

**Implementation of ISO/IEC TR 10167 :
1991**

**Information Technology —
Open systems interconnection
— Guidelines for the
application of Estelle, LOTOS
and SDL**

National foreword

This British Standard reproduces verbatim ISO/IEC TR 10167 : 1991 and implements it as the UK national standard.

This British Standard is published under the direction of the Information Systems Technology Standards Policy Committee whose Technical Committee IST/21 has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international committee any enquiries on interpretation, or proposals for change, and keep UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

NOTE. International and European Standards, as well as overseas standards, are available from BSI Sales Department, BSI, Linford Wood, Milton Keynes, MK14 6LE.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

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)



Documents

Document Sharing Hub

Contents

List of Figures	ix
Foreword	xi
Introduction	xiii
1 Scope	1
2 References	1
3 Terminology	1
3.1 Architectural Terms	2
3.2 FDT Terms	2
4 FDT General Characteristics	2
4.1 Introduction	2
4.2 The Nature and Purpose of FDTs	2
4.2.1 The Purpose of FDTs	2
4.2.2 Use in Development	3
4.2.3 Assessment of FDTs	3
4.3 Estelle	3
4.4 LOTOS	3
4.5 SDL	3
4.6 Benefits of FDTs	4
4.7 Tools for FDTs	4
5 Guide to the Examples	5
5.1 Explanation of the Examples	5
5.1.1 Examples of Basic FDT Concepts	5
5.1.2 Examples of Basic Architectural Concepts	5
5.1.3 Daemon Game	5
5.1.4 Sliding Window Protocol	5
5.1.5 Abracadabra Service and Protocol	5
5.1.6 A Transport Protocol	5
5.2 How to read the Examples	6

© ISO/IEC 1991
All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.
ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland
Printed in Switzerland

6	Examples of Basic FDT Concepts	7
6.1	Abstraction	7
6.1.1	Estelle Representation	7
6.1.2	LOTOS Representation	7
6.1.3	SDL Representation	7
6.2	Information	7
6.2.1	Estelle Representation	7
6.2.2	LOTOS Representation	7
6.2.3	SDL Representation	7
6.3	Action	7
6.3.1	Estelle Representation	7
6.3.2	LOTOS Representation	7
6.3.3	SDL Representation	7
6.4	Interaction	8
6.4.1	Estelle Representation	8
6.4.2	LOTOS Representation	8
6.4.3	SDL Representation	8
6.5	Interaction Point	8
6.5.1	Estelle Representation	8
6.5.2	LOTOS Representation	8
6.5.3	SDL Representation	9
7	Examples of Basic Architectural Concepts	9
7.1	Service Access Point	9
7.1.1	Estelle Representation	9
7.1.2	LOTOS Representation	9
7.1.3	SDL Representation	10
7.2	Endpoint	10
7.2.1	Estelle Representation	10
7.2.2	LOTOS Representation	10
7.2.3	SDL Representation	10
7.3	Service Primitive Parameter	11
7.3.1	Estelle Representation	11
7.3.2	LOTOS Representation	11
7.3.3	SDL Representation	11
7.4	Service Data Unit	11
7.4.1	Estelle Representation	11
7.4.2	LOTOS Representation	12
7.4.3	SDL Representation	12
7.5	Service Primitive	12
7.5.1	Estelle Representation	12
7.5.2	LOTOS Representation	12
7.5.3	SDL Representation	12
7.6	Protocol Entity	12

7.6.1	Estelle Representation	13
7.6.2	LOTOS Representation	13
7.6.3	SDL Representation	13
7.7	Protocol	13
7.7.1	Estelle Representation	13
7.7.2	LOTOS Representation	14
7.7.3	SDL Representation	14
7.8	Protocol Data Unit	15
7.8.1	Estelle Representation	15
7.8.2	LOTOS Representation	15
7.8.3	SDL Representation	15
7.9	Connection	16
7.9.1	Estelle Representation	16
7.9.2	LOTOS Representation	16
7.9.3	SDL Representation	16
7.10	Multiplexing	16
7.10.1	Estelle Representation	16
7.10.2	LOTOS Representation	17
7.10.3	SDL Representation	17
7.11	Splitting	17
7.11.1	Estelle Representation	17
7.11.2	LOTOS Representation	18
7.11.3	SDL Representation	18
7.12	Concatenation	18
7.12.1	Estelle Representation	18
7.12.2	LOTOS Representation	18
7.12.3	SDL Representation	19
7.13	Segmentation	19
7.13.1	Estelle Representation	19
7.13.2	LOTOS Representation	20
7.13.3	SDL Representation	20
7.14	Service	21
7.14.1	Estelle Representation	21
7.14.2	LOTOS Representation	21
7.14.3	SDL Representation	21
8	Daemon Game Example	22
8.1	Informal Description	22
8.2	Deficiencies in the Informal Description	22
8.2.1	Presence of Daemon	22
8.2.2	Login to a Current Game	22
8.2.3	Attempt to play before Login	22
8.2.4	Identification of Players and Games	22
8.2.5	Player Use of System Signals	22

8.2.6	Interruption of Probe or Result	22
8.2.7	Counting of 'Bump' Signals	23
8.3	Estelle Description	23
8.3.1	Architecture of the Formal Description	23
8.3.2	Explanation of Approach	23
8.3.3	Formal Description	23
8.3.4	Alternative Formal Description	25
8.3.5	Subjective Assessment	26
8.4	LOTOS Description	26
8.4.1	Architecture of the Formal Description	27
8.4.2	Explanation of Approach	27
8.4.3	Formal Description	27
8.4.4	Alternative Formal Description	29
8.4.5	Subjective Assessment	31
8.5	SDL Description	31
8.5.1	Architecture of the Formal Description	31
8.5.2	Explanation of Approach	31
8.5.3	Formal Description	32
8.5.4	Subjective Assessment	33
8.6	Assessment of the Application of the FDTs	33
9	Sliding Window Protocol Example	38
9.1	Informal Description	38
9.1.1	Overview	38
9.1.2	Sequence Numbering	38
9.1.3	Transmitter Behaviour	38
9.1.4	Receiver Behaviour	38
9.2	Deficiencies in the Informal Description	39
9.2.1	Underlying Medium	39
9.2.2	Window Size	39
9.2.3	Flow Control	39
9.2.4	Delivery of Corrupted Messages	39
9.2.5	Value of Time-Out Period	39
9.2.6	Consistent Use of NextRequired	39
9.2.7	Receive Window Size	39
9.2.8	Sequence of Operations	39
9.2.9	Transmit Window Size	39
9.2.10	Receive Window Size	39
9.2.11	Corruption of Messages	40
9.2.12	Transfer of Data and Acknowledgements	40
9.2.13	Retransmission on Timeout	40
9.3	Estelle Description	40
9.3.1	Architecture of the Formal Descriptions	40
9.3.2	Explanation of Approach	40

9.3.3	Formal Description of the Protocol	40
9.3.4	Formal Description of the Medium	43
9.3.5	Subjective Assessment	44
9.4	LOTOS Description	44
9.4.1	Architecture of the Formal Descriptions	44
9.4.2	Explanation of Approach	47
9.4.3	Formal Description of the Protocol	47
9.4.4	Formal Description of the Medium	54
9.4.5	Subjective Assessment	56
9.5	SDL Description	56
9.5.1	Architecture of the Formal Descriptions	56
9.5.2	Explanation of Approach	56
9.5.3	Formal Description of the Protocol	57
9.5.4	Formal Description of the Medium	57
9.5.5	Subjective Assessment	57
9.6	Assessment of the Application of the FDTs	57
10	Abracadabra Service and Protocol Example	72
10.1	Informal Description	72
10.1.1	Introduction	72
10.1.2	Service Description	72
10.1.3	Protocol Description	72
10.1.4	Communications Medium Service Description	73
10.1.5	Model	73
10.2	Deficiencies in the Informal Description	73
10.2.1	Flow Control	73
10.2.2	Premature Transmission of DT	74
10.2.3	Stopping Retransmission on Error	74
10.2.4	Retransmission Limit and Period	74
10.2.5	Repeated ConReq	74
10.2.6	DR when Disconnected	74
10.2.7	Connection Refusal	74
10.2.8	Connection Refusal	75
10.2.9	Ignoring Out-of-sequence Data	75
10.3	Estelle Description	75
10.3.1	Architecture of the Formal Descriptions	75
10.3.2	Explanation of Approach	76
10.3.3	Formal Description of the Service	77
10.3.4	Formal Description of the Protocol	79
10.3.5	Subjective Assessment	85
10.4	LOTOS description	85
10.4.1	Architecture of the Formal Descriptions	85
10.4.2	Explanation of Approach	86
10.4.3	Formal Description of the Service	86

10.4.4 Formal Description of the Protocol	91
10.4.5 Subjective Assessment	98
10.5 SDL Description	98
10.5.1 Architecture of the Formal Descriptions	98
10.5.2 Explanation of Approach	99
10.5.3 Formal Description of the Service	99
10.5.4 Formal Description of the Protocol	99
10.5.5 Subjective Assessment	99
10.6 Assessment of the Application of the FDTs	116
11 A Transport Protocol Example	117
11.1 Informal Description	117
11.1.1 Origins	117
11.1.2 Transport Functions	117
11.1.3 Connection Establishment and Termination Procedures	118
11.1.4 Description of Data Transfer Procedures	118
11.1.5 Treatment of Procedure Errors	119
11.1.6 Formats	119
11.1.7 Invalid TPDUs	124
11.2 Deficiencies in the Informal Description	125
11.2.1 Service Definitions	125
11.2.2 Description of Procedures	126
11.2.3 Protocol Classes	126
11.2.4 Missing Definitions	126
11.2.5 Unspecified Functions	127
11.2.6 Non-Use of Concatenation	127
11.2.7 Responding Address	127
11.2.8 Multiple SAP Connections	127
11.2.9 Reaction to Incorrect TCA	127
11.3 Estelle Description	127
11.3.1 Architecture of the Formal Description	127
11.3.2 Explanation of Approach	128
11.3.3 Formal Description	129
11.3.4 Subjective Assessment	138
11.4 LOTOS Description	138
11.4.1 Structure of the Formal Description	138
11.4.2 Explanation of Approach	139
11.4.3 Formal Description	140
11.4.4 Subjective Assessment	171
11.5 SDL Description	171
11.5.1 Architecture of the Formal Description	171
11.5.2 Explanation of Approach	172
11.5.3 Formal Description	172
11.5.4 Subjective Assessment	172

11.6 Assessment of the Application of FDTs	172
Annexes	196
A Bibliography	196
A.1 International Standards	196
A.2 Documents	196
B FDT Characteristics	196
B.1 Specifications and Implementations	196
B.2 Formal Specifications	196
B.3 Levels of Abstraction	196
B.4 FDT Terms	197
B.4.1 Formalisation	197
B.4.2 Abstraction	197
B.4.3 Specification, Description, and Implementation	197
B.4.4 Model	197
B.4.5 Interpretation	198
B.4.6 Constructive	198
B.4.7 Information	198
B.4.8 Action	198
B.4.9 Interaction	198
B.4.10 Composition	198
B.4.11 Non-Determinism	198
C FDT Objectives	198
C.1 Scope of Application	198
C.2 General Requirements	198
C.3 Appropriate Level of Abstraction	199
C.4 Design Support	199
C.5 Implementation Support	199
D Evaluating Formal Descriptions	200
D.1 Layer-Independent Checklists	200
D.1.1 General	200
D.1.2 Service Descriptions	200
D.1.3 Protocol Descriptions	200
D.2 Layer-Independent and FDT-Dependent Checklists	200
D.2.1 General	200
D.2.2 Description of a Single Object	200
D.2.3 Description of Several Interconnected Objects	200
D.2.4 Different Descriptions of Same Object	200
D.3 Verification Methods and Tools	201
D.4 Validation Methods and Tools	201

List of Figures

4.1 Development through Refinement	3
5.1 Typical Layout of an Example	6
8.1 Architecture of the Daemon Game in Estelle	23
8.2 Alternative Architecture of the Daemon Game in Estelle	25
8.3 SDL Specification of Daemon Game	34
9.1 Transmitter Window Parameters	38
9.2 Receiver Window Parameters	38
9.3 Architecture of the Sliding Window Protocol in Estelle	40
9.4 Architecture of the Sliding Window Protocol in LOTOS	45
9.5 Outline Decomposition of the Sliding Window Protocol in LOTOS	45
9.6 Processes of the Sliding Window Protocol in LOTOS	46
9.7 Outline Decomposition of Sliding Window Medium in LOTOS	46
9.8 Processes of Sliding Window Medium in LOTOS	46
9.9 SDL Specification of Sliding Window Protocol	58
9.10 SDL Specification of Sliding Window Medium	69
10.1 Relationship between Abracadabra Service Primitives	72
10.2 Abracadabra Protocol Data Units	72
10.3 Communications Medium Service Primitives	73
10.4 Abracadabra Service and Protocol Model	74
10.5 Architecture of the Abracadabra Service in Estelle	75
10.6 Architecture of the Abracadabra Protocol in Estelle	75
10.7 Outline Decomposition of the Abracadabra Service in LOTOS	85
10.8 Outline Decomposition of the Abracadabra Protocol in LOTOS	85
10.9 SDL Specification of Abracadabra Service	100
10.10 SDL Specification of Abracadabra Protocol	107
11.1 Receiving Terminal Reaction to TCR Addressing Options	119
11.2 Calling Terminal Reaction to TCA Addressing Options	119
11.3 Parameter Element Coding Structure	120
11.4 General Block Structure	120
11.6 Transport Connection Request Block	120
11.5 Transport Layer Block Types	121
11.7 Extended Addressing	122
11.8 Transport Data Block Size Parameter	122
11.9 Transport Connection Accept Block	122
11.10 Transport Connection Clear Block	123
11.11 Additional Clearing Information Parameter	123
11.12 Transport Block Reject Block	123
11.13 Rejected Block Parameter	123

11.14 Transport Data Block 124

11.15 Architecture of A Transport Protocol in Estelle 128

11.16 Constraint-Oriented Decomposition of a Transport Protocol Entity . 139

11.17 Decomposition of Process TPEConnection 139

11.18 SDL Specification of A Transport Protocol 173

B.1 Domain of Applicability of an FDT 197

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.

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).

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

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**.

¹To 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

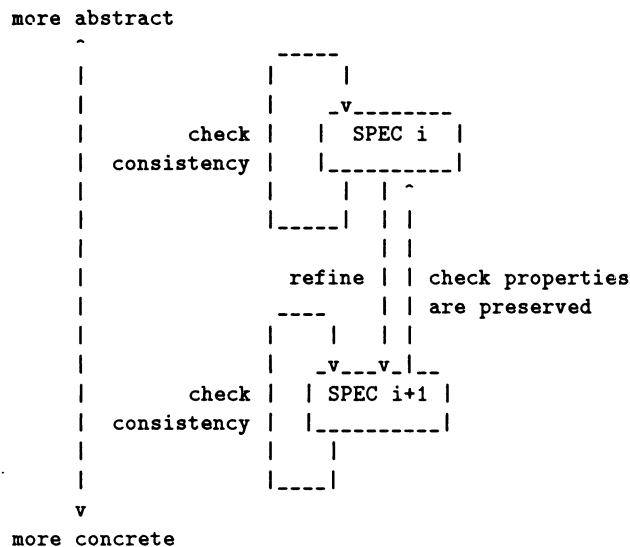


Figure 4.1: Development through Refinement

- 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 step-wise 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/animations, 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:

- 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
- Event1**, etc. are the names of the signals interchanged at the Service Access Point; and
- ServiceUser** is the User of the Service Access Point; and
- 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:

- 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 **ADDRESS**, 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:

```
NEWTYPENAME Typename
ENDNEWTYPENAME;
```

For example:

```
NEWTYPENAME QoSSet Set (QoS)
ENDNEWTYPENAME;
```

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 **ADDRESS**, 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 : UserData_Type);
        DISCONNECTrequest;
    by provider:
        CONNECTindication(Addr1, Addr2, QOS);
        CONNECTconfirm(Addr1, Addr2, QOS);
        DATAindication(UserData : UserData_Type);
        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 **MPTType** 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:

```
NEWTYPER Sdu
INHERITS OctetString ALL;
ENDNEWTYPER;
```

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 checked 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;
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:

- U**, **L1**, and **L2** are the gates for communication at the supported Service and the two underlying Services respectively; and
- USapSet**, **L1SapSet**, and **L2SapSet** are the sets of Service Access point identifiers for the corresponding Services; and
- ProtocolEntities** is the composed behaviour of the individual Protocol Entities; and
- 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 UserData Type ... ENDNEWTYPE;

SIGNAL ConReq, DatReq (UserData Type), ... ;

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:

- USap** is the channel representing the Service Access Point of the Layer, and conveying the signals contained in the signallists **ToUser** and **FromUser**; and
- MSap** is the channel representing the Service Access Point of the underlying Layer, and conveying the signals **UnitReq** and **UnitInd**; and
- 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:

- Boolean** and **OctetString** are the standard library data types; and
- DtPdu** represents a Data Transfer Protocol Data Unit; and
- Dt** is used to construct these Data Transfer Protocol Data Units; and
- DtEot** selects the End of Transfer flag; and
- 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:

```

NEWTYPED 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:

- U** and **L** are the gates for communication at the Upper and Lower Services; and
- USap** refers to the Upper Service Access Point identifier, and **LSap** refers to the Lower Service Access Point identifier; and
- UEp** refers to the Upper Service Endpoint identifier; and
- 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 interaction 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:

- U** and **L** are the gates for communication at the Upper and Lower Services; and
- 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
- Dt** is used to construct a datagram from the source address, destination address, and user data; and
- 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 (
      ConcatenateSdu (Pdu2, Sdu)))

endtype

```

where:

- SduType** defines a Service Data Unit of the underlying Service; and
- OctetString** is the standard library sort, used for Protocol Data Units and Service Data Units of the underlying Service; and
- <>** is the representation of an empty octet string; and
- ConcatenateSdu** appends a Protocol Data Unit to a Service Data Unit, yielding a new Service Data Unit; and
- 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 inter-working 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.

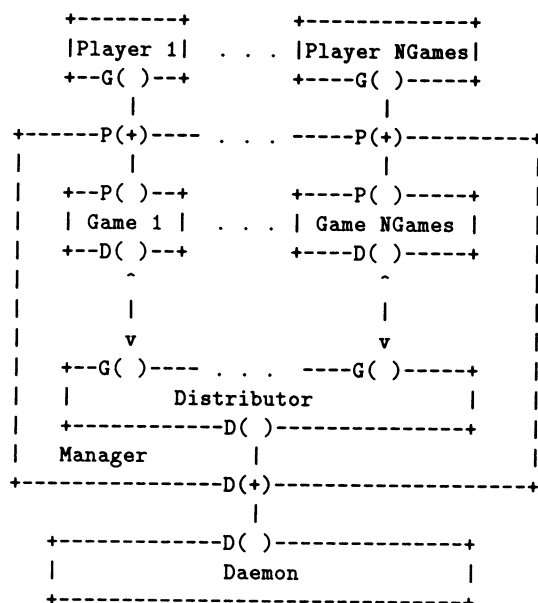


Figure 8.1: Architecture of the Daemon Game in Estelle

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;
    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 }

```

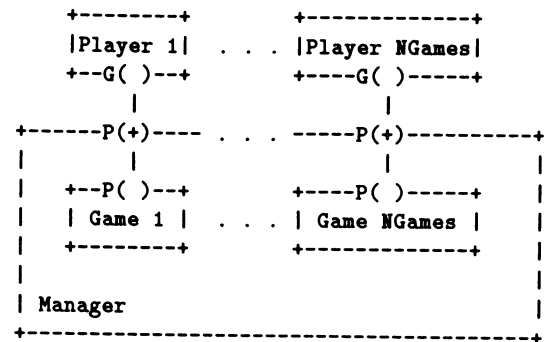


Figure 8.2: Alternative Architecture of the Daemon Game in Estelle

```

    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;
  (
    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 **Datatypesdef** 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 Datatypesdef;

```

```

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:

- How difficult it is to be precise about even simple thing.
- 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.
- How easy it is to be unclear about the responsibilities of different parts of a system, and how these parts should view each other.
- 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.

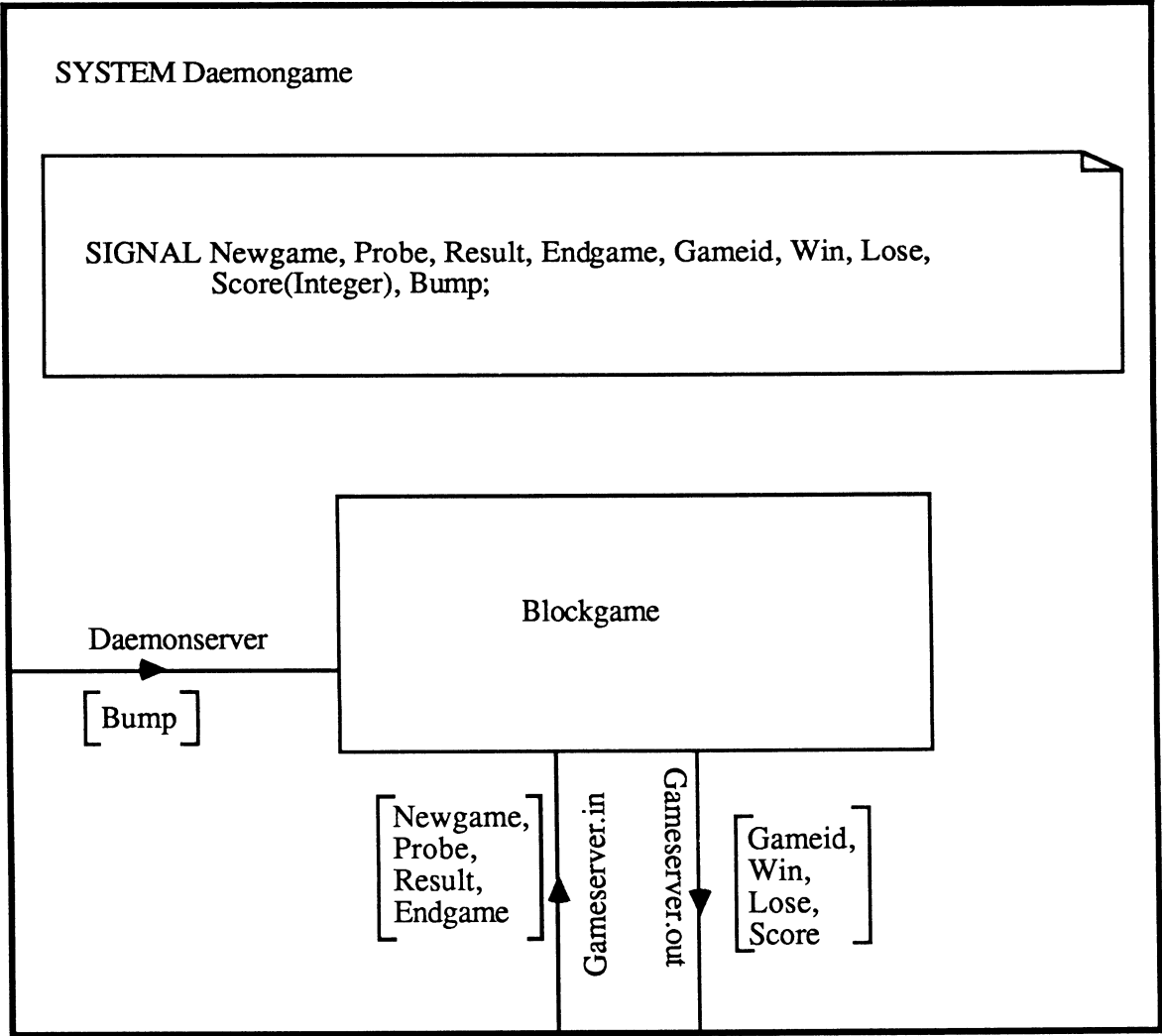


Figure 8.3: SDL Specification of Daemon Game