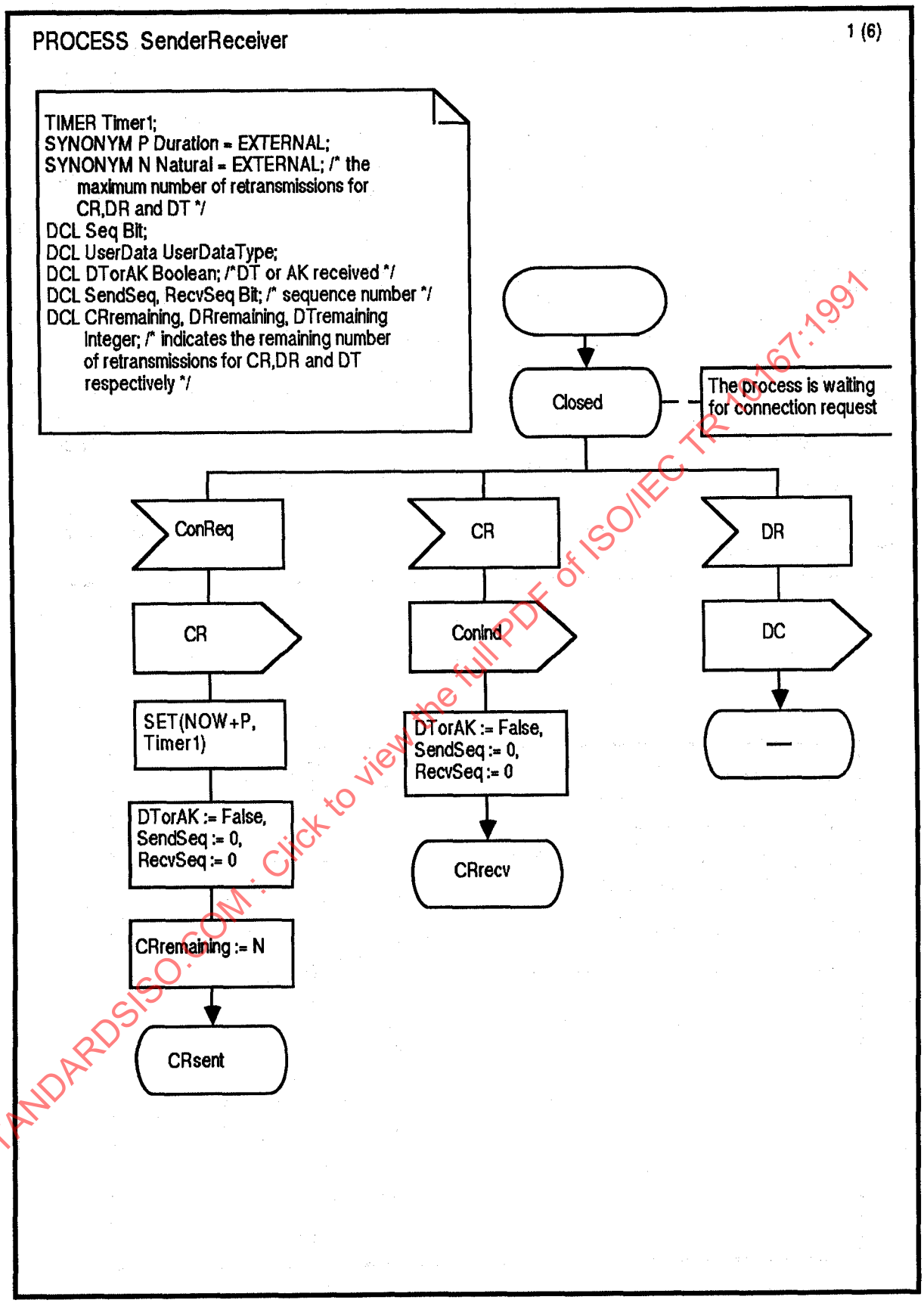


Figure 10.10 *(continued)*

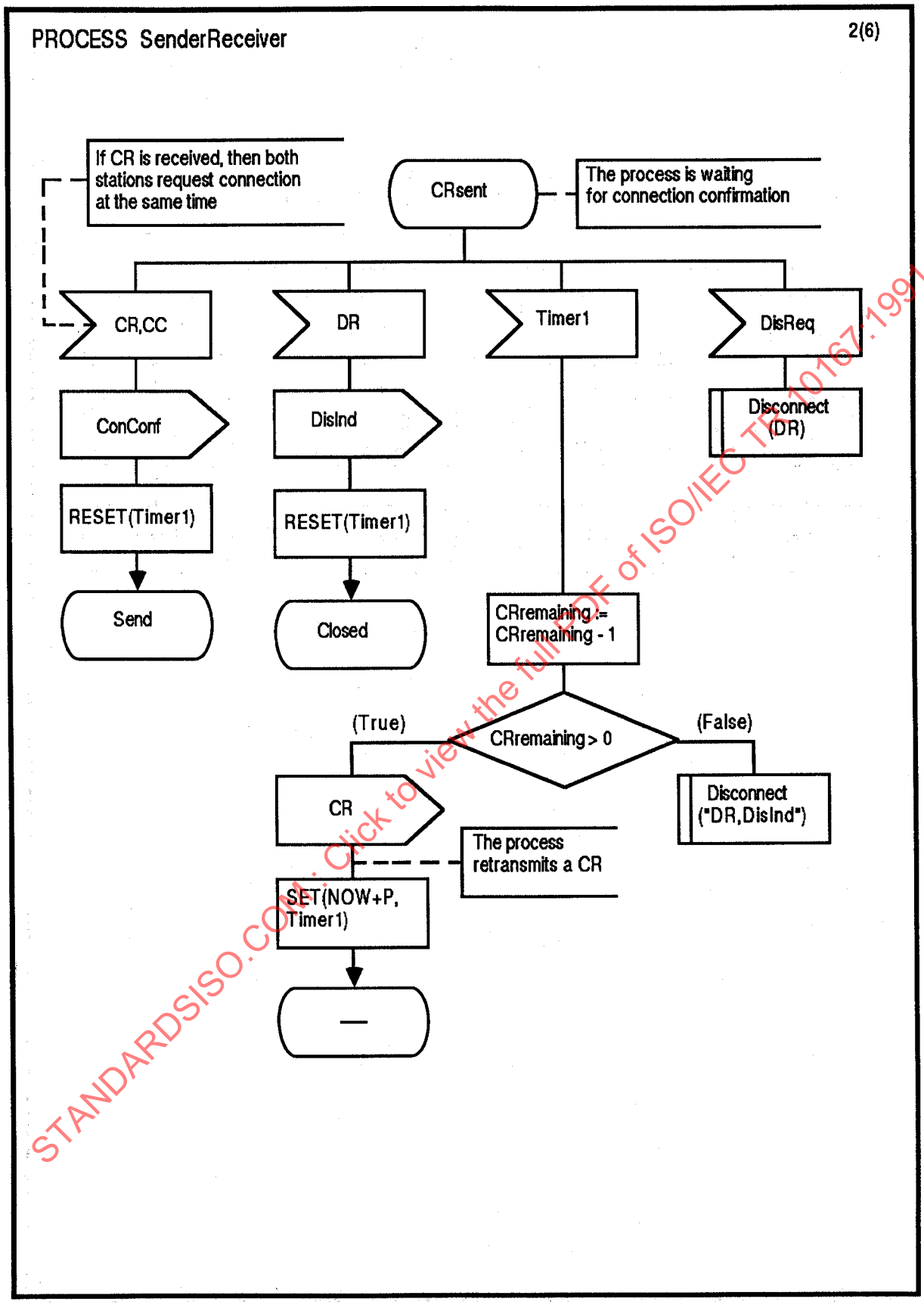


Figure 10.10 *(continued)*

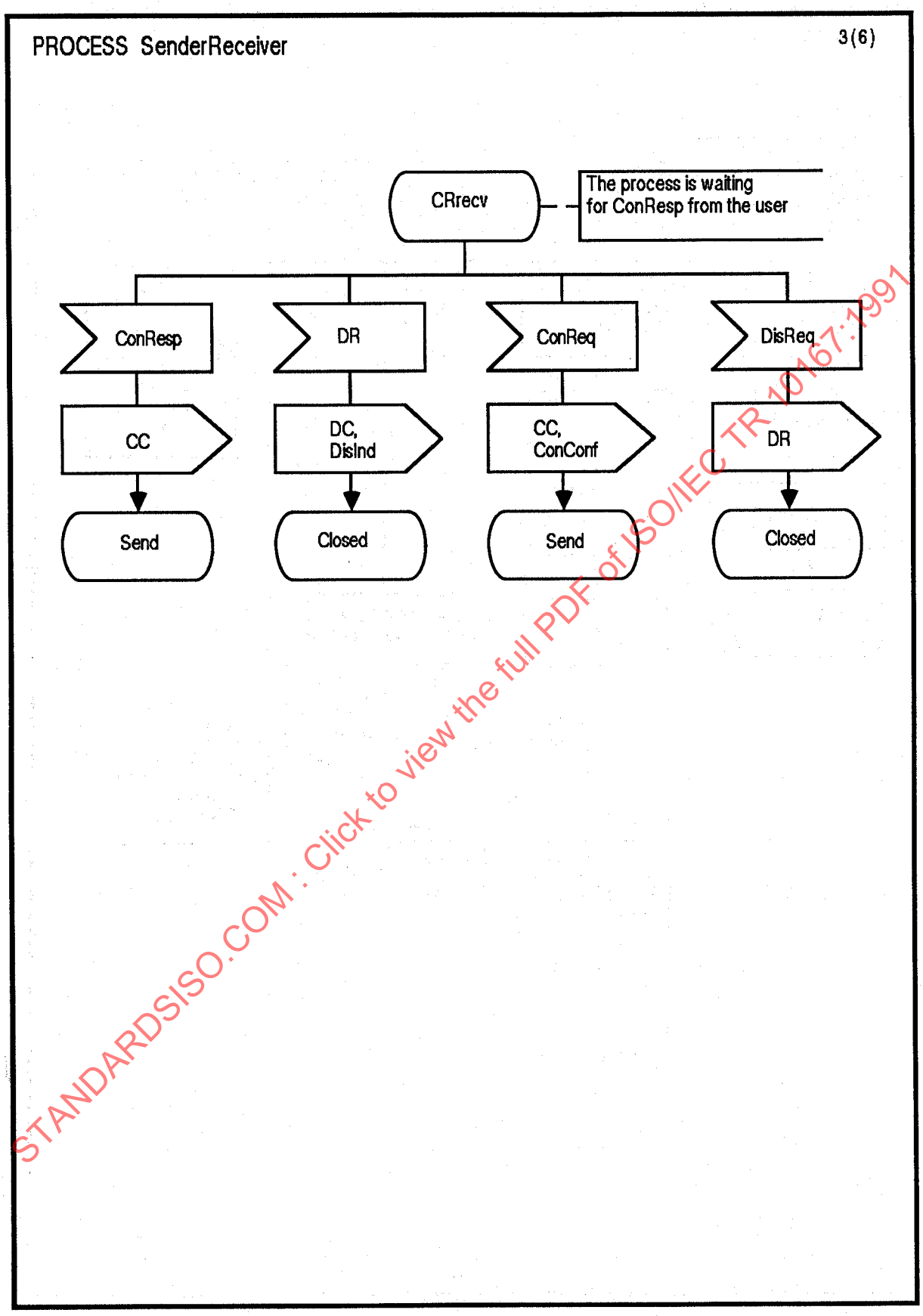


Figure 10.10 *(continued)*

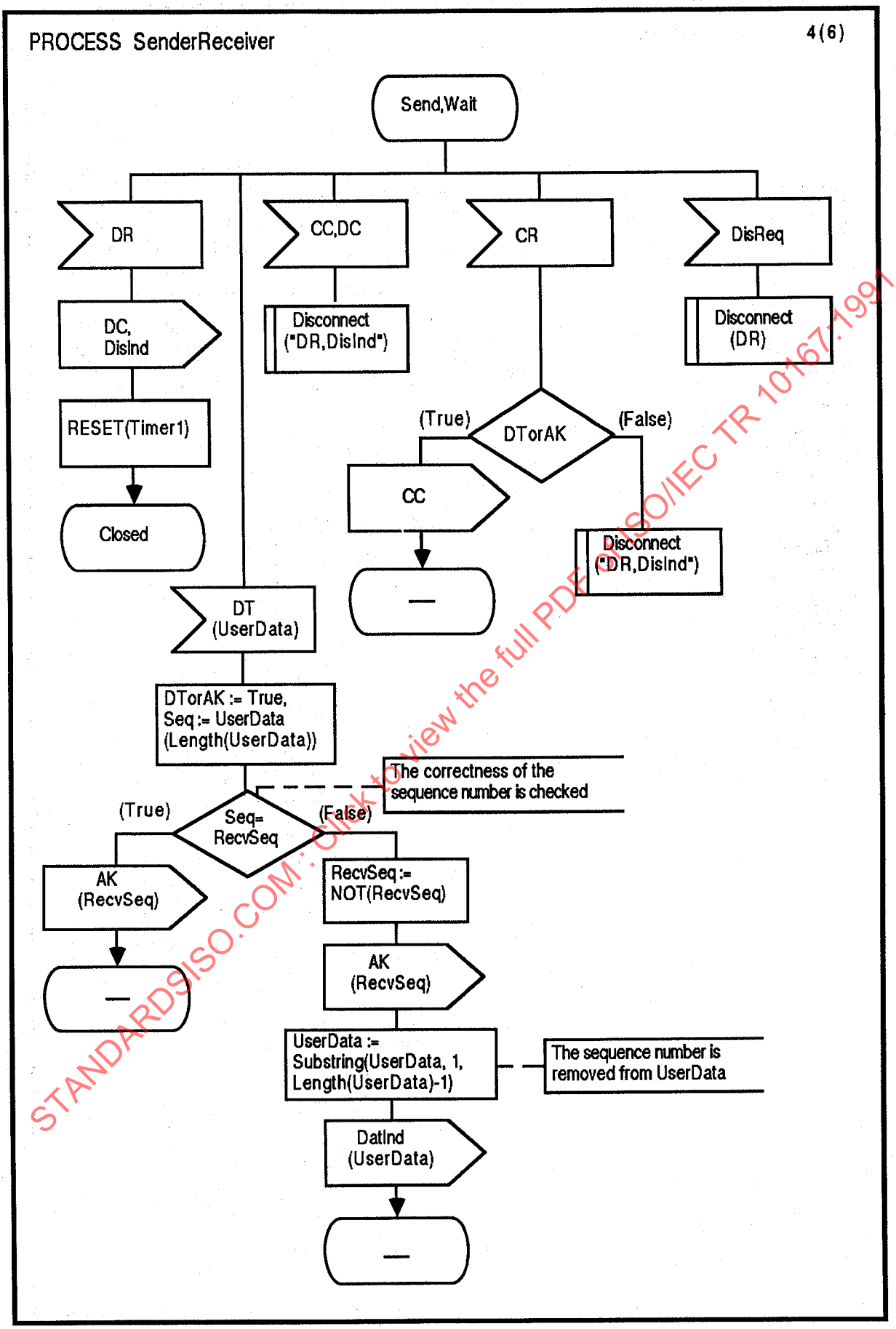


Figure 10.10 *(continued)*

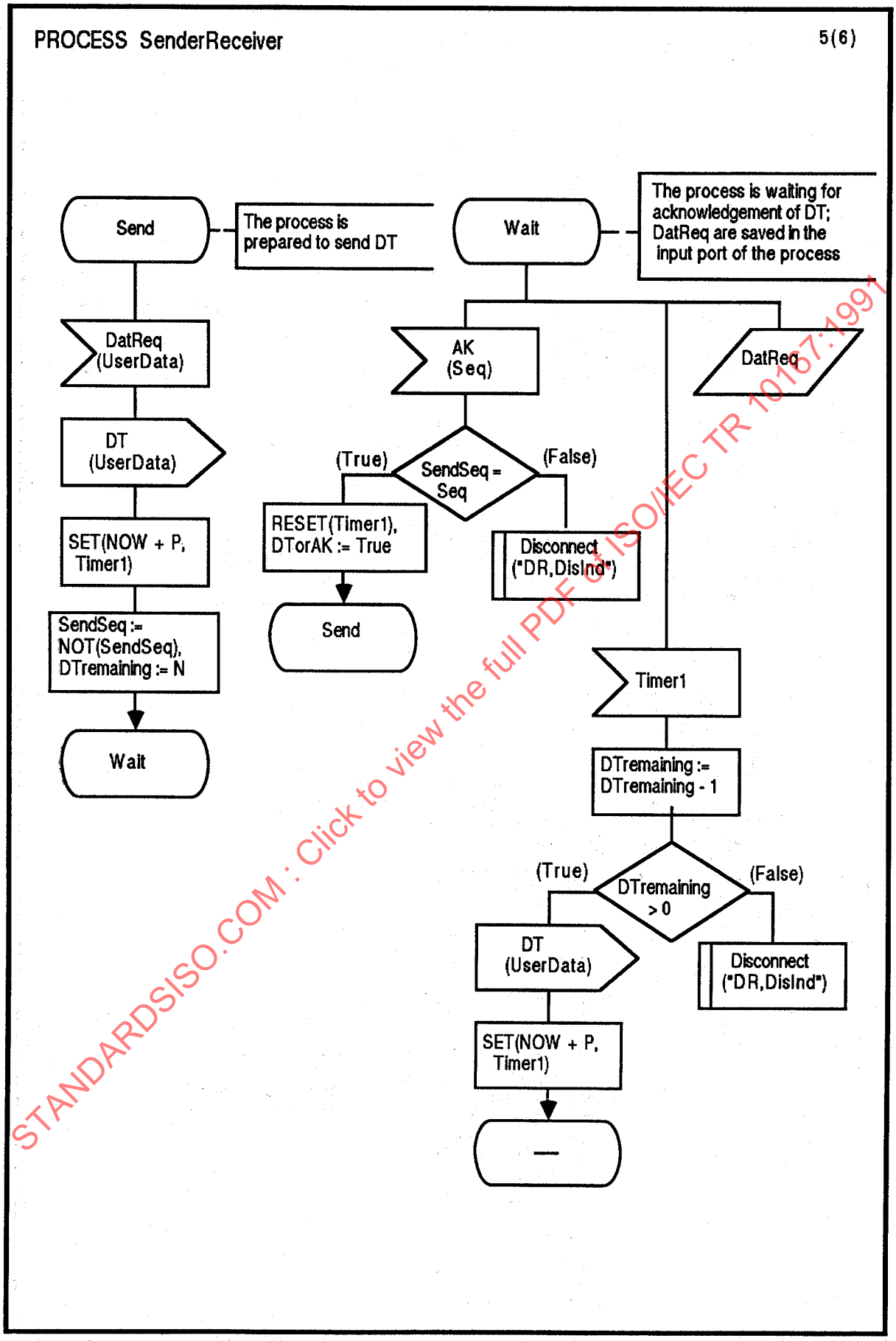
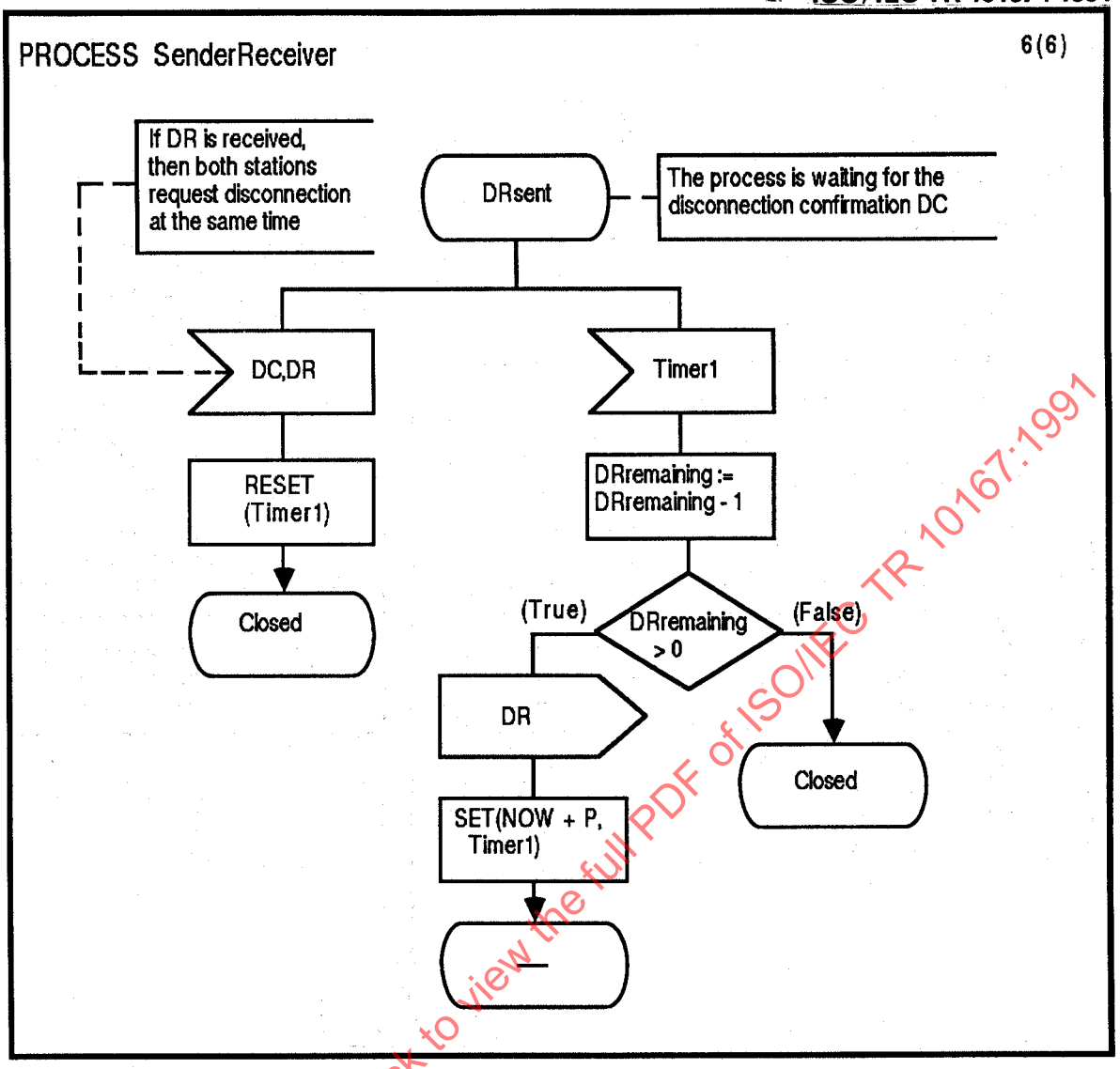


Figure 10.10 *(continued)*



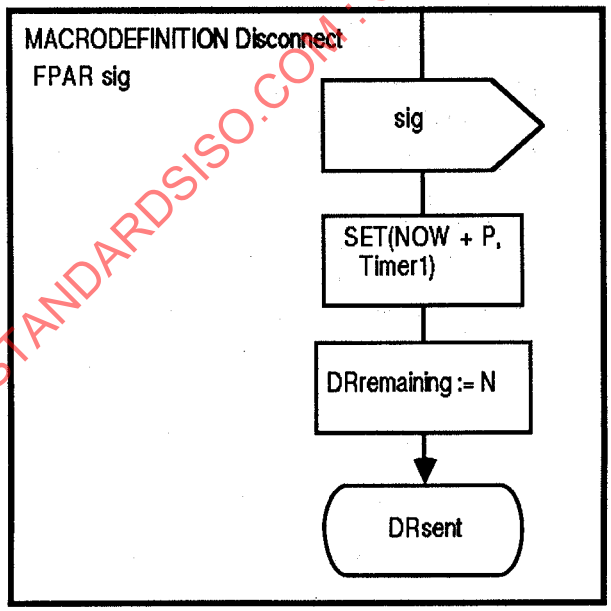


Figure 10.10 *(continued)*

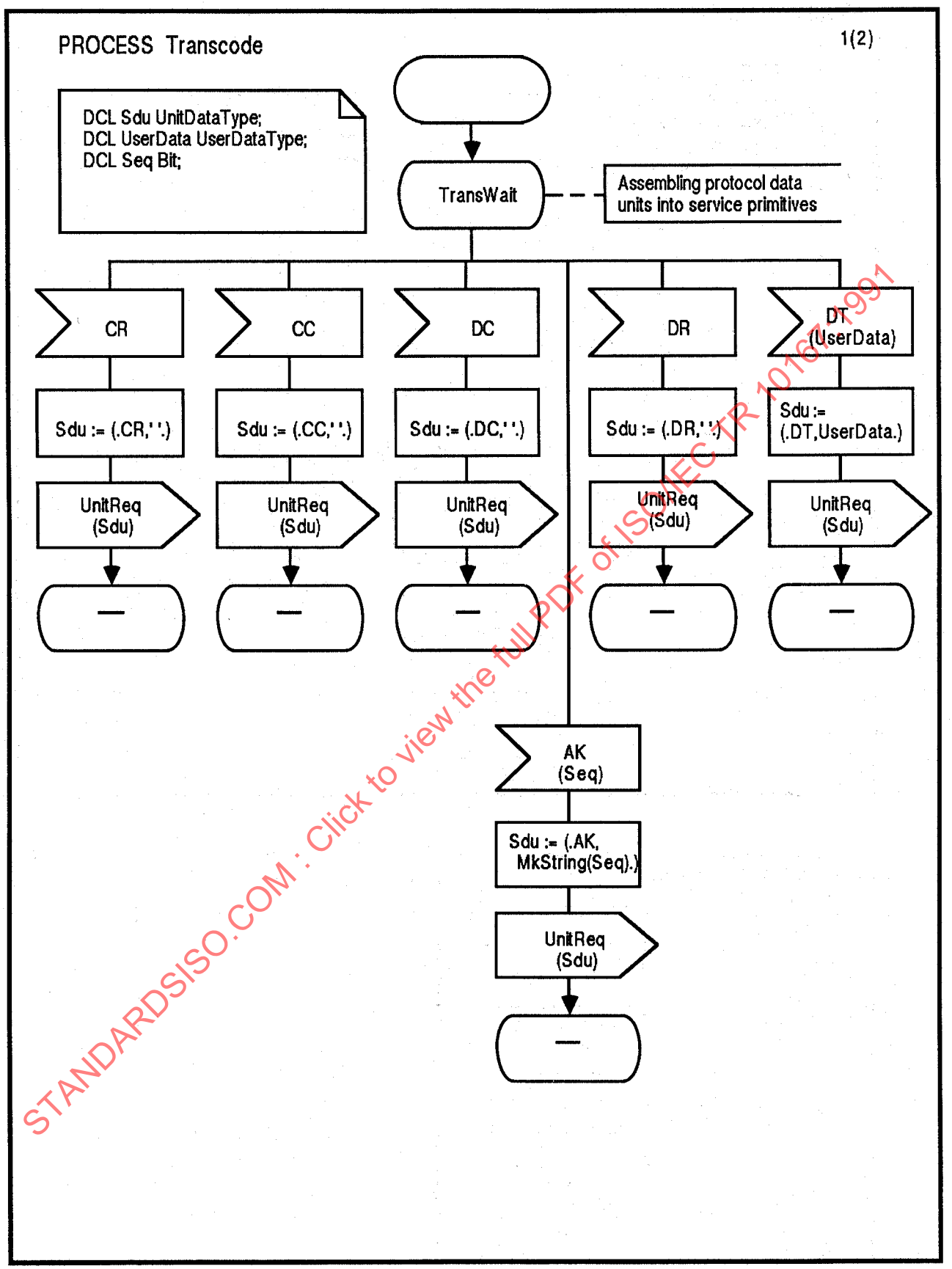


Figure 10.10 *(continued)*

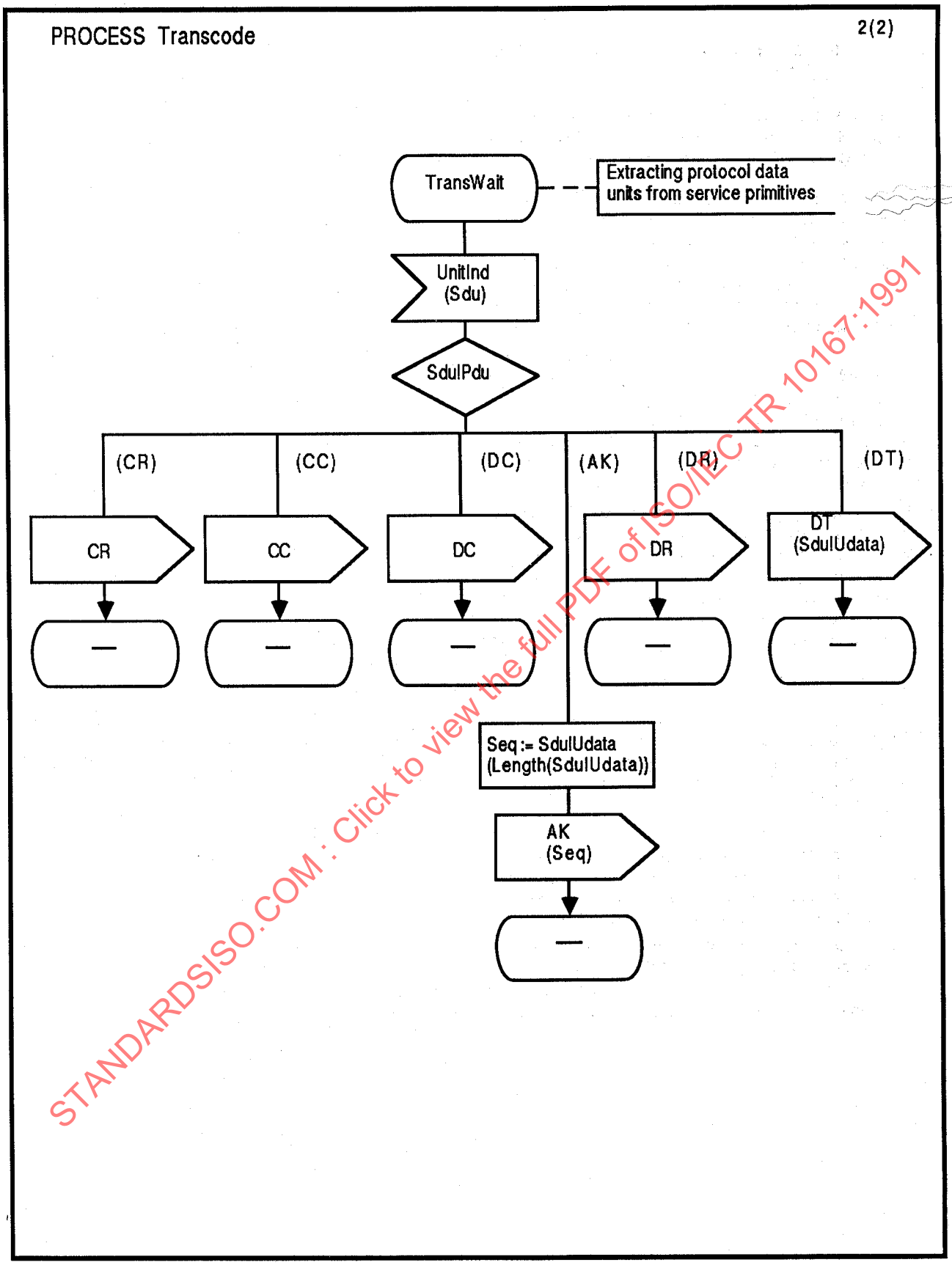


Figure 10.10 *(continued)*

10.6 Assessment of the Application of the FDTs

Considering the comparative complexity of this example, the variety of errors found was quite small. The following interesting, general classes or errors were found:

a) There are often 'phase change' problems, when the boundary between different phases of a Protocol is not clearly delineated. It is frequent in Protocols of this nature to find that in error cases one Protocol Entity has a different view of the state of connection from the other Protocol Entity (e.g. the so-called 'half-open connection'). It is important to describe these cases clearly.

b) It is also important to relate the behaviour of a Service to its underlying Protocol properly. By theoretical verification, it is possible to show using FDTs that a Service is indeed satisfied by its Protocol. A degree of confidence in this may also be established by 'simulation' (symbolic execution) of the Service and Protocol formal descriptions.

c) It is a difficult issue as to how to handle misbehaviour of a Service User. Because Services are not necessarily visible in an implementation, informal Service definitions tend to avoid defining what happens in such cases. However, a formal description has to ascribe some meaning to these cases. A common approach is to omit an explicit description of invalid Service User behaviour. However, the formal description still implicitly has some meaning. The approach depends on the FDT:

   1) In Estelle, such invalid behaviour would result in deadlock. Estelle experts therefore prefer to describe invalid Service User interactions as being accepted but ignored.

   2) In LOTOS, such invalid behaviour would also result in deadlock. However, because LOTOS experts prefer to take as implementation-independent view as possible, no explicit description would normally be included for Service User misbehaviour.

   3) In SDL, such invalid behaviour cannot arise from the point of view of the Service. (The invalid signals cannot be received by the Service: they are discarded before even entering the system.)

11 A Transport Protocol Example

This example is based on the CCITT T.70 Transport Protocol in order to illustrate how real Protocols may be formally described. It is, however, only an example and is not definitive with regard to T.70 as to either the informal or the formal descriptions.

11.1 Informal Description

11.1.1 Origins

The following prose description is an abridgement of CCITT Recommendation T.70. The abridgement has been made by removing or revising references to 'further study' work and other versions of T.70, paragraph numbers, figure numbers, table numbers, state tables, CCITT introductory material, and network considerations. Specifically, this abridgement is based on clause 5 and Table B-4 (E5) of T.70. The abridgement has been made and reproduced in this Technical Report for the following reasons:

a) to provide self-contained text which is integrated into this Technical Report; and

b) to avoid confusion due to references to items for 'further study' or to other versions of T.70; and

c) to emphasise that this example is *not* to be taken as an authoritative statement of T.70.

Except as indicated in 11.2 of this Technical Report, this example is intended to be identical to the Transport Protocol defined in T.70.

11.1.2 Transport Functions

11.1.2.1 General

The Transport Layer will perform all those functions that are necessary to bridge the gap between the services provided by the Network Layer and the services needed by the Session Layer. Therefore, the functions performed are dependent on two criteria: the services provided by the underlying Network Layer and the services required by the Session Layer.

It is the responsibility of the Transport Service User to select a given Quality of Service, which may imply the use of certain Transport Layer functions such as:

a) establishment of a Transport Connection:

1) Transport Connection identification; and
2) Transport Connection multiplexing.

b) data transfer:

1) sequence control; and
2) error detection; and
3) error recovery; and
4) segmenting and reassembling; and
5) flow control; and
6) purge.

c) termination of a Transport Connection.

NOTE — Not all of the above functions will be available in the basic Transport Service (see 11.1.2.3).

11.1.2.2 Transport Protocol Classes

Transport Layer functions are grouped (for ease of negotiation) into a hierarchical system of Transport Protocol Classes whereby Classes occupying superior positions in the hierarchy implement functions of the lower Classes together with the optional functions identified for their own Class. During Transport Connection establishment the use of a given Transport Protocol and optional functions should be negotiated according to the following rules:

a) the calling terminal indicates the Transport Protocol Class and (if applicable) optional functions required; and

b) the called terminal indicates the Transport Protocol Class and (if applicable) optional functions that it is willing to support; and

c) all parameters to be used in the Transport Connection must be explicitly indicated, otherwise default values will apply.

11.1.2.3 The Basic Transport Service (TS)

A limited set of Transport Layer functions is defined for a basic Transport Service. The basic Transport Service is provided by Transport Layer functions which are performed by Transport Layer Protocol Elements. Transport Protocol Data Units (**TPDUs**) carrying Transport Service (**TS**) User information or Control information are called **blocks**. Transport Layer block types are as follows:

a) Transport Connection Request (**TCR**) block; and

b) Transport Connection Accept (**TCA**) block; and

c) Transport Connection Clear (**TCC**) block; and

d) Transport Data (**TDT**) block; and

e) Transport Block Reject (**TBR**) block.

The **TCR** and **TCA** blocks are used to indicate the Protocol Class, and optional functions, applying to a Transport Connection. The **TCC** block is used to indicate the reason for refusing a Connection establishment. The **TDT** block carries information of the Transport Service User. The **TBR** block is used to report procedure errors to the remote terminal.

11.1.2.4 Transport Layer Functions

Basic Class functions and associated Transport Layer Protocol Elements, i.e. blocks, include:

a) Transport Connection establishment, Transport Connection identification, optional extended addressing and optional Transport Data Block Size negotiation (**TCR**, **TCA** and **TCC** blocks); and

b) data delimitation, segmentation/reassembling of arbitrarily long Transport Service Data Units (**TSDUs**). These are contained within **TDT** blocks. The end of a TSDU is indicated by a **TSDU End Mark** in the last data block; and

c) detection and indication of procedural errors (**TBR** block).

Other characteristics of the basic Transport Service are:

d) maintenance of TSDU integrity; and

e) overflow: if the user cannot absorb new data and if the appropriate buffers are not available, flow control is performed at Network or Data Link Layer as appropriate; and

f) error: no mechanism is provided within the Transport Layer to facilitate recovery from detected errors. Where such errors are detected the user of the Transport Service should be informed so that appropriate recovery action may be taken.

11.1.3 Connection Establishment and Termination Procedures

11.1.3.1 General

The Transport Layer Connection Establishment and Termination procedures shall also be used for negotiating Transport Protocol Class and, if applicable, optional Transport Connection functions. For the basic Transport Service, means are provided to establish a Transport Connection using a **TCR** block and a **TCA** block. This exchange provides:

a) a way to negotiate Options; and

b) a Transport Connection identification. The Transport Connection is identified by use of cross-references. Each end of the Connection is responsible for selecting a suitable Transport Connection Identifier.

This mechanism also provides an identification of the Transport Connection independent of any Network Connection identification and therefore provides independence from the life of the Network Connection. The binary value 0 should not be used as an identifier.

11.1.3.2 Transport Connection Request (TCR) Block

The calling terminal shall indicate a Transport Connection Request by transferring a **TCR** block to the remote terminal. The **TCR** block includes the Transport functions (e.g. Source Reference, Class, and optional functions) for negotiation of the characteristics of the Transport Connection being established.

11.1.3.3 Transport Connection Accept (TCA) Block

The called terminal shall indicate its acceptance of the Transport Connection by transferring a **TCA** block to the remote terminal. The **TCA** block includes the Transport parameters applying to the connection and to be used by the calling terminal.

If a terminal receives the request for an optional **TDT** block size it may either:

a) indicate its support by reproducing the requested value in the **TCA** block; or

b) request in the **TCA** block the use of a shorter allowable **TDT** block – the calling side either accepts this size by sending the first **TDT** block or disconnects the Network Connection; or

c) not accept the requested **TDT** Block Size parameter value by sending a **TCA** block without a **TDT** Block Size parameter. Therefore, the standardised **TDT** block size will apply.

A **TCR** requesting an optional **TDT** block size not supported by the called side should not be answered with **TBR**.

11.1.3.4 Transport Connection Clear (TCC) Block

If a Transport Connection cannot be established, the called terminal shall respond to the **TCR** block with a **TCC** block. The clearing cause shall indicate why the connection was not accepted. It is up to the calling side whether the receipt of a **TCC** will cause complete disconnection or whether a new **TCR** with a parameter different from the first one will be sent (e.g. another extended Layer 4 address).

NOTE — There is no explicit Transport Connection termination procedure. Therefore, the life-time of the Transport Connection is directly correlated with the life-time of the supporting Network Connection.

11.1.3.5 Transport Connection Collision

If the calling terminal receives a **TCR** block, it shall transfer a **TBR** block to notify the called terminal of the procedure error.

11.1.3.6 Extended Addressing

The Extended Addressing capability may be used to address terminals in a multi-terminal configuration. The extension addresses for called terminals are optional parameters to **TCR** and **TCA**. The receiving terminal shall respond with a **TCA** as shown in Figure 11.1.

The calling terminal may, when receiving a called terminal address in the **TCA**, act as specified in Figure 11.2.

11.1.4 Description of Data Transfer Procedures

11.1.4.1 General

The data transfer procedure described in the following clauses applies only when the Transport Layer is in the Data Transfer Phase, i.e. after completion of Transport Connection Establishment and prior to Clearing.

NOTE — When a connection is cleared, Transport Data blocks may be discarded. Hence, it is left to the Transport Service User to define Protocols able to cope with the various possible situations that may occur.

Received TCR	Receiver Reaction: Multi-terminal with extended addressing 1)	Receiver Reaction: Stand-alone terminal
Without extended addressing	Send TCA with extended addressing	Send TCA without extended addressing
With extended addressing	Send TCA with extended addressing 2)	Send TCA without extended addressing

NOTES

1 Multi-terminal configuration, with capability for extended addressing.

2 If the called terminal is occupied or out of order, the call should be routed to a default terminal or mailbox. The sender will then be informed of the routing by the extension address of the connected terminal. The receiver of **TCR** may also in this case react by sending **TCC**.

Figure 11.1: Receiving Terminal Reaction to TCR Addressing Options

Sent TCR:	Calling Terminal Reaction: TCA received with extended addressing: Absent	Correct	Incorrect
Without extended addressing	OK	Neglect extension 1)	
With extended addressing	1)	OK	1)

NOTES

1 Reaction left to the discretion of the calling terminal.

2 A terminal may react by releasing the network connection.

Figure 11.2: Calling Terminal Reaction to TCA Addressing Options

11.1.4.2 Transport Data Block (TDT) Length

The standard maximum **TDT** block length to be supported by all terminals is 128 octets including the data block header octets. Other maximum data field lengths may be supported in conjunction with an optional *TDT block size negotiation* connection function (see 11.1.6.4 and 11.1.6.5). Optional maximum data field lengths shall be chosen from the following: 256, 512, 1024, and 2048 octets. If the requested optional **TDT** block size cannot be supported, a shorter allowable **TDT** block size must be selected (see 11.1.3.3). The agreed maximum **TDT** block size should be aimed at for **TDT** blocks having the TSDU End Mark set to 0; a number of octets less than the agreed maximum shall not cause the receiving Transport Entity to reject this **TDT** block.

11.1.4.3 Transport Service Data Unit (TSDU) End

The TSDU End Mark is used to preserve the integrity of the TSDU. The TSDU End Mark is set to binary 1 in the last **TDT** data block carrying information related to a certain TSDU. Exceptionally, this **TDT** block may be sent without carrying user information for an immediate termination of a TSDU in certain error conditions.

In case of a TSDU that comprises a single **TDT** block the TSDU End Mark is also set to 1. In all other cases, the TSDU End Mark is set to zero.

11.1.5 Treatment of Procedure Errors

A terminal shall send a **TBR** block to the remote terminal to report the receipt of an invalid or not implemented block (if not explicitly specified otherwise in this description). During the establishment of a Transport Connection, terminals shall not send a **TBR** block upon the receipt of a **TCR** block whose parameters or parameter values are invalid or not implemented. In this case, terminals shall act as if no errors have occurred and send the appropriate response (if any).

A terminal receiving a **TBR** block shall take appropriate recovery action.

NOTE — A **TBR** whether invalid or valid shall not be answered by sending a **TBR** block.

11.1.6 Formats

11.1.6.1 General

Transport Protocol Data Units (**TPDUs**) carrying Transport Service (**TS**) User information or Control information are called blocks (see 11.1.2.3). All blocks contain an integral number of octets.

Bits of an octet are numbered 8 to 1 where bit 1 is the low order bit and is transmitted first. Octets of a block are consecutively numbered starting from 1 and are transmitted in this order.

TDT block(s) are used to transfer a Transport Service Data Unit (**TSDU**) transparently whilst maintaining the structure of the latter by means of the TSDU End Mark.

Control blocks (**TCR, TCA, TCC, TBR**) are used to control the Transport Protocol functions, including optional func-

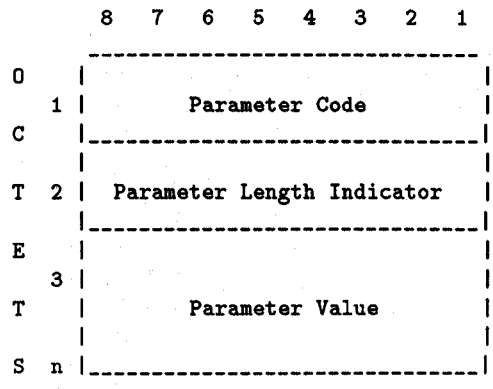
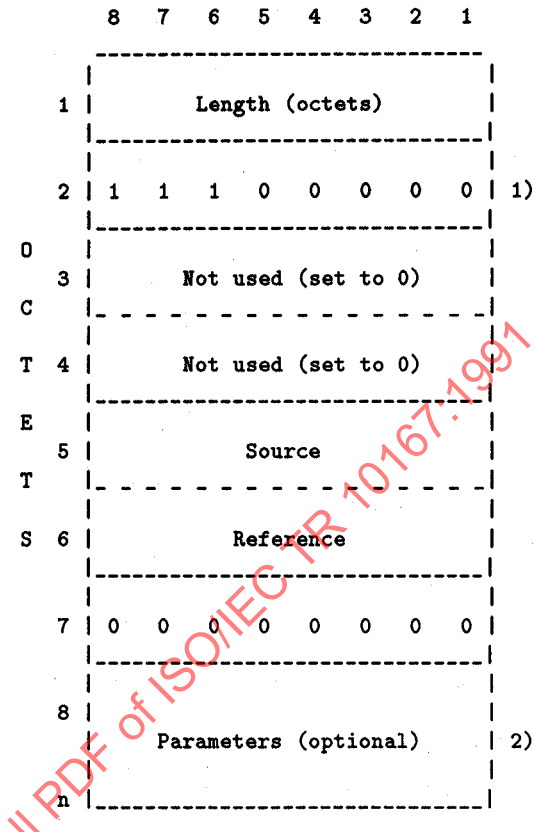


Figure 11.3: Parameter Element Coding Structure

Figure 11.4: General Block Structure



NOTES

1 Block type : TCR

2 The parameter field is present only when the terminal is requesting an optional Transport Connection function.

Figure 11.6: Transport Connection Request Block

tions.

A parameter field is present in all control blocks within the basic Transport Service to indicate optional functions. The parameter field contains one or more parameter elements. The first octet of each parameter element contains a parameter code to indicate the function(s) requested. The general coding structure is as shown in Figure 11.3.

The parameter code field is binary coded and, without extension, provides for a maximum of 255 parameters. Parameter code 11111111 is reserved for extension of the parameter code.

Octet 2 indicates the length, in octets, of the parameter value field. The parameter field length is binary coded and bit 1 is the low order bit of this indicator.

Octet 3 and subsequent octets contain the value of the parameter identified in the parameter code field. The coding of the parameter value field is dependent on the function being requested.

11.1.6.2 Structure of Transport Control and Data Blocks

Figure 11.4 illustrates the general structure of Transport Layer blocks. A summary of Transport Layer blocks is given in Figure 11.5.

Octet 1 contains the length indicator (**LI**). The value of this indicator is a binary number that represents the length in octets of the control block (including parameters) and the header length in octets of data blocks (excluding any subsequent user information). In both cases, this length does not include octet 1. The basic LI value shall be restricted to a maximum of 127 (i.e. a binary value of 01111111). Octet 2 contains the block type code. 1 to 4 of octet 2 are set to 0 for all Transport Layer blocks currently defined. Octet 3 and subsequent octets contain functional codes in a fixed format as per the block type. A parameter field or a data field containing Transport Service (TS) user data may optionally follow the functional code field.

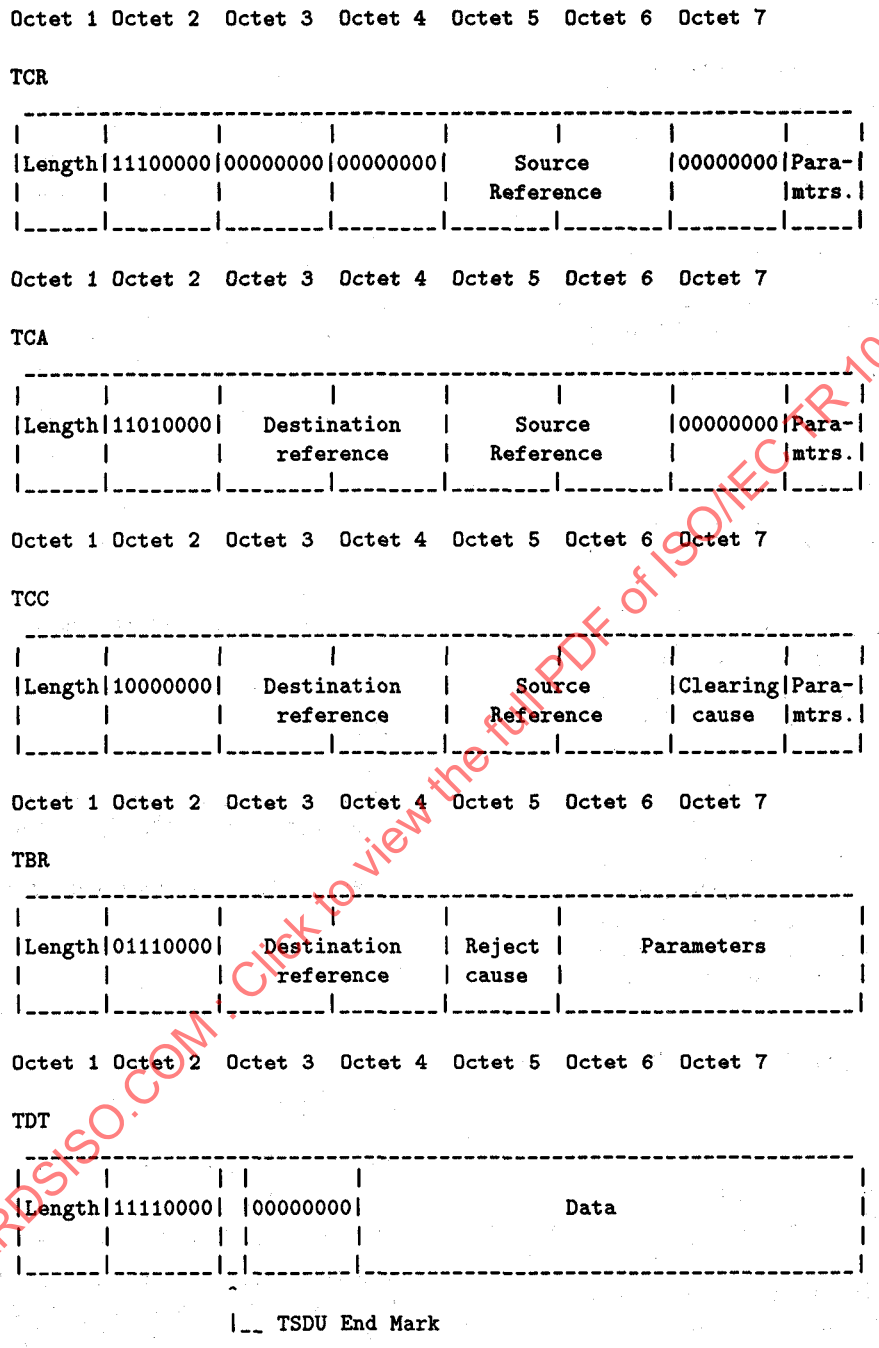
11.1.6.3 Concatenation

Concatenation of Transport Control and/or Transport Data blocks is not applicable.

11.1.6.4 Transport Connection Request (TCR) Block Format

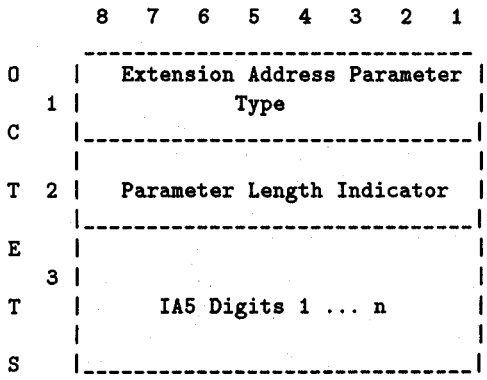
The format of a TCR block is shown in Figure 11.6.

A separate parameter is provided for the indication of Called



NOTE — The terms 'Source' and 'Destination' refer to the initiator and the recipient of the Transport Protocol Data Unit (TPDU), respectively. The value of the Source Reference is a local system parameter. The Source Reference of a received Transport block is to be used as Destination Reference in the response to that Transport block.

Figure 11.5: Transport Layer Block Types



NOTE — The Extension Address Parameter is '11000001' for calling address type or '11000010' for called address type.

Figure 11.7: Extended Addressing

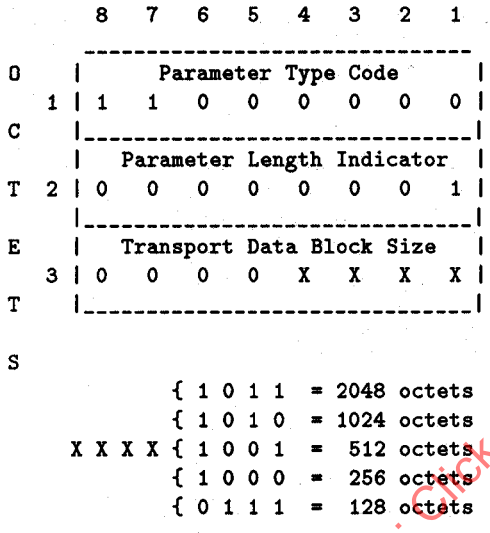


Figure 11.8: Transport Data Block Size Parameter

Extension Addresses. The coding of this parameter is shown in Figure 11.7. The setting of bit 8 for extended addressing should be ignored by the Transport Layer.

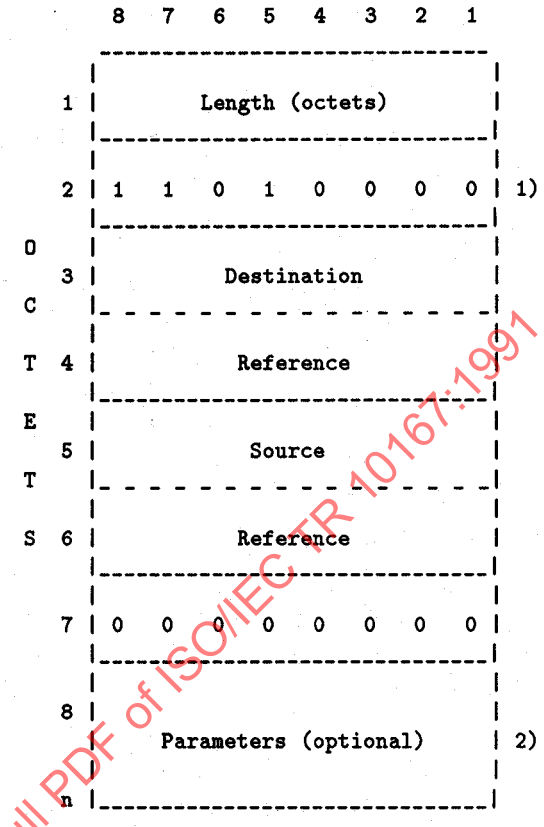
The Transport Data Block Size defines the proposed maximum Transport Data block size (in octets including the Transport Data block header) to be used over the requested Transport Connection. The coding of this parameter is as shown in Figure 11.8.

11.1.6.5 Transport Connection Accept (TCA) Block Format

The format of a **TCA** block is shown in Figure 11.9.

The Extended Addressing parameter is as for **TCR** (see 11.1.6.4).

The Transport Data Block Size parameter is as for **TCR**



NOTES

1 Block type: **TCA**.

2 The parameter field is present only when the terminal is requesting or confirming an optional Transport Connection function.

Figure 11.9: Transport Connection Accept Block

(see 11.1.6.4).

11.1.6.6 Transport Connection Clear (TCC) Block Format

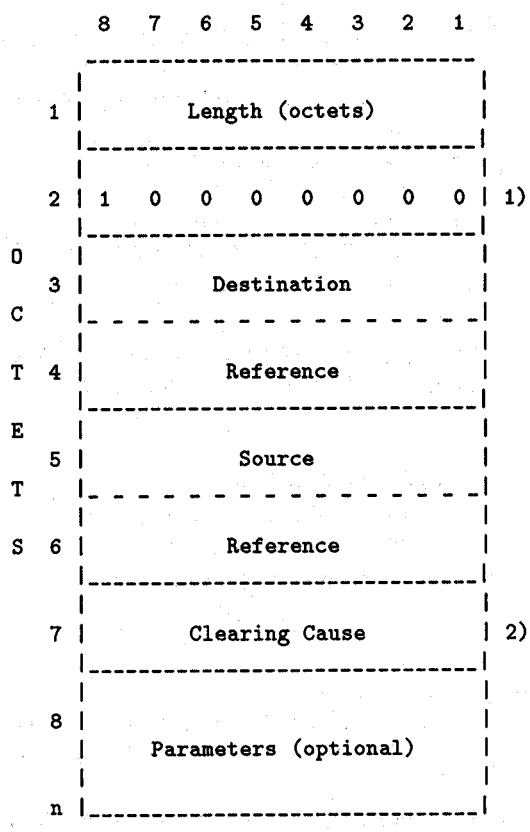
The format of a **TCC** block is shown in Figure 11.10.

The Additional Clearing Information parameter is provided to allow additional information relating to the clearing of the connection. The coding of this parameter is shown in Figure 11.11.

11.1.6.7 Transport Block Reject (TBR) Block Format

The format of a **TBR** block is shown in Figure 11.12.

The mandatory Rejected Block parameter is used to indicate the bit pattern of the rejected block up to and including the octet that caused the rejection. Only the first detected procedural error or parameter which cannot be acted upon shall be indicated by this method. The coding of this parameter is shown in Figure 11.13.



NOTES

1 Block type: **TCC**

2 Clearing Cause:

	8 7 6 5 4 3 2 1
0 - Reason not specified	= 0 0 0 0 0 0 0 0
1 - Terminal occupied	= 0 0 0 0 0 0 0 1
2 - Terminal out of order	= 0 0 0 0 0 0 1 0
3 - Address unknown	= 0 0 0 0 0 0 1 1

Figure 11.10: Transport Connection Clear Block

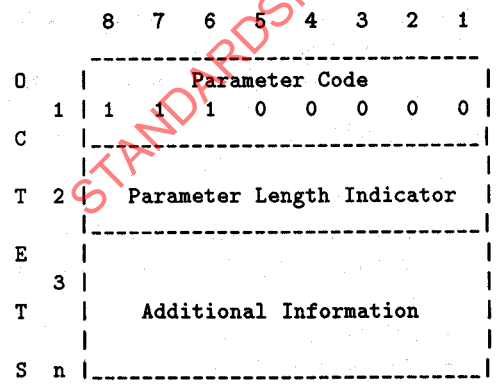
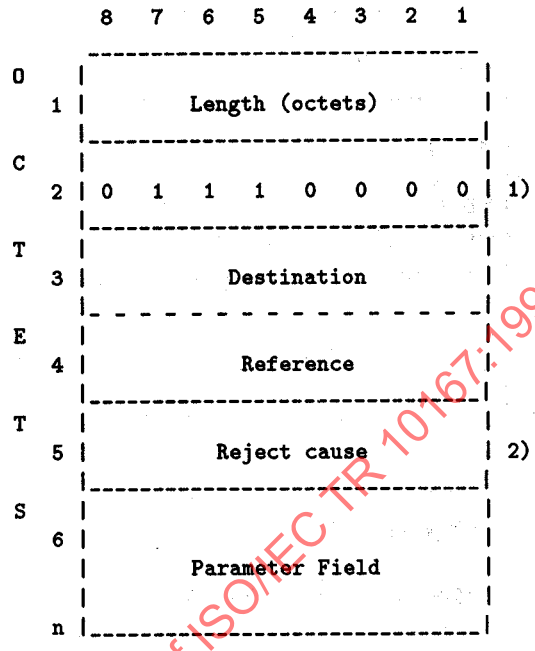


Figure 11.11: Additional Clearing Information Parameter



NOTES

1 Block type: **TBR**

2 Reject Cause:

	8 7 6 5 4 3 2 1
0 - Reason not specified	= 0 0 0 0 0 0 0 0
1 - Function not implemented	= 0 0 0 0 0 0 0 1
2 - Invalid block	= 0 0 0 0 0 0 1 0
3 - Invalid parameter	= 0 0 0 0 0 0 1 1

Figure 11.12: Transport Block Reject Block

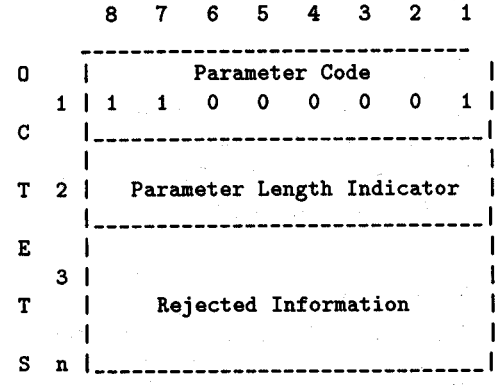
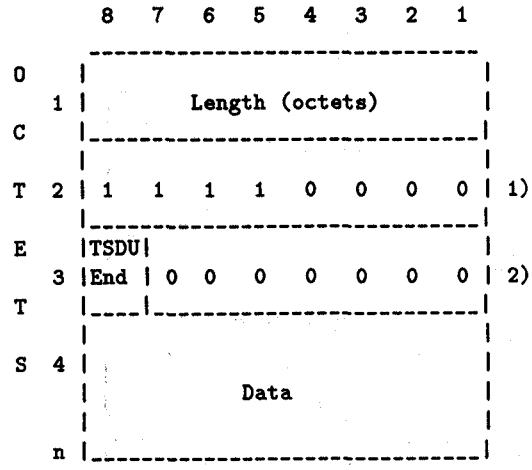


Figure 11.13: Rejected Block Parameter



NOTES

1 Block type: **TDT**.

2 TSDU End: indicates the End of TSDU when set to 1.

Figure 11.14: Transport Data Block

11.1.6.8 Transport Data Block (TDT) Format

The format of a **TDT** block is shown in Figure 11.14.

11.1.7 Invalid TPDUs

11.1.7.1 Invalid TPDUs due to Syntactic Errors

1) TCR

1.1) The value of Octet 1 (LI):

1.1.1) is not equal to the number of the **TCR** block octets minus 1; or

1.1.2) is greater than 127; or

1.1.3) is smaller than 6.

or

1.2) see 6) below.

2) TCA

2.1) The value of Octet 1 (LI):

2.1.1) is not equal to the number of the **TCA** block octets minus 1; or

2.1.2) is greater than 127; or

2.1.3) is smaller than 6.

or

2.2) see 6) below; or

2.3) The values of Octets 3 and 4 are not equal to Octets 5 and 6 respectively of the appropriate **TCR** block; or

2.4) The value of Octet 7 is non-zero; or

2.5) The parameter Transport Data Block Size is present, and:

2.5.1) its value is not equal to 07 (hexadecimal), in response to a **TCR** block without the Transport Data Block Size parameter; or

2.5.2) its value does not correspond to the rules of 11.1.3.3; or

2.5.3) its value is different from the values 07, 08, 09, 0A, 0B (hexadecimal); or

2.5.4) the **PLI** is not equal to 1.

or

2.6) **LI** is not equal to $(6+2\times N+(\text{sum of all PLIs}))$, where **N** is the number of parameters.

3) TCC

3.1) The value of Octet 1 (LI):

3.1.1) is not equal to the number of the **TCC** block octets minus 1; or

3.1.2) is greater than 127; or

3.1.3) is smaller than 6.

or

3.2) see 6) below; or

3.3) The values of Octets 3 and 4 are not equal to Octets 5 and 6 respectively of the appropriate **TCR** block; or

3.4) **LI** is not equal to $(6+2\times N+(\text{sum of all PLIs}))$, where **N** is the number of parameters.

4) TBR

4.1) The value of Octet 1 (LI):

4.1.1) is not equal to the number of the **TBR** block octets minus 1; or

4.1.2) is greater than 127; or

4.1.3) is smaller than 6.

or

4.2) see 6) below; or

4.3) The values of Octets 3 and 4 are not equal to Octets 5 and 6 respectively of the appropriate TC establishment block (**TCR** or **TCA**) received from the peer entity; or

4.4) The value of the **LI** minus 6 is not equal to the value of the **PLI**; or

4.5) The Rejected Block parameter is not present.

See also the NOTE in 11.1.5.

5) TDT

5.1) The value of the **LI** is not equal to 2; or

5.2) The TSDU End Mark is 0 and the information field is empty; or

5.3) The **TDT** block size is larger than negotiated in the establishment phase.

6) No Identified Block

The value of the TPDU Octet 2 is not equal to one of E0, E0, 80, 70, F0 (hexadecimal).

11.1.7.2 Invalid TPDUs due to Procedure Errors

The following are failure cases:

1) After sending a TCR

1.1) Reception of not a **TCA**; or

1.2) Reception of not a **TCC**; or

1.3) Reception of not a **TBR**.

2) After sending a TCA

2.1) Reception of not a **TDT**; or

2.2) Reception of not a **TBR**.

3) After sending a TDT

3.1) Reception of not a **TDT**; or

3.2) Reception of not a **TBR**.

4) After sending a TCC

Reception of not a **TCR**.

5) After sending a TBR

Reception of not a **TDT** while in the Data Phase.

6) After receiving a TDT with EM = 1

Reception of an empty **TDT** with **EM** = 1.

7) After receiving an empty TDT with EM = 1

Reception of an empty **TDT** with **EM** = 1.

8) After N-CONNECT response

Reception of not a **TCR**.

11.2 Deficiencies in the Informal Description

Most of the deficiencies found in the informal description were gaps in information. These have been resolved by reference to existing International Standards, in particular:

a) the OSI Connection-oriented Transport Service, as defined in ISO 8072 and CCITT X.214; and

b) Class 0 of the OSI Connection-oriented Transport Protocol, as defined in ISO 8073 and CCITT X.224; and

c) the OSI Connection-oriented Network Service, as defined in ISO 8348 and CCITT X.213.

As a general criterion, it was agreed that the Protocol is a slight simplification, but also extension, of T.70. The main motivation for introducing simplifications is to be found in the tutorial purpose of these Guidelines. Resolution of deficiencies that could not be achieved by reference to 11.1 was based upon the procedures of the Class 0 of the OSI Transport Protocol, simplified if necessary according to the same motivation.

11.2.1 Service Definitions (Clauses 11.1.2.1, 11.1.2.3 and 11.1.2.4)

11.2.1.1 Deficiency

Following the definitions of the OSI Reference Model, soundness and completeness of a Protocol definition require reference both to the Service provided and to the Service used. Apart from the aforementioned clauses, there is no reference as to which Network Service is used, nor to which Transport Service is provided. In particular no relation is specified between Blocks, on the one hand, and Transport Service Primitives (TSPs) or Network Service Primitives (NSPs) on the other. Actually, the very description of a Service by way of Protocol Data Units (PDUs, termed Blocks in this case) seems debatable. Given this gap, it also appears that even the relation between Blocks only is not complete.

11.2.1.2 Resolution

a) Services

In 11.1.2.1 both the Transport Service provided and the Network Service used should be defined or referenced (if they are standardised). For this Technical Report, the gap is filled by inserting the following paragraphs in 11.1.2.1, after the first paragraph:

> The Transport Service provided to the Session Layer is a subset of the OSI Connection-Oriented Transport Service defined in ISO 8072 and CCITT X.214. In particular, the Transport Expedited Data facility is not provided. For other restrictions see b) below.
>
> The Network Service used is a subset of the OSI Connection-Oriented Network Service defined in ISO 8348 and CCITT X.213. In particular, the Network Expedited Data and Receipt Confirmation facilities are not used. For other restrictions, see b) below.

b) Service Primitives

In 11.1.2.3 the TSPs used in the description should be defined. Also the NSPs used in the description should be defined, perhaps in the same clause. For this Technical Report, the gap is filled by inserting the following paragraphs in 11.1.2.3, at the beginning:

> A basic Transport Service (**TS**) is provided. This is a subset of the TS defined in ISO 8072 and CCITT X.214, according to the following restrictions:
>
> a) Expedited Data services are not provided, and neither Expedited Data Option nor Quality of Service parameters are defined in the **T-CONNECT** Service Primitives; and
>
> b) no User Data parameter is defined in the **T-CONNECT** Service Primitives; and
>
> c) no parameter is defined in the **T-CONNECT response**, **T-CONNECT confirm**, and **T-DISCONNECT** Service Primitives.

To provide the basic Transport Service, this Transport Protocol makes use of a subset of the Network Service, which is defined in ISO 8348 and CCITT X.213, according to the following restrictions:

a) neither Expedited Data nor Receipt Confirmation facilities are used; and

b) in **N-CONNECT request** and **indication** Service Primitives the only parameters defined are the Addresses; and

c) a limited form of **N-RESET** facility is used: only **N-RESET indication** and **response** primitives are used, with no parameters; and

d) no parameters are defined in the **N-CONNECT response**, **N-CONNECT confirm**, and **N-DISCONNECT** Service Primitives.

In 11.1.2.4 both the blocks and the relation between blocks, TSPs and NSPs should be defined. The gap is filled as follows:

a) the current content of 11.1.2.3 should be moved at the beginning of 11.1.2.4; and

b) at the end of existing paragraph a), replace the text in parentheses with:

> **TCR**, **TCA** and **TCC** blocks; **T-CONNECT**, **N-CONNECT** and **N-DATA** service primitives

c) in existing paragraph b), replace the second sentence with:

> These are contained in **T-DATA** service primitives, and each of them is transferred by way of a sequence of **TDT** blocks.

d) at the end of existing paragraph c), replace the text in parentheses with:

> **TBR** block; **T-DISCONNECT**, **N-RESET** and **N-DISCONNECT** Service Primitives.

e) at the end of existing paragraph f), replace the text 'the user ...' through to the end with:

> the TS User is informed by a **T-DISCONNECT indication** and the Network Connection is released.

11.2.2 Description of Procedures (Clause 11.1.3)

11.2.2.1 Deficiency

The relation between the Connection Establishment procedure and the Data Transfer procedure is not defined. Furthermore, the Termination procedure is not described.

11.2.2.2 Resolution

Remove the note in 11.3.4 and append at the end of 11.1.3.1 the following text:

An implicit connection termination procedure is made use of, by which the lifetime of the Transport Connection and the lifetime of the Network Connection are directly correlated. Furthermore, error recovery is not supported, so that the occurrence of an **N-RESET** indication leads to termination of the Transport Connection.

When a **TCR** block is sent, a timer is started. If this timer expires before receipt of a **TCA** or **TCC** block acknowledging the **TCR** block, the TS User is informed by sending a **T-DISCONNECT indication** and the Network Connection is released by an **N-DISCONNECT request**.

11.2.3 Protocol Classes (Clause 11.1.2.2)

11.2.3.1 Deficiency

There is no specification of the Protocol Classes, nor of Class negotiation.

11.2.3.2 Resolution

In fact, the Protocol caters for only one Class; the encoding of the Class and Options parameter in the **TCR** and **TCA** blocks is fixed as a zero-value octet. However, for the sake of compatibility and interoperability with other Protocol Classes, it is required in 11.1.5 that terminals shall not send a **TBR** block upon receipt of a **TCR** block whose parameters or parameter values are invalid or not implemented. For the same reason, TC identification references are exchanged. The following statement should be appended to 11.1.3.2:

> Terminals that comply with this protocol support only Class 0, but may interoperate with terminals that support other Classes and Class negotiation (see 11.1.5).

11.2.4 Missing Definitions (Clause 11.1)

11.2.4.1 Deficiency

There are no definitions of several terms used, including:

a) Transport Layer Protocol Element; and

b) TPDU; and

c) Transport Layer blocks; and

d) Transport Layer functions; and

e) Transport Layer procedures; and

f) Transport Connection collision; and

g) TSDU integrity; and

h) called and calling terminal; and

i) Protocol error; and

j) control block; and

k) mailbox; and

l) sender and receiver.

11.2.4.2 Resolution

All terms should be defined, either in the description itself, or by reference to a document in which they are defined. It was decided not to fill the gap in this Technical Report.

11.2.5 Unspecified Functions (Clause 11.1)

11.2.5.1 Deficiency

The following functions are referenced without explanation as being excluded from the protocol:

a) multiplexing; and
b) sequence control; and
c) error detection; and
d) segmenting; and
e) flow control; and
f) purge; and
g) how to inform the user of an error; and
h) connection identification; and
i) identification of terminals; and
j) routing; and
k) extended addressing; and
l) association of TPDUs with Transport Connections; and
m) recovery after receipt of a **TBR** block; and
n) concatenation (see 11.2.6).

11.2.5.2 Resolution

The functions should be explained directly or by reference, and the use or non-use of each should be specified. In this Technical Report, the gap is filled only for functions that are supported by this protocol, i.e. a) segmentation, b) extended addressing, and c) error detection:

a) Segmenting of a TSDU may only start after the Transport Entity has received the complete TSDU from the TS User in a **T-DATA request**. Similarly, a **T-DATA indication** is executed only after the complete TSDU has been reassembled by the Transport Entity.
b) Multiple, possibly concurrent, Transport Connections may be supported by one Transport Protocol Entity. Use or non-use of extended addressing in the **TCR** and **TCC** blocks is related to this possibility, which is non-deterministic. In the **TCR** and **TCC** blocks, both the Called Address and the Calling Address are optional.
c) When a **TBR** block is sent or received, the TS User is informed by a **T-DISCONNECT indication**. A Transport Entity receiving a **TBR** block releases the Network Connection by means of an **N-DISCONNECT request**. The Transport Entity sending a **TBR** block starts a timer. When this timer expires the Network Connection is released. On receipt of an **N-DISCONNECT indication** the timer is stopped.
Receipt of an **N-RESET indication** is answered with an **N-RESET response**, followed by release of the Network Connection; the TS User is informed by means of a **T-DISCONNECT indication**.

11.2.6 Non-Use of Concatenation (Clause 11.1.6.3)

11.2.6.1 Deficiency

The non-use of concatenation should not be defined in the format section, but be introduced in 11.1.2.4 amongst the functions. The concept of concatenation is not described at all.

11.2.6.2 Resolution

The text of 11.1.6.3 should be moved to 11.1.2.4 as follows:

> g) concatenation: concatenation of Transport control and/or Transport data blocks is not applicable, so that receipt of concatenated blocks is treated as a procedure error (see 11.1.5).

11.2.7 Responding Address (Clause 11.1.3.6 b))

11.2.7.1 Deficiency

It is not clear whether the Called Address parameter in a **TCA** may be different from that in a **TCR** if the call is re-routed.

11.2.7.2 Resolution

It is assumed that they are the same.

11.2.8 Multiple SAP Connections (Clause 11.1.2.3)

11.2.8.1 Deficiency

May a Transport Protocol Entity support more than one Transport Connection at a TSAP or an NSAP?

11.2.8.2 Resolution

It is assumed that this is permissible.

11.2.9 Reaction to Incorrect TCA (Clause 11.1.3 and Figure 11.2)

11.2.9.1 Deficiency

It is not stated which forms of reaction are allowed by the expression: 'left to the discretion of the calling terminal'. Allowing *any* reaction is too vague.

11.2.9.2 Resolution

The interpretation followed in this Technical Report is that an invalid parameter in the **TCA** is treated as for procedure errors (see 11.1.5).

11.3 Estelle Description

11.3.1 Architecture of the Formal Description

The top level architecture of the description is shown in Figure 11.15. This architecture is made up of three kinds of modules:

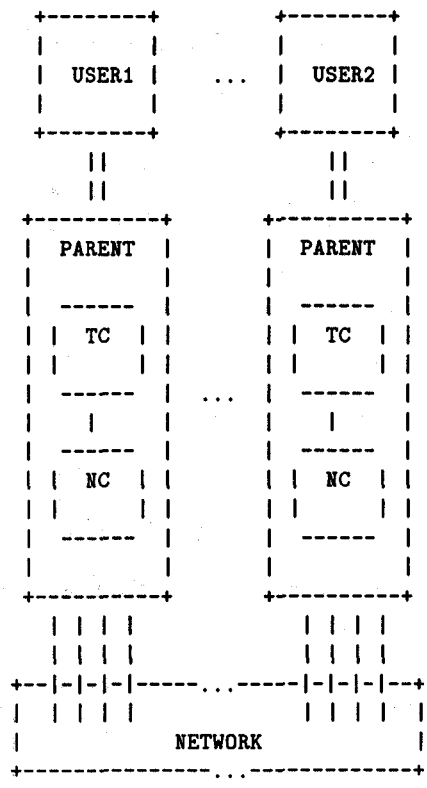


Figure 11.15: Architecture of A Transport Protocol in Estelle

a) two instances of type **USER_T** representing the Transport Service Users; these instances are referred to through module variables **USER1** and **USER2**; and

b) two instances of type **PARENT_T** representing the Transport Protocol Entities; both instances are created using the same module variable **PARENT**; and

c) one instance of type **NETWORK_T** representing the underlying Network Service; this instance is referred to through module variable **NETWORK**.

This architecture remains static once created during the Estelle description initialisation. Because of the nature of the Protocol, the bodies of modules **USERi** and **NETWORK** are not specified. Several external interaction points are defined for representing the inter-layer communication:

a) **TCEP**, which is an array of external interaction points declared in both **USER_T** and **PARENT_T** modules, represents all potential Transport Connections that may be established on behalf of some User The upper bound of the array is defined by constant **MAXTCEP** whose concrete (integer) value is left to be defined (i.e. is implementation-dependent).

b) **NCEP**, which is an array of external interaction points declared in the **PARENT_T** module header, represents all potential Network Connections that may be established from the associated **PARENT** module. The upper bound of the array is defined by constant **MAXNCEP** whose concrete (integer) value is left to be defined (i.e. is implementation-dependent).

The description also illustrates the dynamic structuring capabilities of Estelle. One pair of module instances, whose types are respectively **TC_T** and **NC_T**, is created dynamically by each Transport Protocol Entity during the establishment phase of a Transport Connection. These module instances communicate (i.e. exchange interactions) through their interaction points **TC_IP** and **NC_IP**, which are dynamically connected.

The other external interaction point **TCEP**, declared in the module header **TC_T**, is attached to one of the interaction points introduced by the **TCEP** array declaration within the **PARENT_T** module. (The index within the array represents the Transport Connection Endpoint Identifier.) The result of this attachment is that the interactions related to a Transport Connection are sent directly to the module instances that must process them. Similarly the other external interaction point, declared in the module header **NC_T**, is attached to one of the interaction points declared within the **NCEP** array.

Module instances of type **USER_T**, **PARENT_T** and **NETWORK_T** are system instances that behave asynchronously with respect to each other. Module instances of type **PARENT_T** are **systemactivities**, as the informal description of the Protocol gives no reason for synchronising the behaviour of several Transport Connections managed by the same Transport Protocol Entity. Similar reasoning motivates the use of the **systemactivity** class for module instances of type **USER_T** and **NETWORK_T**.

11.3.2 Explanation of Approach

This approach was motivated by the desire to illustrate some features of Estelle, e.g. structuring mechanisms (both dynamic and static). The **PARENT**, **USER**, and **NETWORK** modules are created (statically) and bound in the **initialize** part of the description. The **TC** and **NC** modules (children within **PARENT**) are created (dynamically) and bound during the Connection Establishment Phase. They are described in the transitions of the **trans** part of **PARENT**. (Note that all the transitions in the description have unique names.) When the Connect Request comes from the local User, the mechanisms required to manage the local connection endpoint are created in one transition (**trans PR2**), i.e. both the **TC** and **NC** modules are created and bound. When the Connect Request comes from the remote User, the mechanisms required to manage the local connection endpoint are created in two transitions. First, the **NC** module is created and the **attach** is performed on **NCEP** in **trans PR3**; then, upon receipt of the **TCR**, the **NC** module requests the **PARENT** to create the **TC** in **trans NC9**. Later, the **PARENT** notices such a request and creates the **TC** module, performing the **connect** of interaction points **TC_IP** and **NC_IP**.

This example also uses the **delay** transitions (**trans TC12** and **TC39**) to illustrate the use of the timer notion and its representation in Estelle.

A User may make a transition override some others, by

giving the **priority** clause. An example of this is **trans NC15** in which the Transport Connection Request of the remote User could not be fulfilled locally, therefore a **TCC** is sent in answer to the **TCR**.

There are several examples of non-deterministic choices of transitions; one significant case is when the Network Connection Indication is either accepted (**trans PR3**) or rejected (**trans PR4**).

11.3.3 Formal Description

```
specification TP;
     default individual queue;
     timescale SECONDS;

     const
          MAXPRIORITY=0;
          NUM_USER=any INTEGER;
          NUM_NETWORK=any INTEGER;
          MAXDATA=any INTEGER;
          MAXCEP=any INTEGER;
          MAXTCEP=MAXCEP;
          MAXNCEP=MAXCEP;
     type
          TADD=array [1..MAXDATA] of CHAR;
          TADDRESS=
               record
                    L:INTEGER;
                    VAL:TADD
               end;
          NADDRESS=...;
          TBR_REASON=(INVALID_PDU);
          OCTET=0..255;
          LEN_T=0..MAXDATA;
          ID_T=1..MAXDATA;
          DATA_T=
               record
                    L:LEN_T;
                    D:array [ID_T] of OCTET
               end;
          REF_T=0..65535;
          TLV_PAR=(P_CALLING,P_CALLED,
               MAX_BL,CL_INFO,REJ_PDU);
          CLEAR_C=(NON_SPEC,OCCUPIED,OUT_OF_ORD,
               UNKNOWN);
          BLOCK_T=(TCR,TCA,TCC,TDT,TBR);
          PDU_T=
               record
                    case CODE:BLOCK_T of
                         TCR,TCA:(DREF1,SREF1:
                                   REF_T;
                              T_CALLING:TADDRESS;
                              T_CALLED:TADDRESS;
                              MAX_BLOCK_SIZE:
                                   INTEGER);
                         TCC:(DREF2,SREF2:REF_T;
                            CAUSE:CLEAR_C;
                            CL_OPT:DATA_T);
                         TDT:(EOT:BOOLEAN;
                              DATA:DATA_T);
                         TBR:(DREF3:REF_T;
                              REASON:TBR_REASON;
                              TPDU:DATA_T)
                    end;

channel TS_INTERFACE(USER,PROVIDER);
     by USER:
          TCON_REQ(T_CALLING,
               T_CALLED:TADDRESS);
          TCON_RESP;
          TDT_REQ(TSDU:DATA_T);
          TDIS_REQ;
     by PROVIDER:
          TCON_IND(T_CALLING,
               T_CALLED:TADDRESS);
          TCON_CONF;
          TDT_IND(TSDU:DATA_T);
          TDIS_IND;

channel NS_INTERFACE(USER,PROVIDER);
     by USER:
          NCON_REQ(N_CALLING,
               N_CALLED:NADDRESS);
          NCON_RESP;
          NDT_REQ(NSDU:DATA_T);
          NRST_RESP;
          NDIS_REQ;
     by PROVIDER:
          NCON_IND(N_CALLING,
               N_CALLED:NADDRESS);
          NCON_CONF;
          NDT_IND(NSDU:DATA_T);
          NRST_IND;
          NDIS_IND;

function TCON_REQ_OK(
     CALLING,CALLED:TADDRESS):
          BOOLEAN;primitive;

function MAP_TADDRESS(ADD:TADDRESS):
     NADDRESS;primitive;

function SET_DTLENGTH(
     MAX_BLOCK_SIZE:INTEGER):
          INTEGER;primitive;

function IS_VALID_TPDU(PHY_PDU:DATA_T):
     BOOLEAN;primitive;

function GET_CODE(PHY_PDU:DATA_T):
     BLOCK_T;primitive;

module PARENT_T systemactivity;
     ip
          TCEP:array [1..MAXTCEP] of
               TS_INTERFACE(PROVIDER);
          NCEP:array [1..MAXNCEP] of
               NS_INTERFACE(USER);
end;

body PARENT_BODY for PARENT_T;
```

```
const
     MAXDTLENGTH=any INTEGER;
type
     REQUEST_T=(UNDEFINED,REJECTED,
          CREATED,REQUESTED);
     TC_IMAGE_T=
          record
               CALLING,CALLED:TADDRESS;
               DREF:REF_T;
               MAX_BLOCK_SIZE:INTEGER
          end;

channel PDU_CHANNEL(TC,NC);
     by TC:
          NDIS_REQ;
     by NC:
          INVALIDPDU(INVALID:DATA_T);
          NCON_CONF;
          NDIS_IND;
     by TC,NC:
          TPDU(PDU:PDU_T);

module TC_T activity(TC_CALLING,
     TC_CALLED:TADDRESS;
     TC_SREF,TC_DREF:REF_T;
     TC_MXBL_S:INTEGER);
     ip
          TCEP:TS_INTERFACE(PROVIDER);
          TC_IP:PDU_CHANNEL(TC);
     export
          TC_RELEASE_REQ:BOOLEAN;
end;

body TC_BODY for TC_T;
     const
          TCR_TIMER=60;
          TBR_TIMER=60;
     state
          IDLE,CLOSED,WFNC,WFND,WFTCA,
          WFTRESP,OPEN,ERROR,
          PRE_RELEASE;
     var
          PDU_SEND:PDU_T;
          TSDU_SEND,TSDU_RCVD:DATA_T;
          ITPDU:DATA_T;

     function D_LENGTH(DATA:DATA_T):
       LEN_T;
          begin
          D_LENGTH:=DATA.L
          end;

     procedure D_NULL(var DATA:DATA_T);
          var
               INDEX:INTEGER;
          begin
          for INDEX:=1 to MAXDATA do
               DATA.D[INDEX]:=0;
          DATA.L:=0
          end;

procedure D_COPY(
  FROM_DATA:DATA_T;
  var TO_DATA:DATA_T);
     var
          INDEX:INTEGER;
     begin
     for INDEX:=1 to MAXDATA do
          TO_DATA.D[INDEX]:=
            FROM_DATA.D[INDEX];
     TO_DATA.L:=FROM_DATA.L
     end;

procedure D_CREATE(
  var DATA:DATA_T;LENGTH:ID_T);
     begin
     D_NULL(DATA);
     DATA.L:=LENGTH
     end;

function D_GET(
  DATA:DATA_T;OFFSET:ID_T):OCTET;
     begin
     if OFFSET>DATA.L then
          D_GET:=0
     else D_GET:=DATA.D[OFFSET]
     end;

procedure D_PUT(
  var DATA:DATA_T;OFFSET:ID_T;
  VALUE:OCTET);
     begin
     if OFFSET<=DATA.L then
          DATA.D[OFFSET]:=VALUE
     end;

procedure D_ASSEMBLE(
  var BASE:DATA_T;
  var ADDITION:DATA_T);
     var
          TOT_LENGTH:INTEGER;
          INDEX:LEN_T;
     begin
     TOT_LENGTH:=
       BASE.L+ADDITION.L;
     if TOT_LENGTH>MAXDATA then
          TOT_LENGTH:=MAXDATA;
     for INDEX:=1 to
       TOT_LENGTH-BASE.L do
          BASE.D[INDEX+BASE.L]:=
            ADDITION.D[INDEX];
     BASE.L:=TOT_LENGTH;
     D_NULL(ADDITION)
     end;

procedure D_FRAGMENT(
  var HEAD:DATA_T;var OLD:DATA_T;
  LEN:LEN_T);
     var
          INDEX,LENGTH:LEN_T;
     begin
     if LEN>OLD.L then
```

```
        LENGTH:=OLD.L
      else LENGTH:=LEN;
      D_CREATE(HEAD,LENGTH);
      if LENGTH>0 then
           begin
           for INDEX:=1 to
             LENGTH do
                HEAD.D[INDEX]:=
                  OLD.D[INDEX];
           for INDEX:=1 to
             OLD.L-LENGTH do
               OLD.D[INDEX]:=
                 OLD.D[
                   INDEX+LENGTH];
           for INDEX:=
             OLD.L-LENGTH+
                to OLD.L do
                OLD.D[INDEX]:=0;
           OLD.L:=OLD.L-LENGTH
           end
      end;

procedure BUILD_TCR(SREF:REF_T;
  CALLING,CALLED:TADDRESS;
  MAX_BLOCK_SIZE:INTEGER;
  var NSDU:DATA_T);
     const
           TCR_CODE=224;
     type
           SIZE_BL=
             (L128,L256,L512,L1024,
               L2048);
           OPT_PAR_T=
             record
               case OPT:TLV_PAR of
                 P_CALLING,
                  P_CALLED:
                   (ADDR:TADDRESS);
                 MAX_BL:
                   (L:SIZE_BL);
                 CL_INFO:
                   (C:CLEAR_C);
                 REJ_PDU:
                   (ER:DATA_T)
              end;
     var
           LENGTH,LG1:INTEGER;
           OCT1,OCT2:OCTET;
           NSDU1:DATA_T;
           OPT_PAR_VAL:OPT_PAR_T;

     function FHS(CODE:INTEGER):
       INTEGER;
          begin
          end;

     procedure ENCODE_REF(
       REF:REF_T;
       var LOW,HIGH:OCTET);
          begin
          end;

function OPT_LENG(P:SIZE_BL):
  OCTET;
     type
          POWER_T=0..4;
     begin
     end;

function OPT_PAR_TYPE(
  OPT:TLV_PAR):OCTET;
     begin
     end;

function CODE_LENGTH(
  S:INTEGER):SIZE_BL;
     begin
     end;

procedure TLV_ENCODE_PAR(
  OP:TLV_PAR;var EOP:DATA_T;
  var L:INTEGER);
     var
          I:INTEGER;
     begin
     case OP of
       P_CALLING,P_CALLED:
         begin
           L:=
             OPT_PAR_VAL.
               ADDR.L
         end;
       MAX_BL:
         begin
           L:=1
         end
     end;
     D_CREATE(EOP,L+2);
     D_PUT(EOP,1,
       OPT_PAR_TYPE(OP));
     D_PUT(EOP,2,L);
     case OP of
       P_CALLING,P_CALLED:
         begin
         for I:=1 to L do
           D_PUT(EOP,2+I,
             ORD(OPT_PAR_VAL.
               ADDR.VAL[I]))
         end;
       MAX_BL:
         begin
         D_PUT(EOP,3,
           OPT_LENG(
             OPT_PAR_VAL.L))
         end
     end;
     L:=L+2
     end;

begin
LENGTH:=FHS(TCR_CODE);
D_CREATE(NSDU,LENGTH+1);
```

```
        ENCODE_REF(0,OCT1,OCT2);
        D_PUT(NSDU,3,OCT1);
        D_PUT(NSDU,4,OCT2);
        ENCODE_REF(SREF,OCT1,OCT2);
        D_PUT(NSDU,5,OCT1);
        D_PUT(NSDU,6,OCT2);
        D_PUT(NSDU,7,0);
        OPT_PAR_VAL.ADDR:=CALLING;
        OPT_PAR_VAL.ADDR:=CALLED;
        OPT_PAR_VAL.L:=
          CODE_LENGTH(
            MAX_BLOCK_SIZE);
        OPT_PAR_VAL.OPT:=P_CALLING;
        TLV_ENCODE_PAR(
          P_CALLING,NSDU1,LG1);
        LENGTH:=LENGTH+LG1;
        D_ASSEMBLE(NSDU,NSDU1);
        OPT_PAR_VAL.OPT:=P_CALLED;
        TLV_ENCODE_PAR(
          P_CALLED,NSDU1,LG1);
        LENGTH:=LENGTH+LG1;
        D_ASSEMBLE(NSDU,NSDU1);
        OPT_PAR_VAL.OPT:=MAX_BL;
        TLV_ENCODE_PAR(
          MAX_BL,NSDU1,LG1);
        LENGTH:=LENGTH+LG1;
        D_ASSEMBLE(NSDU,NSDU1);
        D_PUT(NSDU,1,LENGTH)
        end;

procedure BUILD_TCA(
  DREF,SREF:REF_T;
  CALLING,CALLED:TADDRESS;
  MAX_BLOCK_SIZE:INTEGER;
  var RES:PDU_T);
     external;

procedure BUILD_TCC(
  DREF:REF_T;var RES:PDU_T);
     external;

procedure BUILD_TDT(
  EOT:BOOLEAN;DATA:DATA_T;
  var RES:PDU_T);
     external;

procedure BUILD_TBR(
  DREF:REF_T;REASON:TBR_REASON;
  TPDU:DATA_T;
  var RES:PDU_T);
     external;

procedure TRANSFER(
  var REC_DATA:DATA_T;
  var REC_PDU:PDU_T);
     external;

initialize
     to IDLE
          begin
          TC_RELEASE_REQ:=FALSE;
          D_NULL(TSDU_SEND);
          D_NULL(TSDU_RCVD)
          end;

trans
     from WFNC
       to WFTCA
         when TC_IP.NCON_CONF
           var
               PRE_PDU:DATA_T;
           name TC2:
           begin
           BUILD_TCR(
             TC_SREF,TC_CALLING,
             TC_CALLED,
             TC_MXBL_S,PRE_PDU);
           TRANSFER(
             PRE_PDU,PDU_SEND);
           output TC_IP.TPDU(
             PDU_SEND)
           end;
       to PRE_RELEASE
         when TC_IP.NDIS_IND
           name TC3:
           begin
           output TCEP.TDIS_IND;
           end;
         when TCEP.TDIS_REQ
           name TC1:
           begin
           output TC_IP.NDIS_REQ;
           end;
     from WFTCA
       to PRE_RELEASE
         when TC_IP.NDIS_IND
           name TC5:
           begin
           output TCEP.TDIS_IND;
           end;
         when TCEP.TDIS_REQ
           name TC4:
           begin
           output TC_IP.NDIS_REQ;
           end;
     from WFTCA
       when TC_IP.TPDU
         provided PDU.CODE=TCC
           to PRE_RELEASE
             name TC8:
             begin
             output
               TC_IP.NDIS_REQ;
             output TCEP.TDIS_IND;
             end;
         provided PDU.CODE=TCA
           to OPEN
             name TC7:
             begin
             output
               TCEP.TCON_CONF;
             TC_DREF:=PDU.SREF1;
```

```
                TC_MXBL_S:=
                  PDU.MAX_BLOCK_SIZE
                end;
            provided PDU.CODE=TBR
              to PRE_RELEASE
                name TC10:
                begin
                output
                  TC_IP.NDIS_REQ;
                output TCEP.TDIS_IND;
                end;
            provided PDU.CODE=TDT
              to PRE_RELEASE
                name TC9:
                begin
                output
                  TC_IP.NDIS_REQ;
                output TCEP.TDIS_IND;
                end;
            provided PDU.CODE=TCR
              to ERROR
                name TC6:
                begin
                TC_DREF:=PDU.SREF1;
                D_NULL(ITPDU);
                output TCEP.TDIS_IND
                end;
        from WFTCA
          to PRE_RELEASE
            when TC_IP.INVALIDPDU
              name TC11:
              begin
              D_NULL(INVALID);
              output TCEP.TDIS_IND;
              output TC_IP.NDIS_REQ;
              end;

trans
        from WFTCA
          to PRE_RELEASE
            delay(TCR_TIMER)
              name TC12:
              begin
              output TCEP.TDIS_IND;
              output TC_IP.NDIS_REQ;
              end;

trans
        from WFTRESP
          to PRE_RELEASE
            when TC_IP.NDIS_IND
              name TC15:
              begin
              output TCEP.TDIS_IND;
              end;
            to WFND
              when TCEP.TDIS_REQ
                name TC14:
                begin
                BUILD_TCC(
                  TC_DREF,PDU_SEND);
                output
                  TC_IP.TPDU(PDU_SEND)
                end;
            to OPEN
              when TCEP.TCON_RESP
                name TC13:
                begin
                BUILD_TCA(
                  TC_DREF,TC_SREF,
                  TC_CALLING,TC_CALLED,
                  TC_MXBL_S,PDU_SEND);
                output
                  TC_IP.TPDU(PDU_SEND)
                end;
        from WFTRESP
          to ERROR
            when TC_IP.INVALIDPDU
              name TC21:
              begin
              ITPDU:=INVALID;
              output TCEP.TDIS_IND
              end;
        from WFTRESP
          when TC_IP.TPDU
            provided PDU.CODE<>TBR
              to ERROR
                name TC16_17_18_19:
                begin
                output TCEP.TDIS_IND;
                D_NULL(ITPDU)
                end;
            provided PDU.CODE=TBR
              to PRE_RELEASE
                name TC20:
                begin
                output TCEP.TDIS_IND;
                output
                  TC_IP.NDIS_REQ;
                end;
        from OPEN
          to PRE_RELEASE
            when TC_IP.NDIS_IND
              name TC24:
              begin
              output TCEP.TDIS_IND;
              end;
            when TCEP.TDIS_REQ
              name TC23:
              begin
              output TC_IP.NDIS_REQ;
              end;
          to OPEN
            when TCEP.TDT_REQ
              var
                  DATA:DATA_T;
              name TC22:
              begin
              TSDU_RCVD:=TSDU;
              while D_LENGTH(
                TSDU_RCVD)>
                  TC_MXBL_S-3 do
```

```
            begin
            D_FRAGMENT(
              DATA,TSDU_RCVD,
              TC_MXBL_S-3);
            BUILD_TDT(
              FALSE,DATA,
              PDU_SEND);
            output TC_IP.TPDU(
              PDU_SEND)
            end;
          BUILD_TDT(
            TRUE,TSDU_RCVD,
            PDU_SEND);
          output
            TC_IP.TPDU(PDU_SEND)
          end;
 from OPEN
   when TC_IP.TPDU
   provided(PDU.CODE=TDT) and
    not PDU.EOT
     name TC28_1:
     begin
     D_ASSEMBLE(
       TSDU_SEND,PDU.DATA)
     end;
   provided(PDU.CODE=TDT) and
    PDU.EOT
     name TC28_2:
     begin
     D_ASSEMBLE(
       TSDU_SEND,PDU.DATA);
     output TCEP.TDT_IND(
       TSDU_SEND);
     D_NULL(TSDU_SEND)
     end;
   provided(PDU.CODE<>TDT) and
    (PDU.CODE<>TBR)
     to ERROR
          name TC25_26_27:
          begin
          output TCEP.TDIS_IND;
          D_NULL(ITPDU)
          end;
    provided PDU.CODE=TBR
     to PRE_RELEASE
          name TC29:
          begin
          output TCEP.TDIS_IND;
          output
            TC_IP.NDIS_REQ;
          end;
from OPEN
  to ERROR
  when TC_IP.INVALIDPDU
    name TC30:
    begin
    ITPDU:=INVALID;
    output TCEP.TDIS_IND
    end;
from IDLE
  provided TC_DREF=0
     to WFNC
       begin
       end;
   provided TC_DREF<>0
     to WFTRESP
       begin
       output TCEP.TCON_IND(
         TC_CALLING,TC_CALLED)
       end;
from CLOSED
  to PRE_RELEASE
    when TC_IP.NDIS_IND
      name TC31:
      begin
      end;
from CLOSED
  when TC_IP.TPDU
    provided PDU.CODE=TCR
      to WFTRESP
         name TC32:
         begin
         TC_CALLING:=
           PDU.T_CALLING;
         TC_CALLED:=
           PDU.T_CALLED;
         TC_SREF:=PDU.SREF1;
         TC_MXBL_S:=
           PDU.MAX_BLOCK_SIZE;
         output TCEP.TCON_IND(
           TC_CALLING,
           TC_CALLED)
         end;
    provided PDU.CODE<>TCR
      to PRE_RELEASE
        name TC33_34_35_36:
        begin
        output
          TC_IP.NDIS_REQ;
        end;
  when TC_IP.INVALIDPDU
    name TC37:
    begin
    D_NULL(INVALID);
    output TC_IP.NDIS_REQ;
    end;
from WFND
  to PRE_RELEASE
    when TC_IP.NDIS_IND
      name TC38:
      begin
      end;
  to PRE_RELEASE
    delay(TBR_TIMER)
      name TC39:
      begin
      output TC_IP.NDIS_REQ;
      end;
  to WFND
    when TC_IP.TPDU
      name TC42:
      begin
```

```
                end;
            from ERROR
              to PRE_RELEASE
                name TC41:
                begin
                output TC_IP.NDIS_REQ;
                end;
              to WFND
                name TC40:
                begin
                BUILD_TBR(
                  TC_DREF,INVALID_PDU,
                  ITPDU,PDU_SEND);
                output
                  TC_IP.TPDU(PDU_SEND)
                end;
            from PRE_RELEASE
              name TC43:
              begin
              TC_RELEASE_REQ:=TRUE
              end;
end;

module NC_T activity(
  INITIATOR:BOOLEAN;
  NC_CALLING,NC_CALLED:NADDRESS);
     ip
          NCEP:NS_INTERFACE(USER);
          NC_IP:PDU_CHANNEL(NC);
     export
          TC_CREATE_REQ:REQUEST_T;
          TC_IMAGE:TC_IMAGE_T;
          NC_RELEASE_REQ:BOOLEAN;
end;

body NC_BODY for NC_T;
     state
          IDLE,CLOSED,WFNC,OPEN;
     var
          PDU_SEND,PDU_RCVD:PDU_T;

     procedure ENCODE(
       var D:DATA_T;R:PDU_T);
          begin
          end;

     procedure DECODE(
       D:DATA_T;var R:PDU_T);
          begin
          end;

     procedure BUILD_TCC(
       DREF:REF_T;var RES:PDU_T);
          external;

     initialize
          to IDLE
               begin
               NC_RELEASE_REQ:=FALSE;
               TC_CREATE_REQ:=UNDEFINED
               end;

trans
     from IDLE
       provided INITIATOR
         to WFNC
           name NC1:
           begin
           output NCEP.NCON_REQ(
             NC_CALLING,NC_CALLED)
           end;
       provided not INITIATOR
         to OPEN
           name NC2:
           begin
           end;
     from WFNC
       to OPEN
         when NCEP.NCON_CONF
           name NC3:
           begin
           output NC_IP.NCON_CONF
           end;
       to CLOSED
         when NCEP.NDIS_IND
           name NC4:
           begin
           output NC_IP.NDIS_IND;
           end;
       to CLOSED
         when NC_IP.NDIS_REQ
           name NC5:
           begin
           output NCEP.NDIS_REQ;
           end;
     from OPEN
       to CLOSED
         when NCEP.NDIS_IND
           name NC7:
           begin
           output NC_IP.NDIS_IND;
           end;
       to CLOSED
         when NCEP.NRST_IND
           name NC6:
           begin
           output NCEP.NRST_RESP;
           output NCEP.NDIS_REQ;
           output NC_IP.NDIS_IND;
           end;
       to OPEN
         when NC_IP.TPDU
           var
               DT:DATA_T;
           name NC14:
           begin
           ENCODE(DT,PDU);
           output NCEP.NDT_REQ(DT)
           end;
       to CLOSED
         when NC_IP.NDIS_REQ
           name NC13:
```

```
      begin
      output NCEP.NDIS_REQ;
      end;
from OPEN
  when NCEP.NDT_IND
    provided
     IS_VALID_TPDU(NSDU) and
      (TC_CREATE_REQ=CREATED)
      name NC10:
      begin
      DECODE(NSDU,PDU_RCVD);
      output
        NC_IP.TPDU(PDU_RCVD)
      end;
from OPEN
  when NCEP.NDT_IND
    provided
     IS_VALID_TPDU(NSDU) and
      (TC_CREATE_REQ=
       UNDEFINED) and
        (GET_CODE(NSDU)=TCR)
      name NC9:
      begin
      DECODE(NSDU,PDU_RCVD);
      with PDU_RCVD do
        begin
        TC_IMAGE.CALLING:=
          T_CALLING;
        TC_IMAGE.CALLED:=
          T_CALLED;
        TC_IMAGE.DREF:=SREF1;
        TC_IMAGE.
          MAX_BLOCK_SIZE:=
            MAX_BLOCK_SIZE
        end;
      TC_CREATE_REQ:=
        REQUESTED
      end;
    provided
     IS_VALID_TPDU(NSDU) and
      (TC_CREATE_REQ=
       UNDEFINED) and
        (GET_CODE(NSDU)<>TCR)
      to CLOSED
        name NC8:
        begin
        output NCEP.NDIS_REQ;
        end;
    provided
     not IS_VALID_TPDU(NSDU)
      and (TC_CREATE_REQ=
       CREATED)
      name NC11:
      begin
      output
        NC_IP.INVALIDPDU(
          NSDU)
      end;
    provided
     not IS_VALID_TPDU(NSDU)
      and (TC_CREATE_REQ=
       UNDEFINED)
      to CLOSED
        name NC12:
        begin
        output NCEP.NDIS_REQ;
        end;
    provided TC_CREATE_REQ=
     REJECTED
      priority MAXPRIORITY
        var
            DT:DATA_T;
        name NC15:
        begin
        BUILD_TCC(
         TC_IMAGE.DREF,
         PDU_SEND);
        ENCODE(DT,PDU_SEND);
        output
          NCEP.NDT_REQ(DT);
        TC_CREATE_REQ:=
          UNDEFINED
        end;
  from CLOSED
    name NC16:
    begin
    NC_RELEASE_REQ:=TRUE
    end;
end;

type
    USE_T=(OCCUPIED,FREE);
var
    TCEP_ID:array [1..MAXTCEP] of
      USE_T;
    NCEP_ID:array [1..MAXNCEP] of
      USE_T;
modvar
    TC:TC_T;
    NC:NC_T;

function NCACCEPT(FA,TA:NADDRESS):
  BOOLEAN;
    external;

function TESTUD(U:DATA_T):
  BOOLEAN;
    external;


initialize
    begin
    all I:1..MAXCEP do
        begin
        NCEP_ID[I]:=FREE;
        TCEP_ID[I]:=FREE
        end
    end;

trans
    any I:1..MAXCEP do
        when TCEP[I].TCON_REQ
```

```
        provided not TCON_REQ_OK(
          T_CALLING,T_CALLED)
          name PR1:
          begin
          output TCEP[I].TDIS_IND
          end;
        provided TCON_REQ_OK(
          T_CALLING,T_CALLED)
          name PR2:
          begin
          forone J:1..MAXNCEP
           suchthat NCEP_ID[J]=
            FREE do
            begin
            init NC with NC_BODY(
                TRUE,MAP_TADDRESS(
                  T_CALLING),
                MAP_TADDRESS(
                  T_CALLED));
            attach NCEP[J] to
              NC.NCEP;
            init TC with TC_BODY(
              T_CALLING,T_CALLED,I,
              0,SET_DTLENGTH(
                MAXDTLENGTH));
            connect TC.TC_IP to
              NC.NC_IP;
            attach TCEP[I] to
              TC.TCEP;
            NC.TC_CREATE_REQ:=
              CREATED;
            NCEP_ID[J]:=OCCUPIED;
            TCEP_ID[I]:=OCCUPIED
            end
          otherwise
            begin
            output TCEP[I].TDIS_IND
            end
          end;

trans
     any J:1..MAXNCEP do
          when NCEP[J].NCON_IND
           provided NCACCEPT(
            N_CALLING,N_CALLED)
            name PR3:
            begin
            NCEP_ID[J]:=OCCUPIED;
            output NCEP[J].NCON_RESP;
            init NC with NC_BODY(
              FALSE,N_CALLING,
              N_CALLED);
            attach NCEP[J] to NC.NCEP
            end;
          provided not NCACCEPT(
           N_CALLING,N_CALLED)
            name PR4:
            begin
            output NCEP[J].NDIS_REQ
            end;

trans
     any I:1..MAXTCEP do
          provided exist X:TC_T
           suchthat X.TC_RELEASE_REQ
            name PR5:
            begin
            forone X:TC_T suchthat
             X.TC_RELEASE_REQ do
              begin
              release X;
              TCEP_ID[I]:=FREE
              end
            end;

trans
     any I:1..MAXNCEP do
          provided exist X:NC_T
           suchthat X.NC_RELEASE_REQ
            name PR6:
            begin
            forone X:NC_T suchthat
             X.NC_RELEASE_REQ do
              begin
              NCEP_ID[I]:=FREE;
              release X
              end
            end;

trans
     provided exist X:NC_T suchthat(
      X.TC_CREATE_REQ=REQUESTED)
        name PR7:
        begin
        forone X:NC_T suchthat
         X.TC_CREATE_REQ=REQUESTED do
          begin
          forone I:1..MAXTCEP suchthat
           TCEP_ID[I]=FREE do
            begin
            init TC with TC_BODY(
              X.TC_IMAGE.CALLED,
              X.TC_IMAGE.CALLING,I,
              X.TC_IMAGE.DREF,
              SET_DTLENGTH(
                X.TC_IMAGE.
                MAX_BLOCK_SIZE));
            connect TC.TC_IP to
              NC.NC_IP;
            attach TCEP[I]to TC.TCEP;
            X.TC_CREATE_REQ:=CREATED;
            TCEP_ID[I]:=OCCUPIED
            end
            otherwise
              X.TC_CREATE_REQ:=REJECTED
            end
        end;

end;

module USER_T systemactivity;
     ip
```

```
            TCEP:array [1..MAXTCEP] of
                TS_INTERFACE(USER);
end;

body USER_BODY for USER_T;external;

module NETWORK_T systemactivity;
     ip
          NCEP:array
               [1..NUM_USER,1..MAXNCEP] of
                    NS_INTERFACE(PROVIDER);
end;

body NETWORK_BODY for NETWORK_T;external;

modvar
     USER:array [1..NUM_USER] of USER_T;
     PARENT:PARENT_T;
     NETWORK:NETWORK_T;

initialize
     begin
     init NETWORK with NETWORK_BODY;
     all I:1..NUM_USER do
          begin
          init USER[I] with USER_BODY;
          init PARENT with PARENT_BODY;
          all J:1..MAXTCEP do
               connect USER[I].TCEP[J] to
                    PARENT.TCEP[J];
          all J:1..MAXNCEP do
               connect PARENT.NCEP[J] to
                    NETWORK.NCEP[I,J]
          end
     end;

end.
```

11.3.4 Subjective Assessment

The Estelle description shows clear separation of the declaration part and the transition part. The data may be fully defined in a Pascal style or ignored to a first approximation (e.g. **any** INTEGER). Similarly, the behaviour of a module may either be fully described by the transition part or be postponed to a later stage (e.g. by using **external** for module bodies or **primitive** for functions). This allows the focus to be concentrated upon selected modules in the initial stages of description. Of course, when the description gradually evolves towards an implementation description, all the unspecified parts must be completed.

The Estelle description gives a good insight to the architecture being described in terms of modules and channels. The channel definition gives the list of interactions that it carries, and the **roles** show the direction in which these interactions are carried. The module header describes the view of a module from the outside. The module body describes the internal behaviour in terms of the transition system. The hierarchy of modules is straightforward: a parent module embodies a child module. This style allows the specifier to describe a system in Estelle along the lines of the informal Protocol description.

Although at first glance the description may appear to be in Pascal style, a closer examination reveals that there is a clear distinction between the Pascal constructs and the specific Estelle constructs. In the declaration part, there are Estelle features for data description (e.g. **module variable**, **interaction point**, and **state**). The transitions have a specific Estelle structure with a list of clauses and a transition block (**begin** ... **end**). There are also statements to handle the specific Estelle variables (e.g. the **init** and **release** operations on modules, the **attach** and **connect** operations for binding modules, and the **output** operation to send an interaction from one module).

The description should be regarded as only an example: some choices were made to illustrate selected features of Estelle. Other important features have not been illustrated, just because the Protocol is not suited to these.

11.4 LOTOS Description

```
(*-----------------------------------------------
```

11.4.1 Structure of the Formal Description

The description is designed on basis of two major requirements, which together provide the appropriate framework for the formal representation of the Transport Protocol architecture by means of the formal specification of a generic Transport Protocol Entity:

a) the specification is to be provably consistent with the formal description in LOTOS [ISO TR 10023] of the OSI Connection-Oriented OSI Transport Service [ISO 8072], assuming a correct formal description in LOTOS of the OSI Connection-oriented Network Service [ISO 8348]; and

b) the specification is to apply to any Transport Entity that implements the Protocol.

The main aspects of the specification are:

specification parameters	SAP addresses
global types	sorts of specification parameters, and imported interface data types from Service specifications
behaviour definition	constraint-oriented specification of dynamic conformance.

Already in the first decomposition of the constraints on the Protocol Entity behaviour, Figure 11.16, a remarkable fact arises: some of the components describe constraints that apply to, and depend upon, the behaviour of the Protocol Entity at only one of the two Service boundaries. These constraints will be referred to as Service constraints, whereas the term Protocol constraints will refer to those which are described by the other components. Notice that, in subsequent decomposition steps, the description of a Protocol constraint may reveal further Service constraints among its own components.

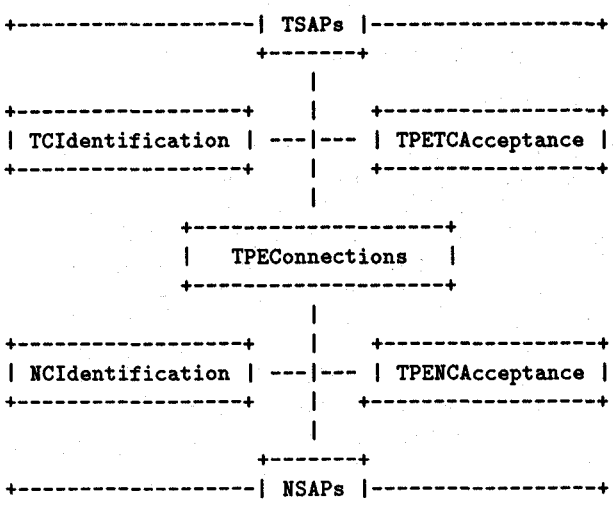


Figure 11.16: Constraint-Oriented Decomposition of a Transport Protocol Entity

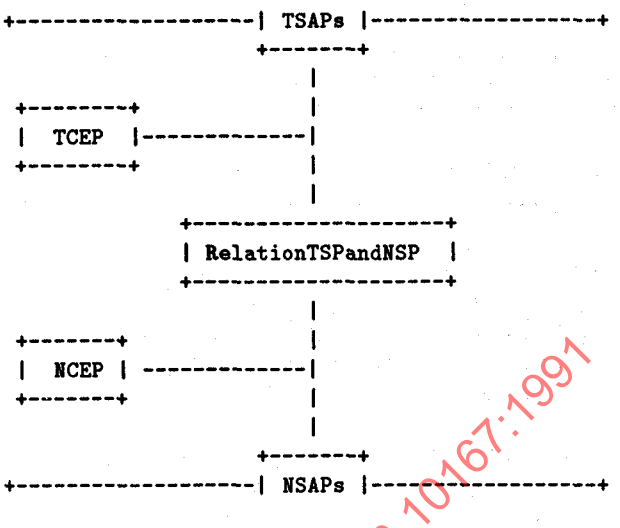


Figure 11.17: Decomposition of Process TPEConnection

The decomposition of the behaviour definition finds its most complex component in process **TPEConnections** which describes the constraints on provision of Transport Connections in relation to usage of Network Connections. Process **TPEConnections** comprises instances of the process **TPEConnection** combined in indefinite number by parallel interleaving. The decomposition of process **TPEConnection** is illustrated in Figure 11.17. It enables a further separation of concerns between:

a) Service constraints relating to a Transport Connection Endpoint considered in isolation; and

b) Service constraints relating to a Network Connection Endpoint considered in isolation; and

c) Protocol constraints that describe the required relation between Transport Service Primitives (**TSPs**) and Network Service Primitives (**NSPs**).

11.4.2 Explanation of Approach

This specification is in a 'constraint-oriented' style. It applies to all valid implementations of the Protocol, as it consists of the formal description of a generic Protocol Entity that can:

a) access any given sets of Transport and Network Service access points; and

b) provide the Transport user with any number of Transport Connections; and

c) make use of any number of Network Connections.

The specification provides the reader with an abstract model of the Protocol behaviour requirements, i.e. those requirements that apply to any instance of communication shown by implementations of the specified Protocol Entity.

This description is sufficiently complete, although a few improvements could still be made. These are indicated by footnotes

11.4.2.1 Service Interactions

Service interactions are described as events at the **t** and **n** gates. These events are of the following form:

t ?ta : TAddress ?tcei : TCEI ?tsp : TSP

and:

n ?na : NAddress ?ncei : NCEI ?nsp : NSP.

11.4.2.2 Block Transfer

Blocks are transferred at the **n** gate in **N-DATA** NSPs on the Network Connection to which the Transport Connection is assigned. There would be no need to use any Connection identification mechanism other than Connection Endpoint identification since neither multiplexing, nor splitting, nor reassignment are supported. Hence every Transport Connection is uniquely related to one Network Connection. Connection identification by References is formally described, however, according to the resolution in 11.2.3.2.

11.4.2.3 Assignment of Transport Connections

The assignment of a Transport Connection to a Network Connection depends upon the rôle of the Entity with regard to the Transport Connection. If the Entity is the initiator of the Transport Connection, it creates a Network Connection to which the Transport Connection is assigned. In this case the assignment ensures that blocks of that Transport Connection are sent on the Network Connection to which the Transport Connection is assigned. If the Entity is the responder of the Transport Connection, the Transport Connection is assigned to the Network Connection on which the first block associated with this Transport Connection is received.

11.4.2.4 Association of Blocks

All blocks received on a given Network Connection are associated with the Transport Connection which is assigned

to it, if any. Receipt of a block that performs assignment generates the creation of a new Transport Connection, with which the block is associated.

11.4.2.5 Negotiation of Class and Options

The only negotiations that take place are:

a) **Class**: The Entity accepts any **TCR** block but a Transport Connection may only be created if Class 0 is a proposed Class. The Entity may only send **TCR** blocks with Class 0 as preferred Class and no alternative Class.

b) **Block Size**: The Entity negotiates the maximum block size according to the negotiation rules defined in the Protocol.

c) **Extended Addressing**: The Entity may negotiate the use of extended addressing.

11.4.2.6 Segmenting and Reassembling

Segmenting and reassembling are described using the data type **TSDUS**, where a value of sort **TSDUS** is a queue of octet strings. Each TSDU carried by a **TDATA request** is added to the queue **TSDUSdown** as one new element (**OctetString**). Blocks are formed from the oldest element of **TSDUSdown**. When this block contains the End-Of-TSDU indication, then it contains all remaining octets (possibly none) of the oldest element, and this element is removed from **TSDUSdown**. Otherwise only the octets contained in the block are removed from the oldest element of **TSDUSdown**. The data octets of a received TDT block are added to the newest element of the queue **TSDUSup**. If the block contains the End-Of-TSDU indication then the newest element is considered to be a complete TSDU for a **T-DATA indication**, and a new and empty newest element is added to **TSDUSup**. A **T-DATA indication** containing data is generated only if **TSDUSup** deprived of its newest element, contains at least one possibly empty element.

11.4.2.7 Actual Block Transfer

Blocks are transferred by means of **N-DATA** NSPs. Functions are provided that specify the encoding of blocks as octet strings, which are the NSDU parameter of **N-DATA** NSPs. A completely general formulation of the encoding of (abstract) block structures as octet strings is presented. This formulation includes boolean functions on octet strings that evaluate to **true** if, and only if, the octet string is a valid encoding of a (possibly given) abstract block.

11.4.2.8 Handling of Protocol Errors

Protocol errors are treated according to the informal description, together with resolution in 11.2.5.2. Whenever an error is detected, a **TBR** block is sent and the Transport and Network Connection are disconnected. Data types determine syntactically invalid blocks and processes determine ordering errors in block sequences are defined.

11.4.3 Formal Description

11.4.3.1 General

The **t (n)** gate represents the Transport (Network) Service boundary accessed by the Entity. It models the totality of TSAPs (NSAPs) at which the Entity interacts with the TS user (NS provider). Each TSAP (NSAP) is uniquely identified by a TSAP address (NSAP address) out of the set **tas (nas)**. Proper cooperation between Session and Transport entities (or Transport and Network entities), within one open system, is ensured by assigning the same **tas** to the Session and Transport entities (and the same **nas** to the Transport and Network entities) residing in the same Open System. This implies that a Transport Entity cannot cooperate with a Session Entity if they reside in different open systems.

```
-----------------------------------------------------*)

specification TEntity[t, n]
 (tas : TAddresses, nas : NAddresses) : noexit

library
  Boolean, Element, Set, String, NaturalNumber,
  NatRepresentations, Bit, Octet, OctetString
endlib

(*-----------------------------------------------
```

11.4.3.2 Service Data Types

This clause defines the specification parameters types. See 11.4.1.

Transport Address: No Transport address structure is specified in the Transport Service standard. The following definition allows representation an infinite number of Transport addresses. See 11.4.3.2 for the definition of **GeneralIdentifier**.

```
-----------------------------------------------*)

type TAddress
is GeneralIdentifier renamedby
sortnames
  TAddress for Identifier
opnnames
  SomeTAddress    for SomeIdentifier
  AnotherTAddress for AnotherIdentifier
endtype (* TAddress *)

(*-----------------------------------------------
```

Transport Address Set: By the following definitions, any value of sort **TAddresses** is a finite set of Transport Addresses.

```
-----------------------------------------------*)

type TAddresses
is Set actualizedby TAddress using
```

```
sortnames
  TAddress   for Element
  TAddresses for Set
  Bool       for FBool
endtype (* TAddresses *)

(*-----------------------------------------------
```

The definition of **Network Address** and **Network Address Set** are similar to those of **Transport Address** and **Transport Address Set**.

```
-----------------------------------------------*)

type NetworkAddress
is GeneralIdentifier renamedby
sortnames
  NAddress for Identifier
opnnames
  SomeNAddress    for SomeIdentifier
  AnotherNAddress for AnotherIdentifier
endtype (* NetworkAddress *)

type NAddresses
is Set actualizedby NetworkAddress using
sortnames
  NAddress   for Element
  Bool       for FBool
  NAddresses for Set
endtype (* NAddresses *)

(*-----------------------------------------------
```

Transport Service Primitive: The **TSP** data type is presented starting with the basic construction of values of sort **TSP**. This construction is a direct formulation of the definition given in Table 3 of the TS standard [ISO 8072], simplified in accordance with the resolution in 11.2.1.2. The functions that yield values of sort **TSP** are referred to as 'TSP constructors'. This definition imports the definitions that relate to TSP parameters.

```
-----------------------------------------------*)

type BasicTSP
is TAddress, OctetString
sorts
  TSP
opns
  TCONreq, TCONind   : TAddress, TAddress -> TSP
  TCONresp, TCONconf : -> TSP
  TDTreq, TDTind     : OctetString -> TSP
  TDISreq, TDISind   : -> TSP
endtype (* BasicTSP *)

(*-----------------------------------------------
```

Transport Service Primitive Classification: A classification of TSPs is defined, that enables to enrich the basic construction with further functions in a simple way.

This classification is based on the type **TSPSubsort**, which consists of a set of constants, each denoting a TSP name in correspondence with Table 3 of the TS standard.

The type **TSPClassifiers** merges the previous constructions and introduces the following functions on TSPs:

a) **Subsort**, that yields the TSP name; and

b) *boolean functions, termed* 'TSP (subsort) Classifiers', defined according to the Classification introduced by **TSPSubsort**.

NOTE — The auxiliary function **h** that maps TSP names to natural numbers is defined in order to simplify the specification of the boolean operations of equality on TSP names (as well as on TSPs). **RicherNaturalNumber** is an extension of NaturalNumber with the **Odd** and **Even** operations. See 11.4.3.3.

```
-----------------------------------------------*)

type TSPSubsort
is RicherNaturalNumber
sorts
  TSPSubsort
opns
  TCONNECTrequest, TCONNECTindication,
  TCONNECTresponse, TCONNECTconfirm,
  TDATArequest, TDATAindication,
  TDISCONNrequest, TDISCONNindication :
    -> TSPSubsort
  h : TSPSubsort -> Nat
  IsRequest, IsIndication : TSPSubsort -> Bool
  _eq_, _ne_ : TSPSubsort, TSPSubsort -> Bool
eqns
forall
  s, s1 : TSPSubsort, n : Nat
ofsort Nat
  h(TCONNECTrequest) = 0;
  h(TCONNECTindication) =
    Succ(h(TCONNECTrequest));
  h(TCONNECTresponse) =
    Succ(h(TCONNECTindication));
  h(TCONNECTconfirm) = Succ(h(TCONNECTresponse));
  h(TDATArequest)    = Succ(h(TCONNECTconfirm));
  h(TDATAindication) = Succ(h(TDATArequest));
  h(TDISCONNrequest) = Succ(h(TDATAindication));
  h(TDISCONNindication) =
    Succ(h(TDISCONNrequest))
ofsort Bool
  IsRequest(s)    = Even(h(s));
  IsIndication(s) = Odd(h(s));
  s eq s1 = h(s) eq h(s1);
  s ne s1 = not(s eq s1)
endtype (* TSPSubsort *)

type TSPClassifiers
is BasicTSP, TSPSubsort
opns
  IsTCON, IsTCON1, IsTCON2, IsTDT, IsTDIS,
  IsTCONreq, IsTCONind, IsTCONresp, IsTCONconf,
  IsTDTreq, IsTDTind,
```

```
  IsTDISreq, IsTDISind, IsTReq, IsTind :
    TSP -> Bool
  Subsort : TSP -> TSPSubsort
eqns
forall
  a, a1 : TAddress, d : OctetString, t : TSP
ofsort TSPSubsort
  Subsort(TCONreq(a, a1)) = TCONNECTrequest;
  Subsort(TCONind(a, a1)) = TCONNECTindication;
  Subsort(TCONresp)       = TCONNECTresponse;
  Subsort(TCONconf)       = TCONNECTconfirm;
  Subsort(TDTreq(d))      = TDATArequest;
  Subsort(TDTind(d))      = TDATAindication;
  Subsort(TDISreq)        = TDISCONNrequest;
  Subsort(TDISind)        = TDISCONNindication
ofsort Bool
  IsTCON(t)     = IsTCON1(t) or IsTCON2(t);
  IsTCON1(t)    = IsTCONreq(t) or IsTCONind(t);
  IsTCON2(t)    = IsTCONresp(t) or IsTCONconf(t);
  IsTDT(t)      = IsTDTreq(t) or IsTDTind(t);
  IsTDIS(t)     = IsTDISreq(t) or IsTDISind(t);
  IsTCONreq(t)  = Subsort(t) eq TCONNECTrequest;
  IsTCONind(t)  =
    Subsort(t) eq TCONNECTindication;
  IsTCONresp(t) = Subsort(t) eq TCONNECTresponse;
  IsTCONconf(t) = Subsort(t) eq TCONNECTconfirm;
  IsTDTreq(t)   = Subsort(t) eq TDATArequest;
  IsTDTind(t)   = Subsort(t) eq TDATAindication;
  IsTDISreq(t)  = Subsort(t) eq TDISCONNrequest;
  IsTDISind(t)  =
    Subsort(t) eq TDISCONNindication;
  IsTReq(t)     = IsRequest(Subsort(t));
  IsTind(t)     = IsIndication(Subsort(t))
endtype (* TSPClassifiers *)

(*-----------------------------------------
```

Transport Service Primitive Selectors: The construction of Transport Service Primitives presented above is enriched with functions that allow to determine the value of individual parameters of TSPs. The Address parameter selectors are defined as boolean functions. The reason for this indirect representation is the completeness of the equational definition.

```
-----------------------------------------*)

type TSPParameterSelectors
is TSPClassifiers
opns
  _IsCallingOf_, _IsCalledOf_ :
    TAddress, TSP -> Bool
  Userdata : TSP -> OctetString
eqns
forall
  a, a1, a2 : TAddress, d : OctetString, t : TSP
ofsort Bool
  a IsCallingOf TCONreq(a1, a2)       = a eq a1;
  a IsCallingOf TCONind(a1, a2)       = a eq a1;
  not(IsTCON1(t)) => a IsCallingOf t  = false;
  a IsCalledOf TCONreq(a1, a2)        = a eq a2;
  a IsCalledOf TCONind(a1, a2)        = a eq a2;
  not(IsTCON1(t)) => a IsCalledOf t   = false
ofsort OctetString
  Userdata(TCONreq(a1, a2)) = <>;
  Userdata(TCONind(a1, a2)) = <>;
  Userdata(TCONresp)        = <>;
  Userdata(TCONconf)        = <>;
  Userdata(TDTreq(d))       = d;
  Userdata(TDTind(d))       = d;
  Userdata(TDISreq)         = <>;
  Userdata(TDISind)         = <>
endtype (* TSPParameterSelectors *)

(*-----------------------------------------
```

Transport Service Primitive Equality: Boolean equality on TSPs is defined as the conjunction of:

a) TSP name equality; and

b) equality of TSP parameters.

```
-----------------------------------------*)

type TSPEquality
is TSPParameterSelectors
opns
  _eq_, _ne_ : TSP, TSP -> Bool
eqns
forall
  a1, a2, a3, a4 : TAddress, t1, t2 : TSP,
  d1, d2 : OctetString
ofsort Bool
  TCONreq(a1, a2) eq TCONreq(a3, a4) =
    (a1 eq a3) and (a2 eq a4);
  TCONind(a1, a2) eq TCONind(a3, a4) =
    (a1 eq a3) and (a2 eq a4);
  TDTreq(d1) eq TDTreq(d2) = d1 eq d2;
  TDTind(d1) eq TDTind(d2) = d1 eq d2;

  not(IsTCON1(t1) or IsTDT(t1) or IsTCON1(t2) or
   IsTDT(t2)) =>
    t1 eq t2 = Subsort(t1) eq Subsort(t2);
  t1 ne t2   = not(t1 eq t2)
endtype (* TSPEquality *)

(*-----------------------------------------
```

Transport Service Primitives Miscellaneous: **IsValidTCON2For** represents TS requirements that apply locally to each TC Endpoint.

```
-----------------------------------------*)

type TransportServicePrimitive
is TSPEquality
opns
  _IsValidTCON2For_ : TSP, TSP -> Bool
eqns
forall
  t1, t2 : TSP
```

```
ofsort Bool
  t2 IsValidTCON2For t1 =
    IsTCONconf(t2) and IsTCONreq(t1) or
      (IsTCONresp(t2) and IsTCONind(t1))
endtype (* TransportServicePrimitive *)

(*-----------------------------------------------
```

Network Service Primitives: The specification of Network Service Primitives is very similar to the definition of Transport Service Primitives above. The complete Network Service is not specified, only the parts used in this description.

```
-----------------------------------------------*)

type BasicNSP
is NetworkAddress, OctetString
sorts
  NSP
opns
  NCONreq, NCONind : NAddress, NAddress -> NSP
  NDTreq, NDTind : OctetString -> NSP
  NCONresp, NCONconf, NDISreq, NDISind,
  NRSTind, NRSTresp : -> NSP
endtype (* BasicNSP *)

type NSPSubsort
is RicherNaturalNumber
sorts
  NSPSubsort
opns
  NCONNECTrequest, NCONNECTindication,
  NCONNECTresponse, NCONNECTconfirm,
  NDATArequest, NDATAindication,
  NDISCONNrequest, NDISCONNindication,
  NRESETindication, NRESETresponse :
    -> NSPSubsort
  h : NSPSubsort -> Nat
  IsRequest, IsIndication : NSPSubsort -> Bool
  _eq_, _ne_ : NSPSubsort, NSPSubsort  -> Bool
eqns
forall
  s, s1 : NSPSubsort, n : Nat
ofsort Nat
  h(NCONNECTrequest)     = 0;
  h(NCONNECTindication) =
    Succ(h(NCONNECTrequest));
  h(NCONNECTresponse)    =
    Succ(h(NCONNECTindication));
  h(NCONNECTconfirm)     =
    Succ(h(NCONNECTresponse));
  h(NDATArequest)        =
    Succ(h(NCONNECTconfirm));
  h(NDATAindication)     =
    Succ(h(NDATArequest));
  h(NDISCONNrequest)     =
    Succ(h(NDATAindication));
  h(NDISCONNindication) =
    Succ(h(NDISCONNrequest));
  h(NRESETindication)    =
    Succ(Succ(h(NDISCONNindication)));
  h(NRESETresponse)      =
    Succ(h(NRESETindication));
ofsort Bool
  IsRequest(s)    = Even(h(s));
  IsIndication(s) = Odd(h(s));
  s eq s1 = h(s) eq h(s1);
  s ne s1 = not(s eq s1)
endtype (* NSPSubsort *)

type NSPClassifiers
is BasicNSP, NSPSubsort
opns
  Subsort : NSP -> NSPSubsort
  IsNCON, IsNCON1, IsNCON2,
  IsNDT, IsNDIS, IsNRST,
  IsNCONreq, IsNCONind, IsNCONresp, IsNCONconf,
  IsNDTreq, IsNDTind, IsNDISreq, IsNDISind,
  IsNRSTind, IsNRSTresp, IsNReq, IsNInd :
    NSP -> Bool
eqns
forall
  a, a1 : NAddress, d : OctetString, n : NSP
ofsort NSPSubsort
  Subsort(NCONreq(a, a1)) = NCONNECTrequest;
  Subsort(NCONind(a, a1)) = NCONNECTindication;
  Subsort(NCONresp)       = NCONNECTresponse;
  Subsort(NCONconf)       = NCONNECTconfirm;
  Subsort(NDTreq(d))      = NDATArequest;
  Subsort(NDTind(d))      = NDATAindication;
  Subsort(NDISreq)        = NDISCONNrequest;
  Subsort(NDISind)        = NDISCONNindication;
  Subsort(NRSTind)        = NRESETindication;
  Subsort(NRSTresp)       = NRESETresponse
ofsort Bool
  IsNCON(n)     = IsNCON1(n)    or IsNCON2(n);
  ISNRST(n)     = IsNRSTind(n)  or IsNRSTresp(n);
  IsNCON1(n)    = IsNCONreq(n)  or IsNCONind(n);
  IsNCON2(n)    = IsNCONresp(n) or IsNCONconf(n);
  IsNDT(n)      = IsNDTreq(n)   or IsNDTind(n);
  IsNDIS(n)     = IsNDISreq(n)  or IsNDISind(n);
  IsNCONreq(n)  = Subsort(n) eq NCONNECTrequest;
  IsNCONind(n)  =
    Subsort(n) eq NCONNECTindication;
  IsNCONresp(n) = Subsort(n) eq NCONNECTresponse;
  IsNCONconf(n) = Subsort(n) eq NCONNECTconfirm;
  IsNDTreq(n)   = Subsort(n) eq NDATArequest;
  IsNDTind(n)   = Subsort(n) eq NDATAindication;
  IsNDISreq(n)  = Subsort(n) eq NDISCONNrequest;
  IsNDISind(n)  =
    Subsort(n) eq NDISCONNindication;
  IsNRSTind(n)  = Subsort(n) eq NRESETindication;
  IsNRSTresp(n) = Subsort(n) eq NRESETresponse;
  IsNReq(n)     = IsRequest(Subsort(n));
  IsNInd(n)     = IsIndication(Subsort(n))
endtype (* NSPClassifiers *)

type NSPParameterSelectors
is NSPClassifiers
opns
  _IsCallingOf_, _IsCalledOf_ :
    NAddress, NSP -> Bool
```

```
  Userdata : NSP -> OctetString
eqns
forall
  a, a1, a2 : NAddress, d : OctetString, n : NSP
ofsort Bool
  a IsCallingOf NCONreq(a1, a2)       = a eq a1;
  a IsCallingOf NCONind(a1, a2)       = a eq a1;
  not(IsNCON1(n)) => a IsCallingOf n  = false;
  a IsCalledOf NCONreq(a1, a2)        = a eq a2;
  a IsCalledOf NCONind(a1, a2)        = a eq a2;
  not(IsNCON1(n)) => a IsCalledOf n   = false
ofsort OctetString
  Userdata(NCONreq(a1, a2)) = <>;
  Userdata(NCONind(a1, a2)) = <>;
  Userdata(NCONresp)        = <>;
  Userdata(NCONconf)        = <>;
  Userdata(NDTreq(d))       = d;
  Userdata(NDTind(d))       = d;
  Userdata(NDISreq)         = <>;
  Userdata(NDISind)         = <>;
  Userdata(NRSTind)         = <>;
  Userdata(NRSTresp)        = <>
endtype (* NSPParameterSelectors *)

type NSPEquality
is NSPParameterSelectors
opns
  _eq_, _ne_ : NSP, NSP -> Bool
eqns
forall
  a1, a2, a3, a4 : NAddress,
  d1, d2 : OctetString, n1, n2 : NSP
ofsort Bool
  NCONreq(a1, a2) eq NCONreq(a3, a4) =
    (a1 eq a3) and (a2 eq a4);
  NCONind(a1, a2) eq NCONind(a3, a4) =
    (a1 eq a3) and (a2 eq a4);
  NDTreq(d1) eq NDTreq(d2)          = d1 eq d2;
  not(IsNCON1(n1) or IsNDT(n1) or IsNCON1(n2) or
   IsNDT(n2)) =>
   n1 eq n2 = Subsort(n1) eq Subsort(n2);
  n1 ne n2  = not(n1 eq n2)
endtype (* NSPEquality *)

type NetworkServicePrimitive
is NSPEquality
opns
  _IsValidNCON2For_ : NSP, NSP -> Bool
eqns
forall
  n1, n2 : NSP
ofsort Bool
  n2 IsValidNCON2For n1 =
    IsNCONconf(n2) and IsNCONreq(n1) or
      (IsNCONresp(n2) and IsNCONind(n1))
endtype (* NetworkServicePrimitive *)

(*-----------------------------------------------
```

11.4.3.3 Auxiliary Service definitions

The following definitions are used in the definitions above. The **RicherNaturalNumber** is an extension of the NaturalNumber with **Odd** and **Even** functions. A **GeneralIdentifier** specifies an infinite number of identifiers[1].

```
-----------------------------------------------*)

type RicherNaturalNumber
is NaturalNumber
opns
  Odd, Even : Nat -> Bool
eqns
forall
  n : Nat
ofsort Bool
  Even(0)       = true;
  Even(Succ(n)) = not(Even(n));
  Odd(n)        = not(Even(n))
endtype (* RicherNaturalNumber *)

type GeneralIdentifier
is Boolean
sorts
  Identifier
opns
  SomeIdentifier : -> Identifier
  AnotherIdentifier : Identifier -> Identifier
  _eq_, _ne_, _lt_ :
    Identifier, Identifier -> Bool
eqns
forall
  a, a1 : Identifier
ofsort Bool
  SomeIdentifier eq SomeIdentifier =
    true;
  SomeIdentifier eq AnotherIdentifier(a) =
    false;
  AnotherIdentifier(a) eq SomeIdentifier =
    false;
  AnotherIdentifier(a) eq AnotherIdentifier(a1) =
    a eq a1;
  a ne a1 = not(a eq a1);
endtype (* GeneralIdentifier *)

(*-----------------------------------------------
```

11.4.3.4 Global Behaviour

The process presented below describes the relationship between provision of TCs and usage of NCs (see TPEConnections), subject to Service constraints at both Service boundaries. This process formally describes the structure shown in Figure 11.16.

The Service constraints ensure, for instance, that the Address component of an interaction at **t** (**n**) is a member of the set **tas** (**nas**), that the identification of a Connection by means of a Connection Endpoint Identifier is unique within

[1] In this type, _lt_ has been included to ease checking with tools.

the scope of any given Address, that the Entity is ready to accept and support at least one Connection, and so on.

The TS constraints are expressed by processes **TCIdentification** and **TPETCAcceptance**: the former is imported from the TS formal description, the latter is very similar to the **TCAcceptance** process of the TS formal description, but for the presence of a bound on the set of Transport Addresses where TCs may be accepted. (This bound is necessary here to represent the fact that the Transport Entity is confined within an end-system.) The NS constraints are similarly expressed by processes **NCIdentification** and **TPENCAcceptance**.

```
-----------------------------------------------*)

behaviour
  TPEConnections [t, n]
||
  GlobalConstraints [t, n]
||
  (
    (
      TCIdentification [t]
    ||
      TPETCAcceptance [t] (tas)
    )
  |||
    (
      NCIdentification [n]
    ||
      TPENCAcceptance [n] (nas)
    )
  )

where

(*-----------------------------------------
```

11.4.3.5 Service Constraints

Connection Identification Data Types: No structure of Connection Endpoint Identifiers is defined by the TS and NS standards. The value **SomeTCEI** is may also be understood as a 'null' value in the case that a Transport Connection Endpoint Identifier is not required.

TCEndpointIdentifier specifies an *infinite number of* TC Endpoint Identifiers to be represented. **TCEIdentification** presents TC Endpoint Identifiers that are global to the whole TS boundary: each of them is a pair <TAddress, TCEI>. See the definition of **Pair** in 11.4.3.9, and the definition of **GeneralIdentifier** in 11.4.3.2. **TCIdentifications** presents finite sets of global TC Endpoint Identifiers.

```
-----------------------------------------------*)

type TCEndpointIdentifier
is GeneralIdentifier renamedby
sortnames
  TCEI for Identifier
opnnames
  SomeTCEI    for SomeIdentifier
  AnotherTCEI for AnotherIdentifier
endtype (* TCEndpointIdentifier *)

type TCEIdentification
is Pair actualizedby TAddress,
  TCEndpointIdentifier using
sortnames
  TAddress for Element
  TCEI     for OtherElement
  Bool     for FBool
  TId      for Pair
opnnames
  TId  for Pair
  TA   for First
  TCEI for Second
endtype (* TCEIdentification *)

type TCEIdentifications
is Set actualizedby TCEIdentification using
sortnames
  TId  for Element
  Bool for FBool
  TIds for Set
endtype (* TCEIdentifications *)

(*-----------------------------------------
```

The definition of **NCEndpointIdentifiers** is very similar to that of **TC EndpointIdentifiers**.

```
-----------------------------------------------*)

type NCEndpointIdentifier
is GeneralIdentifier renamedby
sortnames
  NCEI for Identifier
opnnames
  SomeNCEI    for SomeIdentifier
  AnotherNCEI for AnotherIdentifier
endtype (* NCEndpointIdentifier *)

type NCEIdentification
is Pair actualizedby NetworkAddress,
  NCEndpointIdentifier using
sortnames
  NAddress for Element
  NCEI     for OtherElement
  Bool     for FBool
  NId      for Pair
opnnames
  NId  for Pair
  NA   for First
  NCEI for Second
endtype (* NCEIdentification *)

type NCEIdentifications
is Set actualizedby NCEIdentification using
sortnames
  NId  for Element
  Bool for FBool
```

```
  NIds for Set
endtype (* NCEIdentifications *)

(*-----------------------------------------
```

Connection Identification Processes: Processes **TCIdentification** and **NCIdentification** prescribe that at no Transport or Network address may any Endpoint Identifier be assigned to more than one Connection at any given time.

Any two distinct instances of **TConnection** that concurrently access the same TSAP are distinguishable by the TS user. This is achieved by means of the TC Endpoint Identifier (TCEI), which is passed together with every Service Primitive at every TSAP. It is to be required, therefore, that:

a) at any given TSAP, no TCEI may be assigned to more than one TConnection at any given time; and

b) at each TSAP, every **TConnection** employs the same TCEI for the whole lifetime of the TC it represents.

While the latter constraint can be specified within the definition of **TConnection**, the former constraint has a more global scope, and is represented by process **TCIdentification** as follows.

Track is kept of the TCEP identifiers in use, for each TSAP, by means of the parameter **Use**, which is a finite set of pairs of sort **TId** = <TAddress, TCEI>. (The type definition **TCEIdentifications** above describes such sets.)

Use is initially empty. A pair <**ta**, **tcei**> is to be in **Use** if, and only if, **tcei** is assigned to some TC that accesses the TSAP having address **ta**.

Process **TCIdent** allows any T-CONNECT request or indication to be passed at any given TSAP of address **ta** only with such **tcei** that the pair <**ta**, **tcei**> is not in **Use**. No other TSP is constrained but, upon execution of a T-Disconnect Primitive, the associated <**ta**, **tcei**> is removed from **Use**.

NC identification at the Network Service boundary is very similar.

NOTE — The following technical detail is to be taken into account: $Insert(e, s) = \{e\} \cup s$. Therefore $Insert(e, s) = s$ whenever $e \in s$.

```
-----------------------------------------*)

process TCIdentification[t] : noexit:=
TCIdent [t] ({} of TIds)
endproc (* TCIdentification *)

process TCIdent[t](Use : TIds) : noexit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTCON1(tsp) implies
    (TId(ta, tcei) NotIn Use)];
(let  ti : TId = TId(ta, tcei)
   in
     [not(IsTDIS(tsp))] ->
       TCIdent [t] (Insert(ti, Use))
   []
     [IsTDIS(tsp)] ->
       TCIdent [t] (Remove(ti, Use)))
endproc (* TCIdent *)

process NCIdentification[n] : noexit:=
NCIdent [n] ({} of NIds)
endproc (* NCIdentification *)

process NCIdent[n](Use : NIds) : noexit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNCON1(nsp) implies
    (NId(na, ncei) NotIn Use)];
(let  ni : NId = NId(na, ncei)
   in
     [not(IsNDIS(nsp))] ->
       NCIdent [n] (Insert(ni, Use))
   []
     [IsNDIS(nsp)] ->
       NCIdent [n] (Remove(ni, Use)))
endproc (* NCIdent *)

(*-----------------------------------------
```

Connection Acceptance: At any time the Protocol Entity is allowed to accept establishment of new Connections at an only finite set of Connection Endpoints. At the TS boundary, this is described by process **TPETCAcceptance**, which internally chooses a finite set **tias** of <**ta**, **tcei**> pairs before engaging in any interaction. If the interaction starts a new TC, the Endpoint where the interaction occurs must be among those represented by **tias**. Clearly, the addresses of elements of **tias** must be accessible by the Entity. This is specified by using the function **Addresses**, defined in the data type **TPETCAcceptance**, that when applied to a set **tids** of global TCEIs returns the set of addresses that are addresses of elements of **tids**.

Upon each choice of **tias**, however, the set of Endpoints where new Connections can be started is actually a subset of **tias**, because of the presence of a separate constraint on TC identification. Precisely, a new Connection can only be started with a pair <**ta**, **tcei**> that is in **tias** but not in **Use**. See process **TCIdentification** above. Upon each choice of **tias**, therefore, the set of Endpoints where new Connections can be started is represented by the difference **tias - Use**.

The Protocol Entity is allowed internal non-determinism in the dynamic choice of which and how many Endpoints may be allocated to new Connections, provided the following minimal functionality requirement is met: if no TC is active, the Entity must be able to accept at least one TC, i.e. the subset of **tias** where new TCs can be actually accepted must be non-empty in this case.

Similarly, the following constraints are described by process **TPENCAcceptance**:

a) NS Primitives may only be exchanged at NS addresses accessed by the Entity; and

b) at all times at least one NC Endpoint is provided; this may allow an **N-CONNECT request** or indication to occur.

The **TPENCAcceptance** definitions are very similar.

NOTE — In fact, the minimal functionality requirement is equivalent to the simpler requirement that **tias** be non-empty in any case, if the constraint imposed by **TCIdentification** is taken into account. This is so because:

a) if no TC is active, then **Use** is empty, thus the subset of **tias** where new TCs can be accepted is **tias** itself; whereas

b) if some TC is active, then the choice of a non-empty **tias** still allows that the subset where new TCs are accepted could be empty, i.e. whenever **tias** is included in **Use**.

```
-----------------------------------------------*)

type TPETCAcceptance
is TAddresses, TCEIdentifications
opns
  Addresses : TIds -> TAddresses
eqns
forall
  ta : TAddress, tcei : TCEI, tids : TIds
ofsort TAddresses
  Addresses({}) = {};
  Addresses(Insert(TId(ta, tcei), tids)) =
    Insert(ta, Addresses(tids))
endtype (* TPETCAcceptance *)

process TPETCAcceptance[t]
  (tas : TAddresses) : noexit:=
choice  tias : TIds []
[tias ne {} and
 (Addresses(tias) IsSubsetOf tas)] ->
   i ;
   t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
     [ta IsIn tas and (IsTCON1(tsp) implies
       (TId(ta, tcei) IsIn tias))];
TPETCAcceptance [t] (tas)
endproc (* TPETCAcceptance *)

type TPENCAcceptance
is NAddresses, NCEIdentifications
opns
  Addresses : NIds -> NAddresses
eqns
forall
  na : NAddress, ncei : NCEI, nids : NIds
ofsort NAddresses
  Addresses({}) = {};
  Addresses(Insert(NId(na, ncei), nids)) =
    Insert(na, Addresses(nids))
endtype (* TPENCAcceptance *)

process TPENCAcceptance[n]
  (nas : NAddresses) : noexit:=
choice  nias : NIds []
[nias ne {} and
 (Addresses(nias) IsSubsetOf nas)] ->
  i ;
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [na IsIn nas and (IsNCON1(nsp) implies
      (NId(na, ncei) IsIn nias))];
TPENCAcceptance [n] (nas)
endproc (* TPENCAcceptance *)

(*-----------------------------------------------
```

11.4.3.6 Protocol Constraints

General: TPEConnections describes the relationship between provision of TCs and usage of NCs. It consists of the unsynchronised, i.e. independent, parallel composition of an indefinite number of instances of process **TPEConnection**. The latter describes the relationship between, and constrains the occurrence of, TSP and NSP interactions that relate to a single TC and supporting NC, for their lifetime. These interactions include block transfer actions. **UniqueLocalReferences** ensures for all Connections the use of unique local references in blocks[3].

```
-----------------------------------------------*)

process TPEConnections [t, n] : noexit :=
  TPEConnection [t, n]
|||
  i; TPEConnections [t, n]
endproc (* TPEConnections *)

(*-----------------------------------------------
```

Provision of a Transport Connection: TPEConnection is decomposed into three processes. See figure 11.17.

a) Constraints at the **t** gate only: these are imported from the TS Formal Description. Process **TCEP** specifies the use of the same pair <TSAP Address, TCEP Identifier> until a **T-DISCONNECT** Primitive occurs, and the local TS ordering requirements on TSPs relating to a single TC.

b) Constraints at the **n** gate only: process NCEP specifies the use of the same pair <Network Address, NCEP Identifier> until an **N-DISCONNECT** Primitive occurs, and the local NS ordering requirements on the occurrence of NSPs relating to a single NC.

c) Constraints that relate the events at **t** to those at **n** and *vice versa*; these constraints are formalized by process **RelationTSPandNSP**, which describes the relationships between TSPs, blocks, and NSPs, including the response to Protocol errors.

NOTE — Termination of both of the processes that represent the ends of the Connection is a clearly sufficient representation of the end of the Connection lifetime: this motivates the use of the [> exit construct below.

```
-----------------------------------------------*)

process TPEConnection[t, n] : exit:=
TCEP [t] (TSUCalling) [] TCEP [t] (TSUCalled)
 |[t]|
```

[3] In this process, **|** has been included to ease checking with tools.

```
( RelationTSPandNSP [t, n] [> exit )
|[n]|
NCEP [n] (NSUCalling) [] NCEP [n] (NSUCalled)
endproc (* TPEConnection *)

(*-----------------------------------------------
```

Service UserRoles are 'calling' and 'called', and are defined using **Doublet**, which is a sort with two distinct values.

```
-----------------------------------------------*)

type TSUserRole
is Doublet renamedby
sortnames
  TSUserRole for Doublet
opnnames
  TSUCalling for One
  TSUCalled  for Two
endtype (* TSUserRole *)

type NSUserRole
is Doublet renamedby
sortnames
  NSUserRole for Doublet
opnnames
  NSUCalling for One
  NSUCalled  for Two
endtype (* NSUserRole *)

(*-----------------------------------------------
```

Service Constraints: The definitions that relate to the TS boundary are presented, followed by similar NS definitions.

```
-----------------------------------------------*)

process TCEP[t](role : TSUserRole) : exit:=
  TCEPAddress [t]
||
  TCEPIdentification [t]
||
  TCEPSPOrdering [t] (role)
endproc (* TCEP *)

process NCEP[n](role : NSUserRole) : exit:=
  NCEPAddress [n]
||
  NCEPIdentification [n]
||
  NCEPSPOrdering [n] (role)
endproc (* NCEP *)

(*-----------------------------------------------
```

Throughout the lifetime of a Transport Connection the same Identification, a pair <TAddress, TCEI>, is used. Its value is determined on the first event, in cooperation with the TS User, and thereafter is constant.

NOTE — Both processes may terminate at any time: the end of the local (i.e. at a TCEP) lifetime of the TC is actually determined by the local ordering of Service Primitives.

```
-----------------------------------------------*)

process TCEPAddress[t] : exit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP ;
ConstantTA [t] (ta) [> exit
endproc (* TCEPAddress *)

process ConstantTA[t](ta : TAddress) : noexit:=
t !ta ?tcei : TCEI ?tsp : TSP ;
ConstantTA [t] (ta)
endproc (* ConstantTA *)

process TCEPIdentification[t] : exit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP ;
ConstantTCEI [t] (tcei) [> exit
endproc (* TCEPIdentification *)

process ConstantTCEI[t](tcei : TCEI) : noexit:=
t ?ta : TAddress !tcei ?tsp : TSP ;
ConstantTCEI [t] (tcei)
endproc (* ConstantTCEI *)

process NCEPAddress[n] : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP ;
ConstantNA [n] (na) [> exit
endproc (* NCEPAddress *)

process ConstantNA[n](na : NAddress) : noexit:=
n !na ?ncei : NCEI ?nsp : NSP ;
ConstantNA [n] (na)
endproc (* ConstantNA *)

process NCEPIdentification[n] : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP ;
ConstantNCEI [n] (ncei) [> exit
endproc (* NCEPIdentification *)

process ConstantNCEI[n](ncei : NCEI) : noexit:=
n ?na : NAddress !ncei ?nsp : NSP ;
ConstantNCEI [n] (ncei)
endproc (* ConstantNCEI *)

(*-----------------------------------------------
```

TCEPSPOrdering: This specifies the constraints on the possible sequences of TSPs at one TC Endpoint, applied to a single TC. The TC establishment phase at a TC Endpoint is specified as the sequence of **TCEPConnect1** and **TCEPConnect2**. This is due to the possible release of the TC even before (thus preventing) successful establishment of the TC, but only after the beginning of the TC lifetime.

The **T-CONNECT** Primitive executed in **TCEPConnect1** is relevant information for **TCEPConnect2**, as constraints apply to the **T-CONNECT response/confirm** that depend on the **T-CONNECT indication/request**.

Successful TC establishment enables entering the data

transfer phase, which at each TCEP is specified by **TCEP-DataTransfer**. The behaviour in this phase is independent of the rõle of the TCEP.

TC release at a TC Endpoint consists of a T-Disconnect Primitive execution, as represented by **TCEPRelease**. This can occur at any time after the first **T-CONNECT**.

NOTE — The last alternative in the definition of **TCEPSPOrdering** caters for the possibility that the Network Connection is released without execution of a TSP at this TCEP.

```
-------------------------------------------*)

process TCEPSPOrdering[t](role : TSUserRole) :
  exit:=
TCEPConnect1 [t] (role)
 >> accept  tsp : TSP in
(
  (
    TCEPConnect2 [t] (tsp)
  >>
    TCEPDataTransfer [t]
  )
[>
  TCEPRelease [t]
)
 [] [role eq TSUCalled] -> exit
endproc (* TCEPSPOrdering *)

process TCEPConnect1[t]
 (role : TSUserRole) : exit(TSP):=
[role eq TSUCalling] ->
  t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
    [IsTCONreq(tsp) and (ta IsCallingOf tsp)];
  exit (tsp)
 []
[role eq TSUCalled] ->
  t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
    [IsTCONind(tsp) and (ta IsCalledOf tsp)];
  exit (tsp)
endproc (* TCEPConnect1 *)

process TCEPConnect2[t](tsp1 : TSP) : exit:=
t ?ta : TAddress ?tcei : TCEI ?tsp2 : TSP
  [tsp2 IsValidTCON2For tsp1]; exit
endproc (* TCEPConnect2 *)

(*-------------------------------------------
```

See the definition of the data type **TransportServicePrimitive** for the definition of the boolean function **IsValidTCON2For**.

The following describes the Data Transfer Phase and Release Phase at a TCEP.

```
-------------------------------------------*)

process TCEPDataTransfer[t] : noexit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTDT(tsp)];
TCEPDataTransfer [t]
endproc (* TCEPDataTransfer *)

process TCEPRelease[t] : exit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTDIS(tsp)];
exit
endproc (* TCEPRelease *)

(*-------------------------------------------
```

The definition of **NCEPSPOrdering** is similar to the definition of TCEPSPOrdring. The most significant difference between both definitions is found in the process NCEPDataTransfer where **N-RESET** Primitives may occur between successive **N-DATA** Primitives.

```
-------------------------------------------*)

process NCEPSPOrdering[n]
 (role : NSUserRole) : exit:=
NCEPConnect1 [n] (role)
 >> accept  nsp : NSP in
(
  (
    NCEPConnect2 [n] (nsp)
  >>
    NCEPDataTransfer [n]
  )
[>
  NCEPRelease [n]
)
 [] [role eq NSUCalled] -> exit
endproc (* NCEPSPOrdering *)

process NCEPConnect1[n]
 (role : NSUserRole) : exit(NSP):=
[role eq NSUCalled] ->
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNCONreq(nsp) and (na IsCallingOf nsp)];
  exit (nsp)
 []
[role eq NSUCalled] ->
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNCONind(nsp) and (na IsCalledOf nsp)];
  exit (nsp)
endproc (* NCEPConnect1 *)

process NCEPConnect2[n](nsp1 : NSP) : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp2 : NSP
  [nsp2 IsValidNCON2For nsp1];
exit
endproc (* NCEPConnect2 *)

process NCEPDataTransfer[n] : noexit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDT(nsp)];
NCEPDataTransfer [n]
[]
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNRSTind(nsp)];
```

```
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNRSTresp(nsp)];
NCEPDataTransfer[n]
endproc (* NCEPDataTransfer *)

process NCEPRelease[n] : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDIS(nsp)];
exit
endproc (* NCEPRelease *)

(*-----------------------------------------------
```

Protocol Constraints: The process **RelationTSPandNSP** describes the constraints between TSPs and NSPs, which may contain blocks. It consists of four parts:

a) process **TNCNProvision** describes constraints relating to the assignment of the TC Connection to the NC Connection. It deals with **T-CONNECT request**, **T-CONNECT indication**, **N-CONNECT request** and **N-CONNECT indication** Service Primitives.

b) process **BlockHandling** relates TSPs through valid blocks to NSPs. It specifies which NSPs that carry blocks should occur after a given TSP and vice versa. It deals with **N-CONNECT response**, **N-CONNECT indication**, **T-DATA** and **N-DATA** Service Primitives.

c) process **ErrorDetectionAndHandling** specifies the detection of blocks that are invalid or constitute Protocol errors. Furthermore it specifies what actions are to be taken in case of any error. It deals with **N-RESET** and **N-DATA** Service Primitives.

d) process **TCNCRelease** specifies the de-assignment of Transport and Network Connection. It deals with **T-DISCONNECT** and **N-DISCONNECT** Service Primitives.

```
-----------------------------------------------*)

process RelationTSPandNSP[t, n] : noexit:=
(
  BlockHandling [t, n]
||
  ErrorDetectionAndHandling [t, n]
)
|| RelationTCandNC [t, n]
endproc (* RelationTSPandNSP *)

(*-----------------------------------------------
```

RelationTCandNC describes the constraints relating to assignment of the TC to an NC. It may either create a new Network Connection as a TC initiator or accept a Network Connection as a TC responder. In both cases, the lifetime of the TC will be directly related to the lifetime of the corresponding NC.

Implementations will also relate the parameters of corresponding **T-CONNECT** and **N-CONNECT** Primitives, e.g. addresses or QoS values. This is not formally described, however, since such information cannot be derived from the informal description.

```
-----------------------------------------------*)

process RelationTCandNC[t, n] : noexit:=
(TCNCProvision[t, n] >> IgnoreUntilDIS [t, n])
[> TCNCRelease [t, n]
endproc (* TCNCProvision *)

(*-----------------------------------------------
```

A Network Connection may be created on request of the TS user or on request of the NS provider.

NOTE — The possibility that a Network Connection is accepted, but does not result in a **T-CONNECT indication** is catered for by the ability of an Entity to initiate a NCrelease at any time.

```
-----------------------------------------------*)

process TCNCProvision [t, n] : exit :=
TCNCInitiator [t, n] [] TCNCResponder [t, n]
endproc (* TCNCProvision *)

process TCNCInitiator[t, n] : exit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTCONreq(tsp)];
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNCONreq(nsp)];
exit
endproc (* TCNCInitiator *)

process TCNCResponder[t, n] : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNCONind(nsp)];
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTCONind(tsp)];
exit
endproc (* TCNCResponder *)

(*-----------------------------------------------
```

TCNCRelease describes the constraints relating to de-assignment of the TC from the underlying NC. A Transport Connection release can be initiated in three, possible concurrent ways. The fact that two or more of these cases may occur independently is reflected by the interleaving of the processes. The Service constraints ensure that every Connection is closed only once:

a) **UserRelease** describes the release initiated by the Transport Service User; and

b) **LocalRelease** describes the release initiated by the local Transport Entity; and

c) **RemoteRelease** describes the release initiated by the remote Transport Entity or by the NS provider. No distinction can be made between a NS provider initiated disconnect and a remote Entity initiated disconnect.

```
-------------------------------------------*)
process TCNCRelease[t, n] : noexit:=
  UserRelease [t, n]
|||
  LocalRelease [t, n]
|||
  RemoteRelease [t, n]
endproc (* TCNCRelease *)

(*-------------------------------------------
```

The Transport Service User may initiate a release by the T-DISCONNECT Request Service Primitive.

```
-------------------------------------------*)

process UserRelease[t, n] : noexit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTDISreq(tsp)];
n ?na : NAddress ?ncei : NCEI !NDISreq ; stop
endproc (* UserRelease *)

(*-------------------------------------------
```

The release initiated by the remote Entity or by the NS provider is indicated to the Entity using an **N-DISCONNECT indication** Service Primitive. The TS user is informed using a **T-DISCONNECT indication**.

```
-------------------------------------------*)

process RemoteRelease[t, n] : noexit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
[IsNDISind(nsp)];
t ?ta : TAddress ?tcei : TCEI !TDISind ; stop
endproc (* RemoteRelease *)

(*-------------------------------------------
```

An entity may at any time decide to initiate release of the Transport Connection. This includes the following cases in which no Transport or Network Service Primitives, other than **DISCONNECT**, are allowed:

a) after expiry of the timer set on sending a **TCR** block; and

b) after an **N-RESET** response.

```
-------------------------------------------*)

process LocalRelease[t, n] : noexit:=
t ?ta : TAddress ?tcei : TCEI !TDISreq ; stop
 |||
n ?na : NAddress ?ncei : NCEI !NDISreq ; stop
endproc (* ReleaseTC *)

(*-------------------------------------------
```

BlockHandling: This process relates TSPs and NSPs through blocks. The **TCR**, **TCC** and **TCA** blocks are termed 'direct'. Non-direct blocks are **TDT** which is handled by **DataBlockTransfer**. and **TBR** which is handled by **TBRTransfer**. The usage of the unsynchronised parallel operator is justified by the fact that the sets of events in which the parallel components may engage are disjoint. Requirements on sent blocks, for example the maximum length of a block, are represented by **SendBlockConstraints**

```
-------------------------------------------*)

process BlockHandling[t, n] : noexit:=
  (
    DirectBlockTransfer [t, n]
  |||
    DataBlockTransfer [t, n]
  |||
    TBRTransfer [n ]
  )
|[n]|
  SendBlockConstraints [n]
endproc (* BlockHandling *)

(*-------------------------------------------
```

The Calling and Called addresses of **T-CONNECT** Primitives should be related to the corresponding optional extension addresses of the corresponding **TCR** and **TCA** blocks. The information presented in the informal description is however not sufficient to describe formal constraints relating to this relationship. The two components of **DirectBlockTransfer** have disjoint interaction sets. This justifies their unsynchronised parallel composition.

NOTE — Process **DirectUp** contains an incompatibility between T.70 and ISO Transport Class 0. In Transport Class 0, after receipt of a **TCC** block, the connection must be released, while in T.70 another **TCR** block may be sent.

```
-------------------------------------------*)

process DirectBlockTransfer[t, n] : noexit:=
DirectUp [t, n] ||| DirectDown [t, n]
endproc (* DirectBlockTransfer *)

process DirectDown[t, n] : noexit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTCONreq(tsp)];
n ?na : NAddress  ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and IsTCR(Userdata(nsp))];
DirectDown [t, n]
 []
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTCONresp(tsp)];
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and IsTCA(Userdata(nsp))];
DirectDown [t, n]
endproc (* DirectDown *)

process DirectUp[t, n] : noexit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTind(nsp) and IsTCR(Userdata(nsp))];
```

```
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTCONind(tsp)];
DirectUp [t, n]
 []
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTind(nsp) and IsTCA(Userdata(nsp))];
t ?ta : TAddress  ?tcei : TCEI !TCONconf ;
DirectUp [t, n]
 []
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTind(nsp) and IsTCC(Userdata(nsp))];
(
  t ?ta : TAddress ?tcei : TCEI !TDISind ;
  DirectUp [t, n]
[]
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and IsTCR(Userdata(nsp))];
  DirectUp [t, n]
)
endproc (* DirectUp *)

(*-----------------------------------------------
```

A data structure is needed for the description of the constraints that relate to transfer of data blocks:

This data structure is used for the sequence of TSDUs conveyed by **T-DATA request** and for that of TSDUs to be (possibly) conveyed by **T-DATA indication**.

Basic operations on these sequences follow a FIFO discipline. Thus the definition of **BasicTSDUS** is presented as an instance of a generic, i.e. parameterized **Queue** definition, which introduces the usual operations on FIFO queues. The basic definition is then enriched in **TSDUS** with functions that, taking into account that queue elements are octet strings, allow one to manipulate the top and the bottom element of a queue, using standard **OctetString** operations. These functions enable the description of segmenting and reassembling in a straightforward way (see process **DataBlockTransfer**):

a) **ReplaceTop** relates to the segmenting of the earliest TSDU of **TSDUdown** into outgoing **TDT** blocks; and

b) **AddSegment** relates to the reassembling of the latest TSDU of **TSDUup** from incoming **TDT** blocks.

The definition of **TSDUS** is based on a generic description of **Queue** (see 11.4.3.9) to which the operations **ReplaceTop** and **AddSegment** are added.

```
-----------------------------------------------*)

type BasicTSDUS
is Queue actualizedby OctetString, Boolean using
sortnames
  OctetString for Element
  TSDUS       for Queue
  Bool        for FBool
endtype (* BasicTSDUS *)

type TSDUS
is BasicTSDUS
opns
  ReplaceTop : OctetString, TSDUS -> TSDUS
  AddSegment : OctetString, TSDUS -> TSDUS
eqns
forall
  s, s1, t : OctetString, q : TSDUS
ofsort TSDUS
  ReplaceTop(t, Empty)              = Empty;
  ReplaceTop(t, Add(s, Empty))      =
    Add(t, Empty);
  ReplaceTop(t, Add(s, Add(s1, q))) =
    Add(s, ReplaceTop(t, Add(s1, q)));
  AddSegment(s, Empty)      = Empty;
  AddSegment(s, Add(s1, q)) = Add(s ++ s1, q)
endtype (* TSDUS *)

(*-----------------------------------------------
```

DataBlockTransfer is split into **TransferDown** which describes the segmenting of a **T-DATA request** into outgoing **TDT** blocks and their transfer through the Network Service, and **TransferUp** which describes the reassembling of incoming **TDT** blocks into TSDUs and their transfer to the TS user by means of **T-DATA indication**. The initial value of the parameter of **TransferUp** enables this process to start reassembling of incoming **TDT** blocks. The value of **et** in **TDT** blocks denotes presence of an End-Of-TSDU delimiter.

No internal non-determinism is described in the following processes. The internal non-determinism on the size of outgoing data blocks, and the constraint that follows from the related negotiation, are described in process **SendBlockConstraints**.

```
-----------------------------------------------*)

process DataBlockTransfer[t, n] : noexit:=
    TransferUp [t, n] (Add(<>, Empty))
||| TransferDown [t, n] (Empty)
endproc (* DataBlockTransfer *)

process TransferDown[t, n]
 (down : TSDUS) : noexit:=
t ?ta : TAddress ?tcei : TCEI ?tsp : TSP
  [IsTDTreq(tsp)];
TransferDown [t, n] (Add(Userdata(tsp), down))
 []
(
  choice  et : EOTsdu, ud, rd, s : OctetString []
    [(rd ++ ud IsTopOf down) and
     (s Encodes TDT(et, ud))] ->
      n ?na : NAddress ?ncei : NCEI !NDTreq(s) ;
      (
        [et eq yes] ->
          TransferDown [t, n] (RemoveTop(down))
        []
        [et eq no] ->
          TransferDown [t, n]
            (ReplaceTop(rd, down))
```

```
      )
)
endproc (* TransferDown *)

process TransferUp[t, n](up : TSDUS) : noexit:=
(
  choice  et : EOTsdu, ud : OctetString []
    n ?na : NAddress ?ncei : NCEI ?nsp : NSP
      [TDT(et, ud) Decodes Userdata(nsp)];
    (
      [et eq yes] ->
        TransferUp [t, n]
          (Add(<>, AddSegment(ud, up)))
    []
      [et eq no] ->
        TransferUp [t, n] (AddSegment(ud, up))
    )
)
 []
(
  choice  ud : OctetString []
    [ud IsTopOf up and
     not(RemoveTop(up) eq Empty)] ->
      t ?ta : TAddress ?tcei : TCEI !TDTind(ud) ;
      TransferUp [t, n] (RemoveTop(up)
   )
)
endproc (* TransferUp *)

(*-----------------------------------------
```

The process **TBRTransfer** allows transmission or receipt of a **TBR** block at any time.

```
-----------------------------------------*)

process TBRTransfer[n] : noexit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsTBR(Userdata(nsp))];TBRTransfer[n]
endproc (* TBRTransfer *)

(*-----------------------------------------
```

SendBlockConstraints specifies that the size of an outgoing **TDT** Block shall not exceed the maximum TPDU size negotiated in the Connection establishment.

```
-----------------------------------------*)

process SendBlockConstraints[n] : noexit:=
IgnoreUntilTCA [n]
 [>
(choice  b : Block, ds : DTSize
   []
  [ds IsDTSizeOf b and IsTCA(b)] ->
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [Userdata(nsp) Encodes b];
  MaxLengthSend [n] (ds))
endproc (* SendBlockConstraints *)

process MaxLengthSend[n](ms : DTSize) : noexit:=
IgnoreUntilNDTreq [n]
 [>
 n ?na : NAddress ?ncei : NCEI ?nsp : NSP
   [IsNDTreq(nsp) and
     (Length(Userdata(nsp)) le DTSize(ms))];
   MaxLengthSend [n] (ms)
endproc (* MaxLengthSend *)

(*-----------------------------------------
```

ErrorDetectionAndHandling: This process specifies error detection, determination whether or not a block is valid block, and error handling. Three sources of errors can be identified:

a) errors detected by the Network Service Provider resulting in an **N-RESET** Primitive; and

b) errors detected by the peer Entity resulting in the receipt of a **TBR** block; and

c) errors detected by this Entity.

For each of these sources a process specifies the corresponding constraints on error detection and error handling. The process **NoDIS** specifies that a normal release is never an error.

```
-----------------------------------------*)

process ErrorDetectionAndHandling[t, n] :
 noexit:=
  NetworkDetectedErrors [t, n]
||
  PeerEntityDetectedErrors [t, n]
||
  EntityDetectedErrors [t, n]
endproc (* ErrorDetectionAndHandling *)

(*-----------------------------------------
```

The distinction between error detection and error handling is represented by the use of the >> operator. An Network detected error is always indicated to the NS user using an **N-RESET indication** Primitive which is answered by an **N-RESET response** and the Connection is released.

```
-----------------------------------------*)

process NetworkDetectedErrors[t, n] : noexit:=
  NetworkErrorDetect [n]
|||
  (IgnoreTSPbutDIS [t] [> exit)
 >> NetworkErrorHandling [t, n]
endproc (* NetworkDetectedErrors *)

process NetworkErrorDetect[n] : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNRSTind(nsp)]; exit
endproc (* NetworkErrorDetect *)
```

```
process NetworkErrorHandling[t, n] : noexit:=
n ?na : NAddress ?ncei : NCEI !NRSTresp ; stop
endproc (* NetworkErrorHandling *)

(*-----------------------------------------------
```

Whenever a **TBR** block is received, the Connection is released according to the procedure in 11.2.5.2.

```
-----------------------------------------------*)

process PeerEntityDetectedErrors[t, n] : noexit:=
  PeerEntityErrorDetect [n]
|||
  (IgnoreTSPbutDIS [t] [> exit)
 >>
  PeerEntityErrorHandling [t, n]
endproc (* PeerEntityDetectedErrors *)

process PeerEntityErrorDetect[n] : exit:=
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsTBR(Userdata(nsp))]; exit
endproc (* PeerEntityErrorDetect *)

process PeerEntityErrorHandling[t, n] : noexit:=
stop
endproc (* PeerEntityErrorHandling *)

(*-----------------------------------------------
```

All other errors are detected by the TP Entity. To be able to distinguish them, a data structure **ErrorType** is introduced.

```
-----------------------------------------------*)

type ErrorType
is TenTuplet renamedby
sortnames
  ErrorType for TenTuplet
opnnames
  EncodingError       for One
  ErrorAfterTCR       for Two
  ErrorAfterNCONresp  for Three
  ErrorAfterTDT       for Four
  ErrorAfterTCC       for Five
  ErrorAfterTCA       for Six
  IllegalDTSize       for Seven
  IllegalUserdataSize for Eight
  EmptyTDT            for Nine
  IllegalParameter    for Ten
endtype

(*-----------------------------------------------
```

EntityDetectedErrors deals with all errors, except for **N-RESET** and received **TBR** blocks. **EntityErrorDetect** exports two values of sorts **ErrorType** and **OctetString** respectively. Both values are used when sending **TBR** blocks.

```
-----------------------------------------------*)

process EntityDetectedErrors[t, n] : noexit:=
  EntityErrorDetect [n]
||
  (
    NoNRST [n] || NoTBRReceived [n]
  [>
    exit (any ErrorType, any OctetString)
  )
|||
  (
    IgnoreTSPbutDIS [t]
  [>
    exit (any ErrorType, any OctetString)
  )
>> accept  err : ErrorType, s : OctetString in
     EntityErrorHandling [t, n] (err, s)
endproc (* EntityDetectedErrors *)

(*-----------------------------------------------
```

The Entity detected errors can be divided into three groups:

a) receipt of a block which constitutes a Protocol error; and

b) receipt of a block with a incorrect size; and

c) receipt of a syntactically incorrect block.

```
-----------------------------------------------*)

process EntityErrorDetect[n] :
 exit(ErrorType, OctetString):=
    OrderingConstraints [n]
 || SizeConstraints [n]
 || InvalidBlockConstraints [n]
endproc (* EntityErrorDetect *)

(*-----------------------------------------------
```

All constraints on received blocks are always present. This justifies the full synchronisation between the different error detection processes. Each process may be aborted by the occurrence of an error in one of the other processes. This is represented by '[> **OtherError()**'.

NOTE — When the number of values of sort **ErrorType** is greater than the number of error detection processes, all these processes may exit at any time.

The process **OrderingConstraints** detects the failure cases mentioned in 11.1.7.2[3].

```
-----------------------------------------------*)

process OrderingConstraints[n] :
 exit(ErrorType, OctetString):=
(AfterTCRSend [n] [> OtherError (ErrorAfterTCR))
```

[3] For completeness it would be necessary to add processes to detect if the source reference and destination reference of successive Blocks are correct.

```
 ||
(AfterTCASend [n] [> OtherError (ErrorAfterTCA))
 ||
(AfterTDTSend [n] [> OtherError (ErrorAfterTDT))
 ||
(AfterTCCSend [n] [> OtherError (ErrorAfterTCC))
 ||
(EmptyTDT [n] [> OtherError (EmptyTDT))
 ||
(
  AfterNCONresp [n]
[>
  OtherError (ErrorAfterNCONresp)
)
endproc (* OrderingConstraints *)

(*-----------------------------------------------
```

After sending a **TCR** the receipt of a block which is not a **TCC**, **TCA** or **TBR** constitutes an error.

```
-----------------------------------------------*)

process AfterTCRSend[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilSendTCR [n]
 [>
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and IsTCR(Userdata(nsp))];
 (IgnoreUntilNDTind[n] [>
  (n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and (IsTCA(Userdata(nsp))
      or IsTCC(Userdata(nsp)) or
        IsTBR(Userdata(nsp)))];
   AfterTCRSend [n]
     []
    n ?na : NAddress ?ncei : NCEI ?nsp : NSP
      [IsNDTind(nsp) and not(IsTCA(Userdata(nsp))
        or IsTCC(Userdata(nsp)) or
          IsTBR(Userdata(nsp)))];
    exit (ErrorAfterTCR, Userdata(nsp))))
endproc (* AfterTCRSend *)

(*-----------------------------------------------
```

After sending a **TCA**, the receipt of a block which is not a **TDT** or **TBR** constitutes an error.

```
-----------------------------------------------*)

process AfterTCASend[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilSendTCA [n]
 [>
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and IsTCA(Userdata(nsp))];
  (IgnoreUntilNDTind[n] [>
   (n ?na : NAddress ?ncei : NCEI ?nsp : NSP
     [IsNDTind(nsp) and (IsTDT(Userdata(nsp))
       or IsTBR(Userdata(nsp)))];
     AfterTCASend [n]
       []
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and not(IsTDT(Userdata(nsp))
    or IsTBR(Userdata(nsp)))];
  exit (ErrorAfterTCA, Userdata(nsp))))
endproc (* AfterTCASend *)

(*-----------------------------------------------
```

After sending a **TDT**, receipt of a block which is not a **TDT** or **TBR** is an error.

```
-----------------------------------------------*)

process AfterTDTSend[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilSendTDT [n]
 [>
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and IsTDT(Userdata(nsp))];
  (IgnoreUntilNDTind[n] [>
  (n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and (IsTDT(Userdata(nsp))
    or IsTBR(Userdata(nsp)))]; AfterTDTSend [n]
   []
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and not(IsTDT(Userdata(nsp))
    or IsTBR(Userdata(nsp)))];
  exit (ErrorAfterTDT, Userdata(nsp))))
endproc (* AfterTDTSend *)

(*-----------------------------------------------
```

After sending a **TCC**, receipt of a block which is not a **TCR** or **TBR** is an error.

```
-----------------------------------------------*)

process AfterTCCSend[n]
  : exit(ErrorType, OctetString):=
IgnoreUntilSendTCC [n]
 [>
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and IsTCC(Userdata(nsp))];
 (IgnoreUntilNDTind[n] [>
(n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTind(nsp) and (IsTCR(Userdata(nsp))
    or IsTBR(Userdata(nsp)))]; AfterTCCSend [n]
   []
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and not(IsTCR(Userdata(nsp))
    or IsTBR(Userdata(nsp)))];
  exit (ErrorAfterTCC, Userdata(nsp))))
endproc (* AfterTCCSend *)

(*-----------------------------------------------
```

After receiving a **TDT** with TSDU end mark equal to 1, receipt of an empty **TDT** with End-of-TSDU set to 1 is an error.

```
-----------------------------------------------*)

process EmptyTDT[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilReceivedTDT [n]
 [>
(
  choice  b : block, et : EOTsdu []
    [et IsEndTSDUOf b and IsTDT(b)] ->
      n ?na : NAddress ?ncei : NCEI ?nsp : NSP
        [IsNDTind(nsp) and
          (Userdata(nsp) Encodes b)];
      EmptyTDT2 [n] (et)
)
endproc (* EmptyTDT *)

process EmptyTDT2[n](last : EOTsdu) :
 exit(ErrorType, OctetString):=
IgnoreUntilReceivedTDT [n]
 [>
(
  choice  b : Block, et : EOTsdu,
    os : OctetString []
    [IsTDT(b) and (et IsEndTSDUOf b) and
     (os IsUserdataOf b)] ->
      (
        [Length(os) eq 0 implies
         (last eq yes)] ->
          n ?na : NAddress ?ncei : NCEI ?
           nsp : NSP
            [IsNDTind(nsp) and
              (Userdata(nsp) Encodes b)];
          exit (EmptyTDT, Userdata(nsp))
       []
         [not(Length(os) eq 0 implies
          (last eq yes))] ->
           n ?na : NAddress ?ncei : NCEI ?
            nsp : NSP
             [IsNDTind(nsp) and
               (Userdata(nsp) Encodes b)];
           EmptyTDT2 [n] (et)
      )
)
endproc (* EmptyTDT2 *)

(*-----------------------------------------------
```

After an **N-CONNECT** response, the receipt of not a **TCR** is an error.

```
-----------------------------------------------*)

process AfterNCONresp[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilNCONresp [n]
 [>
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNCONresp(nsp)];
 (IgnoreUntilNDTind[n] [>
(n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTind(nsp) and IsTCR(Userdata(nsp))];
  AfterNCONresp [n]
   []
  n ?na : NAddress ?ncei : NCEI ?nsp : NSP
    [IsNDTind(nsp) and
      not(IsTCR(Userdata(nsp)))];
  exit (ErrorAfterNCONresp, Userdata(nsp))))
endproc (* AfterNCONresp *)

(*-----------------------------------------------
```

The following two constraints are derived from the description in 11.1.6.8.

```
-----------------------------------------------*)

process SizeConstraints[n] :
 exit(ErrorType, OctetString):=
(
  DTSizeConstraint [n]
[>
  OtherError (IllegalDTSize)
)
 ||
(
  TDTLength [n]
[>
  OtherError (IllegalUserdataSize)
)
endproc (* SizeConstraints *)

(*-----------------------------------------------
```

The **DTSize** parameter in a received **TCC** or **TCA** must not be greater than the **DTSize** parameter in the last send **TCR**, see 11.1.4.2.

```
-----------------------------------------------*)

process DTSizeConstraint[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilSendTCR [n]
 [>
(
  choice  b : Block, ds : DTSize []
    [IsTCR(b) and (ds IsDTSizeOf b)] ->
      n ?na : NAddress ?ncei : NCEI ?nsp : NSP
        [IsNDTreq(nsp) and
          (Userdata(nsp) Encodes b)];
      DTSizeConstraint2 [n] (ds)
)
endproc (* DTSizeConstraint *)

process DTSizeConstraint2[n]
 (mds : DTSize) : exit(ErrorType, OctetString):=
  IgnoreUntilReceivedTCC [n]
||
  IgnoreUntilReceivedTCA [n]
||
  IgnoreUntilReceivedTCR[n]
 [>
```

```
(
  choice  b : Block, ds : DTSize []
    [IsTCC(b) or IsTCA(b) and
      (ds IsDTSizeOf b)] ->
      (
        [ds le mds] ->
          n ?na : NAddress ?ncei : NCEI ?
           nsp : NSP
            [IsNDTind(nsp) and
              (Userdata(nsp) Encodes b)];
          DTSizeConstraint2 [n] (mds)
     []
       [ds gt mds] ->
         n ?na : NAddress ?ncei : NCEI ?nsp : NSP
           [IsNDTind(nsp) and
             (Userdata(nsp) Encodes b)];
         exit (IllegalDTSize, Userdata(nsp))
      )
  []
    [IsTCR(b) and (ds IsDTSizeOf b)] ->
      n ?na : NAddress ?ncei : NCEI ?nsp : NSP
        [IsNDTind(nsp) and
          (Userdata(nsp) Encodes b)];
      DTSizeConstraint2 [n] (ds)
)
endproc (* DTSizeConstraint2 *)

(*-----------------------------------------------
```

The total length of a **TDT** Block shall not be greater than the negotiated **DTSize**.

```
-----------------------------------------------*)

process TDTLength[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilReceivedTCA [n]
 [>
(
  choice  b : Block, ds : DTSize []
    [IsTCA(b) and (ds IsDTSizeOf b)] ->
      n ?na : NAddress ?ncei : NCEI ?nsp : NSP
        [IsNDTind(nsp) and
          (Userdata(nsp) Encodes b)];
      TDTLength2 [n] (ds)
)
endproc (* TDTLength *)

process TDTLength2[n]
 (mds : DTSize) : exit(ErrorType, OctetString):=
IgnoreUntilNDTind [n]
 [>
(
  choice  b : block, bss : BlockSubsort,
    os : OctetString []
    [Subsort(b) eq bss and (os Encodes b)] ->
    (
      [Length(os) le DTSize(mds)] ->
        n ?na : NAddress ?ncei : NCEI ?nsp : NSP
          [IsNDTind(nsp) and
            (Userdata(nsp) eq os)];
          TDTLength2 [n] (mds)
    []
      [not(Length(os) le DTSize(mds))] ->
        n ?na : NAddress ?ncei : NCEI ?nsp : NSP
          [IsNDTind(nsp) and
            (Userdata(nsp) eq os)];
        exit (IllegalUserdataSize, Userdata(nsp))
    )
)
endproc (* TDTLength2 *)

(*-----------------------------------------------
```

InvalidBlockConstraints detects which blocks are valid with respect to encoding.

```
-----------------------------------------------*)

process InvalidBlockConstraints[n] :
 exit(ErrorType, OctetString):=
IgnoreUntilReceivedInvalidBlock [n]
 [>
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTind(nsp) and
    not(EncodesABlock(Userdata(nsp)))];
exit (EncodingError, Userdata(nsp))
endproc (* InvalidBlockConstraints *)

(*-----------------------------------------------
```

After an error detected by a TP Entity, a **TBR** block should be sent. Afterwards, incoming Blocks are ignored, see 11.2.5.2. The informal specification states that the bit pattern of the rejected block up to and including the octet that cause the rejection are to be transmitted. For the sake of simplicity, transmission of any part of this bit pattern is allowed.

Readers are invited to extend the description of error detection so that this simplification is removed.

```
-----------------------------------------------*)

process EntityErrorHandling[t, n]
 (err : ErrorType, os : OctetString) : noexit:=
choice  b : Block, s1, s2 : OctetString
 []
[IsTBR(b) and  ((s1 ++ s2) eq os) and
  ( s1 IsRejBlockof b)] ->
n ?na : NAddress ?ncei : NCEI ?nsp : NSP
  [IsNDTreq(nsp) and (Userdata(nsp) Encodes b)];
(OnlyNDTind [n] [> i ; stop )
endproc (* EntityErrorHandling *)

(*-----------------------------------------------
```

The process **OtherError** exits on all errors other than the given one.

```
-----------------------------------------------*)
```

```
process OtherError(e : ErrorType) :
 exit(ErrorType, OctetString):=
choice  x : ErrorType []
  [x ne e] ->
    exit (x, any OctetString)
endproc (* OtherError *)

(*-----------------------------------------------
```

GlobalConstraints: The only global constraint is the uniqueness of TC references passed at the **n** gate.

```
-----------------------------------------------*)

process GlobalConstraints [t, n] : noexit :=
UniqueLocalReferences [n]
|||
IgnoreTSP [t]
endproc (* GlobalConstraints *)

(*-----------------------------------------------
```

Unique References: Usage of local references is to be such that for each TC a unique reference is made use of. Note however that only **TCR**, **TCA** and **TCC** blocks have the source reference parameter. First a non-empty set **lrs** of non-zero references is internally chosen.

```
-----------------------------------------------*)

process UniqueLocalReferences[n] : noexit:=
choice lrs : RefSet []
  [lrs ne {} and (Unassigned NotIn lrs)] ->
    i ; LocalReferences [n] (lrs)
endproc (* UniqueLocalReferences *)

(*-----------------------------------------------
```

LocalReferences: This ensures that the source reference used in an outgoing **TCR** or **TCA** or **TCC** block is not in use for another Connection[4]. A reference can either be **Free** or **Bound**.

```
-----------------------------------------------*)

process LocalReferences[n]
 (lrs : RefSet) : noexit:=
choice  lr : Ref []
  [lr IsIn lrs] ->
    (
      FreeRef [n] (lr)
    |||
      i; LocalReferences [n] (Remove(lr, lrs))
    )
endproc (* LocalReferences *)

process FreeRef[n](lr : Ref) : noexit:=
IgnoreUntilNDT [n]
 [>
(
  choice  b : Block []
    n ?na : NAddress ?ncei : NCEI ?nsp : NSP
      [IsNDTreq(nsp) and
        (Userdata(nsp) Encodes b) and
          (lr IsSrcRefOf b)];
    BoundRef [n] (na, ncei, lr)
)
endproc (* FreeRef *)

process BoundRef[n]
 (na : NAddress, ncei : NCEI, lr : Ref) :
  noexit:=
n !na !ncei ?nsp : NSP [not(IsNDIS(nsp))];
BoundRef [n] (na, ncei, lr)
 []
n !na !ncei ?nsp : NSP [IsNDIS(nsp)];
FreeRef [n] (lr)
endproc (* BoundRef *)

(*-----------------------------------------------
```

[4] In this process, I has been included to ease checking with tools.

11.4.3.7 Block Data Type Definitions

The definitions relating to blocks are presented below in a hierarchical fashion, according to the following outline, where items correspond to the types that follow:

a) basic construction of an 'abstract' (i.e. independent of encoding) block data type; and

b) definition of 'block subsort' values, that correspond to the 'block types' defined in the Protocol: the difference in terminology is to avoid confusion with the (more general) concept of 'type' in LOTOS; and

c) enrichment of the abstract block with functions, termed 'Classifiers', that tell whether or not a given block is of a given subsort; and

d) enrichment of the abstract block with functions, termed 'parameter selectors', that tell whether a given value is the value of a certain parameter of a given block (this indirect representation is convenient, for the sake of completeness of the equational definition, since generally a block parameter is defined only for some, but not all, blocks); and

e) enrichment of the abstract block with boolean functions representing equality and inequality; and

f) definitions relating to individual parameters of (abstract) blocks: each definition includes basic construction and equality enrichments

Auxiliary definitions that are referred to in the following, as well as the encoding of Blocks into OctetStrings, are presented later.

Basic Block Construction: Values of sort **Block**, which represent blocks abstracting from encoding details, are constructed by five functions, each corresponding to a distinct block 'type' (in the sense of the Protocol). The sorts of block

parameters are either standard sorts, such as **OctetString** and **Bool**, or are defined later. Note that absence of optional parameters, e.g. maximum data block size in **TCR** or **TCA** blocks, is only represented in the concrete encoding of blocks, where it is mapped to the parameter values (of the abstract block) that are defined by the Protocol as default values[5].

```
-----------------------------------------------*)

type BasicBlock
is BasicReference, ExtendedAddressing,
  DataBlockSize, ClearingCause, OctetString,
  Boolean, RejectCause, EOTsdu
sorts
  Block
opns
  TCR : Ref, ExtAddress, ExtAddress, DTSize
    -> Block
  TCA : Ref, Ref, ExtAddress, ExtAddress, DTSize
    -> Block
  TCC : Ref, Ref, ClearingCause, OctetString
    -> Block
  TDT : EOTsdu, OctetString -> Block
  TBR : Ref, RejCause, OctetString -> Block
endtype (* BasicBlock *)

(*-----------------------------------------------
```

Block Subsorts: *Sort* **BlockSubsort** *consists of five constants, which represent the block types defined in the Protocol.* See 11.4.3.9 for the type **FiveTuplet**.

```
-----------------------------------------------*)

type BlockSubsort
is FiveTuplet renamedby
sortnames
  BlockSubsort for FiveTuplet
opnnames
  TCR for One
  TCC for Two
  TCA for Three
  TDT for Four
  TBR for Five
endtype (* BlockSubsort *)

(*-----------------------------------------------
```

Block Classifiers: The following definition enriches the combination of the two basic constructions given above with the following functions on blocks:

a) **Subsort**, that yields the block subsort; and

b) the boolean functions **IsTCR**, **IsTCA**, etc. termed **BlockClassifiers**.

[5]The specification does not deal with a Class option in a **TCR** or **TCC**.

```
-----------------------------------------------*)

type BlockClassifiers
is BasicBlock, BlockSubsort
opns
  Subsort : Block -> BlockSubsort
  IsTCR, IsTCA, IsTCC, IsTDT, IsTBR :
    Block -> Bool
eqns
forall
  b : Block, sr, dr : Ref, cga, cda : ExtAddress,
  ds : DTSize, cc : ClearingCause,
  ac : OctetString, et : EOTsdu,
  ud : OctetString, rc : RejCause,
  rb : OctetString
ofsort BlockSubsort
  Subsort(TCR(sr, cga, cda, ds))      = TCR;
  Subsort(TCA(dr, sr, cga, cda, ds)) = TCA;
  Subsort(TCC(dr, sr, cc, ac))        = TCC;
  Subsort(TDT(et, ud))                = TDT;
  Subsort(TBR(dr, rc, rb))            = TBR
ofsort Bool
  IsTCR(b) = Subsort(b) eq TCR;
  IsTCA(b) = Subsort(b) eq TCA;
  IsTCC(b) = Subsort(b) eq TCC;
  IsTDT(b) = Subsort(b) eq TDT;
  IsTBR(b) = Subsort(b) eq TBR
endtype (* BlockClassifiers *)

(*-----------------------------------------------
```

Block Selectors: The following definition presents boolean functions that allow to determine whether a given value is the value of a certain parameter of a given block, for each block parameter defined by the Protocol. The data types relating to parameters of sort other than **OctetString** or **Bool** are defined later.

```
-----------------------------------------------*)

type BlockParameterSelectors
is BlockClassifiers
opns
  _IsSrcRefOf_, _IsDstRefOf_ : Ref, Block
    -> Bool
  _IsCallingAddrOf_, _IsCalledAddrOf_ :
    ExtAddress, Block       -> Bool
  _IsDTSizeOf_ : DTSize, Block -> Bool
  _IsClearingCauseOf_ : ClearingCause, Block
    -> Bool
  _IsAddClearInfOf_ : OctetString, Block -> Bool
  _IsEndTSDUOf_ : EOTsdu, Block -> Bool
  _IsUserdataOf_, _IsRejBlockOf_ :
    OctetString, Block -> Bool
  _IsRejCauseOf_ : RejCause, Block -> Bool
eqns
forall
  b : Block, sr, sr1, dr, dr1 : Ref,
  cga, cga1, cda, cda1 : ExtAddress,
  ds, ds1 : DTSize, cc, cc1 : ClearingCause,
  ac, ac1 : OctetString, et, et1 : EOTsdu,
```

```
  ud, ud1 : OctetString, rc, rc1 : RejCause,
  rb, rb1 : OctetString
ofsort Bool
  sr IsSrcRefOf TCR(sr1, cga, cda, ds) =
    sr eq sr1;
  sr IsSrcRefOf TCA(dr, sr1, cga, cda, ds) =
    sr eq sr1;
  sr IsSrcRefOf TCC(dr, sr1, cc, ac) = sr eq sr1;
  not(IsTCR(b) or IsTCA(b) or IsTCC(b)) =>
    sr IsSrcRefOf b = false;
  dr IsDstRefOf TCA(dr1, sr, cga, cda, ds) =
    dr eq dr1;
  dr IsDstRefOf TCC(dr1, sr, cc, ac) = dr eq dr1;
  dr IsDstRefOf TBR(dr1, rc, rb) = dr eq dr1;
  not(IsTCA(b) or IsTCC(b) or IsTBR(b)) =>
    dr IsDstRefOf b = false;
  cga IsCallingAddrOf TCR(sr, cga1, cda, ds) =
    cga eq cga1;
  cga IsCallingAddrOf
    TCA(dr, sr, cga1, cda, ds) = cga eq cga1;
  not(IsTCR(b) or IsTCA(b)) =>
    cga IsCallingAddrOf b = false;
  cda IsCalledAddrOf TCR(sr, cga, cda1, ds) =
    cda eq cda1;
  cda IsCalledAddrOf
    TCA(dr, sr, cga, cda1, ds) = cda eq cda1;
  not(IsTCR(b) or IsTCA(b)) =>
    cda IsCalledAddrOf b  = false;
  ds IsDTSizeOf TCR(sr, cga, cda, ds1) =
    ds eq ds1;
  ds IsDTSizeOf TCA(dr, sr, cga, cda, ds1) =
    ds eq ds1;
  not(IsTCR(b) or IsTCA(b)) =>
    ds IsDTSizeOf b = false;
  cc IsClearingCauseOf TCC(dr, sr, cc1, ac) =
    cc eq cc1;
  not(IsTCC(b)) =>
    cc IsClearingCauseOf b = false;
  ac IsAddClearInfOf TCC(dr, sr, cc, ac1) =
    ac eq ac1;
  not(IsTCC(b)) =>
    ac IsAddClearInfOf b = false;
  et IsEndTSDUOf TDT(et1, ud) = et eq et1;
  not(IsTDT(b)) =>
    et IsEndTSDUOf b = false;
  ud IsUserdataOf TDT(et, ud1) = ud eq ud1;
  not(IsTDT(b)) =>
    ud IsUserdataOf b = false;
  rc IsRejCauseOf TBR(dr, rc1, rb) = rc eq rc1;
  not(IsTBR(b)) =>
    rc IsRejCauseOf b = false;
  rb IsRejBlockOf TBR(dr, rc, rb1) = rb eq rb1;
  not(IsTBR(b)) =>
    rb IsRejBlockOf b = false
endtype (* BlockParameterSelectors *)

(*-----------------------------------------------
```

Block Equality: Equality of two blocks holds if, and only if, the blocks are of the same subsort and have pairwise equal parameter values. The following definition is an effective formalization of this requirement.

```
-----------------------------------------------*)

type BlockEquality
is BlockParameterSelectors
opns
  _eq_, _ne_ : Block, Block -> Bool
eqns
forall
  b, b1 : Block, sr, sr1, dr, dr1 : Ref,
  cga, cga1, cda, cda1 : ExtAddress,
  ds, ds1 : DTSize, cc, cc1 : ClearingCause,
  ac, ac1 : OctetString, et, et1 : EOTsdu,
  ud, ud1 : OctetString, rc, rc1 : RejCause,
  rb, rb1 : OctetString
ofsort Bool
  Subsort(b) ne Subsort(b1) => b eq b1 = false;
  TCR(sr, cga, cda, ds) eq
   TCR(sr1, cga1, cda1, ds1) =
    (sr eq sr1) and (cga eq cga1) and
     (cda eq cda1) and (ds eq ds1);
  TCA(dr, sr, cga, cda, ds) eq
   TCA(dr1, sr1, cga1, cda1, ds1) =
    (dr eq dr1) and (sr eq sr1) and
     (cga eq cga1) and (cda eq cda1) and
      (ds eq ds1);
  TCC(dr, sr, cc, ac) eq
    TCC(dr1, sr1, cc1, ac1) =
    (dr eq dr1) and (sr eq sr1) and
      (cc eq cc1) and (ac eq ac1);
  TDT(et, ud) eq TDT(et1, ud1) =
   et eq et1 and (ud eq ud1);
  TBR(dr, rc, rb) eq TBR(dr1, rc1, rb1) =
    (dr eq dr1) and (rc eq rc1) and (rb eq rb1);
  b ne b1 = not(b eq b1)
endtype (* BlockEquality *)

(*-----------------------------------------------
```

The following data type definitions specify the parameters of blocks.

Protocol references are to be specified as forming a domain of exactly 65535 distinct values. The easiest way of specifying this is just to let a reference uniquely correspond to an ordered 16-tuple of bits. See 11.4.3.9 for the type **Hextet**. The description of this part of the abstract block structure is therefore very close to that of its encoding, for the sake of simplicity; this fact should be considered as an exception.

```
-----------------------------------------------*)

type BasicReference
is Hextet renamedby
sortnames
  Ref for Hextet
opnnames
  Ref for Hextet
endtype (* BasicReference *)
```